12-2017

The Synthesis of Memristive Neuromorphic Circuits

Austin Richard Wyer
University of Tennessee, Knoxville, awyer@vols.utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Austin Richard Wyer entitled "The Synthesis of Memristive Neuromorphic Circuits." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Garrett S. Rose, Major Professor

We have read this thesis and recommend its acceptance:

James Plank, Mark Dean

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

The Synthesis of Memristive Neuromorphic Circuits

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Austin Richard Wyer

December 2017

I dedicate this work to Elizabeth, who supported me throughout this endeavor.

Acknowledgments

I would like to thank the members of my committee: Dr. Plank, Dr. Dean, and Dr. Rose. Without the incredibly supportive research environment these three have striven to create none of this work would be possible. I would especially like to thank Dr. Rose for challenging me to broaden the way I conduct research, and ensuring that the work I conducted was held to a high scientific standard. Your support and guidance was of the utmost importance.

I would like to thank all of the other students who supported me. Gangotree, Adnan, Sagar, Sherif, Ryan, Mesbah, Majumder, I would like to thank you all. Without your guidance and support I would have been hopeless in modeling any circuits. Grant, John, Adam thank you for patiently explaining the neuromorphic software framework to me, and being willing to help any time I encountered a problem. Thank you to Nick, who helped improve the simulator and allowed me to focus on conducting research. Thank you to Miles, who helped encourage me to pursue research, and helped develop the original simulator. It was a always a joy talking to all of you, and discussing the crazy ideas we all had.

I would especially like to thank Dr. Schuman. Without your work, none of this would be possible. You were always willing to answer any questions I had, and point me in the right direction when I was lost.

Abstract

As Moores Law has come to a halt, it has become necessary to explore alternative forms of computation that are not limited in the same ways as traditional CMOS technologies and the Von Neumann architecture. Neuromorphic computing, computing inspired by the human brain with neurons and synapses, has been proposed as one of these alternatives. Memristors, non-volatile devices with adjustable resistances, have emerged as a candidate for implementing neuromorphic computing systems because of their low power and low area overhead. This work presents a C++ simulator for an implementation of a memristive neuromorphic circuit. The simulator is used within a software framework to design and evaluate these circuits.

The first chapter provides a background on neuromorphic computing and memristors, explores other neuromorphic circuits and their programming models, and finally presents the software framework for which the simulator was developed. The second chapter presents the C++ simulator and the genetic operators used in the generation of the memristive neuromorphic networks. Next, the third chapter presents a verification of the accuracy of the simulator, and provides some analysis of designs. These analyses focus on variation, the Axon-Hillock neuron model, limited programming resolutions, and online learning mechanisms. Finally, the fourth chapter discusses future considerations.

Thus, this thesis presents the C++ simulator as a tool to generate memristive neuromorphic networks. Additionally, it shows how the simulator can be used to understand how the underlying hardware impacts the application level performance of the network.

Table of Contents

1 Introduction and Background **1**

1.1 Neuromorphic Computing 1

1.2 Memristors 2

1.3 Existing Neuromorphic Implementations 3

1.4 Software Framework 5

2 Memristive Neuromorphic Simulator **8**

2.1 Components 9

2.2 Network 13

2.3 Evolutionary Optimization 17

3 Experimental Verification and Analysis **27**

3.1 Verification 27

3.2 Analysis 31

3.2.1 Variation 31

3.2.2 Axon-Hillock Neuron Model 32

3.2.3 Programming Resolution 34

3.2.4 Online Learning 36

4 Conclusions and Future Work **42**

Bibliography **44**

Appendices **49**

A Sample Code 50
A.1 Neuron 50
A.2 Synapse 54
A.3 Delay 57
A.4 Random 58
A.5 Mutate 60
A.6 Crossover 64
B Sample Files 76
B.1 Network File 76
B.2 Parameter File 77
B.3 Input File 81

Vita **84**

List of Tables

3.1 Table of verification metrics for an XOR network 30
3.2 Comparison of networks generated from DANNA, the standard memristive model, and the memristive Axon-Hillock model 34
3.3 Performance of networks with programming resolution of 3 for iris classification application 35
3.4 Performance of networks with programming resolution of 3 for polebalancing application 41

List of Figures

1.1 Interaction of the layers of the software framework 5

2.1 Circuit representation of important network components 9
2.2 A synaptic weight can be realized through multiple combinations of memristances 11
2.3 The delay unit is modeled in the simulator using a programmable cyclic array 12
2.4 Code snippet for apply and update functions for the delay block 14
2.5 A visualization of a network(left) and a string representation of a network(right) 16
2.6 Random takes application specific starting parameters and produces a sparse random network 20
2.7 The move neuron operation optimizes the temporal components of the networks 25
2.8 An example of the crossover operation for two arbitrary memristive networks 26

3.1 Simple network simulated at the high level and circuit level 28
3.2 Output from the C++ simulator plotted using MATLAB(left) and output from Cadence Spectre(right) 29
3.3 Performance of networks with and without LTPD for the classification of different datasets 37
3.4 Accuracy over time for same XOR network with and without STDP enabled 39
3.5 Impact of STDP in networks for the iris classification task 40
3.6 Improvement in performance of networks trained with STDP over those without 41

Chapter 1

Introduction and Background

1.1 Neuromorphic Computing

Neuromorphic computing is an emerging field that seeks to leverage our understanding of the human brain to develop new systems capable of computation unlike that of the Von Neumann architecture. These systems are typically characterized as neurons connected in parallel by synapses. Unlike the Von Neumann architecture, data is stored in the computational components [14]. Thus, a neuron readily has its data available throughout operation, and the neuromorphic system does not suffer the same memory bottleneck as a Von Neumann system.

With a reduction in data movement also comes a reduction in energy consumption. The human brain is estimated to consume about 20 watts, while the average laptop consumes around 60 watts. However, as the size of the Von Neumann systems begins to increase, their power requirements become more unrealizable. It has been estimated that it would require over 500 megawatts, the approximate power output of a nuclear reactor, to power an exascale super computer using the Von Neumann architecture. Neuromorphic architectures do not suffer from the same scaling issues as Von Neumann architectures in this respect, as they do not need to scale their systems for data movement.

The human brain not only consumes less power, but also includes mechanisms for learning. Humans can continue to learn new concepts, and make novel connections[31]. A computer capable of such a feat would revolutionize the world of data science, as we

could allow computers to find new patterns in data that humans might not have considered before. Therefore, exploring computation in a manner similar to the brain provides powerful potential opportunities.

While it is clear that neuromorphic computing has much potential, there are still many challenges in designing such systems. One of the major problems has been determining the appropriate network topology. It is well known that a neural network must contain a hidden neuron to complete even a simple XOR operation[20]. Furthermore, neural networks have seen staggering performance in classification tasks when adding convolutional layers[17]. While it is generally understood why these topological approaches work, there is no clear reason as to how to pick the best topology for an arbitrary task. One approach has been to use genetic algorithms to generate networks that can complete a task up to some standard.

1.2 Memristors

Memristors, memory resistors, were first proposed in 1971 by Leon Chua as a theoretical device that relates charge and flux[6]. This proposed device was given the name memristor as it would have a resistance that would depend on the electrical charge that had passed through the device giving it memory. Their low area and power overhead make them an appealing alternative to traditional CMOS. Additionally, because of their non-volatility, memristors have been proposed as a way to implement new forms of memory, new security primitives, and neuromorphic circuits[32].

While there have been many proposed implementations, there is no standard consensus on what constitutes a neuromorphic system. Many of the proposed models use some form of synapse and neuron. Memristors have emerged as a popular candidate for many synapse implementations. Not only does their low area allow for scalability in the size of the neuromorphic processor, but their low power consumption means that processors can scale in a high performance computing environment, where the Von Neumann architecture currently struggles. Memristors also function well as synapses because of their non-volatility and memory properties. Many of these biologically inspired systems have been looked at for including long term potentiation and depression of synapses in their model. A memristor can

have its memristance adjusted on-chip by simply applying the appropriate voltage, and it will maintain it's new memristance value. In this way, memristors can function as synapses by providing a simple means to adjust the strength of a synapse on-chip.

1.3 Existing Neuromorphic Implementations

Many different neuromorphic systems have been designed and provide different sets of tools to aid developers in building networks for different applications. Therefore, it is important to understand the purpose of the different designs and provide similar tools for programming a memristive neuromorohic circuit.

One of the most well known neuromorphic systems is IBMs TrueNorth. TrueNorth is a digital CMOS implementation where the chips contain many cores, with each core containing many neurons. All computation is handled within the cores which are highly intra-connected [2]. In this way, the TrueNorth design avoids the pitfalls of the Von Neumann architecture, particularly the massive communication and memory overheads. Programmers are able to program networks on the chip using the Corelet language. Unlike traditional programs which provide a sequential set of instructions to execute, the Corelet language allows the user to program TrueNorths individual cores. Once a core is programmed, it can be copied and used elsewhere in the system, or it can be used within a larger core. This allows the user to populate a chip with many special cores functioning in parallel. In this way, TrueNorth provides a new programming paradigm to program networks in a way that is in line with the goals of neuromorphic computing. Despite the freedom these tools provide, the network topology still must be hand selected by the designer of the system. Though there are tools for mapping some known artificial neural networks to TrueNorth, there is no guarantee that the network selected is suitable for the target application [10]. In the case where no good network structure is known, it is possible for a designer to build a network by hand using the Corelet language, but such a process can be tedious. Thus, TrueNorth provides a platform for neuromorphic engineers to implement and develop a variety of networks, but lacks the tools for generating these networks.

In contrast to TrueNorth, which is more biologically inspired, the Human Brain Projects SpiNNaker is focused on bio-mimicry [16]. The ultimate goal of the Human Brain Project is to provide a tool that can simulate the human brain. To aid in this goal, a Python library for neural networks, PyNN, has been produced. PyNN provides both high and low level abstractions to aid in the design of these large scale networks [7]. At the low level, a developer can specify the behavior of synaptic plasticity, while at the high level the user can specify network topology. Additional tools exist for mapping known networks onto the SpiNNaker hardware. Such tools, however, suffer from the same problem as TrueNorths in that they do not provide methods for selecting the ideal topology for a target application.

Within our larger research group, there have been two neuromorphic systems proposed, NIDA and DANNA. NIDA, Neuroscience-inspired dynamic architecture, is a software implementation of a spiking time dependent neural network. Neurons are embedded in a three dimensional space, and their distance in this space determines the temporal delay along their synapses [25]. NIDA provides tools for developers to generate networks for specific applications with specific performance characteristics. For example, if one wanted a NIDA network for a polebalancing task, they could specify how long they want the pole to be balanced for, and let NIDAs generation produce the network for them. The network generated could be a recurrent neural network or a convolutional neural network, and it will meet a minimum performance standard for a target application. In this way, NIDA provides tools for selecting efficient network topologies. DANNA, dynamic adaptive neural network arrays, provide a hardware implementation on FPGAs, field programmable gate arrays [26]. In this way, DANNA can potentially act as a low power solution to a variety of applications. To aid in this effort, DANNA provides a simulator to leverage a similar evolutionary optimization to NIDA. A DANNA network can quickly be generated in software for a target application, and be directly mapped to the hardware. Thus, DANNA and NIDA provide tools to generate networks with known performance characteristics unlike SpiNNaker and TrueNorth [23] [15].

From reviewing these architectures, it is clear that it is necessary to provide tools to aid in the development of neuromorphic systems. Like SpiNNaker and TrueNorth, the tools should support abstraction both at lower and higher levels of the network. It should be possible

for a developer to understand the impact of using specific memristors at the lower level on the performance of networks at an application level. Additionally, the tools should provide a way for users to generate networks for target applications as in NIDA and DANNA.

1.4 Software Framework

There is an ongoing effort to create software tools to aid developers in neuromorphic computing. One of the efforts has been led by Plank et al [22]. in their design of the University of Tennessees neuromorphic software library. This software stack divides neuromorphic implementations into three parts: application, evolutionary optimization, and model. Each part is self contained, and has specific interfaces to communicate with the other parts. In this way, the implementations for the individual parts can be changed without having to adjust the other parts. An application developer does not need to develop an application for one specific neuromorphic model such as DANNA, but instead develops the application for any model that uses the neuromorphic interface.

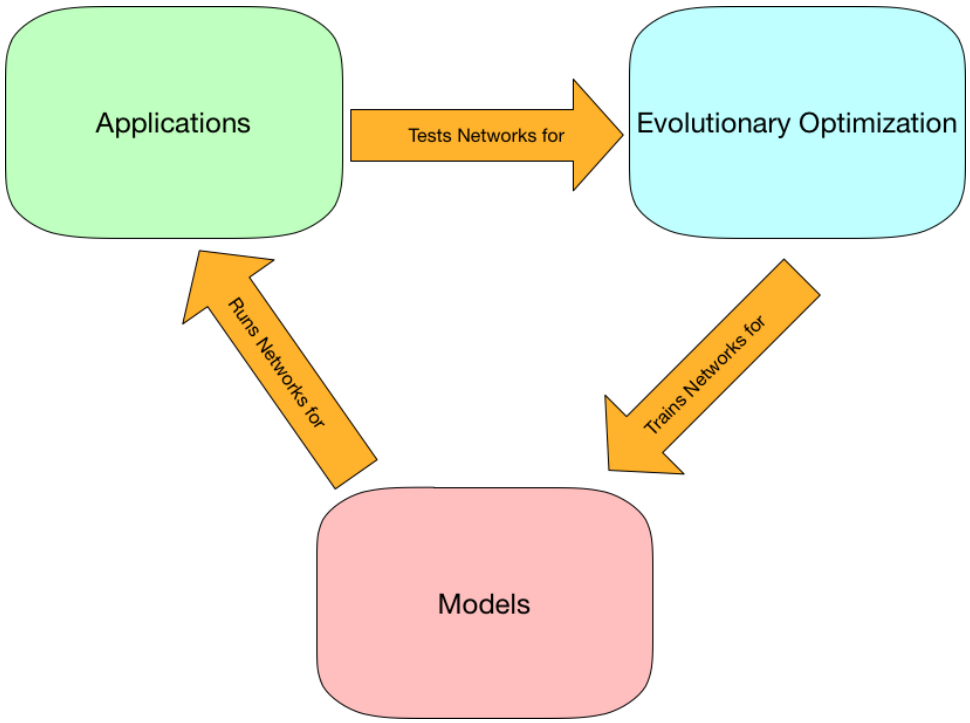


Figure 1.1: Interaction of the layers of the software framework

Models specify the behavior of the neuromorphic architecture. They provide functions to read and write representations of a network. Additionally, they provide functions to apply input into, and read output from a given network, and simulate the network for some specified time. Besides modeling the networks functional behavior, models are also required to provide implementations of the genetic operators random, mutate, and crossover. These operators facilitate the evolutionary optimization to create and modify networks.

Applications generate and run networks in order to accomplish some task such as classification, XOR, or balancing a pole[8]. To accomplish this an application encodes input into a network, interprets output from a network, and updates the state and performance statistics for the task the network is meant to accomplish. In addition to this, the application is responsible for measuring the fitness of the network. A networks fitness is a quantification of how well the network performs the task. For a classification application, fitness is usually represented with accuracy. For control applications, fitness is usually a measure of how long the network can perform the task before failure.

The evolutionary optimization piece of the software framework is used to generate networks for different applications. Using application defined parameters, it generates an initial population of networks from the models random function. It then tests the fitness of each network in the initial population. The networks with higher fitness are saved for the next generation of networks, while low performing networks are discarded. Next, the population is refilled by applying mutations to the high fitness networks in addition to performing crossover between two of the high fitness networks. This new generation of networks is then tested in the same way that the previous one was. Generations of networks continue to be generated until either a stop fitness value is reached, or the maximum number of generations is tested.

While the framework provides effective support for developers in neuromorphic computing, there are still many questions that make designing applications and systems difficult. First, the application decides how to apply input into the network, and how to interpret network output. This can be arbitrary, and there might not be an optimal encoding for all networks. Some neuromorphic models might benefit from more inputs, while others might benefit from less. If the output is looked at in a window, then it might be necessary to

understand what sort of average delay occurs within the model. Despite these difficulties, one of the advantages of the framework is the robustness of evolutionary optimization. Should a network require a longer than average delay to be successful, then evolutionary optimization will tend to select those networks as they should perform better. Though this does not solve the problems, it allows a developer to generate networks for specific applications, and to compare the performance of that network with other networks, even those generated without evolutionary optimizaiton.

Chapter 2

Memristive Neuromorphic Simulator

In order to study the behavior and efficacy of our particular memristive neuromorphic circuit, it was necessary to design a simulator of such a circuit. It was not feasible to use previous approaches as their designs were too different from our target hardware design and did not fit into our software framework. While there are tools for general hardware simulation such as Cadence Spectre, these tools are relatively slow, and do not scale well for large networks. Additionally, they also are not incorporated into the software framework described previously, so they are unable to examine the neuromorphic systems from an application level. Therefore, a new simulator was needed that could accurately capture the behavior of our design, as well as take advantage of the software framework and its applications and network generating ability.

Within the software framework, the memristive neuromorphic simulator acts as a model. Thus, the simulator provides tools for all of the functions of a model including the genetic operators. Additionally, the simulator provides tools to collect information about network activity, so that energy use can be estimated. In order to work with the software framework, it was necessary to write the simulator in C++. Because C++ is a compiled language, there is an advantage in speed over alternative interpreted languages. This advantage is important, as evolutionary optimization is computationally intensive.

The simulator source code is divided into component code and network code. Components represent the hardware units that make up a memristive neuromorphic circuit. The network code is responsible for setting up the topology of the network, simulating the network,

collecting activity information, and facilitating evolutionary optimization. This separation allows for behavioral changes to be made at a component level, but not at the network level. Therefore, it is simple to evaluate design decisions that occur at the component level without requiring changes at the network level. Similarly, design decisions at the network level can be evaluated with no consequence to the components.

2.1 Components

There are three major components that require simulation within the memristive neuromorphic circuits: neurons, synapses, and delay blocks. Each of these components has functionality to manage its own state information, communicate with other components, and collect any data to monitor performance if needed. Components are meant to be contained within networks, and are not meant to be accessed directly. In this way, it is not possible for an application to interfere with components at run time. Instead, the network code provides ways for applications to put input into, or retrieve output from the network, as well as set flags to collect information.

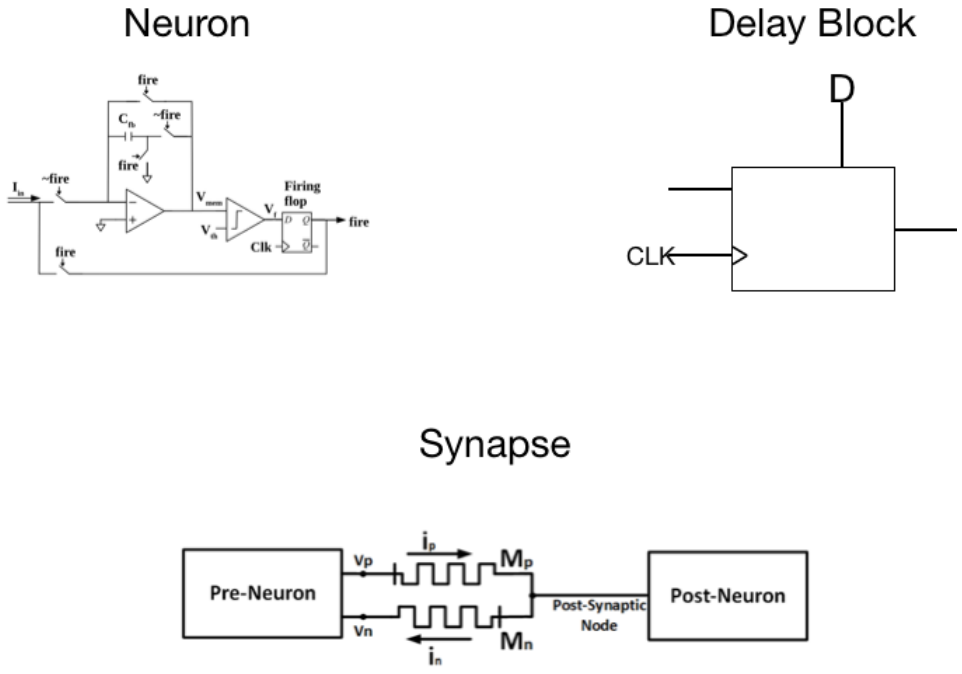


Figure 2.1: Circuit representation of important network components

Neuron

Neurons accumulate weight changes from their incoming synapses, and broadcast fire events to all of their outgoing synapses once the accumulated charge goes over some threshold [5]. Incoming synaptic fires, though they come in separately within the simulator, are handled as if they are simultaneous. The threshold is an integer, and it is assumed that there is some known ratio between the incoming synaptic weight and threshold. A neuron fire is not immediate, but instead occurs two cycles after the threshold is exceeded. During the first cycle, the neuron state variables are reset appropriately, and during the second cycle the neuron fires, sending an alert to its outgoing connections. A neuron cannot accumulate charge during any of these cycles as they constitute a refractory period.

When a neuron fires it goes into a refractory period. This is an integer number of cycles for which that neuron cannot accumulate any charge. After the refractory period ends a neuron is free to continue to accumulate charge. The charge accumulated by the neurons is a double value that is expected to be both positive and negative, and cannot go below a specified parameter.

Neurons also collect activity about how many cycles a neuron was active or passive, which cycles a neuron fired, and how much stored accumulation a neuron had in each cycle. This collection is optional, and can be turned on or off by setting the appropriate parameter in the parameters file. By default, collection is turned off. In order to facilitate online learning mechanisms, it is necessary for the neuron to also store information related to previous synaptic fires.

Synapse

Synapses are responsible for collecting their activity information and maintaining their internal state. The activity information measures whether a neuron was active or passive, and like the neurons collection, can be turned on or off. A synapses state is used to determine the effective synaptic weight. Synaptic weights are initialized as integers, but can take on real values up to a maximum magnitude if they are updated through online learning. Without online learning, the synaptic weight only needs to be mapped to the threshold of a neuron.

Typically, this ratio is kept one-to-one for simplicity in reasoning about the network, but might need to be updated to maintain integrity to the behavior of the hardware.

To facilitate online learning it is necessary for a synapse to maintain information about the state and properties of its memristors. The hardware implementation of the synapse uses two memristors to allow for both positive and negative weights. As a consequence, online learning is achieved by updating both memristors [24]. This update, however, causes a nonlinear change on the weight, and requires knowledge about the memristors previous states. Therefore, when online learning is being used, the simulator requires a mapping from the state of the memristors to an effective synaptic weight. This mapping is implemented when a synapse is first instantiated. First, a normalization factor is derived by the following equation: With this normalization factor, it is now possible to initialize a synapse to any desired synaptic weight with appropriate memristance values. Furthermore, when online learning occurs, a synapse can now update its memristance values, calculate its new effective conductance, and then normalize that value to the new synaptic weight. The switching behavior of the memristors in the synapse is based on a model proposed by Amer et al. [1].

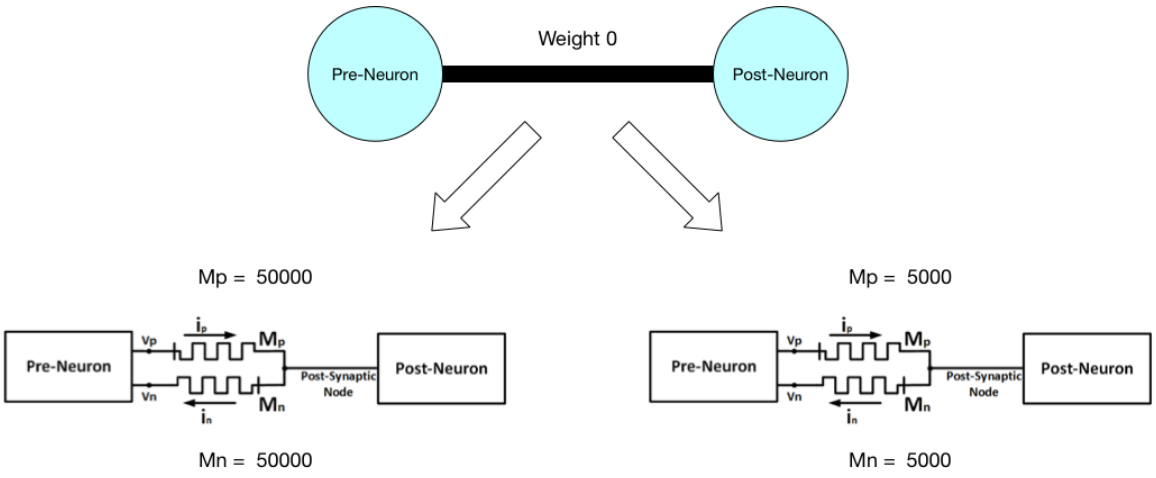


Figure 2.2: A synaptic weight can be realized through multiple combinations of memristances

Delay

The delay unit is the last major component, and is responsible for adding cycles between the time a neuron fires to its outgoing connections, and the time that signal is received by

the outgoing neurons. Though they are separate entities in the hardware, the delay unit can be thought of as part of the synapse, as each synapse has its own delay unit. The time delay is an integer number of clock cycles. If a delay were long enough, it is possible to have multiple neuron fires being processed simultaneously by the same delay unit. The signals are processed using a cyclic array, and when a signal reaches the end of that delay, a signal is sent to the delay units corresponding synapse. The synapse then passes this signal through to an outgoing neuron, which will process the signal appropriately. In this way, synapses are passive in the way they handle fire signals, as they are passed through the synapse when they are received. Besides acting as a temporal delay, delay units collect activity information for how many charges pass through the unit. This statistic is used along with the cycle length of the delay unit to estimate energy consumption.

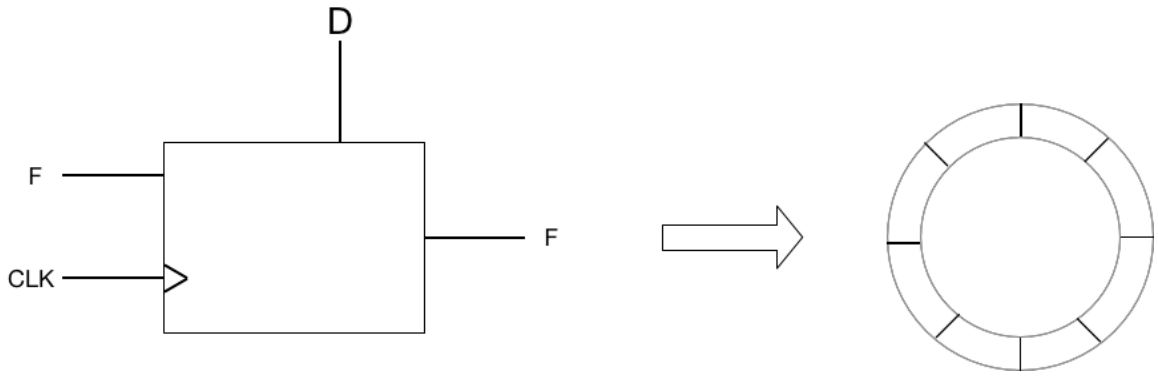


Figure 2.3: The delay unit is modeled in the simulator using a programmable cyclic array

Besides the three major components there is an additional component known as an output element. Output elements are purely abstract and do not have any hardware implementation. These components are added as an outgoing connection to any output neuron. When an output neuron fires this event is considered to be visible to outside entities. Thus, output elements record these fires in a network level data structure so that it is possible to track when output neurons fire.

2.2 Network

The network code for the memristive neuromorphic simulator is responsible for managing the topology of the components, advancing the state of the simulation, reading and writing networks, and facilitating evolutionary optimization. In these roles, the network serves as a layer above the components. While it might be necessary to get the state of the components, this is done through and facilitated by the network.

Topology is handled within the network through the element interface. There is a class named element within the components code that specifies which actions components can take to communicate between one another. The primary two functions in this role are apply and update. Update is a function that the network calls to advance the state of the simulation for each required component. For example, a neuron in its refractory period will move out of the refractory period if update is called, and the appropriate amount of cycles have passed. Updating a delay will cause any signals in the delay pipeline to move one stage forward. If a signal reaches the end of the pipeline, then the delay unit will alert its outgoing synapse. This is done with the apply function. Apply is the function that components use to alert each other of a spike being passed between one another. For example, when a synapse calls apply for its outgoing neuron, the neuron will update its stored accumulation based on the weight of the synapse that made the call, and prepare itself to fire if it crosses the threshold. An example of the code for the apply and update functions is displayed in Fig. 2.4.

Besides apply and update, the element interface provides functions for resetting a component, getting activity information, or facilitating online learning. Resetting is simple, and is initiated from the network level. The network broadcasts to all components to reset. The implementation of reset is different for each component, but they are all similar in that they reset any state information about the component to its initial state. Activity collection is triggered by the network in the same way a reset is. The implementation for each component is described earlier. Finally, the element interface provides a way for networks to provide feedback to one another. Normally, components are only concerned with propagating signals to their outgoing connections. However, online learning in synapses requires the neuron to send information to its incoming synaptic connections.

```
//Apply - How other elements signal the delay block to add a new charge
void delay::apply(float w, int clockCycle, element* sender){
    chargesPassed++; //energy estimation

    //Put spike into cyclic array
    if(head != 0){
        charges[head - 1] = w;
    }
    else{
        charges[charges.size()-1] = w;
    }
}

//Update - How the network signals the delay block to advance the simulation by 1 cycle
void delay::update(int clockCycle){

    //Pass on spike if it has reached end of delay block
    if(charges[head] != 0.0){
        for(ElementMap::iterator it = outputElements.begin(); it != outputElements.end(); ++it){
            it->second->apply(charges[head],clockCycle,this);
        }
    }
    charges[head] = 0.0; //reset value

    //manage cyclic array
    if(head != (charges.size() - 1)){
        head++;
    }
    else{
        head = 0;
    }

}
```

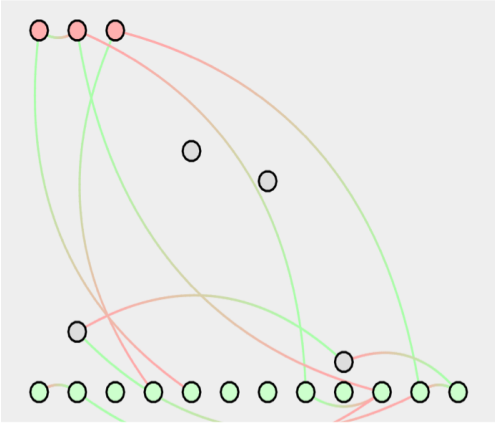
Figure 2.4: Code snippet for apply and update functions for the delay block

One of the most important functions of the network code is to manage and run the simulation of the circuit. Simulators typically accomplish this task in either a cycle-accurate simulation or an event simulation. A cycle-accurate simulation simulates a system one cycle at a time, and processes all of the activity for that cycle in the necessary order. In contrast, event driven simulators only simulate activity that defined by events. Events are placed in a queue, and processed chronologically. An event can be added to the queue through some external stimulus on the system, or can be added as a result of processing other events. Both simulations have distinct advantages and disadvantages [18]. Event simulators only need to simulate activity within the network. If a simulations activity is particularly sparse, then there can be significant performance benefits from only simulating the active parts. Cycle-accurate simulators, in contrast, simulate every cycle whether there is activity or not. While event simulators tend to outperform cycle-accurate simulators in simulation time on average, they tend to perform better for dense activity. For these cases, the event simulator incurs additional overhead for processing the events, which the cycle-accurate simulator

would default to anyway. Additionally, optimizations can be made at the cycle level in cycle-accurate simulators where they cannot be made in an event based simulation.

In designing the simulator, one of the most important design requirements was that the simulator should simulate the hardware behaviorally with little to no error. The simplest way to capture the behavioral information of the networks is looking at whether or not a neuron fired on any particular cycle. By ensuring that all of the neurons are firing on the same cycles in both the simulator and hardware, we can be confident that our simulator is accurate. Further verification that the simulator is accurate requires a deeper look at the components. For further analysis, the simulator state of each component can be compared to the state of the hardware. These states can be accumulated voltages, memristance values, or any other important state variable within the network. With this state information, it is also much easier to understand differences between the simulator and the hardware. For example, there might be some edge case where the hardware causes a neuron to fire due to variance, but the simulator would not fire. Analyzing these states provides insight into the differences. Therefore, in order to achieve an accurate simulation, a cycle-accurate simulator was chosen. This made it simple to obtain cycle by cycle information about components, while also not hurting the performance in such cases. An event simulator would require collection events to be injected and processed at each cycle, or would have required interpolation of those states if possible. With a cycle-accurate simulator it is easy to advance the state of the network. For each cycle, the network simply calls update for all of the components, and records activity if necessary. The network continues until the maximum number of cycles to run for is reached.

In order to pass networks throughout the software framework, it is necessary to represent them as text strings. Thus, the network has code to both read and write networks to and from a string or file. The format is intended to be human readable, and an example network is displayed in Fig. 2.5. The first line specifies how many dimensions the network is embedded into, which will be described later in chapter 3 on evolutionary optimization, and should be an integer number. On the next line are the maximum lengths of each dimension. These are expressed as real values. The third and fourth lines specify the number of input and output neurons. Lines starting with either an I, O, or N mark the start of a neuron declaration. An



```
Embedded: 2
MaxDims: 6.000000 6.000000
In: 4
Out: 2
I 0 0.000000 0.000000  Refrac: 3 Thres: 3
 S 4 D 0
 D 2 W -9 I  2.000000 0.000000
O 0 0.000000 6.000000  Refrac: 3 Thres: 4
I 1 1.000000 0.000000  Refrac: 3 Thres: 1
 S 2 D 0
 D 6 W 5 O  0.000000 6.000000
 D 1 W 3 N  2.000000 1.000000
 D 3 W 4 N  3.000000 3.000000
O 1 1.000000 6.000000  Refrac: 3 Thres: 1
 D 1 W -2 O  0.000000 6.000000
I 2 2.000000 0.000000  Refrac: 3 Thres: 1
 S 2 D 0
N 2.000000 1.000000  Refrac: 3 Thres: 3
 D 5 W 4 O  1.000000 6.000000
 D 1 W -7 I  2.000000 0.000000
 D 1 W 5 I  3.000000 0.000000
I 3 3.000000 0.000000  Refrac: 3 Thres: 7
 S 8 D 0
 D 7 W 4 O  0.000000 6.000000
N 3.000000 3.000000  Refrac: 3 Thres: 8
 D 2 W 3 N  2.000000 1.000000
 D 2 W 6 N  5.000000 1.000000
```

Figure 2.5: A visualization of a network(left) and a string representation of a network(right)

I means the neuron is an input neuron, an O means the neuron is an output neuron, and an N means the neuron is a hidden neuron. For input and output neurons the next term on the line is an integer representing that neurons input or output ID. Input and output IDs are used to ensure that input and output is consistent. For hidden neurons there is no term for ID. The next n terms, where n is the number of dimensions, are real coordinates of that neurons place in the embedding. These values are used in evolutionary optimization, and serve no functional role in the network. The next two terms on the line specify a neurons refractory period, and the final two terms specify that neurons threshold. Lines following a neuron are used to specify synapses going out from that neuron. There can be an arbitrary number of synapses, and the list is discontinued when the next neuron line is reached or the end of the file is reached. An input neurons first synapse line will always start with an S to indicate the weight and delay of the input synapse. Note that S is only used to represent an input synapse, and a different character is used to represent a regular synapse. The next term is an integer value indicating the weight of the synapse. After the synaptic weight is the character D followed by an integer to specify the delay. This delay is set to 0 by default, and the synaptic weight is defaulted to be a weight that will force the input neuron to fire

if there is no negative accumulation. For non input synapses, the lines in the network file begin with the D character followed by the integer delay value. The next two terms are the character W followed by the synaptic weight of the synapse. Finally, the last n terms indicate the coordinates of the neuron that the synapse fires to. Networks are both read and written in this format.

Networks are meant to function in two distinct ways. The first is acting as a model in the neuromorphic software framework. This function is effective for generating and measuring the performance of networks. However, it might be necessary to test the network on a set of predefined inputs, or test a network that was generated by hand. For such cases, the network provides support for loading a network from a file, and simulating for an arbitrary number of clock cycles with inputs from a user provided file. These input files contain a line for each clock cycle of simulation. The first two terms specify which clock cycle the line represents. The next 2n terms, where n is the number of input neurons, are used to specify whether or not there is input on each of the inputs.

2.3 Evolutionary Optimization

Evolutionary optimization has proven that it can be an effective way to generate networks for a variety of tasks without having to fix a topology before training [28]. In this way, the developer of an application or a model do not have to worry about selecting topologies ahead of time. If an application can find a way to define inputs and outputs, then a genetic algorithm can begin to look for candidate solutions without any knowledge of what a good solution might look like. This is in contrast to other methods of training neural networks, where a topology and some learning parameters are selected before training. Multiple approaches were considered, but eventually evolutionary optimization for the memristive neuromorphic model was implemented using an embedding similar to NIDA. This embedding is managed by the network and neurons, and is an integral part of the random, mutation, and crossover operations.

One of the challenges of implementing the genetic operators was determining what an effective approach might look like, and determining whether that approach can function

within the software framework. The first method considered was an approach that treated the network as a graph made of vertices and edges. Neurons would act as vertices within the graph, and an edge would be composed of a synapse and a delay unit. With this approach, implementing the random operator would be simple, as there are many well studied methods for generating a random graph such as Erdos-Renyi and the Gilbert method to randomly generate arbitrary graphs [9] [12]. Mutation would also be simple for these graphs, as it would be trivial to add or delete vertices and edges from the graph, or change the properties of a random vertex or edge. The challenge, however, with this approach is the implementation of the crossover operation. Using graphs for crossover is intuitively desirable, because graphs inherently maintain structure, while the goal of crossover in our neuromorphic model is to combine two proven computational structures to achieve a superior computational structure. Graph partitioning is a particularly difficult problem, however, and most of the study of the problem has focused on deterministic methods for partitioning the graph [3]. While deterministic methods are desirable in many domains such as clustering, it is desirable that the crossover implementation be random for our implementation. If not, then performing crossover multiple times will yield the same structure, while we would like to be able to test many different structures. One idea to remedy this was to select a random partition of the graph, but this was determined to be too computationally difficult. Therefore, even though graph partitions present an intuitive potential way to do crossover, they were determined to be too difficult to implement in practice.

Another method considered was one based on the Neuroevolution of Augmenting Topologies(NEAT) algorithm [30]. Unlike other neuromorphic genetic algorithms, NEAT uses a speciated approach to ensure diversity in its candidate solutions. In this way, NEAT tracks the history of networks within the population, and groups them based on how similar their structures are. One of the motivations for NEAT is that by preserving species, it is possible for some candidate networks to succeed where they otherwise would be unable to. For example, it is possible that one network structure starts at a relatively high fitness, but does not offer much room for growth. We can then consider another structure which, in contrast, might start out at a relatively lower fitness, but has the potential to become a near optimal network. A typical genetic algorithm would discard networks of the second type,

and populate itself with networks of the first. NEAT, however, would preserve both networks due to their unique structures, and allow both to advance toward candidate solutions. In this way, we can be confident that NEAT will allow for topological diversity. Despite these advantages of the NEAT algorithm, there are challenges in implementing NEAT for memristive networks. The first major challenge comes in the temporal components that are present in the memristive networks, but are absent from NEAT. This can be overcome by averaging the clock cycle delay for any common synapses in the two networks being crossed over, or simply taking delays from network for one of the offspring, and another for the other. One of the other challenges with NEAT is that NEAT was designed for artificial neural networks which do not generally have a spiking implementation. As our neural networks are spiking networks, it is unclear what the impact would be on NEATs effectiveness. The primary challenge with NEAT comes from the fact that it is not simple to include in the software framework. The evolutionary optimization piece of the framework currently does not have support for speciation. This approach would require an implementation of NEAT that adheres to this framework. In addition, model developers would now be forced to support speciation with NEAT by providing compatible network representations. Because there were still many open questions about the feasibility of the hardware implementation, it was decided that the focus should be on making the memristive model compatible with the framework, instead of adding to the framework to support the model. Therefore, despite the appeal of the NEAT algorithm, it was ultimately decided that the implementation was not feasible at the time.

The approach that was ultimately selected was based on the two other models in the neuromorphic software framework, NIDA and DANNA [27]. NIDA networks are entirely abstract, and have no hardware implementation. The DANNA model, however, models the FPGA implementation of DANNA. To facilitate evolutionary optimization, NIDA networks are embedded in a three dimensional space. In contrast, DANNA maps its networks directly onto the FPGA. When designing the genetic operators for the memristive model there was no sense of how hardware components would be mapped. Therefore, it was decided that an approach similar to NIDA would be better. Not only has this approach been shown to work, but the networks could be generated, and then later mapped onto the hardware with some

other software. In this way, the memristive model facilitates evolutionary optimization by embedding the networks it generates into a two dimensional grid. This approach allows for simple and proven implementations of the genetic operators, and can easily be incorporated into the software framework.

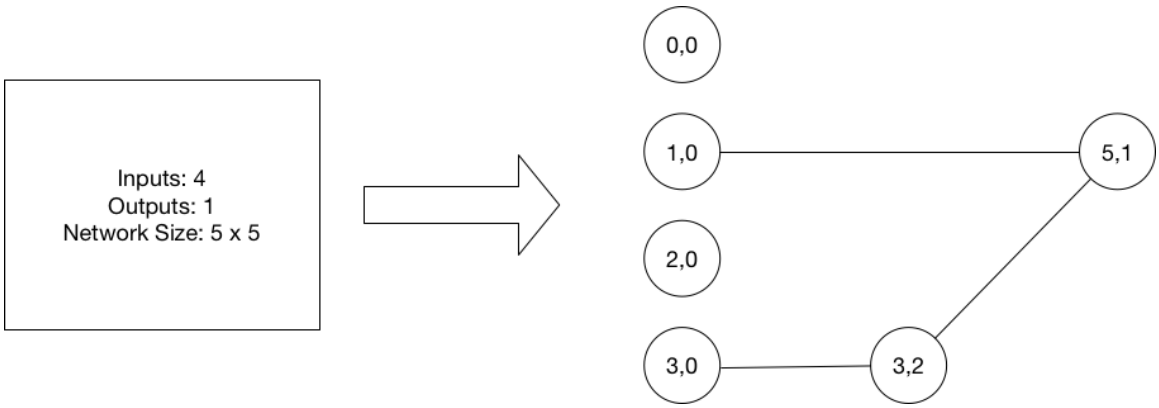


Figure 2.6: Random takes application specific starting parameters and produces a sparse random network

Random

The random operation is the simplest genetic operator implemented for the memristive model. Random creates a sparse random network that is meant to fill the initial population. This operation requires that the number of inputs and outputs be set by the network constructor using a configuration string. Typically, the application will be responsible for creating this string and initializing the parameters. Using this function outside of the software framework requirez that the constructor be invoked manually by the developer. Input neurons are placed along the first dimensions, and are placed one unit of distance apart from one another. All other coordinates for the neuron are zero. Output neurons are placed along the opposite end of the second dimension, such that they are placed at the maximum value for that dimension. They are spaced similarly to input neurons for the first dimension. From there, between one and five hidden neurons are added into the network. These neurons cannot be placed where input and output neurons are located, and are placed at integer coordinate pairs. Thus, the number of neurons in a network is limited by the dimensions. Because the dimensions are defined by application parameters, it is up

to an application developer to pick a parameter such that the dimensions allow for feasible networks. All of the neurons generated by this function have a random threshold between one and the maximum threshold specified in the parameter file. The refractory period is randomly generated from parameters within the parameter file. These parameters allow for variable and fixed refractory periods of tunable durations. After placing the random neurons on the grid, between one and ten synapses are initially placed into the network. This ensures that our initial networks are sparse, and allows the genetic algorithm to build up candidate networks. The source and destination neuron are selected at random. The magnitude of the weight of the synapse is a random integer between one and the maximum value specified in the parameter file. The sign of the weight is then determined randomly. Setting the delay of the synapse is done by calculating the Euclidean distance between the neurons, and rounding up to the nearest integer. In this way, the delays inside the network are a function of the distance of the neurons. This allows for local structures to emerge which helps in the crossover operation.

Mutate

Mutation is the second genetic operator models are required to implement in order to make use of the evolutionary optimization provided by the software framework. The implementation for mutation for all models should take a network and attempt to modify it in some nontrivial way. The memristive neuromorphic model will perform one of seven random mutations. Each mutation has the same probability of occurring. The first mutation is adding a neuron to the network. This mutation generates a random pair of integer coordinates, and attempts to place the neuron there. It will continue trying to place a neuron at other destinations one hundred times if a coordinate pair is occupied. After placing the neuron, two neurons are randomly chosen from the existing network. One synapse is added from the first existing neuron to the new neuron, while another is added from the new neuron to the second existing neuron. This helps ensure that the neuron plays a functional role in the network. Otherwise, synapses would need to be added to this neuron through another mutation. In contrast to the first mutation, the second mutation deletes a neuron from the network. For this operation a hidden neuron is randomly selected, and deleted from the

network. With the deletion of the neuron, all of its incoming and outgoing synapses are deleted as well. It is important that this operation does not delete input or output neurons, as they are considered static. One future approach, however, might allow for the deletion of these neurons from the functional network, but allow them to continue as virtual. This might desirable if applications are developed for which there are many inputs, and it is not clear which inputs are valuable. In this case, deleting an input neuron that causes a negative impact on performance might be beneficial. Besides adding and deleting neurons, mutations exist for adding and deleting synapses as well. These implementations are much more straightforward, as adding or deleting a synapse has less of a topological impact on the network, and can make more of an immediate impact. Adding a synapse simply involves randomly selecting two unique neurons, generating a random synaptic weight as described in the random operation, and calculating the appropriate delay as described in the random operation. For deletion, a synapse is selected at random, and is deleted from the network.

Besides adding or deleting structures, the only other mutation that alters the topology of the network is a mutation that moves a neuron to a new location in the embedded space. While this mutation does not affect the connectivity of the network, it does have an impact on the temporal part of the network. First, the network tries up to one hundred times to find a new integer coordinate pair that is not currently occupied. The neuron is then moved to that location, maintaining any previous connections it had. Finally, the clock cycle delay is recalculated for all of the neurons corresponding incoming and outgoing synapses. Thus, though there is no change in the connections of the network, the delays will be altered affecting the temporal behavior of the network.

The final two mutations adjust the threshold and synaptic weights of existing neurons and synapses. For adjusting thresholds, a random neuron is selected, and has its threshold randomly set as if it were being generated in the random operation. For input neurons, the weight of the corresponding input synapse is adjusted appropriately to force a fire assuming no negative accumulation as was mentioned previously. Adjusting a synaptic weight is simple as well. A synapse is selected at random, and its weight is determined in the same way as the random operation. Currently, input synapses are excluded from this operation, as well as the deletion operation. This ensures that their behavior is uniform, and keeps them

as static elements in the network. However, allowing mutations for input synapses might serve similar purpose as virtual input neurons for some applications, and could be worthy of further exploration.

Crossover

The final genetic operator that a model must implement is the crossover operation. For all models, the crossover operation takes two known networks as its input, and creates two networks as its output. The two output networks should be generated as some meaningful combination of the input networks. For a graph approach this might be combining partitions from two graphs into one structure. For NIDA, a random plane is generated to cut the two networks in half, and combines one half of the first network, with the other half of the second. Then, the other half of the first network is combined with the unused half of the second. The halves are chosen such that dimensionality of the two new networks will be the same as the parents. This approach will preserve any local structure on either half of the cutting plane, and hopefully will combine two high fitness structures into a network that has a higher fitness than both of the parents. For the memristive model a similar approach has been adopted. Because memristive networks are embedded in two dimensions instead of three, a line is used to divide the network in half. Furthermore, this line is parallel to either of the two boundaries of the space. The direction that the line is parallel to is chosen at random, and the location along the dimension is random as well. The same line divides both parent networks. In this way, the crossover operation is similar to DANNA, which is also laid out on a grid, but DANNAs grid embedding is functional while the memristive models embedding is purely abstract to facilitate evolutionary optimization. After selecting the line to divide the network, inputs and outputs from one parent network are placed into one of the offspring networks, and inputs and outputs from the other parent network are placed into the other offspring network. As these components are static, with the exception of their thresholds, across networks it will not disrupt the crossover.

After the static components are instantiated inside of each of the offspring networks, copies of the hidden neurons from each parent network are divided among the offspring networks based on which side of the cut line they are on. For example, a child will get copies

of one set of neurons from one side of the cut from the first parent network, and copies of a second set of neurons from the other side of the cut from the second parent. These two sets of copied neurons, along with the static neurons input and output neurons, will form the full set of neurons for this new child network. Synapses are then added into the network after the neurons. First, input synapses are added with appropriate weights for their corresponding input neurons. Then, for each neuron in the new child network, its copy in the parent network is examined. For each outgoing synapse, the destination neuron is found, and it is determined whether or not a copy of that neuron exists in the child. If a copy does exist, then the synapse is added with the same weight and delay value as the parent network. If a copy does not exist, then the potential synapse is added to a list to be processed after all other potential synapses are examined. Finally, the synapses whose destination neurons did not exist in the child network are processed. For each of these synapses, the closest neuron to the source neuron is found. If a synapse does not already exist between these two neurons, then a synapse is added between them with the same synaptic weight as the parent network. The delay for this synapse is determined to be the Euclidean distance between the two neurons. Note that if a synapse is connected to a static neuron then it will remain in the network despite where a cut occurs, as these neurons are always present. After this process is done for both child networks, crossover is complete.

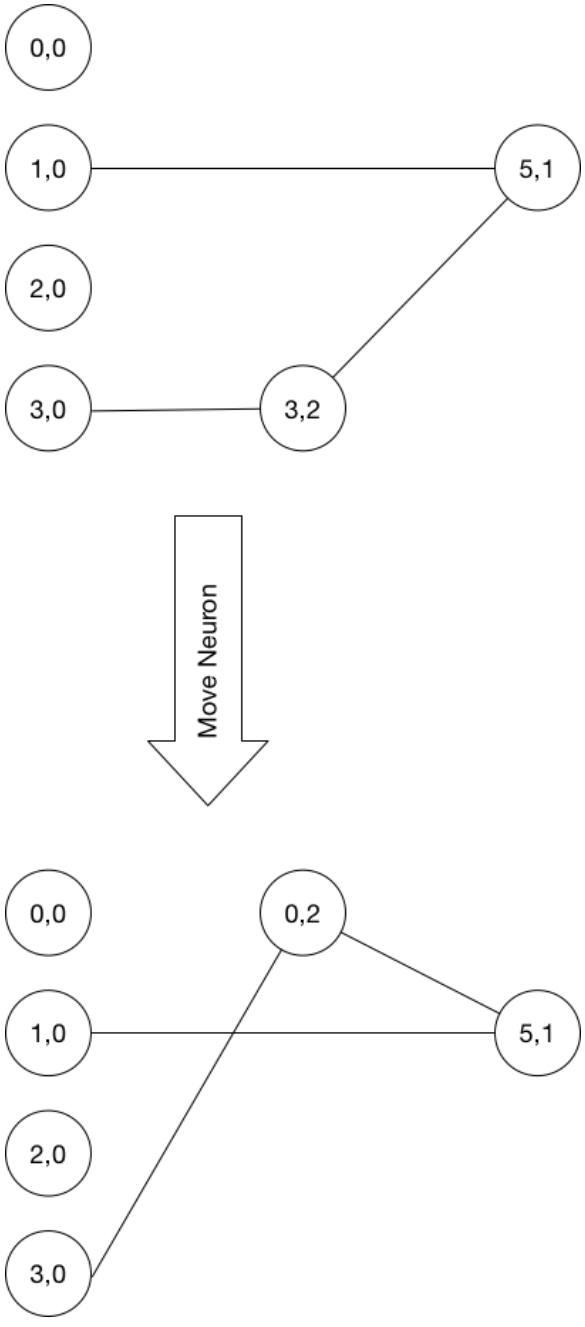


Figure 2.7: The move neuron operation optimizes the temporal components of the networks

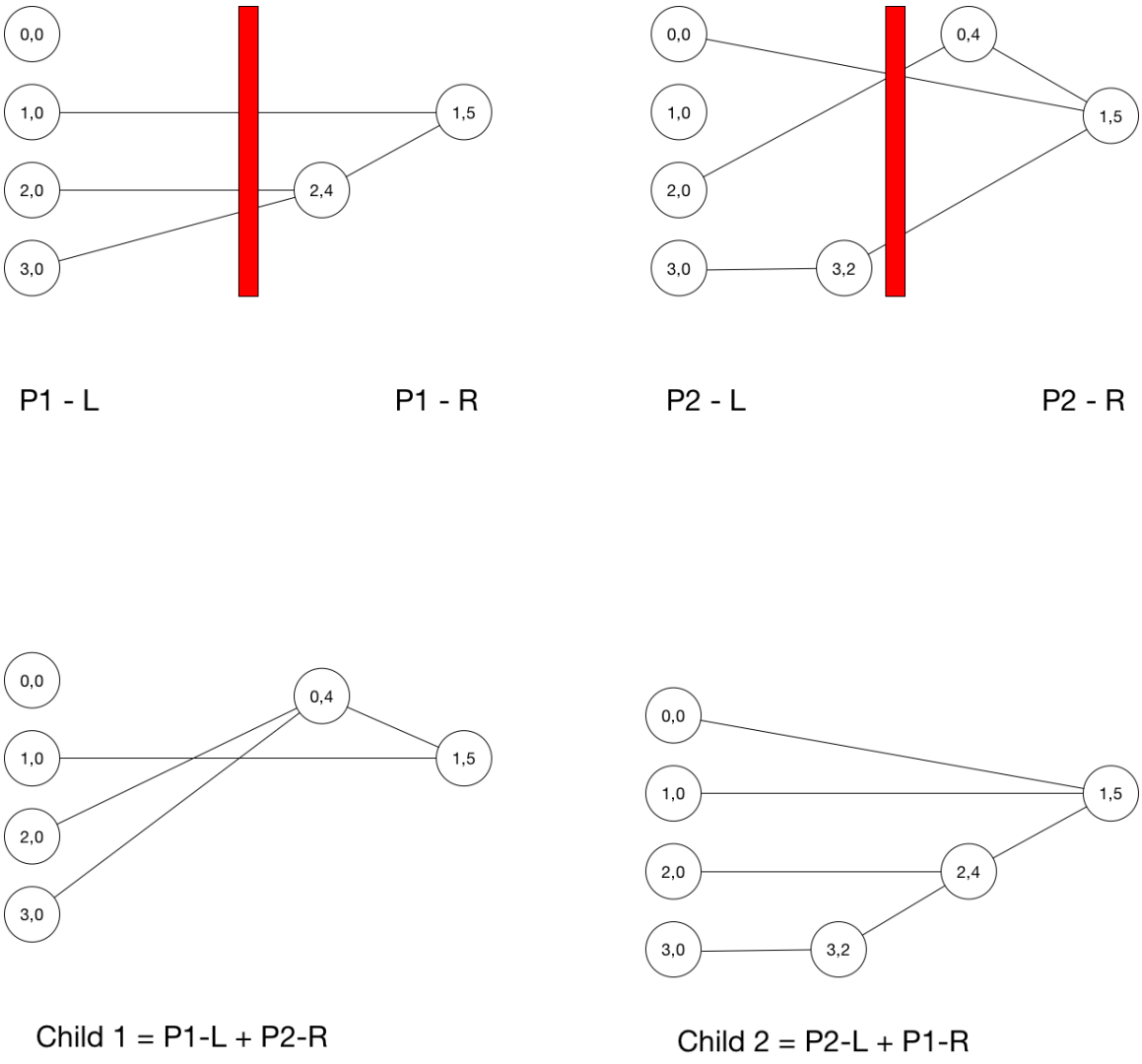


Figure 2.8: An example of the crossover operation for two arbitrary memristive networks

Chapter 3

Experimental Verification and Analysis

One of the major benefits of incorporating the memristive neuromorphic simulator as a model in the software framework is the ability to understand how the design fairs at an application level. Before making any design evaluations based on the results of the simulator, it is important to verify that the simulator is accurate with respect to the hardware. For this purpose, networks were tested both in the high level model simulation, and the low level Cadence Spectre simulation. The fidelity of the networks was evaluated with a suite of metrics. After the simulator was confirmed to be an accurate representation of the hardware implementation, the simulator was then used to answer questions about the feasibility of design parameters, alternative neuron designs, and comparisons to other models in the software framework.

3.1 Verification

The memristive C++ simulator gains speed over traditional hardware simulators such as Cadence Spectre by sacrificing granularity. Whereas the hardware simulator must simulate each transistor in the circuit, the C++ simulator is only concerned with capturing the behavior of the higher level circuit components. However, it is important the behavior predicted by the C++ simulator is the same as that of the hardware simulator. Otherwise,

any conclusions we draw from the C++ simulator could not be made for the hardware. Therefore, four metrics were used to verify that the behavior of networks in both simulators was the same. The four metrics measure the accuracy of neurons in the simulator. To measure the behavior of neurons we look at whether or not a neuron fired in a given clock cycle. For that neuron on that clock cycle, the bit 0 is assigned if the neuron did not fire, and the bit 1 is assigned otherwise. Then, a bit string for that neuron is formed by placing that neurons bits in chronological order. These bit strings are generated from both simulators. For simple networks with sparse activity, the outputs of the simulators can be compared by hand. For example, the simple network in Fig. 3.1 was created to verify the behavior of the high level simulator was consistent with the circuit level simulator when online learning was enabled. Sample output waveforms are displayed in Fig. 3.2. From the figure it is clear that potentiation and depression in both simulators changes the network behavior over time. When initially two sets of fires on the input neurons were needed to produce an output neuron fire, after two fires on the output neuron only one set of fires on the input neurons is needed.

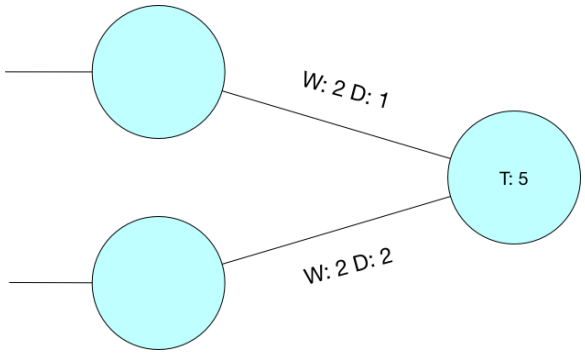


Figure 3.1: Simple network simulated at the high level and circuit level

The simplest metric used to compare neuron bit strings is the matching percentage. This metric is the percentage of cycles that both simulators agree about a neurons behavior. An ideal score for this metric is one hundred percent, or that the behavior matches perfectly. While this metric is simple, and ideally it would always be perfect, there are quirks in the hardware behavior that are not captured by the simulator. Variations and noise margins can either cause a neuron to fire or not fire when it is close to a threshold and the opposite behavior would be expected. In some extreme cases this can cause this metric to make it

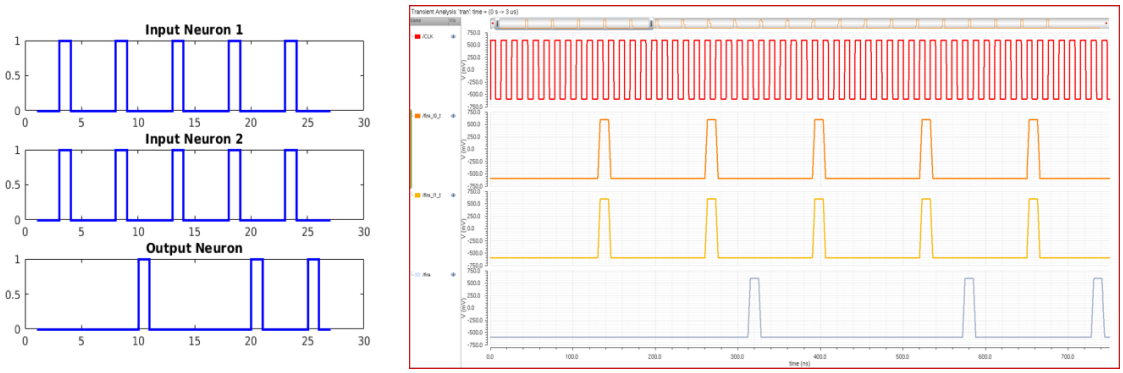


Figure 3.2: Output from the C++ simulator plotted using MATLAB(left) and output from Cadence Spectre(right)

look as if the C++ simulator is highly inaccurate when this is not true. Consider the case of a highly active neuron that fires every other cycle. If early on the hardware had a glitch or variation that shifted the fire by a cycle, then the strings would have a nearly zero match percentage. Despite the low score from the match percentage metric, both simulators display almost identical behavior. Additionally, most applications track the output in either some window or the number of fires over some specific time. Therefore, because the applications are already resistant to such noise, it is helpful to have a metric that is resistant to this noise as well. To combat this case, the edit distance metric was incorporated into the verification tools [33]. Edit distance is a well known way to measure the difference between two strings. There is a cost associated with making an addition or deletion to alter the strings so that they match. We set these costs to be one, and in the case that both strings match the edit distance will be zero. For the example presented above, a low cost addition the beginning of one string, and the end of the other string will make the strings match. In this way, even though the strings displayed a high mismatch percentage, they will also display a low edit distance. Because of the possibility of variation that the C++ simulator cannot account for, edit distance provides another revealing metric about the accuracy of the simulator.

The errors that necessitate metrics such as edit distance are a reality of the limited granularity of the C++ simulator. In order to characterize the errors that occur, two more metrics were proposed. The first is time to error. This metric measures how many clock cycles occur before a difference in the two simulations arises. Thus, with this metric it is possible to characterize how long it takes before errors begin occurring for different applications. This

metric, much like the match percentage, also suffers from the problem of characterizing some cases. If we consider a network that has one error at the start of the simulation, and one other error at the end, then we might think our simulation is inaccurate because it immediately makes an error. By introducing a metric to track the average time between errors, however, we can see that the errors that occur are far between. In fact, such a network would have a low edit distance value as well as a high match percentage. Thus, it is important that all of the metrics presented be used to characterize the error that occurs. Not only do these metrics provide a tool to verify that the C++ simulator is accurate, but they also help in characterizing errors, so that the simulator can be fixed, or the error can be classified as benign.

In addition to verification on a simple handcrafted network, the simulator was also verified for a network generated using evolutionary optimization. A network was generated to perform the XOR operation, and tested in both the simulator and Cadence Spectre. Table 3.1 contains the results. All but one of the neurons displayed good fidelity to the hardware. The seventh neuron, however, displayed multiple errors. When looking back at the output files from both simulators, the errors on this neuron occurred as the result of a hardware glitch. Particularly, this neuron fired two cycles in a row, and the error cascaded in the behavior in later cycles. Thus, the verification scheme can be used to find errors in the hardware implementation as well as characterizing error.

Table 3.1: Table of verification metrics for an XOR network

Neuron	1	2	3	4	5	6	7
Percent Match	99%	100%	100%	100%	99%	100%	83%
Time To Error	96	N/A	N/A	N/A	98	N/A	13
Time Between Error	3	N/A	N/A	N/A	1	N/A	4.11
Edit Distance	1	0	0	0	1	0	17

3.2 Analysis

Having verified the correctness of the C++ simulator, it is now possible to use the simulator to explore the impact of certain parameters at the hardware level on the viability of networks at the application level. For example, one might wonder whether the device characteristics of a particular memristor allow for the generation of networks for a particular application. If the evolutionary optimization were able to find high fitness networks using that device, it would suggest that the device is a viable candidate for implementing the memristive neuromorphic circuit. If, on the contrary, networks were unable to be generated, it would be revealing that those devices would be inadequate. Not only can a designer test the viability of devices, but they can also measure the impact certain tunable parameters such as the clock frequency have on the types of networks generated. One could even quantify the difference in average power consumption of networks generated using both frequencies. Most of the analysis will focus on measuring the feasibility of specific designs and choice of devices in comparison to existing models, particularly DANNA.

3.2.1 Variation

As an emerging technology, memristors are particularly prone to process variations. These variations have been shown to affect the extreme resistance states, as well as the switching characteristics of the memristors [21]. Besides process variation, there is also variation on-chip while the devices are running. The pulses used to switch the memristors are not exact, and it cannot be expected that a memristor can be set to a precise memristance value. While there are techniques to understand the impact of these variations at the circuit level, it was uncertain whether these variations would affect the neuromorphic system at the application level.

To understand variation at the application level, the simulator was expanded to include adjustable parameters that add variability to memristor parameters. Particularly, when the simulator instantiates a synapse it normally assumes the nominal resistance states and switching properties. When the appropriate variability parameter is set "ON", however, these parameters are initialized to a normally distributed random value. Therefore,

these variability options can allow the user to understand the application performance of memristive networks given uncontrollable variations.

The first application explored in this way was the simple XOR application. While not a particularly difficult application, it can be important in characterizing the impact of a change to the model. If the variation made it impossible to complete an XOR, then this would be informative about the types of applications that this neuromorphic device could be used for. There were two cases considered in testing networks with the XOR application. First, networks were trained with no variation, and then tested with noise. This set of tests was used to measure whether the trained networks were resistant to variation in their synaptic weights. Second, networks were trained and tested with noise. This was done in parallel with the first test case with the intention to understand whether training with variation provided any performance advantage at an application level. Ten networks were trained for each test case. For the first case, networks were tested with three different variation profile: cycle-to-cycle variation only, process variation only, both cycle-to-cycle and process variation. For all three noise profiles across all ten of the XOR networks there was no impact on the ability of the networks to perform the XOR operation. All ten networks trained with noise also tested perfectly.

Next, the impact of variation was measured for the iris classification application. The iris classification application involves training a network to classify iris flowers from the well known iris flower dataset [19]. To measure the impact of variation, ten networks were trained with and without variation. After training, the classification accuracy was then measured on the test set for the different variation profiles. Like in the case of the XOR application, the variation did not impact the performance of any of the networks generated. Thus, for simple tasks, it can be concluded that memristive networks generated using evolutionary optimization are resistant to both process variation and cycle to cycle variation.

3.2.2 Axon-Hillock Neuron Model

Not only can the simulator be used to test alternative design parameters, but it can also be used to test alternative architectures. Particularly, an alternative neuron model was explored. The neuron described in previous sections is an integrate and fire neuron. However,

the Axon-Hillock neuron has been put forth as an alternative to this model [34]. The Axon-Hillock neuron is different in three primary ways from the integrate and fire neuron. First, the amount of charge accumulated by the neuron is non-linear, and based on the current accumulation. Second, parallel synaptic fires cause an average accumulation from the synapses as opposed to a sum of accumulations. Third, the weight of the synaptic fire moves the accumulation towards an associated asymptote.

The Axon-Hillock model tested only included the average accumulation and asymptotic behavior of the neurons. This was done to avoid the complexity of a non-linear model, and to test the viability of the other two behavioral features. Before implementing the model, it was determined that the accumulation to threshold ratio of the default neuron would require at least four synaptic fires to cause a neuron to fire. Because application input spikes are typically sparse, this would make it impossible for the Axon-Hillock neuron model to complete even the most simple tasks such as an XOR. This was confirmed when the genetic algorithm was unable to find a viable network when the Axon-Hillock model was enabled.

While the default accumulation behavior made it impossible for the Axon-Hillock neuron to complete the simple XOR task, it is possible to increase the amount accumulated by adding gain in the hardware. Adding gain to the Axon-Hillock neuron not only increases the amount accumulated, but also changes the asymptotes to which a particular synaptic weight will try to accumulate to. The simulator was thus modified to support an adjustable gain parameter for the Axon-Hillock model. With a gain of two, the model was able to generate networks for the XOR application and the polebalancing application.

The results for the networks generated by the genetic algorithm are displayed in Table 3.2. For the polebalancing application networks are trained to balance a pole continuously for five minutes. The runs complete metric specifies how many runs of the genetic algorithm completed. The average epochs to completion specifies how many generations on average it took before a run completed, not counting runs that failed to complete. From the table it is clear that while the Axon-Hillock neuron was not able to generate as many networks, it was still a feasible model. One particular point of interest is the difference in the size of the networks generated. Networks with Axon-Hillock models were, on average, much larger than networks without. It would be possible to use the simulator to calculate the average

energy of networks using either model and their different energy requirements to determine if there would be a significant difference in energy consumption.

Table 3.2: Comparison of networks generated from DANNA, the standard memristive model, and the memristive Axon-Hillock model

Architecture	Percent Runs Completed	Average Epochs to Complete	Number of Neurons in Complete Networks	Number of Synapses in Complete Networks	Average Fitness of All Networks
DANNA	83.5%	42.5	18.8	55	14095
mrDANNA 21 Levels with LTPD	35.3%	58.6	22	37.3	8604.5
mrDANNA Axon-Hillock	22.7%	43.3	30.9	59.2	4404.3

3.2.3 Programming Resolution

While memristors can theoretically take on any memristance value between their high resistance state and low resistance state, it is difficult to program to an arbitrary memristance state on-chip. As the programming granularity increases, so does the overhead of the circuitry to support that programming. Thus, it was questioned whether reducing the programming granularity would impact the performance of the networks that could be generated with evolutionary optimization.

In order to describe this limited programming, the term programming resolution was used. The number of integer weights a synapse can be programmed to is said to be its programming resolution. A programming resolution of 21 means a synapse can be programmed to 21 distinct integer weights. These weights are typically symmetric. A programming resolution of 3 symmetric weights would mean a synapse can be programmed to either 1, 0, or -1. In both cases note that the maximum and minimum weights correspond to the same effective conductance, but networks with a higher programming resolution will be able to achieve more combinations of weights thusly increasing the potential solution space.

For the initial chip design, it was desirable to reduce the overhead as much as possible. In this way, the memristors would be limited to only being programmed to either the high resistance state or low resistance state. This extremely limited programming corresponds to a programming resolution of 3. It was unclear if networks with this limited resolution would be able to perform complex applications. To understand the impact of this resolution, networks were generated with this resolution for both the iris classification and polebalancing tasks. The results are displayed in Tables 3.3 and 3.4.

Table 3.3: Performance of networks with programming resolution of 3 for iris classification application

Architecture	Best Fitness	Average Epochs to Complete	Number of Neurons in Complete Networks	Number of Synapses in Complete Networks	Average Fitness of All Networks
DANNA	97.3%	261.2	13.8	7.3	86.9%
mrDANNA 21 Levels	97.3%	223.6	11.6	23.1	90.7%
mrDANNA 3 Levels	98.7%	160.4	12	24.8	96.4%

From the tables it is clear that programming with a limited resolution is feasible, as networks are still able to perform successfully in both the polebalancing and iris classification tasks. For the polebalancing task, networks take longer to train, and are less likely to successfully balance the pole for five minutes, but some networks can still be generated, and these networks are of comparable size to networks with a larger resolution. For the iris classification application, the genetic algorithm actually produces better networks with the limited resolution. Not only was the best network better than the other architectures considered, but the average network was better as well. This might be due to the simplicity of the iris classification task, however. With the reduced solution space, the genetic algorithm might be able to find a solution more quickly, and optimize the synaptic weights more easily.

3.2.4 Online Learning

Online learning is an unsupervised learning technique to improve the performance of a network dynamically. Unlike training which is conducted using a genetic algorithm, online learning occurs as the network is in use. There have been many proposed systems for implementing online learning. One of the main ones, and the one implemented in the memristive model, is potentiation and depression of synapses. In this way, synaptic weights are either increased or decreased based on the function of that synapse.

There are many motivations for including online learning in a neuromorphic model. Online learning has been shown to be a suitable mechanism for training in some problems, meaning that a neuromorphic system with these feature could be trained without supervision [11]. Additionally, online learning has been shown to combat process variation, which is important in a memristive system due to their variability [4]. Finally, online learning can be implemented in a way that is separate from the programming of the memristors such that it can achieve memristance states online that a network cannot be programmed to. Because online learning only adjusts a synaptic weight by a small amount, this can be implemented in hardware with little overhead. Because the initial chip design was considering an extremely limited programming resolution, this is particularly important as online learning can provide additional resolution online thusly increasing the potential solution space.

The first online learning mechanisms considered were a simple long term potentiation and depression model. Synaptic weights that occurred the cycle before a neuron fired, and thus contributed to the neuron firing, were potentiated, while synaptic weights that occurred in a neuron's refractory period were depressed [13]. These dynamic updates of the weights would permanently affect that synapse until the network was reset. The update in a synaptic weight is determined by the switching characteristics of the memristor, and the length of the pulse to cause the switch. These values can be set as parameters within the simulator.

To study the impact of this simple potentiation and depression scheme, networks were generated for the iris classification task. Networks were then tested both with and without online learning enabled, and their accuracy was compared. The results are displayed in Fig. 3.3. These results make it clear that these online learning mechanisms are able to

dynamically improve the performance of networks, as there is a significant drop in accuracy when they are disabled.

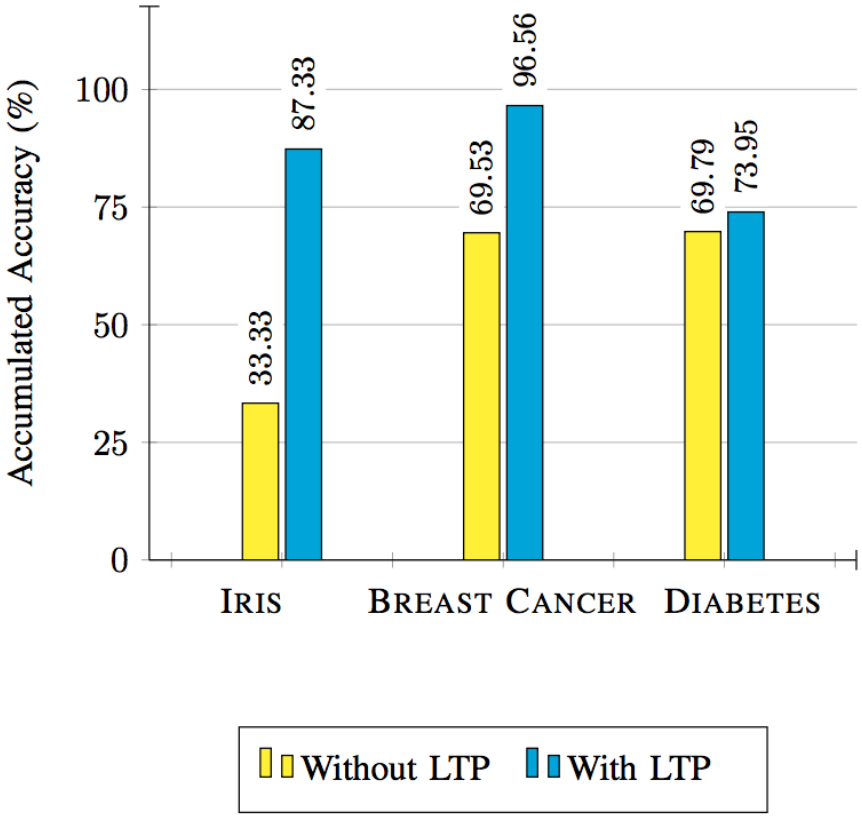


Fig. 11. Average accumulated accuracy for classification task with/without learning.

Figure 3.3: Performance of networks with and without LTPD for the classification of different datasets

After seeing the improvement from simple potentiation and depression mechanisms, a more biologically accurate mechanism was explored in spike-timing-dependent plasticity [29]. Unlike the simple mechanism described before, spike-timing-dependent plasticity considers synapses that contribute to a neuron fire in larger time windows. Not only is the window of time larger, but the magnitude of the potentiation and depression are affected based on where a synaptic fire occurs within that window. Synapses that occur immediately before and after a neuron fires are potentiated and depressed the most, while those at the edge of the window are adjusted the least. The difference in magnitude in both of these cases is exponential with respect to the temporal distance in that window.

The XOR application was the first considered to understand the impact of spike-timing-dependent plasticity on network performance. A network was generated with spike-timing-dependent plasticity enabled, and then tested in two cases. In the first case spike-timing-dependent plasticity was disabled, and the synaptic weights were restricted to their original programmed values. In the second case, spike-timing-dependent plasticity was enabled, and the synaptic weights were potentiated and depressed based on their contributions to a neuron's fire within a three cycle window before and after the fire. The accuracy of the network over several runs in each test case is displayed in Fig. 3.4.

From Fig. 3.4 it is clear that the inclusion of spike-timing-dependent plasticity in the network is vital to the network's success. Without spike-timing-dependent plasticity the network can only achieve approximately 60% accuracy. However, the network with spike-timing-dependent plasticity, though it begins around the same accuracy, eventually converges to a network that perfectly does the XOR operation. The improvement provided by spike-timing-dependent plasticity is different than the improvement from evolutionary optimization as spike-timing-dependent plasticity is unsupervised.

These improvements lend credence to the idea that not only can spike-timing-dependent plasticity improve performance dynamically and unsupervised, but also suggest that spike-timing-dependent plasticity might be able to be used as a mechanism to expand the solution space. Even though the network could only be programmed to a limited number of integer values, spike-timing-dependent plasticity enabled the network to achieve new weights it might not have been able to before. Thus, spike-timing-dependent plasticity can increase the granularity of synaptic weights in systems with limited resolution.

After confirming that spike-timing-dependent plasticity was able to dynamically optimize a network for the XOR application, it was necessary to investigate the performance on other applications. While XOR can be a good application to gain insight on the viability of a design, it is still much too simple to quantify the benefits or detriments of that design. To expand the results, the impact of spike-timing-dependent plasticity was measured on the iris classification application. Iris classification allows us to more easily quantify the impact of spike-timing-dependent plasticity through the classification accuracy. For the XOR application, a network is considered a failure if it is unable to reliably perform the

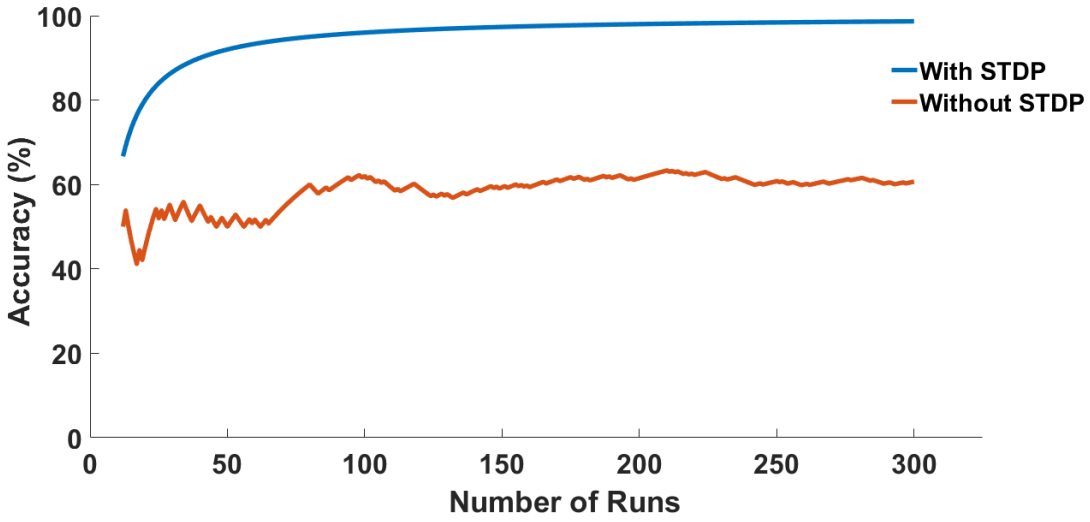


Figure 3.4: Accuracy over time for same XOR network with and without STDP enabled

operation. The iris classification task also allows us to measure whether spike-timing-dependent plasticity provides any generalization, improved performance on data the network was not trained on.

In order to study the impact of spike-timing-dependent plasticity on the memristive model, two sets of networks were trained. The first set of networks was trained without spike-timing-dependent plasticity, and the second set was trained with spike-timing-dependent plasticity. These sets were further divided by the resolution for which the synaptic weights were allowed to be programmed to. Networks were trained with programming resolutions of 3, 7, and 21.

After the networks were generated, their accuracy was evaluated on a testing set that is separate from the training set. Networks that were trained without spike-timing-dependent plasticity were tested without STDP as well. Networks that were trained with STDP, however, were tested both with spike-timing-dependent plasticity on and off. In doing so the effect of STDP on these specific networks can be observed. It is possible that evolutionary optimization was able to find networks in spite of spike-timing-dependent plasticity, and that spike-timing-dependent plasticity did nothing to improve the performance of these networks. Testing the network with spike-timing-dependent plasticity both on and off reveals whether or not spike-timing-dependent plasticity actually does anything within these networks.

The results from the iris classification are displayed in Fig. 3.5 and Fig. 3.6. From Fig. 3.5 it is clear that spike-timing-dependent plasticity is important to the success of networks that are trained with it. When spike-timing-dependent plasticity is turned off during testing, the accuracy drops on average from 94% to 79%. Thus, like in the XOR task, spike-timing-dependent plasticity is able to take networks that are initially programmed to suboptimal values, and improve the network fitness by adjusting the synaptic weights online. Additionally, the inclusion of spike-timing-dependent plasticity in the model improves the performance on average. Fig. 3.6 shows that, while small, STDP improves the performance on the testing set for networks with 3, 7, and 21 programmable levels. Additionally, the improvement from the inclusion of spike-timing-dependent plasticity in the model is increased as the resolution decreases. This is likely a result of the ability of spike-timing-dependent plasticity to increase granularity more when the resolution is limited most.

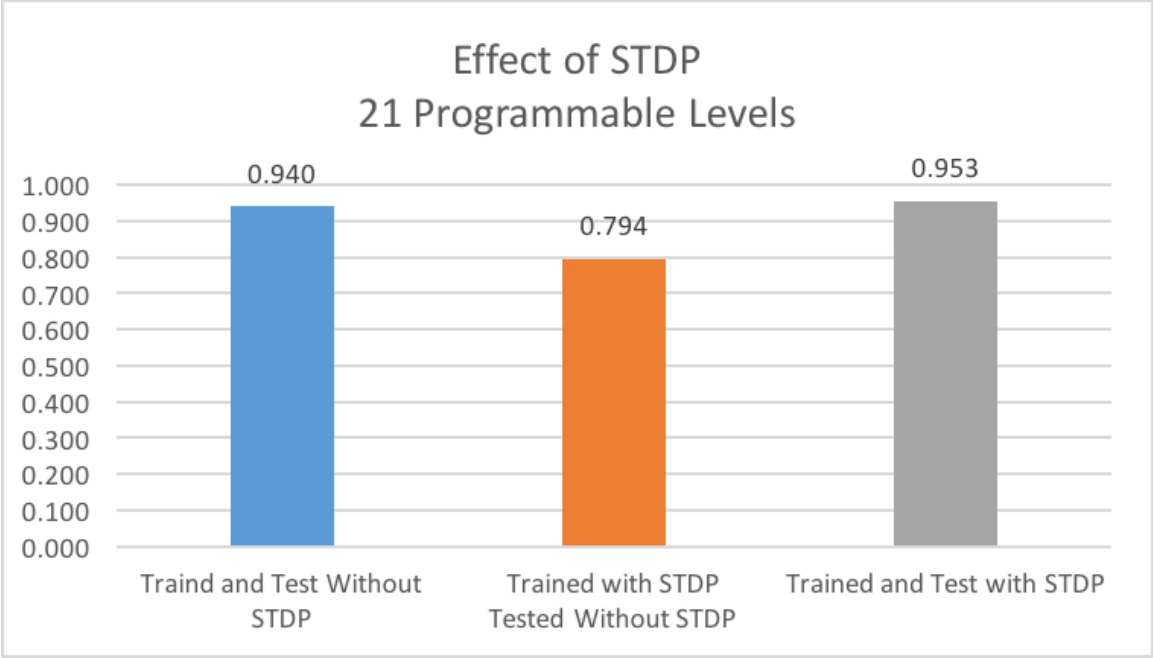


Figure 3.5: Impact of STDP in networks for the iris classification task

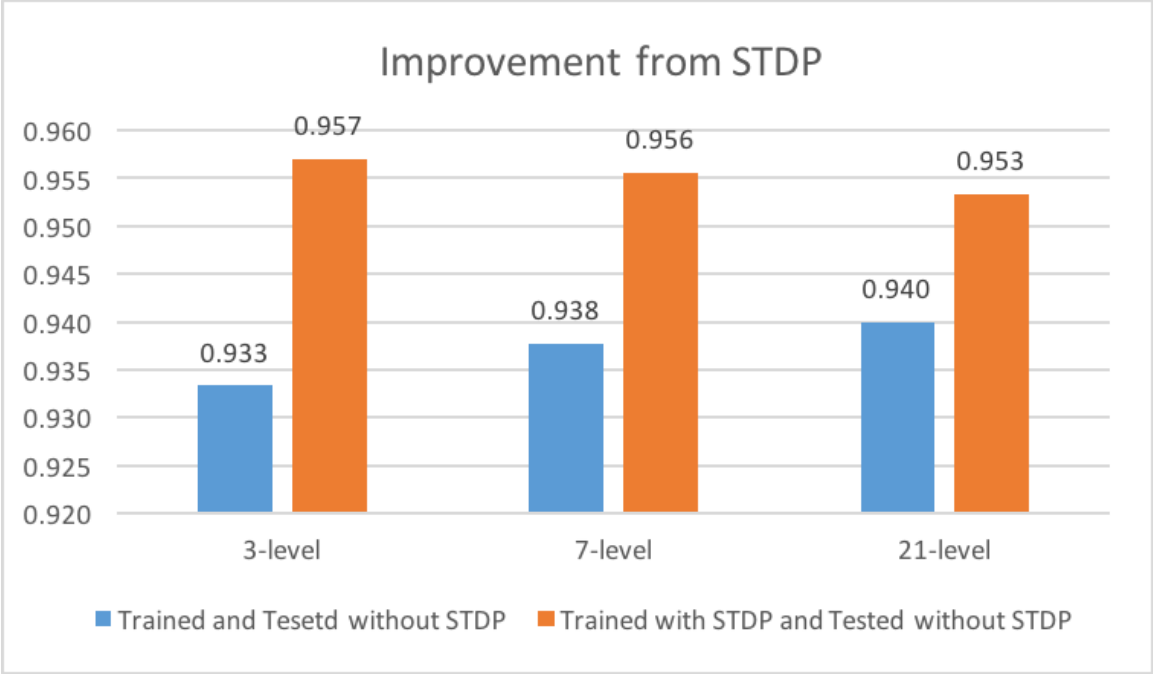


Figure 3.6: Improvement in performance of networks trained with STDP over those without

Table 3.4: Performance of networks with programming resolution of 3 for polebalancing application

Architecture	Percent Runs Completed	Average Epochs to Complete	Number of Neurons in Complete Networks	Number of Synapses in Complete Networks	Average Fitness of All Networks
DANNA	83.5%	42.5	18.8	55	14095
mrDANNA 21 Levels with LTPD	35.3%	58.6	22	37.3	8604.5
mrDANNA 3 Levels	27.3%	66.9	23.4	42.3	5319.5

Chapter 4

Conclusions and Future Work

In this work a high level C++ simulator is presented to efficiently model memristive neuromorphic circuits. The simulator is built into a software framework that allows for the generation of networks using evolutionary optimization, and achieves faster simulation by sacrificing granularity for speed. The simulator is verified using a simple handcrafted network, and a network to perform the XOR operation. Additionally, three genetic operators were implemented to facilitate evolutionary optimization: random, mutate, and crossover. These operators are facilitated by a planar embedding.

With the capabilities of the simulator to understand the impact circuit design parameters on the application level performance of the network, various design decisions were examined. The impact of memristor process variation was examined, and determined to be negligible for simple applications. An alternative neuron model, the Axon-Hillock model, was simplified and modeled. The circuit design was updated to produce viable networks at an application level based on results obtained from the simulator. A limited programming resolution was added to the model, and it was determined that the resolution did not negatively impact the networks generated. Finally, online learning mechanisms were evaluated, and it was shown that not only can online learning improve the performance of networks dynamically, but it can also increase generalization in networks for classification tasks.

While this work focuses on the model piece of the software framework, there are many questions about the application and evolutionary optimization pieces that have yet to be answered. First, it is unclear exactly how the way an application defines

inputs and outputs affects the performance of different models. For example, would the Axon-Hillock model benefit from an I/O scheme that includes more spikes into input neurons. Additionally, further work can be done to understand whether alternative forms of evolutionary optimization such as speciation can improve the networks generated.

Additional work can also be done to more thoroughly study online learning and spike-timing-dependent plasticity. In this work a three cycle window was considered. However, it is not clear how the impact of spike-timing-dependent plasticity might change as that window is expanded. Further work can also be done to understand the impact of spike-timing-dependent plasticity in other models, as well as for other applications.

Bibliography

[1] Amer, S., Sayyaparaju, S., Rose, G. S., Beckmann, K., and Cady, N. C. (2017). A practical hafnium-oxide memristor model suitable for circuit design and simulation. In *ISCAS: International Symposium on Circuits and Systems*, Baltimore, MD. 11

[2] Amir, A., Datta, P., Risk, W., Cassidy, A., A. Kusnitz, J., Esser, S., Andreopoulos, A., M. Wong, T., Flickner, M., Alvarez-Icaza, R., McQuinn, E., Shaw, B., Pass, N., and S. Modha, D. (2013). Cognitive computing programming paradigm: A corelet language for composing networks of neurosynaptic cores. 3

[3] Andreev, K. and Räcke, H. (2004). Balanced graph partitioning. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 120–124, New York, NY, USA. ACM. 18

[4] Cameron, K., Boonsobhak, V., Murray, A., and Renshaw, D. (2005). Spike timing dependent plasticity (stdp) can ameliorate process variations in neuromorphic vlsi. *IEEE Transactions on Neural Networks*, 16(6):1626–1637. 36

[5] Chakma, G., Dean, M. E., Rose, G. S., Beckmann, K., Manem, H., and Cady, N. (2016). A hafnium-oxide memristive dynamic adaptive neural network array. In *International Workshop on Post-Moore's Era Supercomputing (PMES)*, Salt Lake City, UT. 10

[6] Chua, L. (1971). Memristor-the missing circuit element. *IEEE Transactions on circuit theory*, 18(5):507–519. 2

[7] Davison, A., Brderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., and Yger, P. (2009). Pynn: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2:11. 4

[8] Disney, A., Reynolds, J., Schuman, C. D., Klibisz, A., Young, A., and Plank, J. S. (2016). DANNA: A neuromorphic software ecosystem. *Biologically Inspired Cognitive Architectures*, 9:49–56. 6

[9] ERDdS, P. and R&WI, A. (1959). On random graphs i. *Publ. Math. Debrecen*, 6:290–297. 18

[10] Esser, S. K., Merolla, P. A., Arthur, J. V., Cassidy, A. S., Appuswamy, R., Andreopoulos, A., Berg, D. J., McKinstry, J. L., Melano, T., Barch, D. R., di Nolfo, C., Datta, P., Amir, A., Taba, B., Flickner, M. D., and Modha, D. S. (2016). Convolutional networks for fast, energy-efficient neuromorphic computing. *CoRR*, abs/1603.08270. 3

[11] Florian, R. V. (2007). Reinforcement learning through modulation of spike-timing-dependent synaptic plasticity. *Neural Computation*, 19(6):1468–1502. 36

[12] Gilbert, E. N. (1959). Random graphs. *Ann. Math. Statist.*, 30(4):1141–1144. 18

[13] Hebb, D. O. (1949). *The organization of behavior: A neuropsychological approach*. John Wiley & Sons. 36

[14] Indiveri, G. (2015). *Neuromorphic Engineering*, pages 715–725. Springer Berlin Heidelberg, Berlin, Heidelberg. 1

[15] Jin, X., Galluppi, F., Patterson, C., Rast, A., Davies, S., Temple, S., and Furber, S. (2010). Algorithm and software for simulation of spiking neural networks on the multi-chip spinnaker system. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. 4

[16] Khan, M. M., Lester, D. R., Plana, L. A., Rast, A., Jin, X., Painkras, E., and Furber, S. B. (2008). Spinnaker: Mapping neural networks onto a massively-parallel chip multiprocessor. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 2849–2856. 4

[17] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc. 2

[18] Lavagno, L., Martin, G., and Scheffer, L. (2006). *Electronic Design Automation for Integrated Circuits Handbook - 2 Volume Set*. CRC Press, Inc., Boca Raton, FL, USA. 14

[19] Lichman, M. (2013). UCI machine learning repository. 32

[20] Minsky, M. L. and Papert, S. A. (1988). *Perceptrons: Expanded Edition*. MIT Press, Cambridge, MA, USA. 2

[21] Niu, D., Chen, Y., Xu, C., and Xie, Y. (2010). Impact of process variations on emerging memristor. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 877–882, New York, NY, USA. ACM. 31

[22] Plank, J. S., Rose, G. S., Dean, M. E., and Schuman, C. D. (2017). A CAD system for exploring neuromorphic computing with emerging technologies. In *42nd Annual GOMACTech Conference*, Reno, NV. 5

[23] Sawada, J., Akopyan, F., Cassidy, A. S., Taba, B., Debole, M. V., Datta, P., Alvarez-Icaza, R., Amir, A., Arthur, J. V., Andreopoulos, A., Appuswamy, R., Baier, H., Barch, D., Berg, D. J., Nolfo, C. d., Esser, S. K., Flickner, M., Horvath, T. A., Jackson, B. L., Kusnitz, J., Lekuch, S., Mastro, M., Melano, T., Merolla, P. A., Millman, S. E., Nayak, T. K., Pass, N., Penner, H. E., Risk, W. P., Schleupen, K., Shaw, B., Wu, H., Giera, B., Moody, A. T., Mundhenk, N., Van Essen, B. C., Wang, E. X., Widemann, D. P., Wu, Q., Murphy, W. E., Infantolino, J. K., Ross, J. A., Shires, D. R., Vindiola, M. M., Namburu, R., and Modha, D. S. (2016). Truenorth ecosystem for brain-inspired computing: Scalable systems, software, and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 12:1–12:12, Piscataway, NJ, USA. IEEE Press. 4

[24] Sayyaparaju, S., Chakma, G., Amer, S., and Rose, G. S. (2017). Circuit techniques for online learning of memristive synapses in CMOS-memristor neuromorphic systems. In *27th ACM Great Lakes Symposium on VLSI*, pages 479–482, Banff, Alberta, Canada. ACM. 11

[25] Schuman, C. D., Birdwell, J. D., and Dean, M. (2014). Neuroscience-inspired inspired dynamic architectures. In *Proceedings of the 2014 Biomedical Sciences and Engineering Conference*, pages 1–4. 4

[26] Schuman, C. D., Disney, A., and Reynolds, J. (2015). Dynamic adaptive neural network arrays: A neuromorphic architecture. In *Workshop on Machine Learning in HPC Environments, Supercomputing*, Austin, TX. 4

[27] Schuman, C. D., Disney, A., Singh, S. P., Bruer, G., Mitchell, J. P., Klibisz, A., and Plank, J. S. (2016a). Parallel evolutionary optimization for neuromorphic network training. In *Machine Learning in HPC Environments, Supercomputing 2016*, Salt Lake City. 19

[28] Schuman, C. D., Plank, J. S., Disney, A., and Reynolds, J. (2016b). An evolutionary optimization framework for neural networks and neuromorphic architectures. In *International Joint Conference on Neural Networks*, Vancouver. 17

[29] Song, S., Miller, K. D., and Abbott, L. F. (2000). Competitive hebbian learning through spike-timing-dependent synaptic plasticity. *Nature neuroscience*, 3(9):919–926. 37

[30] Stanley, K. O. and Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evol. Comput.*, 10(2):99–127. 18

[31] Tang, Y., Nyengaard, J. R., De Groot, D. M., and Gundersen, H. J. G. (2001). Total regional and global number of synapses in the human brain neocortex. *Synapse*, 41(3):258–273. 1

[32] Uddin, M., Majumder, M. B., Rose, G. S., Beckmann, K., Manem, H., Alamgir, Z., and Cady, N. C. (2016). Techniques for improved reliability in memristive crossbar puf circuits. In *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, pages 212–217. IEEE. 2

[33] Wagner, R. A. and Fischer, M. J. (1974). The string-to-string correction problem. *J. ACM*, 21(1):168–173. 29

[34] Weiss, R., Chakma, G., and Rose, G. S. (2017). A synchronized axon hillock neuron for memristive neuromorphic systems. In *60th IEEE International Midwest Symposium on Circuits and Systems*, Boston, MA. 33

Appendices

A Sample Code

This section contains samples of code relevant to the functional behavior of the network. The neuron code presented is responsible for accumulating and firing. The synapse code is an implementation of online learning modeled from the underlying memristors. The delay code is responsible for managing the cyclic array. The random code is the complete code to generate a random network for evolutionary optimization. The mutation code highlights the move neuron mutation. The crossover code performs the entire crossover operation.

A.1 Neuron

```
void mrNeuron::apply(float w, int clockCycle, element* sender)
{
  double THRESHOLDLIMIT = NeuroUtils::ParamsGetDouble("mrdanna","
     THRESHOLDLIMIT",true);
  string PRINTEVENTS = NeuroUtils::ParamsGetString("mrdanna","
     PRINTEVENTS",true);
  if(PRINTEVENTS == "TRUE"){
    string e;
    for(NeuroIOMap::iterator it = inElements.begin(); it !=
       inElements.end(); ++it){
      if(it->second == sender){
        if(it->first[0] == -1.0){
          e = "Time: " + to_string(clockCycle) + " Type:
             Pulse_Neuron Location: " + printCoords();
          break;
        }
        else{
          e =  "Time: " + to_string(clockCycle) + " Type:
             Synapse_Fire Synapse Location: " + printCoords(it->
             first) + " --> " + printCoords();
```

```
            break;
          }
        }
      }
      cout << e <<endl;
      /*
      events.push_back(e);
      eventTimes.push_back(clockCycle);
      */
    }
    if(storedCycle != clockCycle){
      update(clockCycle);
    }
    if(inRefracPeriod){
      depressSynapse(sender,(clockCycle-cycleLastFired + 1));
      return;
    }
    if(resetCycle){
      depressSynapse(sender,(clockCycle-cycleLastFired + 1));
      return;
    }
    if(signals[signals.size()-1].empty()){
      accumInput++;
    }
    signals[signals.size()-1].push_back(sender);
    accumulator += w;
    //Set a lower limit for negative - wont exist for positive
    if(accumulator < (-1.*THRESHOLDLIMIT)){
      accumulator = (-1.0*THRESHOLDLIMIT);
    }
```

```
  if(accumulator < threshold){
    fireNextCycle = false;
  }
  else if(accumulator >= threshold){
    fireNextCycle = true;
  }

}

/* Making a functional change to match the hardware
 * after a neuron passes its threshold it enters a reset cycle
 * fires cycle after that
 * also all delays are doubled as a result(handling that in delay)
 */
void mrNeuron::update(int clockCycle)
{
  //collectAccums.push_back(accumulator);

  string PRINTEVENTS = NeuroUtils::ParamsGetString("mrdanna","
     PRINTEVENTS",true);
  if(storedCycle == clockCycle){
    return;
  }
  totalCycles++;
  storedCycle = clockCycle;
  if(fireNextCycle){
    resetCycle = true;
    fireNextCycle = false;
  }
  else if(resetCycle){
```

```
  for(NeuroIOMap::iterator it = outElements.begin(); it !=
      outElements.end(); ++it){
    it->second->apply(1.0,clockCycle,this);
  }
  if(PRINTEVENTS == "TRUE"){
    string e;
    e = "Time: " + to_string(clockCycle) + " Type: Neuron_Fire
        Neuron Location: "+ printCoords();
    cout<< e <<endl;
  //  events.push_back(e);
  //  eventTimes.push_back(clockCycle);
  }
  inRefracPeriod = true;
  fireNextCycle = false;
  cycleLastFired = clockCycle;
  accumulator = 0.0;
  for(unsigned int i = 0; i < signals.size(); i++){
    for(unsigned int j = 0; j < signals[i].size(); j++){
      potentiateSynapse(signals[i][j],signals.size()-i);
    }
  }

  for(unsigned int i = 0; i < signals.size() - 1; i++){
    signals[i] = signals[i + 1];
  }
  signals[signals.size() - 1].clear();
  fired++;
  //collectFireds.push_back(clockCycle);
  resetCycle = false;
}
```

```
  else{
    for(unsigned int i = 0; i < signals.size() - 1; i++){
      signals[i] = signals[i + 1];
    }
    totalAccum += accumulator;
    if(inRefracPeriod){
      if(cycleLastFired + refractoryPeriod == clockCycle){
        inRefracPeriod = false;
      }
    }
  }
}
```

A.2 Synapse

```
void mrSynapse::potentiateSTDP(int val)
{
  double LEARNINGPARAM = NeuroUtils::ParamsGetDouble("mrdanna","
     LEARNINGPARAM",true);
  int USECYCVARIATION = NeuroUtils::ParamsGetInt("mrdanna","
     USEMEMVARIATION",true);
  if(USECYCVARIATION){
    double PDRDEV = NeuroUtils::ParamsGetDouble("mrdanna","PDRDEV
       ","true");
    double NDRDEV = NeuroUtils::ParamsGetDouble("mrdanna","NDRDEV
       ","true");
    default_random_engine generator;
    normal_distribution<double> PDRdis(pdelR,pdelR*(PDRDEV/100.0))
       ;
```

```
    normal_distribution<double> NDRdis(ndelR,ndelR*(NDRDEV/100.0))
        ;
    double pdelRVar = PDRdis(generator);
    double ndelRVar = NDRdis(generator);
    Rn += (ndelRVar*(LEARNINGPARAM-val + 1));
    Rp -= (pdelRVar*(LEARNINGPARAM-val + 1));
  }
  else{
    Rn += (ndelR*(LEARNINGPARAM-val + 1));
    Rp -= (pdelR*(LEARNINGPARAM-val + 1));
  }
//  Rn += (delR/val);
//  Rp -= (delR/val);
  Geff = (1.0/Rp) - (1.0/Rn);
  weight = Geff/normG;
  if(weight > max){
    weight = max;
    Geff = GeffMax;
    Rn = HRS;
    Rp = LRS;
  }
  eWeight = weight;
}

/*Depressin based on how the device will do it*/
void mrSynapse::depressSTDP(int val)
{
  double LEARNINGPARAM = NeuroUtils::ParamsGetDouble("mrdanna","
      LEARNINGPARAM",true);
```

```
    int USECYCVARIATION = NeuroUtils::ParamsGetInt("mrdanna","
        USEMEMVARIATION",true);
    if(USECYCVARIATION){
      double PDRDEV = NeuroUtils::ParamsGetDouble("mrdanna","PDRDEV
          ","true");
      double NDRDEV = NeuroUtils::ParamsGetDouble("mrdanna","NDRDEV
          ","true");
      default_random_engine generator;
      normal_distribution<double> PDRdis(pdelR,pdelR*(PDRDEV/100.0))
          ;
      normal_distribution<double> NDRdis(ndelR,ndelR*(NDRDEV/100.0))
          ;
      double pdelRVar = PDRdis(generator);
      double ndelRVar = NDRdis(generator);
      Rn -= (ndelRVar*(LEARNINGPARAM_val + 1));
      Rp += (pdelRVar*(LEARNINGPARAM_val + 1));
    }
    else{
      Rn -= (ndelR*(LEARNINGPARAM_val + 1));
      Rp += (pdelR*(LEARNINGPARAM_val + 1));
    }

//  Rn -= (delR/val);
//  Rp += (delR/val);
    Geff = (1.0/Rp) - (1.0/Rn);
    weight = Geff/normG;
    if(weight < (-1.0*max)){
      Geff = -1.0*GeffMax;
      Rn = LRS;
      Rp = HRS;
```

```
  }
  eWeight = weight;

}
```

A.3 Delay

```
//Apply - How other elements signal the delay block to add a new
   charge
void delay::apply(float w, int clockCycle, element* sender){
  chargesPassed++; //energy estimation

  //Put spike into cyclic array
  if(head != 0){
    charges[head - 1] = w;
  }
  else{
    charges[charges.size()-1] = w;
  }
}

//Update - How the network signals the delay block to advance the
   simulation by 1 cycle
void delay::update(int clockCycle){

  //Pass on spike if it has reached end of delay block
  if(charges[head] != 0.0){
    for(ElementMap::iterator it = outputElements.begin(); it !=
       outputElements.end(); ++it){
      it->second->apply(charges[head],clockCycle,this);
    }
```

```
  }
  charges[head] = 0.0; //reset value

  //manage cyclic array
  if(head != (charges.size() - 1)){
    head++;
  }
  else{
    head = 0;
  }

}
```

A.4 Random

```
 void mrNetwork::Random()
{
  inCount = 0;
  outCount = 0;
  clockcycle = 0;
  outputVector.resize(nOut);

  mrNeuron* n;
  mrSynapse* s;
  outputElement* o;
  //delay* d;
  vector<double> cIn;
  vector<double> cOut;
  cIn.resize(embeddedDimension);
  cOut.resize(embeddedDimension);
  for(int i = 0; i < embeddedDimension; i++){
```

```
    cIn[i] = -1.0;
    cOut[i] = -2.0;
  }
  inputComponents.resize(nIn);
  for(int i = 0; i < nIn; i++){

    n = new mrNeuron(maxDims,maxThreshold, "I",inCount);
    //n = new mrNeuron(maxDims,maxThreshold, "I",inCount,nIn);
    s = new mrSynapse(n->getThreshold() + 1,"S");
    delay *d = new delay(0,"D");
    delays.insert(pair<uint64_t,element*>((uint64_t)d,(element*)d)
        );
    d->addOutputElement((element*) s);
    s->addInputElement((element*) d);
    s->addOutputElement((element*)n);
    n->addInputElement((element*)s, cIn);
    inCount++;
    inputComponents[i] = (element*)d;
    mrNetworkGrid[n->coords] = n;
  }
  for(int i = 0; i < nOut; i++){
    n = new mrNeuron(maxDims,maxThreshold,"O",outCount);
    //n = new mrNeuron(maxDims,maxThreshold,"O",outCount,nOut);
    o = new outputElement(i,&outputVector);
    outCount++;
    n->addOutputElement((element*)o,cOut);
    mrNetworkGrid[n->coords] = n;
  }

  int hiddenSize = rand()%5 + 1;
```

```
  hiddenSize = max(hiddenSize,0);
  for(int i = 0; i < hiddenSize; i++){
    n = new mrNeuron(maxDims, maxThreshold, "N");
    int full = 0;
    while(mrNetworkGrid.find(n->coords) != mrNetworkGrid.end()){
      delete n;
      if(full == 100){
        n = NULL;
      }
      n = new mrNeuron(maxDims, maxThreshold, "N");
      full++;

    }
    //if(n != NULL){
      mrNetworkGrid[n->coords] = n;
    //}
  }
  //double connect;
  //int weight;
  int numSynapses = rand()%10 + 1;
  //Using mutation - might cause trouble
  for(int i = 0; i < numSynapses; i++){
    mutationAddSynapse();
  }

}
```

A.5 Mutate

```
void mrNetwork::mutationMoveNeuron()
{
```

```
int randVal = rand()%mrNetworkGrid.size();
NeuronGrid::iterator randNeuron = mrNetworkGrid.begin();
advance(randNeuron,randVal);
if(randNeuron->second->getType() =="O" || randNeuron->second->
    getType() == "I"){
  for(int i = 0; i < 100; i++){
    randVal = rand()%mrNetworkGrid.size();
    randNeuron = mrNetworkGrid.begin();
    advance(randNeuron,randVal);
    if(randNeuron->second->getType() == "N"){
      break;
    }
    else if(i == 99){
      return;
    }
  }
}
auto w = randNeuron->second->getOutgoingWeights();
auto edgeO = randNeuron->second->getOutgoing();
auto edgeI = randNeuron->second->getIncoming();
auto w2 = randNeuron->second->getIncomingWeights();
mrNeuron* n = new mrNeuron(randNeuron->second);
/*What needs to happen -
 * update shuffle to be grid coords(comment out)
 * check the result
 */
int full = 0;
n->shuffle(maxDims);
//Look at this loop closer if theres an issue
while(mrNetworkGrid.find(n->coords) != mrNetworkGrid.end()){
```

```
    full++;
    if(full == 100){
      deleteNeuron(randNeuron->second);
      delete n;
      return;
    }
    n->shuffle(maxDims);
  }
  deleteNeuron(randNeuron->second);
  mrNetworkGrid.insert(pair<vector<double>, mrNeuron*>( n->coords,
     n));

  /*
  for(int i = 0; i < coords.size(); i++){
    coords[i] = randNeuron->second->coords[i];
  }
  */

  NeuronGrid::iterator it;
  double dist;
  delay* d;
  mrSynapse* s;
  for(unsigned int i = 0; i < edgeO.size(); i++){
    it = mrNetworkGrid.find(edgeO[i]);
    if(it->first != edgeO[i]){
      cout<<"key-coord mismatch" <<endl;
      cout<<"going to: ";
      for(unsigned int j = 0; j < it->first.size(); j++){
        cout<<it->first[j] <<" ";
      }
```

```
      cout<<endl;
      cout<<"me: " <<n->printCoords() <<endl;
    }
    dist = n->distanceTo(it->second);
    d = new delay((int)round(dist),"D");
    delays.insert(pair<uint64_t,element*>((uint64_t)d,(element*)d)
        );
    s = new mrSynapse(w[i],"W");
    d->addOutputElement((element*) s);
    s->addInputElement((element*) d);
    s->addOutputElement((element*)it->second);
    n->addOutputElement((element*)d,edgeO[i]);
    it->second->addInputElement((element*)s,n->coords);
}
/*
for(NeuronGrid::iterator it2 = mrNetworkGrid.begin();it2 !=
    mrNetworkGrid.end(); ++it2){
    if(it2->second->isConnection(coords)){
        it2->second->remove(coords);

        dist = it2->second->distanceTo(randNeuron->second);
        d = new delay((int)round(dist),"D");
        delays.insert(pair<uint64_t,element*>((uint64_t)d,(element
            *)d));
        s = new mrSynapse(w2[i],"W");
        d->addOutputElement((element*) s);
        s->addInputElement((element*) d);
        randNeuron->second->addInputElement((element*)s,edgeI[i]);
        it->second->addOutputElement((element*)d,randNeuron->
            second->coords);
```

```
    }
  }
  */

  for(unsigned int i = 0; i < edgeI.size(); i++){
    it = mrNetworkGrid.find(edgeI[i]);
    if(it == mrNetworkGrid.end()){
      cout<<"cant find known input neuron" <<edgeI[i][0] << " " <<
          edgeI[i][1] <<endl;

    }
    else{

      dist = it->second->distanceTo(n);
      d = new delay((int)round(dist),"D");
      delays.insert(pair<uint64_t,element*>((uint64_t)d,(element*)
          d));
      s = new mrSynapse(w2[i],"W");
      d->addOutputElement((element*) s);
      s->addInputElement((element*) d);
      s->addOutputElement((element*)n);
      n->addInputElement((element*)s,edgeI[i]);
      it->second->addOutputElement((element*)d,n->coords);
    }
  }


}
```

A.6 Crossover

```
vector<mrNetwork*> mrNetwork::CrossoverRC(mrNetwork* partner)
{

  //cout<<"***DOING CROSSOVER***" <<endl;
  //Add check for network dimensions

  mrNetwork* child1 = new mrNetwork(inCount,outCount,maxDims,
     connectivity,weightMaxMag,maxThreshold,avgHidden);
  mrNetwork* child2 = new mrNetwork(inCount,outCount,maxDims,
     connectivity,weightMaxMag,maxThreshold,avgHidden);
  //mrNetwork* child2 = new mrNetwork(inCount,outCount,maxDims);
  //HERE - set the EO parameters for the children



  /* Quick pseudo
   * Add to neurogrids of new nets
   * add fixed structs and neurons from p1 and p2
   * get list of connections for each neuron
   * resolve all of the connections
   * do again for opposite sides of cut
   */

  map< vector<double>,vector< vector <double> > > edgeContainer1;
  map< vector<double>, vector< vector <double> > > edgeContainer2;
  map< vector<double> ,vector<int> > weightContainer1;
  map< vector<double>, vector<int> > weightContainer2;

  //If 0 we cut vertical 1 horizontal
  int RowOrCol = rand()%2;
```

```
double cut;
if(RowOrCol){
  cut = (rand() / (RAND_MAX / maxDims[0]));
}
else{
  cut = (rand() / (RAND_MAX / maxDims[1]));
}

for(NeuronGrid::iterator it = mrNetworkGrid.begin(); it!=
    mrNetworkGrid.end(); ++it){
  if(it->second->getType() == "O"){
    child1->mrNetworkGrid[it->second->coords] = new mrNeuron(it
       ->second);
    int pos = it->second->getOutputPos();
    edgeContainer1[it->second->coords] = it->second->getOutgoing
       ();
    weightContainer1[it->second->coords] = it->second->
       getOutgoingWeights();
    //vector<bool> *oV =outputVector;
    outputElement* oe = new outputElement(pos,&child1->
       outputVector);
    vector<double> outC;
    outC.resize(embeddedDimension);
    for(int i = 0; i < embeddedDimension; i++){
      outC[i] = -2.0;
    }
    child1->mrNetworkGrid[it->second->coords]->addOutputElement
       ((element*)oe,outC);
    child1->mrNetworkGrid[it->second->coords]->IOid = pos;
  }
```

```
else if(it->second->getType() == "I"){

  child1->mrNetworkGrid[it->second->coords] = new mrNeuron(it
     ->second);
  edgeContainer1[it->second->coords] = it->second->getOutgoing
     ();
  weightContainer1[it->second->coords] = it->second->
     getOutgoingWeights();
  int pos = it->second->getInputPos();
  child1->mrNetworkGrid[it->second->coords]->IOid = pos;
  mrSynapse* is = new mrSynapse(it->second->getThreshold() +
     1,"S");
  delay* idy = new delay(0,"D");
  delays.insert(pair<uint64_t,element*>((uint64_t)idy,(element
     *)idy));
  idy->addOutputElement((element*)is);
  is->addInputElement((element*)idy);
  is->addOutputElement((element*)child1->mrNetworkGrid[it->
     second->coords]);
  child1->inputComponents[pos] = (element*)idy;
  vector<double> inC;
  inC.resize(embeddedDimension);
  for(int i = 0; i < embeddedDimension; i++){
    inC[i] = -1.0;
  }
  child1->mrNetworkGrid[it->second->coords]->addInputElement((
     element*)is,inC);

}
else if(it->second->coords[RowOrCol] < cut){
```

```
      child1->mrNetworkGrid[it->second->coords] = new mrNeuron(it
         ->second);
      edgeContainer1[it->second->coords] = it->second->getOutgoing
         ();
      weightContainer1[it->second->coords] = it->second->
         getOutgoingWeights();
    }
    else{
      child2->mrNetworkGrid[it->second->coords] = new mrNeuron(it
         ->second);
      edgeContainer2[it->second->coords] = it->second->getOutgoing
         ();
      weightContainer2[it->second->coords] = it->second->
         getOutgoingWeights();
    }
  }

  for(NeuronGrid::iterator it = partner->mrNetworkGrid.begin(); it
     != partner->mrNetworkGrid.end(); ++it){
    if(it->second->getType() == "O"){
      child2->mrNetworkGrid[it->second->coords] = new mrNeuron(it
         ->second);
      edgeContainer2[it->second->coords] = it->second->getOutgoing
         ();
      weightContainer2[it->second->coords] = it->second->
         getOutgoingWeights();
      int pos = it->second->getOutputPos();
      outputElement* oe = new outputElement(pos,&child2->
         outputVector);
      vector<double> outC;
```

```
    outC.resize(embeddedDimension);
    for(int i = 0; i < embeddedDimension; i++){
      outC[i] = -2.0;
    }
    child2->mrNetworkGrid[it->second->coords]->addOutputElement
        ((element*)oe,outC);
    child2->mrNetworkGrid[it->second->coords]->IOid = pos;
  }
  else if(it->second->getType() == "I"){
    child2->mrNetworkGrid[it->second->coords] = new mrNeuron(it
        ->second);
    edgeContainer2[it->second->coords] = it->second->getOutgoing
        ();
    weightContainer2[it->second->coords] = it->second->
        getOutgoingWeights();
    int pos = it->second->getInputPos();
    child2->mrNetworkGrid[it->second->coords]->IOid = pos;
    mrSynapse* is = new mrSynapse(it->second->getThreshold() +
        1,"S");
    delay* idy = new delay(0,"D");
    delays.insert(pair<uint64_t,element*>((uint64_t)idy,(element
        *)idy));
    idy->addOutputElement((element*)is);
    is->addInputElement((element*)idy);
    is->addOutputElement((element*)child2->mrNetworkGrid[it->
        second->coords]);
    child2->inputComponents[pos] = (element*)idy;
    vector<double> inC;
    inC.resize(embeddedDimension);
    for(int i = 0; i < embeddedDimension; i++){
```

```
        inC[i] = -1.0;
      }
      child2->mrNetworkGrid[it->second->coords]->addInputElement((
          element*)is,inC);

    }
    else if(it->second->coords[RowOrCol] > cut){
      child1->mrNetworkGrid[it->second->coords] = new mrNeuron(it
          ->second);
      edgeContainer1[it->second->coords] = it->second->getOutgoing
          ();
      weightContainer1[it->second->coords] = it->second->
          getOutgoingWeights();
    }
    else{
      child2->mrNetworkGrid[it->second->coords] = new mrNeuron(it
          ->second);
      edgeContainer2[it->second->coords] = it->second->getOutgoing
          ();
      weightContainer2[it->second->coords] = it->second->
          getOutgoingWeights();
    }
}

mrSynapse* s;
delay* d;

for(NeuronGrid::iterator it = child1->mrNetworkGrid.begin(); it
    != child1->mrNetworkGrid.end(); ++it){
```

```
for(unsigned int i = 0; i < edgeContainer1[it->first].size();
    i++){
  if(child1->mrNetworkGrid.find(edgeContainer1[it->first][i])
     == child1->mrNetworkGrid.end()){
    mrNeuron* closest;
    double dist = (maxDims[0]*maxDims[0]) + (maxDims[1]*
       maxDims[1]);
    for(NeuronGrid::iterator it2 = child1->mrNetworkGrid.begin
       (); it2 != child1->mrNetworkGrid.end(); ++it2){
      if(it != it2){
        if(dist > it->second->distanceTo(it2->second)){
          closest = it2->second;
          dist = it->second->distanceTo(it2->second);
        }
      }
    }
    if(!it->second->isConnection(closest->coords)){
      d = new delay((int)round(dist),"D");
      delays.insert(pair<uint64_t,element*>((uint64_t)d,(
         element*)d));
      s = new mrSynapse(weightContainer1[it->first][i],"W");
      d->addOutputElement((element*)s);
      d->addInputElement((element*)it->second);
      s->addInputElement((element*)d);
      s->addOutputElement((element*)closest);
      child1->delays.insert(pair<uint64_t,element*>((uint64_t)
         d,(element*)d));
      it->second->addOutputElement((element*)d,closest->coords
         );
```

```
            closest->addInputElement((element*)s,it->second->coords)
                ;
          }
        }
        else{
          NeuronGrid::iterator match = child1->mrNetworkGrid.find(
              edgeContainer1[it->first][i]);
          if(!it->second->isConnection(match->second->coords)){
            double dist = it->second->distanceTo(match->second);
            d = new delay((int)round(dist),"D");
            delays.insert(pair<uint64_t,element*>((uint64_t)d,(
                element*)d));
            s = new mrSynapse(weightContainer1[it->first][i],"W");
            d->addOutputElement((element*)s);
            d->addInputElement((element*)it->second);
            s->addInputElement((element*)d);
            s->addOutputElement((element*)match->second);
            child1->delays.insert(pair<uint64_t,element*>((uint64_t)
                d,(element*)d));
            it->second->addOutputElement((element*)d,match->second->
                coords);
            match->second->addInputElement((element*)s,it->second->
                coords);
          }
        }
      }
    }

for(NeuronGrid::iterator it = child2->mrNetworkGrid.begin(); it
    != child2->mrNetworkGrid.end(); ++it){
```

```
for(unsigned int i = 0; i < edgeContainer2[it->first].size();
    i++){
  if(child2->mrNetworkGrid.find(edgeContainer2[it->first][i])
     == child2->mrNetworkGrid.end()){
    mrNeuron* closest;
    double dist = (maxDims[0]*maxDims[0]) + (maxDims[1]*
       maxDims[1]);
    for(NeuronGrid::iterator it2 = child2->mrNetworkGrid.begin
       (); it2 != child2->mrNetworkGrid.end(); ++it2){
      if(it != it2){
        if(dist > it->second->distanceTo(it2->second)){
          closest = it2->second;
          dist = it->second->distanceTo(it2->second);
        }
      }
    }
    if(!it->second->isConnection(closest->coords)){
      d = new delay((int)round(dist),"D");
      delays.insert(pair<uint64_t,element*>((uint64_t)d,(
         element*)d));
      s = new mrSynapse(weightContainer2[it->first][i],"W");
      d->addOutputElement((element*)s);
      d->addInputElement((element*)it->second);
      s->addInputElement((element*)d);
      s->addOutputElement((element*)closest);
      child2->delays.insert(pair<uint64_t,element*>((uint64_t)
         d,(element*)d));
      it->second->addOutputElement((element*)d,closest->coords
         );
```

```
            closest->addInputElement((element*)s,it->second->coords)
                ;
          }
        }
        else{
          NeuronGrid::iterator match = child2->mrNetworkGrid.find(
              edgeContainer2[it->first][i]);
          if(!it->second->isConnection(match->second->coords)){
            double dist = it->second->distanceTo(match->second);
            d = new delay((int)round(dist),"D");
            delays.insert(pair<uint64_t,element*>((uint64_t)d,(
                element*)d));
            s = new mrSynapse(weightContainer2[it->first][i],"W");
            d->addOutputElement((element*)s);
            d->addInputElement((element*)it->second);
            s->addInputElement((element*)d);
            s->addOutputElement((element*)match->second);
            child2->delays.insert(pair<uint64_t,element*>((uint64_t)
                d,(element*)d));
            it->second->addOutputElement((element*)d,match->second->
                coords);
            match->second->addInputElement((element*)s,it->second->
                coords);
          }
        }
      }
    }

    vector<mrNetwork*> children;
    children.push_back(child1);
```

```
    children.push_back(child2);
    return children;

}
```

B Sample Files

B.1 Network File

A sample network file from an XOR network. The first 4 lines specify the embedding and the number of I/O neurons. The rest of the lines describe the neurons and their outgoing synapses. Synapse lines starting with S specify an input synapse for an input neuron, unlike other synapses.

```
Embedded: 2
MaxDims: 5.000000 5.000000
In: 4
Out: 2
I 0 0.000000 0.000000 Refrac: 1 Thres: 8
 S 9 D 0
 D 5 W 5 O 1.000000 5.000000
 D 2 W 4 I 2.000000 0.000000
 D 3 W 7 I 3.000000 0.000000
O 0 0.000000 5.000000 Refrac: 1 Thres: 6
I 1 1.000000 0.000000 Refrac: 1 Thres: 2
 S 3 D 0
N 1.000000 1.000000 Refrac: 1 Thres: 1
 D 1 W -1 I 1.000000 0.000000
 D 1 W 4 I 2.000000 0.000000
O 1 1.000000 5.000000 Refrac: 1 Thres: 1
 D 5 W 5 I 0.000000 0.000000
I 2 2.000000 0.000000 Refrac: 1 Thres: 5
 S 6 D 0
 D 2 W 5 I 0.000000 0.000000
 D 5 W -6 O 1.000000 5.000000
I 3 3.000000 0.000000 Refrac: 1 Thres: 1
 S 2 D 0
```

B.2 Parameter File

A sample parameter file. Sets of parameters include comments above them. Comment lines start with #. The comments describe the valid values for that parameter, and explain how that parameter affects the networks.

```
#This is the maximum magnitude of the synaptic weights
#If the value is set to k, then the weights can range from -k to k
#Also note that if Real LTP/LTD is used, then the model will still
#be constrained by the same LRS -> HRS range
#Should be an integer value

MAXLEVEL: 10,

#This is the maximum threshold value of the neuron
#Notice the relationship between the weight and threshold
#If a synapse has a weight of 1, and neuron has a threshold of 2
#then it will take 2 synaptic fires to produce 1 neuron fire
#Should be an integer value

MAXTHRESHOLD: 10,

#This defines the behavior of the threshold
#If VAR then threshold varies from 1 to MAXTHRESHOLD
#If FIXED threshold is fixed at MAXTHRESHOLD

THRESHOLDTYPE: VAR,

#This specifies the dimension that the network is embedded in
#Currently, the model is only capable of 2D embeddings
#Arbitrary embeddings are a consideration for future work
```

```
#Should be an integer value

DIMENSION: 2,

#This specifies whether any learning should be done
#If on, then learning based on LEARNINGTYPE will be used
#Should be a string ON or OFF

LEARNING: ON,

#This specifies the learning type and controls LTP/LTD
#CONSTANT - increase/decrease weight by constant value, specified
   by
#the LEARNINGPARAM variable
#REAL - perform learning that accurately models the device
#STDP - not implemented yet
#Should be a string specified above

LEARNINGTYPE: REAL,

#This is a value used by the the potentiation/depression functions
#Constant - value to increase/decrease by
#Real - unused, might make this the delR value
#STDP - the integer number of cycles to track for STDP
#Should be floating point value

LEARNINGPARAM: 3.0,

#This sets the maximum value for a neuron's refractory period
#Should be an integer
```

```
MAXREFRACPERIOD: 1,

#This sets the minimum value for a neuron's refractory period
#This is important for STDP
#with STDP this value should be at least the learning param
#if REFRACTYPE is FIXED this value is not used
#not sure how this will work for 0
#should be an integer

MINREFRACPERIOD: 1,

#This determines how the refractory periods should vary
#FIXED - All neurons will have the same refractory period
#          this refractory period will be MAXREFRACPERIOD
#VAR - The refractory periods will be uniform random
#        from 1 to MAXREFRACPERIOD
#Should be a string

REFRACTYPE: FIXED,

#These are the synapse/memristor parameters
#These facilitate LTP/STDP
MEMHRS: 50000,
MEMLRS: 5000,
CLOCKPERIOD: 40E-9,
POSITIVESWITCHTIME: 1E-6,
NEGATIVESWITCHTIME: 1E-6,
POSITIVETHRESHOLDVOLTAGE: .75,
NEGATIVETHRESHOLDVOLTAGE: .75,
```

```
LEARNINGVOLTAGE: 1.2,

#this sets a lower limit on the accumulation
#ex. you have a neuron receive 3 charges from a synapse with a
    weight -5
#this neuron will accumulate to -12, not -15
#only used for lower limit, upper limit is irrelevant as threshold
    handles that
#Note, this param is very important for accuracy between model and
    hardware
THRESHOLDLIMIT: 12,

#Parameters we're using for variability analysis
#use variation as 1 or 0(on or off)
#deviations are read as percents
USEMEMVARIATION: 0,
HRSDEV: 20.0,
LRSDEV: 10.0,
PTDEV: 10.0,
NTDEV: 10.0,
PSTDEV: 5.0,
NSTDEV: 5.0,

USECYCVARIATION: 0,
PDRDEV: 10.0,
NDRDEV: 10.0,

#make this TRUE if you want to have events dynamically printed
    during runtime
PRINTEVENTS: FALSE,
```

```
#make this TRUE if you want to collect activity for energy
    estimation
COLLECT_ACTIVITY: FALSE,

#Parameters for energy estimation
#Specify energy consumed for each operation of component
#All units are assumed to be the same
NEURON_FIRE: 1.0,
NEURON_ACCUM: 1.0,
NEURON_PASSIVE: 1.0,
SYNAPSE_ACTIVE: 1.0,
SYNAPSE_POT: 1.0,
SYNAPSE_DEP: 1.0,
DELAY_MOVESPIKE: 1.0,
DELAY_PASSIVE: 1.0,

}
```

B.3 Input File

A sample 40 cycle input for an XOR network. Every 10 cycles the input neurons are pulsed.

```
CC 0 I 0 I 1 I 0 I 1
CC 1 I 0 I 0 I 0 I 0
CC 2 I 0 I 0 I 0 I 0
CC 3 I 0 I 0 I 0 I 0
CC 4 I 0 I 0 I 0 I 0
CC 5 I 0 I 0 I 0 I 0
CC 6 I 0 I 0 I 0 I 0
CC 7 I 0 I 0 I 0 I 0
```

CC 8 I 0 I 0 I 0 I 0

CC 9 I 0 I 0 I 0 I 0

CC 10 I 0 I 1 I 1 I 0

CC 11 I 0 I 0 I 0 I 0

CC 12 I 0 I 0 I 0 I 0

CC 13 I 0 I 0 I 0 I 0

CC 14 I 0 I 0 I 0 I 0

CC 15 I 0 I 0 I 0 I 0

CC 16 I 0 I 0 I 0 I 0

CC 17 I 0 I 0 I 0 I 0

CC 18 I 0 I 0 I 0 I 0

CC 19 I 0 I 0 I 0 I 0

CC 20 I 1 I 0 I 0 I 1

CC 21 I 0 I 0 I 0 I 0

CC 22 I 0 I 0 I 0 I 0

CC 23 I 0 I 0 I 0 I 0

CC 24 I 0 I 0 I 0 I 0

CC 25 I 0 I 0 I 0 I 0

CC 26 I 0 I 0 I 0 I 0

CC 27 I 0 I 0 I 0 I 0

CC 28 I 0 I 0 I 0 I 0

CC 29 I 0 I 0 I 0 I 0

CC 30 I 1 I 0 I 1 I 0

CC 31 I 0 I 0 I 0 I 0

CC 32 I 0 I 0 I 0 I 0

CC 33 I 0 I 0 I 0 I 0

CC 34 I 0 I 0 I 0 I 0

CC 35 I 0 I 0 I 0 I 0

CC 36 I 0 I 0 I 0 I 0

CC 37 I 0 I 0 I 0 I 0

CC 38 I 0 I 0 I 0 I 0

CC 39 I 0 I 0 I 0 I 0

Vita

Austin Wyer was born in Clarksville, TN, to Robb and Karen Wyer. He is the older of two siblings, the other being Amy Wyer. He graduated from West Creek High School in 2012. After graduation, he went on to attend the University of Tennessee Knoxville to pursue his bachelor's degree. Initially studying aerospace engineering, a study abroad experience with Dr. Michael Berry in London sparked an interest in computer science. He quickly changed majors in 2014, seeking to conduct research in this new area of study. He found an opportunity to study hardware security with Dr. Garrett Rose. Eventually, he branched into neuromorphic computing, which he then conducted research on for his Masters of Science degree in Computer Science under Dr. Rose, which he completed in December 2017. He is now studying graph algorithms while pursuing a PhD with Dr. Michael Langston at the University of Tennessee Knoxville.