Hardware neuromorphic learning systems utilizing memristive devices

Michael Soltiz

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation
Hardware Neuromorphic Learning Systems Utilizing Memristive Devices

by

Michael Soltiz

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Engineering

Supervised by

Assistant Professor Dr. Dhireesha Kudithipudi
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August 2012

Approved by:

Dr. Dhireesha Kudithipudi, Assistant Professor
Thesis Advisor, Department of Computer Engineering

Dr. Shanchieh Jay Yang, Associate Professor
Committee Member, Department of Computer Engineering

Dr. Zhaolin Lu, Assistant Professor
Committee Member, Department of Microsystems Engineering
Abstract

Hardware Neuromorphic Learning Systems Utilizing Memristive Devices

Michael Soltiz

Supervising Professor: Dr. Dhireesha Kudithipudi

As the efficiency of neuromorphic systems improves, biologically-inspired learning techniques are becoming more and more appealing for various computing applications, ranging from pattern and character recognition to general purpose reconfigurable logic. Due to their functional similarities to synapses in the brain, memristors are becoming a key element in the hardware realization of perceptron-based learning systems. By pairing memristive devices with a perceptron-based neuron model, previous work has shown that an efficient and low area neural logic block (NLB) can be developed. However, the use of a simple threshold activation function has limited the set of learnable functions for a single block, resulting in the need for multiple layers to implement certain functions. This complicates the training process, decreases the scalability of the system, and increases the overall energy and delay of large networks.

In this work, three novel NLB designs are presented that overcome the limitations of previous hardware NLBs. First, an Adaptive Neural Logic Block (ANLB) and Robust Adaptive Neural Logic Block (RANLB) are proposed. By integrating an adaptive activation function into a perceptron model, these designs are capable of rapidly learning any function in a single layer. Next, a Multi Threshold Neural Logic Block (MTNLB) is proposed in which a static activation function is used to obtain the same functionality with minimal overhead.
Using a Verilog-AMS model of a physical memristor, the proposed NLBs are applied to implement both reconfigurable logic and an Optical Character Recognition (OCR) system. When considering the MTNLB as a building block for ISCAS-85 benchmark circuits, it provides EDP improvements of over 90 percent over a standard LUT implementation on all benchmark circuits and up to a 99 percent improvement over a threshold NLB implementation. As a compromise, the ANLB and RANLB provide less of an EDP improvement in a static system, but achieve faster training convergence times for all functions. To show how the proposed design can simplify an OCR application, a simple 8x8 digit recognition system is developed. Using only four 16-input NLBs for each digit, the system is able to develop a model of each digit in only 90 us and correctly classify the majority of test images.
Contents

Abstract ................................................................. iii

1 Introduction ......................................................... 1
  1.1 Background on Biological Models ......................... 1
  1.2 Neuromorphic Learning Systems ......................... 4
    1.2.1 Limitations on the Learnable Set ................. 7
    1.2.2 Synapse Implementation ........................... 9
  1.3 Memristive Devices ......................................... 9
    1.3.1 Memristors as Synapses .......................... 14
  1.4 Thesis Objective ........................................ 14

2 Related Work ..................................................... 16
  2.1 Memristor Integration Into NLB Designs ............... 16
    2.1.1 TTGA Block ..................................... 17
  2.2 NLBs With Enhanced Learning Capabilities ........... 19
    2.2.1 Adaptive Activation Function .................... 20
  2.3 Summary .................................................. 22

3 Proposed Neural Logic Block Designs ..................... 23
  3.1 Adaptive Neural Logic Block (ANLB) ..................... 27
  3.2 Robust Adaptive Neural Logic Block (RANLB) ......... 28
  3.3 Multi-Threshold Neural Logic Block (MTNLB) ......... 30
  3.4 Comparison .............................................. 32
List of Tables

1.1 Truth table for an NLB implementation of an AND function 5
1.2 Weight changes while training an NLB to OR functionality 6
1.3 Truth table for an NLB implementation of an OR function 6
1.4 Nonlinearly separable two-input functions 7
1.5 Memristor parameters 12

3.1 Summary of proposed NLB designs 33
3.2 Transistors count comparison 33

4.1 4-input NLB training times 38

5.1 NLB outputs for classifying Test Image 1 51
5.2 NLB outputs for classifying Test Image 2 51
5.3 NLB outputs for classifying Test Image 3 51
5.4 NLB outputs for classifying Test Image 4 51
5.5 NLB outputs for classifying Test Image 5 51
5.6 NLB outputs for classifying Test Image 6 51
5.7 NLB outputs for classifying Test Image 7 52
5.8 NLB outputs for classifying Test Image 8 52
5.9 NLB outputs for classifying Test Image 9 52
5.10 NLB outputs for classifying Test Image 10 52
List of Figures

1.1 A basic perceptron model ........................................ 2
1.2 Hebbian Learning Theory example ............................... 3
1.3 Stochastic gradient descent training methodology ............. 4
1.4 Linearly separable vs. nonlinearly separable functions ......... 6
1.5 Multilayer XOR implementation .................................. 8
1.6 Fundamental circuit elements and variables ..................... 10
1.7 Regions of a memristor ............................................ 12
1.8 I-V curve for our memristor model .............................. 12

2.1 TTGA block schematic ........................................... 17
2.2 TTGA block with training circuitry ............................... 18
2.3 Adaptive activation function ..................................... 20
2.4 Activation function for XOR implementation .................... 21

3.1 Block diagram of a single NLB .................................. 24
3.2 Weighting/Range Select circuit for proposed NLB designs ... 25
3.3 Current comparator circuit ....................................... 25
3.4 Ideal activation function shapes ................................ 27
3.5 Activation function circuit for ANLB ............................ 28
3.6 Activation function circuit for RANLB ......................... 29
3.7 Activation function curve for MTNLB ......................... 30
3.8 Activation function circuit for MTNLB ......................... 31

4.1 Example training waveform .................................... 38
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2 Procedure to obtain training waveforms</td>
<td>39</td>
</tr>
<tr>
<td>5.1 Procedure to obtain EDP data</td>
<td>43</td>
</tr>
<tr>
<td>5.2 EDP results for ISCAS-85 benchmarks</td>
<td>44</td>
</tr>
<tr>
<td>5.3 Average power for 45nm and 16nm implementations</td>
<td>46</td>
</tr>
<tr>
<td>5.4 OCR block in which each NLB analyzes one row of pixels</td>
<td>47</td>
</tr>
<tr>
<td>5.5 OCR block in which each NLB analyzes one quadrant of pixels</td>
<td>49</td>
</tr>
<tr>
<td>5.6 Set of images used to train the OCR system</td>
<td>49</td>
</tr>
<tr>
<td>5.7 Set of images used to test the OCR system</td>
<td>50</td>
</tr>
<tr>
<td>A.1 ANLB learning an XOR function</td>
<td>56</td>
</tr>
<tr>
<td>A.2 ANLB learning a NAND function</td>
<td>57</td>
</tr>
<tr>
<td>A.3 ANLB learning a COUT function</td>
<td>58</td>
</tr>
<tr>
<td>A.4 RANLB learning an XOR function</td>
<td>59</td>
</tr>
<tr>
<td>A.5 RANLB learning a NAND function</td>
<td>59</td>
</tr>
<tr>
<td>A.6 RANLB learning a COUT function</td>
<td>60</td>
</tr>
<tr>
<td>A.7 MTNLB learning a NAND function</td>
<td>60</td>
</tr>
<tr>
<td>A.8 MTNLB learning a COUT function</td>
<td>61</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In 1990, Carver Mead speculated that the human brain is a factor of $10^9$ more efficient than the digital technology of the time and a factor of $10^7$ more efficient than the best strictly-digital technology imaginable [16]. In an attempt to reduce this gap, Mead introduced the concept of neuromorphic systems – electronic systems containing analog circuits designed to mimic neurobiological architectures present in the nervous system. By using analog circuit elements as computational primitives and replicating structures found in biological systems, neuromorphic systems are designed to achieve higher levels of robustness, fault tolerance, and adaptivity within computing applications. Today, a large subset of neuromorphic systems aim to implement biologically-inspired learning algorithms, such as Hebbian Learning [11] and Brain-State-In-A-Box [9]. By mimicking the learning process found in the human brain, these systems are capable of adapting their functionality in real-time. This ability is very appealing for various applications, such as Optical Character Recognition (OCR) and general purpose reconfigurable logic.

1.1 Background on Biological Models

The most recent implementation of a biological model of the human brain is comprised of 100 billion neurons interconnected through a vast network of 100 trillion synapses [6]. Each individual neuron has the ability to produce
Figure 1.1: A basic perceptron model, summarizing the functionality of a neuron.

an electrical spike at its output that can excite or inhibit other neurons or bodily functions. The primary logical function of a neuron is to determine when to produce these electrical spikes. Each input signal is connected to a neuron through a synapse that has a weight associated with it. At its input node, a neuron performs the weighted summation of all input signals. This can be modeled using the equation

$$s = \sum_{i=0}^{n} w_i \times x_i,$$

where $s$ is the weighted summation of the inputs, $n$ is the number of inputs, $x_i$ is the voltage at input $i$, and $w_i$ is the weight of synapse $i$. Next, an activation function, $Y(s)$, is used to determine whether or not the neuron should fire an output voltage spike. For computational simplicity, the activation function is generally modeled as a threshold function,

$$Y(s) = \begin{cases} 0 & s < T \\ 1 & s \geq T \end{cases},$$

where $T$ is some threshold value. This common neuron model, known as the perceptron, is summarized in Fig. 1.1.

In 1949, Donald Hebb proposed a theory that learning is achieved solely
through the adjustment of synaptic weights associated with connections between neurons in the brain. According to this learning theory, Hebbian Learning, when a given neuron repeatedly and persistently excites another neuron, the synaptic connection between the two neurons is strengthened [11]. Anti-Hebbian Learning extends this theory to state that synaptic connections also weaken over time if a neuron’s spiking does not correlate with the excitation of another neuron [2].

A simple example of how learning is achieved through this procedure can be described using Pavlov’s classic conditioning experiment [22]. Naturally, a dog salivates when it tastes food, in order to aid in digestion. In this experiment, Pavlov rang a bell prior to feeding his dogs for an extended period of time. Eventually, the dogs were conditioned to begin salivating whenever they heard a bell because it was associated with food in their mind. Fig. 1.2 shows how this interesting phenomenon can be explained using Hebbian Learning theory. During this experiment, the spiking of bell detector neurons was correlated to the natural spiking of salivation activator neurons every time the dogs were fed. As a result, the synaptic weights between bell detectors and salivation activators were strengthened over time. Eventually, these connections grew strong enough that the weighted input from a bell detector alone was greater than a salivation activator’s activation threshold. As a result, the salivation activator was excited whenever the bell
detector produced a spike, regardless of whether or not the food detectors spiked.

In this configuration, the human brain is able to achieve levels of adaptivity, robustness, and efficiency that are unimaginable in computing and digital technology. From a neuromorphic perspective, computing systems can be designed to mimic this architecture and obtain improvements in a variety of computing applications.

### 1.2 Neuromorphic Learning Systems

In biologically-inspired neuromorphic systems, the functionality of a single neuron with synapses at each input is modeled in a neural logic block (NLB). These NLBs are interconnected in large networks to implement the desired functionality for a specific computing application.

Using an error-based training mechanism, it is fairly straightforward to train an individual NLB to implement different logic functions. Fig. 1.3 outlines the stochastic gradient descent process, a simple error minimization algorithm that can be applied to train a single NLB [17]. During this training process, the NLB is given all possible combination of inputs for a given
amount of time, coupled with the expected output, $Y_{\text{exp}}$. If the actual output, $Y$, is different from the expected output, $Y_{\text{exp}}$, the synaptic weights corresponding to the high inputs are adjusted. This process is repeated until all input combinations produce the correct output.

Consider a two-input NLB with a threshold activation function where $T = 4$ (Eq. 1.2). Assume that each synaptic weight, $w_i$, is initially set to 2, ranging from 1 to 4, and can be incremented or decremented by 1 in a single training cycle. As Table 1.1 shows, the NLB implements a two-input AND function in this initial state. However, through stochastic gradient descent, this NLB can be modified to implement an OR function. In each training cycle, the weights corresponding to a high input are modified using the following equations:

$$\Delta W = \begin{cases} 
-1 & Y > Y_{\text{exp}} \\
0 & Y = Y_{\text{exp}} \\
1 & Y < Y_{\text{exp}} 
\end{cases} \quad (1.3)$$

$$W_{i,\text{new}} = W_{i,\text{old}} + \Delta W. \quad (1.4)$$

Table 1.2 shows the weight adjustments during the training process. After looping through all input sets twice, both input weights are set to 4. With these weights, the NLB implements a two-input OR function, as shown in Table 1.3.
Table 1.2: Weight changes while training a two-input NLB to OR functionality through the stochastic gradient descent algorithm.

<table>
<thead>
<tr>
<th>Step</th>
<th>Inputs $X_1$</th>
<th>Inputs $X_2$</th>
<th>Target $Y_{exp}$</th>
<th>Output $Y$</th>
<th>Weight Changes $\Delta w_1$</th>
<th>Weight Changes $\Delta w_2$</th>
<th>New Weights $w_1$</th>
<th>New Weights $w_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1.3: Truth table for a two-input NLB with final input weights of 4 and an activation threshold of 4, implementing an OR function.

<table>
<thead>
<tr>
<th>Inputs $x_1$</th>
<th>Inputs $x_2$</th>
<th>Weights $w_1$</th>
<th>Weights $w_2$</th>
<th>Weighted Sum $\Sigma x_i \times w_i$</th>
<th>Output $Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 1.4: Representation of (a) linearly separable function and (b) non-linearly separable function. In linearly separable functions, when each input is plotted on an axis, only one hyperplane can separate input combinations with a high output from low input combinations.
Table 1.4: Truth table for all non-linearly separable two-input functions.

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Function Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(A \oplus B)</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0 0 1 1 1</td>
</tr>
<tr>
<td>0 1</td>
<td>1 1 0 1 0 0</td>
</tr>
<tr>
<td>1 0</td>
<td>0 1 0 0 0 1</td>
</tr>
<tr>
<td>1 1</td>
<td>1 0 0 1 1 1</td>
</tr>
</tbody>
</table>

1.2.1 Limitations on the Learnable Set

While perceptron-based NLBs have proven to show high success rates and fast training convergence, they also have their shortcomings. The major shortcoming of these systems is that the choice of an activation function can greatly limit the set of learnable functions for a single NLB. As previously described in Section 1.2, a two-input NLB with a threshold activation function can easily learn an AND or OR function. However, in order for a function to be learnable by this NLB, it must meet two requirements. First of all, when both inputs are low, the output must be low. Because the weighted summation of the inputs is guaranteed to be 0 when all inputs are 0, it is impossible to meet the threshold. For this reason, NAND and NOR functions cannot be learned in a single block unless inverted inputs are available. Furthermore, in order to learn a function in a single NLB, the function can only have one decision boundary. This property, known as linear separability, requires that the value of the output only changes once as the weighted sum of the inputs increases. If uniform input weights are assumed, an OR function’s output switches from low to high when one input goes high, then remains high as additional inputs become high. Because there is only one change in the output as the number of high inputs increases, this function is linearly separable. However, an XOR function’s output switches from low to high when one input becomes high, then switches from high to low when two inputs become high. This function, on the other hand, is not linearly separable and cannot be learned by an NLB with a threshold activation function. A graphical representation of this comparison is given in Fig. 1.4. In this representation, each input is plotted on a separate axis. In order for the function to be linearly separable, a single hyperplane must be able to...
Figure 1.5: A four-input XOR function, implemented using only four-input NLBs with threshold activation functions.

separate all low outputs from high outputs on this graph. Table 1.4 lists all two-input functions that are not linearly separable, and thus cannot be learned by a single NLB with a threshold activation function. If it is assumed that inverted inputs are also available, this list is reduced to simply XOR and XNOR functions. However, as the number of inputs increases, the number of non-linearly separable functions increases drastically.

To overcome the limitations on the learnable set of functions for a single NLB, perceptron-based systems generally implement non-linearly separable functions in multiple layers. For example, if we assume inverted inputs are available, a two-input XOR function, $A \oplus B$, can be implemented in three NLBs by decomposing the function into $AB' + A'B$. For a four-input NLB, the worst-case function is a four-input XOR function. As Fig. 1.5 shows, this function requires the use of eleven blocks connected in three layers. This requirement has several negative impacts on overall system performance. First of all, the need for multiple layers of NLBs increases both latency in a static system and overall training time during adaptation.
Furthermore, the need for multiple NLBs to implement a single function introduces the need for significantly more NLBs overall. In a large neuromorphic system, the need for more NLBs can increase the overall power dissipation and area overhead. This, in turn, limits the scalability of the system. Because the training of a single NLB is not guaranteed to converge for all general functions, the complexity of training logic is also likely to be increased when considering a full neuromorphic system.

1.2.2 Synapse Implementation

While the concept of neuromorphic systems was originally aimed at hardware implementation, circuit design challenges have steered the majority of work with perceptron-based systems toward a software approach. Because synapses are so vast and vital within this type of system, it is essential for these components to be modeled with very low area, power, and timing overhead in hardware. Unfortunately, this requirement is a non-trivial task using active CMOS components. In software, limited system resources and sequential execution put a large damper on training convergence time and the scalability of perceptron-based learning systems. To more accurately model the functionality of a brain and truly see the benefits of a large neural network, it is desirable to develop a hardware implementation with robust, efficient, and fully-functional synapse models. As the following section shows, the recent realization of the memristor removes this bottleneck by providing neuromorphic systems with the ability to model a synapse with a single passive, two-terminal device.

1.3 Memristive Devices

In 1971, Leon Chua speculated that, by principles of symmetry, a fourth passive, two terminal circuit element must exist [5]. Given the four fundamental circuit variables (electric current $i$, voltage $v$, charge $q$, and magnetic flux
Figure 1.6: The four fundamental two-terminal circuit elements (resistor \([R]\), capacitor \([C]\), inductor \([L]\), and memristor \([M]\)) and how they relate the four fundamental circuit variables (current \([i]\), voltage \([v]\), charge \([q]\), and magnetic flux \([\phi]\)).

\[\delta q = i \times \delta t\]  \hspace{1cm} (1.5)

\[\delta \phi = v \times \delta t\]  \hspace{1cm} (1.6)

Out of the four remaining pairs of fundamental circuit variables, three are related linearly by the fundamental circuit elements, resistors, capacitors, and inductors, as described by the equations

\[\delta v = R \times \delta i\]  \hspace{1cm} (1.7)

\[\delta q = C \times \delta v\]  \hspace{1cm} (1.8)

\[\delta \phi = L \times \delta i,\]  \hspace{1cm} (1.9)
where \( R \) is resistance, \( C \) is capacitance, and \( L \) is inductance. After defining these relationships, it follows that a fourth fundamental circuit element should exist to define the relationship between the final pair of fundamental variables, electric charge and magnetic flux (Fig. 1.6). Due to the theoretical functionality of such element, Chua called the fourth circuit element the memristor (an abbreviation for memory resistor).

Assuming that the memristor behaves in the same manner as the three existing circuit elements, the memristor would relate magnetic flux to electric charge as follows,

\[
\delta \phi = M \times \delta q, \tag{1.10}
\]

where \( M \) represents an arbitrary quantity called memristance. From this equation, one can see that the memristance of a device is controlled by the electrical charge on the device. Based on the definition of electric current (Eq. 1.5), the electric charge at a given time, \( t_0 \), is the time integral from \( t = -\infty \) to \( t = t_0 \) of the current passing through an element. As a result, the memristance of a device is dependent upon the past history of the current passing through the device. Furthermore, using Eq. 1.5 and Eq. 1.6, Eq. 1.10 can be modified to the following form:

\[
v(t) = M(q) \times i(t) \tag{1.11}
\]

In this form, it becomes apparent that instantaneous memristance has the same units and physical effect as resistance. However, as it is dependent on the time integral of current, the value of the memristance changes as current passes through the device. The result is that a memristor will act as a variable resistor with a natural memory capability, whose resistance can be increased or decreased by applying a negative or positive voltage across the device.

While Chua’s observations were largely ignored for the remainder of the 20th century, a new interest in memristors was sparked in 2008 when researchers at Hewlett-Packard research laboratories discovered a device with
properties that matched Chua’s description of a memristor while working with thin-films of titanium dioxide [28]. Based on their findings, HP Labs fabricated the first physical memristor on a semiconductor film that consists of a region with a high concentration of dopants (low resistance, $R_{on}$) and a region with a low concentration of dopants (high resistance, $R_{off}$) connected in series. When an external bias voltage, $v(t)$, is applied across the device, the charged dopants naturally drift, moving the boundary between the two regions (Fig. 1.7).

HP’s initial analysis of memristive device behavior led to a simple model

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{on}$</td>
<td>1 MΩ</td>
</tr>
<tr>
<td>$R_{off}$</td>
<td>50 MΩ</td>
</tr>
<tr>
<td>$V_{th,pos}$</td>
<td>1.25 V</td>
</tr>
<tr>
<td>$V_{th,neg}$</td>
<td>-1.00 V</td>
</tr>
<tr>
<td>$t_{pos}$</td>
<td>2 ms</td>
</tr>
<tr>
<td>$t_{neg}$</td>
<td>0.1 ms</td>
</tr>
</tbody>
</table>
with ohmic electronic conductance and linear ionic drift in a uniform field. However, more extensive research on the behavior of memristors revealed that ionic drift within memristive devices is truly non-linear in nature and more accurate models were developed to account for this. Our research group has developed a nonlinear piecewise Verilog-AMS model of a memristor based on a physical metal-oxide device. Experimental data has shown that the fabricated device’s memristance does not change until the magnitude of the voltage drop across the memristor, \( V_m \), exceeds certain threshold voltages, \( V_{th,pos} \) and \( V_{th,neg} \). When the threshold voltages are exceeded, the memristance changes non-linearly based on the magnitude and timing of the bias voltage pulse [23]. To match the I-V curve of a physical memristor, the memristance change of the model is characterized by the equation

\[
M = \begin{cases} 
M - \frac{(\delta r \times \delta t \times V_m)}{(t_{pos} \times V_{th,pos})} & : V_m \geq V_{th,pos} \\
M + \frac{(\delta r \times \delta t \times V_m)}{(t_{neg} \times V_{th,neg})} & : V_m \leq V_{th,neg},
\end{cases}
\]

(1.12)

where \( \delta r \) is \( R_{off} - R_{on} \), \( \delta t \) is the minimum time step interval, \( V_{th,pos} \) is the positive voltage threshold of the device, \( V_{th,neg} \) is the negative voltage threshold of the device, \( t_{neg} \) is the time required to increase the memristance from \( R_{on} \) to \( R_{off} \), and \( t_{pos} \) is the time required to decrease the memristance from \( R_{off} \) to \( R_{on} \). Table 1.5 gives the values considered for each parameter in order to match physical device properties presented in [12]. The resulting I-V curve produced by the model is shown in Fig. 1.8. The hysteretic pattern of the curve is a result of the changing memristance that relates voltage to current. After a positive voltage has been applied to the memristor, the memristance is low, resulting in a larger flow of current. On the contrary, the memristance is high after a negative voltage is applied, resulting in a smaller flow of current. Because a voltage drop of 0V across the memristor will always result in no flow of current, the I-V curve of the memristor will always pass through the origin.
1.3.1 Memristors as Synapses

The natural memory capability of memristors has made these devices very appealing for various applications, such as non-volatile memory [15], signal processing [19], and control systems [18]. However, the functional similarities between memristors and biological synapses make them especially appealing for neuromorphic applications.

When a memristor is held in a constant state, it acts as a resistor in which the current through the device is described as:

\[ I = \frac{V}{M}. \] (1.13)

Conceptually, a memristor can be thought of as a synapse with an input of \( V \), an output of \( I \), and a weight of \( \frac{1}{M} \) [27]. Just as the weight of a synapse can be modified to strengthen or weaken a connection, the memristance of a memristor can be adjusted by applying a positive or negative super-threshold voltage drop across its terminals. As a result, a biological synapse is modeled accurately using a single, passive two-terminal device. By successfully modeling a synapse with low area and power overhead, the use of memristors in NLBs makes the hardware implementation of large neuromorphic systems practical and easily scalable.

1.4 Thesis Objective

The integration of memristive devices into perceptron-based NLB designs provides systems with an efficient, low area, and low power synapse implementation. However, the use of a threshold activation function within perceptron-based NLBs limits the set of learnable functions for a single NLB to linearly separable functions. This limitation results in multiple layers to implement certain functions and, in turn, complicates the training process, decreases scalability, and increases overall delay and energy of large-scale hardware-based neuromorphic systems. To truly see the benefits
of hardware implementations of neuromorphic learning systems, it is critical to develop an NLB with both efficient synapse implementations and a neuron implementation that is capable of learning any function in a single layer.

Recent research in neuroscience suggests that the presence of neuromodulators in the brain modify the activation function of individual neurons during the learning process [25], [7]. This work leverages that observation, proposing three hardware implementations of perceptron-based NLBs, which are capable of learning all logic functions in a single layer. All three designs combine memristive synapses with a novel perceptron design that eliminates the limitations on the trainable set of functions. First, a perceptron-based NLB that utilizes a second layer of memristors to represent an adaptive activation function is proposed. Then, a second perceptron-based NLB is proposed that implements the same functionality using digital values to represent the activation function. Finally, a third perceptron-based NLB is proposed with a static activation function and multiple activation thresholds. A demonstration of the proposed designs in the implementation of reconfigurable logic and a simple OCR application for handwritten digits.

To show the benefits of these NLBs, demonstrations of the proposed designs in the implementation of reconfigurable logic and a simple OCR application for handwritten digits are given. The resulting systems show low power, delay, and training time and offer significant simplifications to overall systems.
Chapter 2

Related Work

2.1 Memristor Integration Into NLB Designs

To exploit the natural memory capabilities of memristors, several groups have explored integrating these components into CMOS systems to develop reconfigurable fabrics. In [3], [13], and [29], neural logic blocks are implemented using memristive nanowire crossbar structures. Shifting the focus away from biologically-inspired models, these designs implement a look-up table in which the state of a memristor is used to represent a logical high or low output. While this allows an NLB to implement any logic function, this type of structure is prone to several issues. Sneak path currents within the crossbar result in the need for additional hardware for reading and writing accurately. Memristive crossbar structures also face fabrication difficulties due to their high density. Furthermore, applying small adaptations to the system essentially requires a memory writing process with significant delay. One of the fundamental benefits of neuromorphic systems is the ability to make small modifications to the system’s functionality without requiring the whole system to be reconfigured. This benefit is largely lost by using a LUT-based implementation.
2.1.1 TTGA Block

In [14], Manem et al. propose a perceptron-based NLB that exploits the similarities between memristors and biological synapses. The proposed design is intended for use in a Trainable Threshold Gate Array (TTGA) for reconfigurable logic. Each individual TTGA block uses memristors as synapses and simple CMOS components to implement a perceptron model, as shown in Fig. 2.1. In this design, each input is connected to a perceptron component through a trainable memristor, coupled with an NMOS current mirror to produce an identical current flowing into the summation node from ground. A PMOS current mirror is then used to produce a reference current, $I_{\text{ref}}$, flowing out of the summation node. Based on Kirchhoff’s Law, if the sum of the input currents exceeds the reference current, the voltage at the summation node will be low. A series of inverters is connected to the output to invert this voltage and ensure that TTGA blocks can be cascaded without the inputs of the next stage affecting the current at the output of a block. If, and only if, the total input current from the inputs exceeds the threshold, $I_{\text{ref}}$, the TTGA block outputs a logical high value.

Manem et al. also propose a training mechanism for the TTGA block based
on stochastic gradient descent (Fig. [1.3]). To implement this training mechanism with minimal overhead, [14] offers a training mechanism that is broken up into global and local components, as shown in Fig 2.2. Each input requires its own local trainer to adjust its individual synaptic weight, while only one global trainer is required per perceptron. Consequently, an ideal system would have the majority of the training circuitry in the global trainer with minimal overhead in the local trainers. To work towards this goal, the global trainer compares the actual output to the desired output and determines if training is necessary and the direction of training. The local trainer, on the other hand, simply receives these control signals and a synapse input, $A$. If $A$ is high, indicating that the corresponding input is affecting the current perceptron output, and the global training select signal is high, the local trainer routes the training pulses to the memristor terminals for one period. Otherwise, it simply routes $A$ to the perceptron through the memristor to allow for standard operation. In [23], a similar training mechanism is designed using sub-threshold voltage levels to minimize the power consumption of the system. The results show that this type of system is capable of reaching an energy consumption on the order of femto-joules.

When compared to standard Look-Up Table (LUT) and Capacitive Threshold Logic (CTL) [20] implementations, the TTGA achieves a lower Energy-Delay Product on benchmark circuits. Furthermore, each individual TTGA block uses less than half of the area overhead associated with an LUT.
However, this design also has its shortcomings. Because it is a strictly perceptron-based model and each TTGA block uses a threshold activation function, each individual block can only implement linearly separable functions. As previously described in Section 1.2.1, this has a negative impact on training time, delay, power, and area overhead in a large system. In turn, the TTGA block is impractical to scale to a large system and fails to reach the full potential of a perceptron-based neuromorphic system.

### 2.2 NLBs With Enhanced Learning Capabilities

In software applications, several solutions to the limitations on the learnable set of functions for perceptron-based NLBs have been proposed. In [1], a complex-value neuron (CVN) is proposed that introduces an imaginary component to each synaptic weight. By adding complexity to synapse models, non-linearly separable functions, such as XOR and XNOR, are learnable using a single NLB with a simple threshold activation function. In [24], an architecture is proposed that constructs decision trees comprised of linear threshold units to learn non-linearly separable functions in multiple layers of NLBs. Similarly, [8] proposes an algorithm to train multiple layers of NLBs to implement non-linearly separable functions.

While these solutions prove to learn all logic functions in software applications, each has its shortcomings when considering a hardware implementation. In [26], a circuit that implements an NLB with complex-valued synaptic weights is presented. In order to achieve the desired functionality, two separate weights, $a_i$ and $b_i$ are applied to each input, representing the real and imaginary components of the weight, respectively. Then, two separate summations, $\Sigma a$ and $\Sigma b$ are calculated, each summation is squared, and the squared values are summed together and input to the activation function. While this successfully applies a complex weight to each input, a large amount of overhead is required to implement this functionality in hardware. Furthermore, no training circuitry or algorithm is presented. The training of complex-valued synaptic weights would likely require significantly more
logic, and thus more overhead. Multi-layer solutions proposed in hardware also require an impractical amount of overhead. These solutions still require multiple NLBs to implement a single logic function, and the training algorithms are more complex and difficult to implement in hardware.

2.2.1 Adaptive Activation Function

The most appealing theoretical solution to the limitations of perceptron-based NLBs is the use of an adaptive activation function, as proposed in [21]. This scheme is inspired by recent research in neuroscience that suggests that neuromodulators exist within the brain and aid in learning by modifying the activation function of individual neurons [25], [7]. To replicate this behavior, the activation function is modeled as a piecewise continuous function comprised of the interpolation between \( m \) points, as shown in Fig. 2.3.

![Graph showing adaptive activation function](image)

Figure 2.3: Adaptive activation function, as proposed in [21]

The training algorithm for this activation function is a simple modification of stochastic gradient descent, in which individual points in the activation function are trained up or down in parallel with the synaptic weights, as outlined in Algorithm 1. By training individual points in the activation function up or down, the shape of the activation function can be modified to match the desired function. For example, an XOR function can be implemented by modifying the shape of the activation function to that shown in Fig. 2.4, where the output voltage is only high if the input current is between two distinct threshold values.
Algorithm 1 Training an NLB with an adaptive activation function

\( \alpha = \) Learning rate.

\( f = \) Array of points \((x,y)\) that comprise the activation function. The activation function is modeled as a continuous function constructed by interpolating between these points.

while Output error, \( E > 0 \) do

\( E \leftarrow Y_{exp} - Y \)

for Each input, \( x_i \), with input weight, \( w_i \) do

\( \Delta w_i = \alpha \times x_i \times E \)

\( w_i \leftarrow w_i + \Delta w_i \)

end for

\( x_{total} \leftarrow \sum x_i \times w_i \)

\( i \leftarrow \) index of the point in \( f \) whose \( x \) value is closest to \( x_{total} \)

\( \Delta f = \alpha \times E \)

\( f[i] = f[i] + \Delta f \)

end while

---

Figure 2.4: Activation function for implementing a two-input XOR function.
An adaptive activation function comprised of \( m \) points can be trained to implement any function with \((m - 1)\) decision boundaries. When considering a 4-input logic block, this means an adaptive activation function consisting of 5 points can implement any possible function. Furthermore, the training algorithm for an NLB with an adaptive activation function is extremely efficient. By modifying both the synaptic weights and the shape of the activation function, an NLB is able to learn a desired function in significantly fewer training cycles than if just the synaptic weights were adjusted. When considering a hardware implementation, this scheme becomes even more appealing. In software, adjusting the weight and activation function is a two-step process that increases the time of each training cycle. However, in hardware, these two adjustments can be made in parallel using the same training period. As a result, the adaptation of the activation function does not have any negative impacts on the training time.

### 2.3 Summary

Previous work has shown that the functional similarities between biological synapses and memristive devices can easily be exploited to implement neuromorphic systems efficiently in hardware. However, by using a threshold activation function, previous hardware implementations of biologically-inspired NLBs require multiple layers of NLBs to implement nonlinearly separable functions. While many solutions to this limitation have shown significant improvements in software implementations of neuromorphic systems, very little work has been performed to utilize these solutions in hardware systems. Without these improvements, perceptron-based reconfigurable hardware is difficult to scale for use in a realistic application, such as large-scale reconfigurable logic or pattern or character recognition. However, by integrating these or similar techniques into a robust and area-efficient hardware system, performance can be improved and scalability to larger systems becomes feasible.
Chapter 3

Proposed Neural Logic Block Designs

The ability of memristors to accurately emulate biological synapses with a single passive device makes them very appealing for the hardware implementation of neuromorphic systems. However, to fully exploit the benefits of memristors in this domain, these devices must be combined with a fully functional and efficient neuron model. While a simple perceptron with a threshold activation function can easily learn linearly separable functions in a very low area NLB, the limitations of this model have a negative impact on overall system performance, area, and scalability. To improve these factors, three novel NLB designs are proposed. Each proposed NLB is capable of learning both linearly separable and nonlinearly separable functions in a single layer with minimal area overhead. By eliminating the need for decomposition of nonlinearly separable functions while keeping low area per block, these NLBs not only simplify large-scale neuromorphic systems to improve scalability drastically, but also improve overall energy, delay, and training time by reducing the number of blocks.

The functionality of the proposed NLBs is broken up into two major components, Weighting/Range Select and the Activation Function, as shown in Fig. 3.1. Each input is first passed into the Weighting/Range Select component. This component applies an adjustable weight to each input, calculates the weighted summation of the inputs, and determines which of \( m \) ranges the input current falls into. To indicate the active range, \( m \) active-low select lines are given as output. Then, based on these select signals, the Activation
Function component determines the value of the digital output. Each component also receives training pulses from an external Global Trainer, which indicate when and how to modify the NLB’s functionality during training.

The limitation on the learnable set of functions for an NLB with a threshold activation function stems from the fact that a single comparator is used on the input current range. This essentially divides the input current, \( i \), into two ranges: \( i < I_{ref} \) and \( i \geq I_{ref} \). However, the number of ranges that the input current is divided into directly correlates to the number of decision boundaries the NLB can implement. An NLB with \( m \) input current ranges can learn any function with \( m - 1 \) decision boundaries. The maximum number of decision boundaries an \( n \)-input function can have is \( n + 1 \), present in an \( n \)-input XOR function. So, in order for it to be possible for an \( n \)-input NLB to learn any possible function, the input current must be broken up into \( n + 1 \) ranges.

The hardware realization of a Weighting/Range Select component that overcomes this limitation is given in Fig. 3.2. Within this component, each individual input is passed through a single memristor that is trained to some memristance, \( M \), ranging from \( R_{on} \) to \( R_{off} \). The current flowing through each memristor represents the input voltage weighted by a factor of \( \frac{1}{M} \). Then, all of the weighted inputs are given to a chain of comparators.

An individual comparator is modeled as shown in Fig. 3.3. Each weighted
Figure 3.2: Weighting/Range Select circuit for proposed NLB designs.

Figure 3.3: Current comparator circuit.
input is connected to an NMOS current mirror to produce an identical current flowing from ground to a summation node. This avoids the possibility of current flowing in the reverse direction through the memristors and ensures that each comparator receives the same input current. To reduce the transistor count, the first transistor in each current mirror is shared among all comparators. After passing though the current mirrors, each input current is connected to a common summation node. By Kirchhoff’s Current Law, the total current flowing into this node is equal to the sum of the input currents. Next, a reference current, \( I_{ref} \), is connected to the same node through a PMOS current mirror to produce a current flowing out of the node to \( V_{dd} \). If the reference current is exceeded by the input current, the voltage at this node will be low, and vice versa. This voltage is passed through a series of inverters for buffering, producing both an active-high and active-low signal indicating when the reference current is exceeded.

In the Weighting/Range Select component, the inputs are passed into a chain of \( m - 1 \) comparators with monotonically increasing reference currents, \( I_{ref,1} > I_{ref,2} > I_{ref,m-1} \). At the output of the comparators, a simple thermometer code is obtained. If the output of \( C_i \) is high and the output of \( C_{i+1} \) is low, then the input current, \( i \), is within the range \( I_{ref,i} < i < I_{ref,i+1} \). Using this logic, a series of active-low range select signals are obtained using simple CMOS logic gates.

The introduction of additional input current ranges complicates the activation function of an NLB. In an NLB with a threshold activation function, the output is simply high if \( i \geq I_{ref} \). However, additional complexity is required to determine the output based on \( m \) input current ranges. Each of the proposed NLB designs implements a different activation function. First, an Adaptive Neural Logic Block (ANLB) is proposed that implements an adaptive activation function in hardware by introducing a second layer of memristors in the activation function. Next, a Robust Adaptive Neural Logic Block (RANLB) is proposed that implements the same functionality but uses flip flops to store each point in the activation function instead of memristors. Finally, a Multi-Threshold Neural Logic Block (MTNLB) is proposed that implements a static activation function that is capable of
learning any logic function.

3.1 Adaptive Neural Logic Block (ANLB)

As described in Section 2.2.1, the use of an adaptive activation function has proven to efficiently overcome the limitations of a threshold neural logic block in software. An adaptive activation function has previously been modeled as a piecewise continuous function, represented as the interpolation of \( m \) points, each of which has a floating point value ranging from 0.0 to 1.0. The value of the activation function for a given input current is rounded to the nearest integer to determine the value of the digital output.

When considering a hardware implementation of this functionality, the activation function can be simplified to associate a value with each of \( m \) ranges. Fig. 3.4 shows the ideal shapes of the activation function for different four-input logic functions. If uniform input weights are assumed, the input current, \( I_{in} \) is analogous to the number of high inputs. By dividing the activation function up into \( m \) ranges and shifting the value associated with each range up or down, the activation function can easily be trained to match any function with less than \( m \) decision boundaries. If \( m \) is greater than \( n \), any \( n \)-input logic function’s ideal activation function curve can be matched.

In order to implement this simplified adaptive activation function in hardware, the circuit shown in Fig. 3.5 is introduced at each range select output.
For each current range, an additional memristor is introduced with a static resistor in a simple voltage divider circuit. If the memristance, $M$, of a given memristor is less than the reference resistance, the voltage at the output of the memristor will be greater than $\frac{V_{dd}}{2}$. If the memristance is much higher than the reference resistance, the voltage at the output of the memristor will be close to zero. This output voltage represents the value of the activation function for the corresponding input current range. By passing this value through a buffer, it is implicitly rounded to a digital high or low value. Finally, the digital signal is passed into a transmission gate. Because it is guaranteed that only one range select signal is high at a given time, this forms a simple multiplexer at the output with minimal area overhead. As a result, the activation function is represented as a piecewise function consisting of $m$ ranges, each of which can be trained to a value ranging from 0 to $V_{dd}$.

### 3.2 Robust Adaptive Neural Logic Block (RANLB)

While the Adaptive Neural Logic Block (ANLB) successfully implements an adaptive activation function in hardware with minimal area overhead, this design has two major shortcomings. First of all, the value associated with each input current range is an analog voltage that must be used for a digital output. If the value associated with a given range is close to the
threshold voltage of CMOS components, the digital value may become ambiguous and very sensitive to small changes. Next, the overall trainability of an ANLB is very sensitive to device parameters and timing. Because the activation function and input weights are both constantly changing during training, it is easy for the ANLB to enter an oscillatory state in which it never converges to the correct functionality, if memristances change too quickly.

In order to overcome these shortcomings, a more robust representation of an adaptive activation function is presented in a Robust Adaptive Neural Logic Block (RANLB). Based on the fact that the output of a digital logic block can only be a logical high or low value, the representation of an adaptive activation function can be simplified further. Rather than associating an analog value with each input current range, a simple digital high or low can be associated with each individual range. Furthermore, to avoid the activation function constantly changing during training, an additional training signal, $\text{clk}_{af}$ is introduced. When this input is high, the activation function swaps its value. Otherwise, the activation function remains constant. For example, consider a $\text{clk}_{af}$ signal that is configured to be high for one in every three training cycles. In this scenario, the input weights are given two training cycles to attempt to match the expected output with a static activation function. If the system is unable to match the expected output, the activation function is modified for one training cycle and the system attempts to train the block again for two more clock cycles. This modification to the procedure can improve training convergence time substantially.
3.3 Multi-Threshold Neural Logic Block (MTNLB)

While the use of an adaptive activation function proves to be an efficient solution to the limitations of perceptron-based systems, the corresponding circuitry introduces added complexity that increases the area overhead of each individual NLB significantly. In some large neural networks in which area is a constraint, this may be undesirable. However, the ability of a logic block to learn nonlinearly separable functions is crucial to the scalability of a neuromorphic system.

In order to improve area overhead, a third NLB design is proposed that uses a static activation function, but is still capable of learning any logic
function. As shown previously in Fig. 3.4, the ideal activation function for different logic functions varies. However, by increasing the weights of each input, one can limit the input current range to span only a small portion of the overall activation function curve. Based on this principle, all logic functions’ ideal activation functions can be realized by limiting the input current range on a single, static activation function curve, as shown in Fig. 3.7(a). If the input current spans the whole range of the curve, the ideal activation function for a four-input XOR function is obtained. If the input current is limited to a smaller range of the curve, the ideal activation function for other functions can be obtained. However, when all inputs are low, the input current is guaranteed to be in the lowest range. Functions that require the output to be high in this scenario, such as NAND, NOR, and XNOR functions, would not be trainable to this activation function. To enable the learning of these functions, a simple bias signal can be introduced to internally invert the activation function, as shown in Fig. 3.7(b). The proposed Multi-Threshold Neural Logic Block (MTNLB) design implements this functionality in its activation function.

The hardware realization of the activation function for an MTNLB is given in Fig. 3.8. In this design, the output will always be high if an even-numbered current range is active. In order to implement this functionality, a NAND function of these active-low signals is used to find the logic block output. However, inverted functions (NAND, NOR, XNOR) require that the opposite be true. To accommodate for this, a bias signal is stored using a single flip-flop. When all the input are low, the expected output is written
to bias. When bias is high, the activation function is implicitly inverted using two simple steps. First, the expected output for training is inverted. This will cause the system to learn the inverted logic function. For example, when learning a NAND function, the system will simply be trained to an AND function. Then, the overall system output is inverted to compensate. These two simple inversions are done by XORing the $Y_{act}$ and $Y_{exp}$ signals with the bias signal.

In many applications, the use of a bias signal is not necessary. For example, consider an image recognition application in which pixel values are inputs to the system and the system is trained to recognize specific images. In this case, it is unlikely that a blank image, in which all inputs are 0, will ever be considered a match to the target image and require a high output. As a result, the activation function of an MTNLB can simply be reduced to a NAND gate with all even range select signals as inputs.

### 3.4 Comparison

In order to provide each of the described NLBs with the ability to learn different logic functions, each of them is paired with the Global/Local training circuitry proposed in [14]. This minimal-overhead stochastic gradient descent based training circuitry consists of a single global trainer, and a local trainer on each memristor in the design. For the ANLB, this includes both input memristors and activation function memristors. The global trainer is comprised of 42 transistors, while each local trainer is comprised of 20 transistors.

Table 3.1 gives a summary of the key features of each proposed NLB design. Furthermore, Table 3.2 gives the total area of each proposed NLB with training circuitry. First, the general equation for an $n$-input NLB capable of learning functions with $m$ decision boundaries is given. Then, the exact transistor count of a 4-input block that is capable of learning any logic function is given. For comparison to other reconfigurable fabrics, a TTGA block with a threshold activation function and a LUT are included.
Table 3.1: Summary of the key features of each proposed NLB design.

<table>
<thead>
<tr>
<th>Feature</th>
<th>ANLB</th>
<th>RANLB</th>
<th>MTNLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synapse Implementation</td>
<td>Single memristor</td>
<td>Single memristor</td>
<td>Single memristor</td>
</tr>
<tr>
<td>Activation Function</td>
<td>Piecewise adaptive activation function. Each point is an analog value represented by a memristor's state</td>
<td>Piecewise adaptive activation function. Each point is a digital value stored in a flip-flop.</td>
<td>Static activation function using multiple thresholds</td>
</tr>
<tr>
<td>Output Decoding</td>
<td>Activation function point routed to the output based on active current range</td>
<td>Activation function point routed to the output based on active current range</td>
<td>Output is high if an even-numbered current range is selected</td>
</tr>
<tr>
<td>Additional Overhead</td>
<td>None</td>
<td>Additional input training signal required ( (clk_{af}) )</td>
<td>Biasing circuitry required to invert output to implement certain functions</td>
</tr>
</tbody>
</table>

Table 3.2: Transistor counts of various reconfigurable logic block implementations.

<table>
<thead>
<tr>
<th>Logic Block Type</th>
<th>Transistor Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n-input block, m current ranges</td>
</tr>
<tr>
<td>ANLB</td>
<td>2nm+20n+34m+26</td>
</tr>
<tr>
<td>RANLB</td>
<td>2nm+18n+32m+28</td>
</tr>
<tr>
<td>MTNLB (with bias)</td>
<td>2nm+22n+12m+64+2\text{ceil}(m/2)</td>
</tr>
<tr>
<td>MTNLB (without bias)*</td>
<td>2nm+20n+12m+30+2\text{ceil}(m/2)</td>
</tr>
<tr>
<td>Single TTGA block**</td>
<td>22n+50</td>
</tr>
<tr>
<td>Multi-Layer TTGA***</td>
<td>11(22n+50)</td>
</tr>
<tr>
<td>LUT</td>
<td>-</td>
</tr>
</tbody>
</table>

* Requires inverted inputs in order to learn functions in which the output is high when all of the inputs are low.

** Can only learn linearly separable functions.

*** This represents the minimum transistor count for a multi-layer TTGA network that is capable of learning all logic functions.
As the results show, all of the proposed NLB designs require significantly more area overhead than a single TTGA block. However, when considering the number of TTGA blocks required to learn the worst-case $n$-input XOR function, the proposed NLBs offer a much larger area improvement. This is due to the fact that the proposed NLBs can learn any function in a single block, while a TTGA block requires up to 11 blocks to learn one non-linearly separable function. When compared to a standard LUT, the proposed NLBs offer a small area improvement. However, the ability of neural logic blocks to learn and make minor adaptations to their functionality makes them much more desirable than an LUT for many applications, such as computer vision and pattern recognition.

When considering the area overhead of an NLB that is capable of learning any logic function, the value of $m$ is set to $(n + 1)$. However, the worst-case $n$-input XOR function is fairly rare for NLBs with more than two inputs. For this reason, it may be acceptable in some applications to limit $m$ to a smaller value. For example, a four-input logic block with $m = 3$ cannot learn a three or four-input XOR function in a single layer. However, it can still learn the more common two-input XOR function, and requires much less area overhead per block. As the number of inputs increases, it becomes more practical to limit the number of decision boundaries a learnable function can have, improving the scalability of the proposed NLBs.

The MTNLB has the lowest area overhead per block. However, there are several benefits to the ANLB and RANLB designs. Assuming that an ANLB is designed with proper device parameters and timing, this design would be expected to achieve the fastest training convergence, because both the activation function and input weights are constantly trained towards a target function during training. While the activation function is modified less frequently in an RANLB, this design still contains this feature that decreases training time. Furthermore, because the MTNLB’s training process requires input currents to be limited to small ranges, the MTNLB generally requires memristors with a larger range of possible resistance values. In general, a four-input ANLB or RANLB requires the memristors to have a minimum of four resistance states. An MTNLB, on the other hand, requires a minimum
of eight resistance states. This not only imposes additional requirements on the timing and memristor parameters in an MTNLB, but also results in the need for more training cycles to train a system from the highest resistance states to the lowest resistance states.

In summary, each NLB design is beneficial for different applications. The ANLB is theoretically capable of reaching the fastest training times, but requires the most area overhead per block and is the least robust. The MTNLB, on the other hand, offers the lowest area overhead, but has the slowest training times. Finally, the RANLB is a compromise that offers more robustness and less area than the ANLB, but requires longer training times.
Chapter 4

Training

The training process for a neural logic block with a threshold activation function is fairly straightforward using the stochastic gradient descent algorithm. In this scenario, the direction to train the input weights corresponds directly to the value of the output. If the output is high, memristances should be increased during training in order to decrease synaptic weights. If the output is low, memristances should be decreased to increase synaptic weights. However, when complexity is added to the activation function, the direction to train input weights becomes more ambiguous. In this work, it was determined that the best approach to determine the direction of training is to ignore the output value and base the decision upon the active input current range. If $\bar{sel}_1$ is low, indicating that the input current is in the highest range, memristances are increased. Otherwise, memristances are decreased. This functionality can be implemented with no additional training overhead by modifying the global trainer to accept an external $neg$ signal and connecting $sel_1$ to it.

As the remainder of this section shows, this training algorithm is capable of learning any logic function rapidly in a single layer. However, this training algorithm also imposes restrictions on the device parameters of the memristors in the NLB designs. Because the memristances only increase when the input current is in the maximum current range, memristances are decreased much more often than they are increased. To compensate for this, the rate at which memristances increase must be faster than the rate at which memristances decrease. In general, this is likely to be true for fabricated devices
because more current flows through the device at low-resistance states, increasing the rate of change of memristance. This functionality can also be encouraged by designing memristors to have a lower magnitude negative threshold voltage, $V_{th,neg}$ than the positive threshold voltage, $V_{th,pos}$ or to have a smaller negative timing parameter, $t_{neg}$, than positive timing parameter, $t_{pos}$. As an alternative, the training algorithm itself could be modified to increase memristances if the input current is in range $m/2$ or lower. This can also be achieved with little-to-no additional area overhead by using the output of $C_{m/2}$ as the $neg$ input to the global trainer (Fig. 3.2). However, this configuration is more likely to lead to cases in which a memristance alternates between increasing and decreasing. When this occurs, the overall training convergence time is increased and it is possible for the system to enter an infinite loop and fail at learning a function.

### 4.1 Examples

In order to show the ability of the proposed designs to learn various logic functions, a simple 4-input NLB was implemented using ANLB, RANLB, and MTNLB designs. In all three cases, the training circuitry was supplied with a read voltage ($V_{read}$) of 1V, a write voltage ($V_{write}$) of 1.5V, and a training clock signal with a period of 150 ns. During training, each input combination is given for 150 ns, coupled with the desired output, $Y_{exp}$, for the input vector. For a four-input block, each full cycle through all input combinations takes 1200 ns. The RANLB design is supplied with an additional $clk_{af}$ signal with a period of 3600 ns and a 33 percent duty cycle. This allows the activation function to be modified once every three full training cycles. When training is not occurring, the training clock signals are held constant at 0 V. The memristors used in the NLB designs were modeled using the parameters given in Table 1.5. For the ANLB and RANLB designs, the comparator reference currents were chosen to be $10nA$, $100nA$, $400nA$, and $800nA$. For the MTNLB design, the comparator reference currents were chosen to be $50nA$, $400nA$, $800nA$, and $1.2\mu A$. 
Figure 4.1: Example waveform of an MTNLB learning a 4-input XOR function. The output, $Y$, is trained to match the expected output, $Y_{\text{exp}}$, after 12us.

Table 4.1: 4-input neural logic block training times

<table>
<thead>
<tr>
<th>Function</th>
<th>RANLB</th>
<th>MTNLB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Num Cycles</td>
<td>Time (us)</td>
</tr>
<tr>
<td>AND</td>
<td>7</td>
<td>16.8</td>
</tr>
<tr>
<td>OR</td>
<td>1</td>
<td>2.4</td>
</tr>
<tr>
<td>XOR</td>
<td>4</td>
<td>9.6</td>
</tr>
<tr>
<td>NAND</td>
<td>4</td>
<td>9.6</td>
</tr>
<tr>
<td>NOR</td>
<td>1</td>
<td>2.4</td>
</tr>
<tr>
<td>XNOR</td>
<td>4</td>
<td>9.6</td>
</tr>
</tbody>
</table>
Figure 4.2: An outline of the procedure used to obtain waveforms of the training process for an NLB.

To test the NLB designs, each was trained to implement all standard logic functions from a random initial state. The procedure used to obtain waveforms for each of the NLB designs is outlined in Fig. 4.2. Verilog-AMS models representing the system and testbench were developed. Using these models, PTM model files, and run files, the Cadence irun script was used to create a waveform database. The produced waveforms were viewed using Cadence Simvision. An example waveform of an MTNLB learning a 4-input XOR function is given in Fig. 4.1. As the waveform shows, the 4-input XOR function is learned in $12us$. Appendix A gives the waveforms for several other cases. After the training process, the output waveform matches the shape of the expected output waveform. However, it should be noted that there are also voltage spikes at some points in the waveform. This occurs when the input current range changes. These voltage spikes can be avoided by putting a capacitor on the output or developing a training process that results in a system with minimal changes to the input current range.

Table 4.1 gives a comparison of the training times for an RANLB and MTNLB learning each function. As the results show, the training times can vary drastically for different target functions. Several factors affect the training time of a single NLB. First of all, the overall distance that the memristance needs to be changed affects the training time. For example, if the initial state of a memristor is $R_{on}$ and a final state of $R_{off}$ is required, it will
take more training cycles to train the NLB than if only a small memristance change was needed. In addition to this, the order in which input pairs are presented and the nature of the target function affect the training time. Consider an NLB whose output is always low in its initial state. When this block is trained to an AND function, the memristances are only modified for a single input vector, in which all inputs are high. In this case, the training time is increased because a large amount of time is taken in which other input combinations are presented but no training actually occurs. Similarly, if the input vectors are given in an order for which the expected output frequently changes from 0 to 1, it is likely that weights will oscillate from low values to high values during training. While eventually the system will reach a stable state and learn the function, these oscillations elongate the training process. From a design perspective, the period of training pulses, memristor timing parameters, and memristor threshold voltages play a large role in training time. These parameters determine how fast the memristances are changed each time training occurs. If the memristances change too quickly, the system may fail to learn some functions. However, if the memristances change too slowly, the training time can be increased.

In a large-scale system, the unpredictable training times can be handled in one of two ways. First of all, the worst case training time could be calculated based on the number of inputs in an NLB and the possible functions it could be trained to. Each time training occurs, the system could allow this worst case time for training. While this method will result in long training times, it requires very little training logic. As an alternative, additional training circuitry could be introduced to monitor $clk_{sel}$, the training signal that only goes high when the output does not match the expected output. Once this signal remains low for all input combinations, training is complete. Because it is difficult to predict training time, the use of an NLB would be most beneficial in an application in which small adaptations are often made and the ability to reconfigure the system in real-time is desired. An NLB can be completely reprogrammed by looping through all input combinations during training, but an NLB truly excels when it is preprogrammed and the output only needs to be modified for certain input vectors. In this case, only
the critical input vectors need to be presented during training and the sys-
tems state is only modified slightly. As a result, the system is able to learn adaptations very rapidly. This is shown in an OCR application in Section 5.2.
Chapter 5

Applications

In general, the proposed NLBs can be used as building blocks to design large-scale systems for a variety of neuromorphic and reconfigurable applications. The novel ability of each individual NLB to learn any logic function in a single layer provides significant improvements to the efficiency and scalability of overall systems in nearly all applicable domains. In order to show the magnitude of such improvements, we consider two common applications. First, a general-purpose reconfigurable logic network is synthesized and the energy and delay of the overall system is compared to a TTGA block and standard LUT implementation. Then, a simple digit optical character recognition (OCR) system is designed to show how the proposed NLBs can simplify the design of neuromorphic systems.

5.1 Reconfigurable Logic

In reconfigurable logic units, such as FPGAs, LUTs are used as building blocks in large networks to implement complex functionality. In [14], Manem et al. show that the overall area of these systems can be improved by replacing LUTs with trainable NLBs with threshold activation functions. While the authors also show potential for energy and delay improvements,
the limitations of each individual NLB result in an average-case Energy-Delay Product (EDP) that remains on par with a standard LUT implementation. By removing these limitations and reducing the number of NLBs required to implement a given functionality, this work shows that the overall Energy-Delay Product of large-scale networks can be improved drastically.

To analyze the efficiency of the proposed NLBs, the ISCAS-85 benchmark suite was considered \[10\]. The ISCAS-85 benchmark suite contains ten combinational networks that implement various computing functions, such as a 16-bit Multiplier, ALU and control, and Priority Decoder. The circuits range in complexity from 160 gates to 3512 gates. Verilog-AMS models of four-input RANLB and MTNLB designs were trained from a random initial state to implement each common gate (AND, OR, XOR, NAND, NOR, XNOR, AB+BC+AC). In each case, the final memristances within each trained NLB were recorded. The theoretical best-case memristances to implement each function using a four-input ANLB were also obtained. For comparison to previous work, 45 nm low power predictive technology models were used. Then, using the recorded resistance values, static blocks
were implemented in which each memristor was replaced with a resistor of the corresponding resistance. A block was developed to represent ANLB, RANLB, and MTNLB implementations of each individual gate. Each block was modeled in Verilog-AMS and a .scs testbench file was developed. Using these files, Cadence Specre was used to produce a waveform database that is viewable in Cadence Analog Environment. From the output waveforms, the average power and delay of each block was calculated. Based on this information, a library of the power and delay of each individual gate was created for each NLB design. Finally, Berkeley SIS [4] was used to synthesize the benchmark circuits using the power and delay library and .blif files representing each circuit. This procedure is outlined in Fig. 5.1.

Using the overall power and delay of each benchmark circuit, the energy-delay product (EDP) was measured and compared to a TTGA block and a standard LUT implementation. The results are shown in Fig. 5.2. It should be noted that each NLB can implement each individual function in a number of different memristance states. In [14], the benchmark circuits were synthesized using the power and delay metrics of TTGA blocks with the
maximum possible resistance values and minimum possible resistance values. For this reason, the results are given as the best-case EDP, $TTGA_{\text{min}}$, and the worst-case EDP, $TTGA_{\text{max}}$. Using this same procedure, the best-case EDP for an ANLB implementation, $ANLB_{\text{min}}$ is given. However, the results for the RANLB and MTNLB implementations are based on the actual memristance values obtained by training from a random initial state. For this reason, they can be considered the average-case results.

As the results show, all of the proposed designs show significant improvements over both a standard LUT implementation and a TTGA block. When compared to a TTGA block, each individual ANLB, RANLB or MTNLB has higher delay and energy consumption. However, by reducing the required number of logic blocks, the proposed designs achieve less overall delay and energy in large-scale systems. Because an individual MTNLB is capable of learning all logic functions using fewer components than the ANLB and RANLB, each individual MTNLB requires less power, resulting in a lower overall EDP. The average-case RANLB implementation’s EDP is up to 86% lower than a standard LUT implementation and up to 84% lower than the best-case TTGA implementation. Furthermore, the MTNLB implementation’s EDP is 92% to 99% lower than a standard LUT implementation and 48% to 99% lower than a TTGA implementation on all ISCAS-85 benchmark circuits. The best-case EDP for an ANLB is also up to 97.8% lower than the best-case EDP for a TTGA block and 78.08% to 97.42% lower than a standard LUT implementation on all ISCAS-85 benchmarks.

In order to show scalability to different technology, an RANLB and MTNLB were designed using 16 nm low power predictive technology models. Fig. 5.3 shows a comparison of the average power of several gate implementations using 16 nm and 45 nm technology.

Aside from the area and EDP improvements, the use of NLBs in reconfigurable logic units is desirable because it gives the system the ability to adapt in real-time. Consider the process of making a small change to an FPGA design. On a standard FPGA, this requires stopping the whole system from running and reprogramming the whole FPGA, effectively overwriting all of the information stored in each active LUT. Now consider if each LUT
Figure 5.3: Average power of various gate implementations using 16nm and 45nm LP PTM models.

was replaced by an NLB. Rather than reprogramming the whole system, a group of NLBs can be trained to different functionality while the remainder of the system remains static. The development of a procedure to partition and adapt parts of the system is out of the scope of this work. However, it should be noted that this is another potential advantage to using NLBs in reconfigurable logic.

5.2 Optical Character Recognition

Optical Character Recognition (OCR) is a computer vision process in which images of handwritten, typewritten, or printed text are classified to specific character data. In general, OCR systems are given a set of training images of known characters and develop a model of each character. Then, OCR systems are able to match any input image to the most similar character model to make a best guess at what character is presented in the image. Neuromorphic systems are commonly considered for this application because their learning capability is appealing for developing a model of each character. In general, the limitations on the trainable set of perceptron-based NLBs require an additional layer of hidden nodes in OCR systems. The number of hidden nodes is often chosen at random through a trial and error process and a poor choice can limit the success of the system. However, by using an NLB with a larger trainable set, the need for this hidden layer can be
Figure 5.4: An OCR block for recognizing a single character / digit, in which each NLB analyzes one row of pixels.

To show the benefits of the proposed NLBs, a simple 8x8 pixel digit recognition system was developed for the numbers 0 through 9. The general structure of an OCR system is fairly straightforward. For each possible character in the system, a single block accepts all pixel data as inputs and produces a logical output, indicating whether or not the corresponding character is recognized. However, there are several different approaches to how each OCR block analyzes the pixels of an image. The most straightforward approach is to use a single 64-input NLB. However, an NLB with this many inputs would be required to learn very complex functionality in order to develop a model of a character. As a result, longer training times would be required and a more complex training algorithm may be necessary to ensure training
convergence. Another approach is to limit each NLB to look at a single row or column of pixels. For example, consider the system shown in Fig. 5.4. In this case, an 8-input NLB is used to analyze a single row of pixels, and produces a high output if the image appears to be the corresponding digit. If the majority of the NLBs detect a digit, it can be assumed that the corresponding digit is present in a given image and the overall output should be high. This functionality can be realized using a simple comparator at the output. However, this approach also has its shortcomings. Consider the first row of pixels in an 8x8 image. In general, the numbers 2, 3, 5, and 7 all have a solid line in this row. During the training process, the NLBs to recognize each of these digits in the first row will be given contradictory inputs. The same pixel pattern will be given with both a high expected output and low expected output at the same time. In this case, the NLB associated with the corresponding row will not be able to develop an accurate model. Furthermore, as the number of characters in an OCR system increases, the probability of this occurring becomes greater. To avoid these shortcomings, the approach outlined in Fig. 5.5 was used in this work. In this case, four 16-input NLBs are used to recognize each character and each NLB analyzes a quadrant of pixels. By reducing the number of inputs compared to using a single NLB, this approach reduces the complexity of training and training time. By looking at a larger window of pixels compared to a row or column approach, it avoids the presentation of contradictory data to a single NLB.

Because of its low power consumption and delay, an MTNLB was chosen for this design. Each NLB was designed using six input current ranges, allowing it to learn functions with up to five decision boundaries. While more input current ranges could be used, it would be very unlikely for any character model to require more than five decision boundaries. Fig. 5.6 shows a set of images used to train the digit recognition system, containing four variants of each target digit. In order to develop models for each digit, these training images were given as inputs to the system, paired with the expected outputs, for 90\(\mu\)s. After this training process, the system was given a set of different test images to classify, shown in Fig. 5.7. While the test images are similar to the training data, none of the test images are identical to any training data. The system was configured so that if all four quadrants
Figure 5.5: An OCR block for recognizing a single character / digit, in which each NLB analyzes one quadrant of pixels.

Figure 5.6: Set of images used to train the OCR system.

produced a high output for a given character, the overall output would be high.

Tables 5.1-5.10 show the output of each neural logic block for each test image. As the results show, all of the test images were classified correctly except for Test Image 7 and Test Image 8. Consider the first quadrant of pixels in the set of training images. The pattern of pixels in an image of a 7 is identical or very similar to the pattern of pixels in some images of the
digits 2 and 3. For this reason, the NLB designed to recognize a 7 in the first quadrant is given contradictory data during training. When a 2 or 3 is presented, the expected output is low. Then, when the same set of inputs is given to represent a 7, the expected output is high. As a result, this NLB fails to develop an accurate model of a 7. In order to avoid this type of discrepancy, the architecture of the OCR system itself would need to be modified. For example, rather than having each NLB analyze a quadrant, each NLB could analyze a random set of pixels in the image. Or, as an alternative, a single 64 input NLB could be used and given a much longer training time.

Next, consider Test Image 8. While this image clearly depicts the digit 8, it also has very similar features to a 2, 3, and 9. For this reason, it is difficult for the system to develop an accurate model of an 8 and avoid misclassifying an image of an 8 to a 2, 3, or 9. When given Test Image 8, the OCR blocks for detecting a 2, 3, and 9 all produced a high output. This issue could be mitigated by using a higher resolution. Rather than representing images as a 8x8 pixel grid, images could be represented as a larger pixel grid, such as 12x12 or 16x16. In this case, greater detail could be depicted in each image, allowing the curves in an 8 to be more defined. As a result, the image of an 8 would then differ more from other digits and be easier to model in an OCR block.
Table 5.1: NLB outputs for classifying Test Image 1.

<table>
<thead>
<tr>
<th>NLB</th>
<th>&quot;1&quot;</th>
<th>&quot;2&quot;</th>
<th>&quot;3&quot;</th>
<th>&quot;4&quot;</th>
<th>&quot;5&quot;</th>
<th>&quot;6&quot;</th>
<th>&quot;7&quot;</th>
<th>&quot;8&quot;</th>
<th>&quot;9&quot;</th>
<th>&quot;0&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadrant 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.2: NLB outputs for classifying Test Image 2.

<table>
<thead>
<tr>
<th>NLB</th>
<th>&quot;1&quot;</th>
<th>&quot;2&quot;</th>
<th>&quot;3&quot;</th>
<th>&quot;4&quot;</th>
<th>&quot;5&quot;</th>
<th>&quot;6&quot;</th>
<th>&quot;7&quot;</th>
<th>&quot;8&quot;</th>
<th>&quot;9&quot;</th>
<th>&quot;0&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadrant 1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Quadrant 3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.3: NLB outputs for classifying Test Image 3.

<table>
<thead>
<tr>
<th>NLB</th>
<th>&quot;1&quot;</th>
<th>&quot;2&quot;</th>
<th>&quot;3&quot;</th>
<th>&quot;4&quot;</th>
<th>&quot;5&quot;</th>
<th>&quot;6&quot;</th>
<th>&quot;7&quot;</th>
<th>&quot;8&quot;</th>
<th>&quot;9&quot;</th>
<th>&quot;0&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadrant 1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Quadrant 3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Overall</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.4: NLB outputs for classifying Test Image 4.

<table>
<thead>
<tr>
<th>NLB</th>
<th>&quot;1&quot;</th>
<th>&quot;2&quot;</th>
<th>&quot;3&quot;</th>
<th>&quot;4&quot;</th>
<th>&quot;5&quot;</th>
<th>&quot;6&quot;</th>
<th>&quot;7&quot;</th>
<th>&quot;8&quot;</th>
<th>&quot;9&quot;</th>
<th>&quot;0&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadrant 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Quadrant 4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.5: NLB outputs for classifying Test Image 5.

<table>
<thead>
<tr>
<th>NLB</th>
<th>&quot;1&quot;</th>
<th>&quot;2&quot;</th>
<th>&quot;3&quot;</th>
<th>&quot;4&quot;</th>
<th>&quot;5&quot;</th>
<th>&quot;6&quot;</th>
<th>&quot;7&quot;</th>
<th>&quot;8&quot;</th>
<th>&quot;9&quot;</th>
<th>&quot;0&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadrant 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Quadrant 4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Overall</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.6: NLB outputs for classifying Test Image 6.

<table>
<thead>
<tr>
<th>NLB</th>
<th>&quot;1&quot;</th>
<th>&quot;2&quot;</th>
<th>&quot;3&quot;</th>
<th>&quot;4&quot;</th>
<th>&quot;5&quot;</th>
<th>&quot;6&quot;</th>
<th>&quot;7&quot;</th>
<th>&quot;8&quot;</th>
<th>&quot;9&quot;</th>
<th>&quot;0&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadrant 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Quadrant 4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Overall</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 5.7: NLB outputs for classifying Test Image 7.

<table>
<thead>
<tr>
<th>NLB</th>
<th>&quot;1&quot;</th>
<th>&quot;2&quot;</th>
<th>&quot;3&quot;</th>
<th>&quot;4&quot;</th>
<th>&quot;5&quot;</th>
<th>&quot;6&quot;</th>
<th>&quot;7&quot;</th>
<th>&quot;8&quot;</th>
<th>&quot;9&quot;</th>
<th>&quot;0&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadrant 1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Quadrant 3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.8: NLB outputs for classifying Test Image 8.

<table>
<thead>
<tr>
<th>NLB</th>
<th>&quot;1&quot;</th>
<th>&quot;2&quot;</th>
<th>&quot;3&quot;</th>
<th>&quot;4&quot;</th>
<th>&quot;5&quot;</th>
<th>&quot;6&quot;</th>
<th>&quot;7&quot;</th>
<th>&quot;8&quot;</th>
<th>&quot;9&quot;</th>
<th>&quot;0&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadrant 1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Quadrant 3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Overall</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.9: NLB outputs for classifying Test Image 9.

<table>
<thead>
<tr>
<th>NLB</th>
<th>&quot;1&quot;</th>
<th>&quot;2&quot;</th>
<th>&quot;3&quot;</th>
<th>&quot;4&quot;</th>
<th>&quot;5&quot;</th>
<th>&quot;6&quot;</th>
<th>&quot;7&quot;</th>
<th>&quot;8&quot;</th>
<th>&quot;9&quot;</th>
<th>&quot;0&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadrant 1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Quadrant 2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Quadrant 3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Quadrant 4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Overall</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.10: NLB outputs for classifying Test Image 10.

<table>
<thead>
<tr>
<th>NLB</th>
<th>&quot;1&quot;</th>
<th>&quot;2&quot;</th>
<th>&quot;3&quot;</th>
<th>&quot;4&quot;</th>
<th>&quot;5&quot;</th>
<th>&quot;6&quot;</th>
<th>&quot;7&quot;</th>
<th>&quot;8&quot;</th>
<th>&quot;9&quot;</th>
<th>&quot;0&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quadrant 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Quadrant 2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Quadrant 3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Quadrant 4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Overall</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusions

In this work, it is proven that the scalability and efficiency of hardware-based neuromorphic systems can be improved drastically by adding complexity to neural logic block (NLB) designs. Three NLB designs are presented, each of which integrates memristive synapses into a novel perceptron model that is capable of learning any logic function in a single layer. First an Adaptive Neural Logic Block (ANLB) and Robust Adaptive Neural Logic Block (RANLB) are presented, each of which implements an adaptive activation function, which is designed for fast training convergence times. A four-input RANLB is capable of learning any function from a random state in 2.4 us to 19.2 us. Next, a Multi Threshold Neural Logic Block (MTNLB) is proposed that uses a static activation function to learn any function. While this method achieves slower training convergence times, a four-input block can be implemented using as few as 216 transistors.

To show the significance of the efficiency improvements the proposed NLBs achieve, a general-purpose reconfigurable logic application was considered. When compared to an LUT implementation, the MTNLB was capable of achieving an EDP 92% to 99% lower on all ISCAS-85 benchmark circuits. Furthermore, the MTNLB’s EDP was 48% to 99% lower than the EDP for a previous NLB implementation with a threshold activation function. Similar improvements were obtained for the RANLB and ANLB designs.

Finally, the OCR application domain was explored to show improvements in a large-scale system. Using four 16-input NLBs per digit, a simple digit
recognition system was developed. The system was able to recognize eight out of ten test images of digits on an 8x8 pixel grid and required only 100 us for training. By increasing the pixel grid size, increasing the training time, and modifying the structure of the OCR system, the accuracy would be expected to improve further.

Future work could include improving the NLB designs and applying them to new application domains. The proposed NLB designs have many parameters that affect both the reliability and training convergence time in each NLB. The choice of comparator reference currents, which act as boundaries between current ranges in the NLBs, must be carefully chosen to ensure that all functions are learnable. However, spacing these reference currents differently could improve training time and simplify training for certain functions. Similarly, the period and magnitude of training pulses and memristor timing and voltage threshold parameters affect how fast training occurs. If memristances change too fast, the system loses reliability. On the other hand, if the memristances change too slow, training time is increased. By conducting an analysis of the effects of changing these parameters or by applying a genetic algorithm, optimal values could be obtained to improve the proposed NLB designs.

In software, neuromorphic systems have been used for a wide variety of applications, in fields such as pattern recognition, control systems, and signal processing. By improving scalability and efficiency in hardware NLB designs, this work opens the door to create hardware based neural networks for these applications. Because the training of NLBs is highly parallel, a very significant speedup would be expected in hardware implementations. The use of NLBs in reconfigurable logic could be expanded by developing large scale training and partitioning schemes to allow portions of the system to adapt while other portions remain static and functional. Similarly, the proposed OCR system could be expanded and applied to any alphabet, set of symbols or characters, or simple image set.
Appendix A

Additional Training Examples
Figure A.1: Waveform of an ANLB learning a 4-input XOR function.
Figure A.2: Waveform of an ANLB learning a 4-input NAND function.
Figure A.3: Waveform of an ANLB learning a full adder carry out function (AB+BC+AC).
Figure A.4: Waveform of an RANLB learning a 4-input XOR function.

Figure A.5: Waveform of an RANLB learning a 4-input NAND function.
Figure A.6: Waveform of an RANLB learning a full adder carry out function (AB+BC+AC).

Figure A.7: Waveform of an MTNLB learning a 4-input NAND function.
Figure A.8: Waveform of an MTNLB learning a full adder carry out function (\(AB+BC+AC\)).
Bibliography


