AFRL-RY-WP-TR-2021-0144

LowPy: SIMULATION PLATFORM FOR MACHINE LEARNING ALGORITHM REALIZATION IN NEUROMORPHIC RRAM-BASED PROCESSORS (Preprint)

Andrew J. Ford (University of Cincinnati)
University of Cincinnati

JUNE 2021
Final Report

Approved for public release; distribution is unlimited.

See additional restrictions described on inside pages

STINFO COPY

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS**.

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
June 2021	Thesis/Dissertation	1 June 2021 –1 June 2021

4. TITLE AND SUBTITLE
LowPy: SIMULATION PLATFORM FOR MACHINE LEARNING ALGORITHM REALIZATION IN NEUROMORPHIC RRAM-BASED PROCESSORS (Preprint)

5a. CONTRACT NUMBER
FA8650-14-D-17240004/CCF-1718428/ECCS-1926465

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
69199F/62204F

6. AUTHOR(S)
Andrew J. Ford

5d. PROJECT NUMBER
2002

5e. TASK NUMBER
N/A

5f. WORK UNIT NUMBER
Y18V

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Cincinnati
2600 Clifton Ave.
Cincinnati, OH 45221

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

National Science Foundation (NSF)
2415 Eisenhower Ave.
Alexandria, VA 22314

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYDT

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2021-0144

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
PAO case number AFRL-2021-1666, Clearance Date 1 June 2021. Submitted to the Graduate School of the University of Cincinnati in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering. Report contains color.

14. ABSTRACT

A novel compilation of non-ideal characteristics which accompany hardware realizations of machine learning algorithms in the form RRAM-based neuromorphic ASIC processors is presented within a convenient, simple, and powerful simulation library named LowPy for use with Python. Simulations results are shown for four different networks; SLP, MLP, CNN, and LSTM. Each is subjected to six different GPU-accelerated nonideality functions backed by experimentally gathered data, as well as data provided by external research. Of the six nonideality functions, a spread of selected parameters is chosen such that any performance impacts of the algorithm are easily observed across several orders of magnitude. Main aspects differentiating LowPy from other neuromorphic simulation platforms include functions not yet implemented by other platforms, the most non-ideal functions provided by any platform known by the author to date, an event-driven architecture that provides the user full control over which and when nonideality functions are executed during training and testing, as well as its ability to wrap around an existing well-documented, popular, GPU-accelerated machine learning library.

15. SUBJECT TERMS
neuromorphic computing, on-chip machine learning, python, RRAM

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Vipul Patel
Unclassified	Unclassified	Unclassified	SAR	95	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

LowPy: Simulation Platform for Machine Learning Algorithm Realization in Neuromorphic RRAM-Based Processors

by
Andrew J. Ford
B.S. Electrical Engineering, University of Cincinnati, 2019

A Thesis submitted to the
Graduate School of the University of Cincinnati
In partial fulfillment of the requirements
for the degree of

Master of Science

in

Electrical Engineering

Department of Electrical Engineering and Computing Systems

College of Engineering and Applied Science

March 8, 2021

Committee Chair: Rashmi Jha, Ph.D.
Committee Member: Ranga Vemuri, Ph.D.
Committee Member: John Emmert, Ph.D.

Approved for public release; distribution is unlimited.

Dedications

Thank you to my parents, Evan and Kristy Ford, for providing me with the means to pursue my educational endeavors at the University of Cincinnati, and pushing me to be the best I can be. Thank you to my research advisor, Rashmi Jha, for the opportunity to pursue graduate coursework and research in her lab, for securing funding so I can focus on research, and for being understanding and motivating while I took longer than either of us would have liked to choose a research topic. Thank you to my lab mates, whose curiosity and passion for learning have rubbed off on me. And thank you to my girlfriend, Kailey Waal, who has become lovingly accustomed to the response ”No I can’t do that, I need to do research”.

Approved for public release; distribution is unlimited.

Acknowledgements

This work is supported by the Rindsberg Fellowship to Andrew J. Ford, by the National Science Foundation under awards #CCF-1718428 and #ECCS-1926465, and by the United States Air Force Research Lab and Edaptive Computing Inc. under award #FA8650-14-D-1724 0004.

Approved for public release; distribution is unlimited.

Abstract

A novel compilation of non-ideal characteristics which accompany hardware realizations of machine learning algorithms in the form RRAM-based neuromorphic ASIC processors is presented within a convenient, simple, and powerful simulation library named LowPy for use with Python. Simulations results are shown for four different networks; SLP, MLP, CNN, and LSTM. Each is subjected to six different GPU-accelerated nonideality functions backed by experimentally gathered data, as well as data provided by external research. Of the six nonideality functions, a spread of selected parameters is chosen such that any performance impacts of the algorithm are easily observed across several orders of magnitude. Main aspects differentiating LowPy from other neuromorphic simulation platforms include functions not yet implemented by other platforms, the most non-ideal functions provided by any platform known by the author to date, an event-driven architecture that provides the user full control over which and when nonideality functions are executed during training and testing, as well as its ability to wrap around an existing well-documented, popular, GPU-accelerated machine learning library.

4
Approved for public release; distribution is unlimited.

Table of Contents

Approved for public release; distribution is unlimited.

Approved for public release; distribution is unlimited.

Approved for public release; distribution is unlimited.

Approved for public release; distribution is unlimited.

List of Figures

Approved for public release; distribution is unlimited.

Approved for public release; distribution is unlimited.

Approved for public release; distribution is unlimited.

Approved for public release; distribution is unlimited.

List of Tables

Approved for public release; distribution is unlimited.

Chapter 1

Background

Year over year, machine learning becomes increasingly relevant across many sectors, with general market spending (around $1.5 billion in 2017) projected to exceed $20 billion by 2024, assuming its continued cumulative annual growth rate of over 44% [1]. This rapid increase in relevance of and demand for machine learning is closely matched by a continual accumulation of machine learn-able data, ready to be processed.

Large-scale data processing is often done in data centers [2], which are comprised of many High Performance Computing (HPC) clusters. Each of these HPC clusters are typically comprised of CPUs and GPUs, currently operating in the "10-300W range in terms of power utilization" [3], considering that "300W is the upper limit for a PCI-based accelerator card" [3]. Machine learning tends to be a power-intensive process, and some networks utilizing these data centers contain parameters on the order of billions [4]. Responsible for a much smaller energy demand, battery powered smart devices also benefit greatly from low-power machine learning processors. Not everyone has blindly surged forward in an attempt to meet the high demand of machine learning, however. The coupling of energy-hungry data centers operating around the clock with battery powered smart devices that call for increased energy efficiency to provide longer operational duration has created fiscal, ecological, and duration-based motivations to reduce the power consumed by computing processors, culminating in the field of green computing.

Green computing re-imagines portions of the computing process that result in power consumption, including data center design, product longevity, software optimization, material recycling, power management, and more. This work operates within a specific category of green computing called neuromorphic computing, which has been responsible for recent developments towards a number of Ultra Low Power (ULP) Very Large-Scale Integration (VLSI) circuits and processors [5, 6, 7]. These processors offer substantial progression towards the goal of reduced power

Approved for public release; distribution is unlimited.

machine learning, and were accompanied by a fairly involved design process. LowPy streamlines a vital portion of this design process, and offers insights when assessing the fundamental devices that these neuromorphic processors are built upon.

1.1 Machine Learning

Traditionally, software developers are tasked with bridging the gap between some set of input data, and producing some output, via computer code. They accomplish this with the invention, implementation, and refinement of a set of rules for a computer to follow when presented with that input data, until only the desired output is produced. Machine learning requires a paradigm shift, in the sense that a network is now responsible for the creation and implementation of the rules. The role of machine learning developers is to build and tune these networks such that they produce desired outputs when faced with input data.

1.1.1 Datasets

The input and output data previously mentioned can come in a variety of formats such as stock fundamentals, audio files, videos, pictures, and many more, are all contained in datasets. In the case of supervised machine learning, these datasets are conventionally comprised of input data and their corresponding desired outputs. Unsupervised machine learning may not include these outputs, and depends on the network to create rules without the supervision of any desired outputs. Developers may utilize an existing dataset, or create their own. Existing datasets are available across the internet, and some are used as popular benchmarks to compare performance. One of these datasets is the Modified National Institute of Standards and Technology's (MNIST) set of handwritten numerical digits [8]. Figure 1 shows nine of the digits from MNIST. The entire dataset is comprised of 60,000 28x28 pixel images. Each image pixel's intensity is represented by a value 0 to 255, often normalized between 0 and 1 for machine learning purposes. The dataset also contains 60,000 labels corresponding to the digits, 0 to 9. The MNIST dataset is so popular, that it has inspired variants including fashion MNIST, sign language MNIST, spoken MNIST, and

Approved for public release; distribution is unlimited.

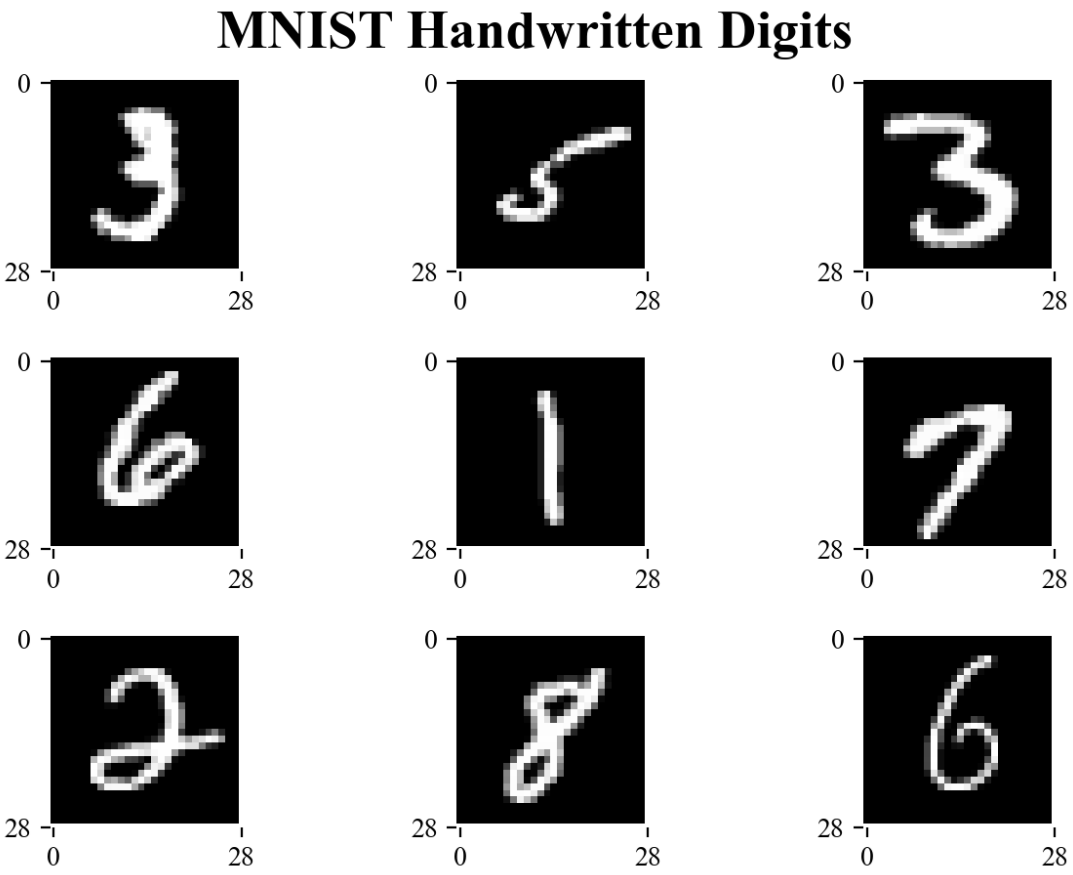


Figure 1

Nine digits from the MNIST Handwritten Digits dataset [8]. Dataset consists of 60,000 digits and labels corresponding to which number is written, 0 to 9. Each image is comprised of 28x28 pixels with values between 0 and 255 representative of pixel intensity. Often scaled down from 0 to 1 for neural networks.

more. MNIST has also been quite popular within the neuromorphic community, with a number of neuromorphic processor using it to, in a way, prove their worth, regarding machine learning capabilities [9, 10, 11].

The MNIST dataset [8] is a static dataset, where the inputs do not change over time, with the alternative being time series datsets. A stock price over a given time frame can be used as a time series dataset. Natural Language Processing (NLP) datasets would be categorized as time series, since the words occur in a sequential order over time. One such example would be the IMDB movie reviews dataset [12], shown in Figure 2. The IMDB dataset is comprised of 50,000 movie

Approved for public release; distribution is unlimited.

```
Although I didn't like Stanley & Iris tremendously as a film,
I did admire the acting.  Jane Fonda and Robert De Niro are
great in this movie.  I haven't always been a fan of Fonda's
work but here she is delicate and strong at the same time.  De
Niro has the ability to make every role he portrays into acting
gold.  He gives a great performance in this film and there is
a great scene where he has to take his father to a home for
elderly people because he can't care for him anymore that will
break your heart.  I wouldn't really recommend this film as
a great cinematic entertainment, but I will say you won't see
much bette acting anywhere.
```

Figure 2

Example input from the IMDB movie reviews dataset [12]. Labels are 0 or 1, corresponding to the negative or positive sentiment found in the review, respectively. Dataset contains 50,000 text files of move reviews and sentiment labels.

reviews stored in text files. Each text file has a corresponding label 0 or 1, indicating whether the review is negative or positive, respectively. Many reviews contain misspellings, varying dialects, and other factors that could make sentiment discernment difficult. This NLP sentiment predicting often work better with networks designed to handle time series data.

1.1.2 Packages

Popular coding languages offer powerful and well documented packages or libraries built for machine learning. Python, one of, if not the, most popular machine learning language has many packages available for developers. TensorFlow [13], created by Google, is a symbolic math library built for machine learning in Python. TensorFlow offers the user automatic differentiation, linear algebra operations, and parallelized, graph based executions that can be run on Nvidia's CUDA [14] enabled GPUs. The Keras [15] API is built with TensorFlow's functions, and included within TensorFlow. Keras provides the user with models, optimizers, loss functions, and a multitude of layer types and functions for building and using neural networks. LowPy wraps around the Keras model, and offers the user access to functions that selectively manipulate the weights in the Keras

Approved for public release; distribution is unlimited.

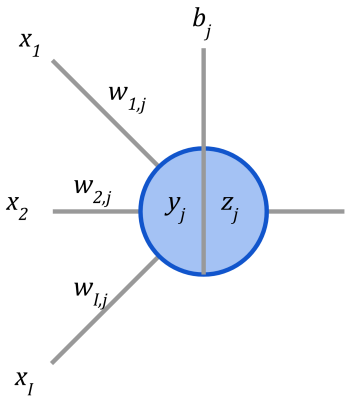


Figure 3

Diagrammatic representation of a perceptron. Input edges from the left are weighted $w_{i,j}$, with the first subscript i indicating the input index, and the second subscript j indicating the neuron index. Each neuron has one incoming connection from the top known as its bias b_j. The perceptron can execute a Multiply And Accumulate (MAC) function to produce an output, y_j. This output y_j can then be activated using a variety of activation functions to produce an activated output z_j. The neuron's activated output z_j, can then be linked to subsequent, or previous neurons, depending on the layer architecture.

layers to emulate properties introduced by neuromorphic processors. In order to better understand the results gathered by LowPy's non-ideal device characteristic simulations, it is important to understand the background behind the layers. The chosen layers for this paper include Single Layer Perceptrons (SLPs), Multi-Layer Perceptrons (MLPs), Convolutional Neural Networks (CNNs), and Long Short-Term Memory (LSTM) networks. These network types were chosen based on their popularity, contrast relative to one another, and to provide a mix of networks predisposed to perform well on image recognition datasets as well as time series data.

1.1.3 The Perceptron

It can be said that the perceptron is the building block of several neural networks [16]. In a neural network, the perceptron is analogous to a neuron, and when interconnected, functions as a connectionist network. Shown in Figure 3, it consists of several incoming connections on the left, one from the top, and can have one or more output to the right. The weights and bias linked to each neuron can be tuned and optimized to produce desired outputs specified by the developer, and are

Approved for public release; distribution is unlimited.

tuned via some sort of learning algorithm like regression or backpropagation. Each perceptron can be forward propagated via the Multiply and Accumulation (MAC) function, shown in (Equation 1).

$$y_j = \sum_{i=0}^{I} (w_{i,j} \cdot x_i) + b_j \tag{1}$$

The MAC function sums the product of the incoming weights and their corresponding inputs, and adds the bias to produce the perceptron's output y_j. Usually, perceptrons will also contain an activation function, most commonly the sigmoid (Equation 2), tanh (Equation 3), ReLu (Equation 4), or Linear (Equation 5) functions.

$$z_j = \frac{1}{1+e^{-y_j}} \tag{2}$$

$$z_j = tanh(y_j) \tag{3}$$

$$z_j = max(0, y_j) \tag{4}$$

$$z_j = y_j \tag{5}$$

These functions often prevent the change in weight values from vanishing (becoming too small) or exploding (becoming too large).

1.1.4 Single Layer Perceptron

The Single Layer Perceptron (SLP) [17] consists of one input layer fully connected to an output layer, as shown in Figure 4. The five blue circles on the left are representative of the inputs (pixel values of MNIST [8], words from IMDB dataset), while the orange circles represent the outputs (MNIST number prediction, IMDB movie review sentiment). 'Fully connected' refers to the weights, represented in Figure 4 by the gray lines, that connect the input and output layers. The diagram in Figure 4 represents the forward propagation (Equation 1). It is built of perceptrons

Approved for public release; distribution is unlimited.

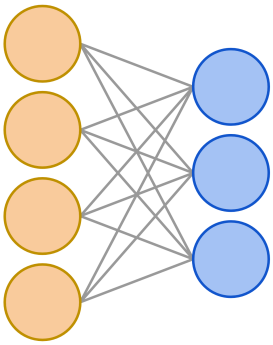


Figure 4

Fully connected diagrammatic representation of a Single Layer Perceptron (SLP). Consists of one input layer (orange) fully connected with one output layer (blue). Connections (gray lines) represent weights, linking to perceptrons shown in Figure 3.

(blue), while the orange input layer simply offers one input value per input neuron (an MNIST pixel value, an IMDB movie review word index). The SLP is lightweight, relative to most other network architectures, and often struggles when extracting features from complex data. On MNIST, it is not uncommon to see SLPs capable of exceeding 90% accuracy, though this is likely not desirable for most applications.

1.1.5 Multi-Layer Perceptron

An SLP may transition into the category of deep learning with the addition of hidden layers. Referred to as Multi-Layer Perceptrons (MLPs) [17], these SLPs + Hidden Layer(s) are comprised of the same perceptrons that make up an SLP, and provide the network with the ability to learn significantly more complex features from the dataset. A diagrammatic representation is shown in Figure 5, where the same orange and blue layers are now outside of a green hidden layer. The number of neurons in the hidden layer is treated as a hyperparameter, and while there are recommendations on how many should be included, the number of hidden layers and number of neurons that produces the highest accuracy for a network will depend on a number of factors.

Approved for public release; distribution is unlimited.

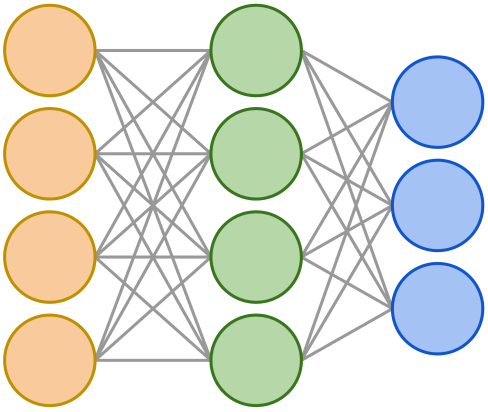


Figure 5

Fully connected diagrammatic representation of a Multi-Layer Perceptron (MLP). Consists of one input layer (orange), fully connected with one hidden layer (green), connected to one output layer (blue). Connections (gray lines) represent weights, linking to perceptrons shown in Figure 3.

1.1.6 Convolutional Neural Network

SLP and MLPs can be manipulated to create other layers, or can be prepended with data preproccessing layers. One such preprocesssing layer is the convolutional layer. Convolutional Neural Networks (CNNs) utilize matrices of coefficients called filters to expose and hide certain features of data. There are filters to expose edges of objects within images, remove noise, expose vertical, horizontal, or other angled features, and more. They can be treated as another hyperparameter for tuning a neural network to work well on a specific dataset.

The degree to which convolutional filters expose the features of the input data is high enough that the inputs can be reduced in size to allow for high accuracy with smaller classification networks. This dimensionality reduction is done with pooling layers, which preserve typically a sliding average of the image inputs, or a sliding maximum of the images. For the scope of this paper, it is only necessary to understand that these convolutional filters expose features via sliding matrices of fixed coefficients, and the pooling layers slide across the inputs to reduce dimensionality.

Approved for public release; distribution is unlimited.

1.1.7 *Recurrent Neural Network*

The first major modification to consider for a MLP is recurrent handling of the perceptron connections, to create a Recurrent Neural Network (RNN) better disposed to handle sequential or time series data. While there are a multitude of ways to visualize this, the simplest is to consider the outputs of the orange layer in Figure 5 feeding back as inputs into the green layer in Figure 5. Their feed forward equations are found in (Equation 6).

$$\begin{aligned} h_t = \sigma_h(i_t) = \sigma_h(U_h x_t + V_h h_{t-1} + b_h) \\ z_t = \sigma_y(a_t) = \sigma_y(W_y h_t + b_h) \end{aligned} \tag{6}$$

While RNNs are a good first step towards handling time series data, they have drawbacks that make them often insufficient for most complex datasets.

The largest issue with RNNs is their proclivity to experience vanishing gradients. As sequences grow in length, the likelihood that the backpropagation or regression algorithm will fail to produce any change to be applied to the RNN's weights becomes higher. The IMDB dataset [12], for instance, contains reviews with hundreds of words, and this is often too many for RNNs to handle.

1.1.8 *Long Short-Term Memory Network*

One such modification to RNNs seeking to rectify the vanishing gradient issue is the Long Short-Term memory network. Proposed 1999 [18], LSTMs are still widely used today for NLP [19] and many other applications. They contain some modifications to RNNs in their forward propagation equations, found in (Equation 7), (Equation 8), (Equation 9), (Equation 10), (Equation 11), and (Equation 12).

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{7}$$

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{8}$$

Approved for public release; distribution is unlimited.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{9}$$

$$\hat{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_c) \tag{10}$$

$$C_t = f_t \circ C_{t-1} + i_t \circ \hat{C}_t \tag{11}$$

$$h_t = o_t \circ \tanh C_t \tag{12}$$

The forget gate in (Equation 7) is responsible for deciding which information from inputs to retain, and which to ignore. The input gate in (Equation 8) annd the hidden state (Equation 12) are used to update the cell state in (Equation 11). Lastly, the output gate (Equation 9) is responsible for deciding what the next hidden state will be.

Another popular modification the Gated Recurrrent Unit (GRU), which, along with the RNN, is easy to simulate with Keras [15] and LowPy, but is not examined within this paper.

1.1.9 Loss Functions

Intuitively it makes sense that our neural networks all have the same goal of achieving the highest accuracy on a chosen dataset, but this is achieved in a roundabout way. Neural networks most often tune their weights based on guidance from an algorithm seeking to minimize a network's error. Commonly referred to as cost, loss, and error functions, a few of the more popular loss functions are listed in (Equation 13) and (Equation 14).

$$MSE = \sum_{i=1}^{n} \frac{(y_i - \hat{y}_i)^2}{n} \tag{13}$$

$$CrossEntropyLoss = -(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)) \tag{14}$$

Approved for public release; distribution is unlimited.

While these loss functions all report a measurement of how well a neural network models given data, they can have a significant impact on how the weights of a neural network are updated.

1.1.10 Optimizers

Now that several common network types and loss functions have been defined, everything is tied together with optimizers. Most commonly, optimizers offer up the numerical adjustments that should be applied to neural network weights so that the network can minimize the loss reported by the chosen loss function. Optimizers calculate these numerical adjustments, commonly referred to as gradients, using a variety of algorithms.

1.1.10.1 Gradient Descent Seeking to minimize some function $J(\theta)$, where θ refers to a neural network's trainable parameters, and J refers to a loss function, gradient descent continually solves (Equation 15), where η is an adjustable coefficient representative of the rate at which the network learns.

$$\theta = \theta - \eta \nabla_\theta J(\theta) \tag{15}$$

Stochastic Gradient Descent (SGD) [20] takes in one input from the dataset at a time, and based on the corresponding label, and the outputs from the neural network, makes adjustments to the trainable parameters θ.

Batch Gradient Descent (BGD) looks at the entire dataset at once, and adjusts the weights in one large swoop. While this method demands less processing time, datasets often don't fit on GPUs/RAM cards, and it does not update as frequently as SGD.

Mini-Batch Gradient Descent (MBGD) breaks the dataset up into smaller batches, and applies updates to the weights based on each individual batch. In this work, we use Mini-Batch Gradient Descent (MBGD), as it offers a good compromise between the two. While it might be more difficult to model in hardware than SGD, simulation times are orders of magnitude quicker.

Approved for public release; distribution is unlimited.

1.1.10.2 Adam Optimizer

Adaptive Moment Estimation (Adam) [21], an adaptation on the SGD learning algorithm, was introduced in 2017, and adapts the learning rate based on the calculated loss value (Equation 16).

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{16}$$

1.2 Neuromorphic Computing

With sufficient background in the basics of machine learning, it is then easier to understand where neuromorphic computing fits into the picture. It can be argued that machine learning algorithms themselves take inspiration from the learning methods which biological organisms employ, such as repeated input stimulus, neuronal potentiation and depression, and some algorithms, such as LSTMs, forgetting irrelevant information as it learns. Neuromorphic computing takes inspiration from biology, neurology, and nature to create algorithms implementable in hardware in pursuit of low power, efficient machine learning, more closely replicating the way brains function. For instance, the neurons in our brains pass spikes of voltage around for learning and actions, unlike the previously discussed algorithms, which take a more analog approach. This more discrete format results in much lower power consumption, and has been been favored by the neuromorphic community, as many neuromorphic processors implement these Spiking Neural Networks (SNNs) [22, 23, 24].

There are many other methods utilized by neuromorphic Very Large-Scale Integration (VLSI) systems to mimic neuro-biological architectures, most of which are realized with oxide-based memristors, spintronic memories, and other forms of transistors.

While the inspiration drawn from other scientific fields is currently has been prevalent in the neuromorphic computing field, it currently offers a unique solution to the memory bandwidth problem plaguing traditional computing architectures. Since floating point operations are executed currently in devoted processors such as CPUs which draw information to and from Hard Drive

Approved for public release; distribution is unlimited.

Disks (HDDs), Solid State Drives (SSDs), and Random Access Memory (RAM), the PCIe lines present a delay in processing time unavoidable with traditional computing architectures. Neuromorphic processors offer a simpler, more elegant solution: In-Memory Computing [25]

In-Memory Computing [25] takes full advantage of the data retention characteristics after the removal of power from RRAM cells. The same memory which is used to store data and information is used to execute computational operations, instead of relying on multiple off-chip memory banks and waiting for their information transfer times to complete. Therefore, a neuromorphic CPU may one day no longer require the use of RAM and other memory based peripherals.

1.2.1 Resistive Random Access Memory

Postulated in 1971 [26], the actual memristor was first tangibly discovered in 2008 [27]. Put simply, it is a variable resistor that retains its resistance value, even without power. Compared to previously known fundamental circuit components like the resistor, capacitor, and inductor, this new component offers attributes not previously available to circuit designers. For many neuromorphic processors, this readily serves as a fundamental building block of neuromorphic devices.

Memristors have now opened the door to very low power computations. They can function as Random Access Memory (RAM) does, donning the name Resistive Random Access Memory (ReRAM or RRAM). In terms of neuromorphic RRAM-based machine learning algorithm hardware realizations, the layered perceptrons have been realized in many ways [28, 29, 30], along with RRAM-based LSTMs [31, 32], with all proudly reporting power consumption orders of magnitudes lower than their CPU/GPU implemented counterparts.

1.2.1.1 Single and Multi State RRAM

In terms of states, it should be mentioned that RRAM can be used in a few different ways. At its simplest, RRAM can behave as a nonlinear variable resistor, functioning in an analog manner, typically used in approximate computing applications [33]. This form of RRAM, referred to as Single Bit RRAM (SBRRAM), contains a Low Resistance State (LRS), as well as a High Resistance State (HRS). In this paper, the term

Approved for public release; distribution is unlimited.

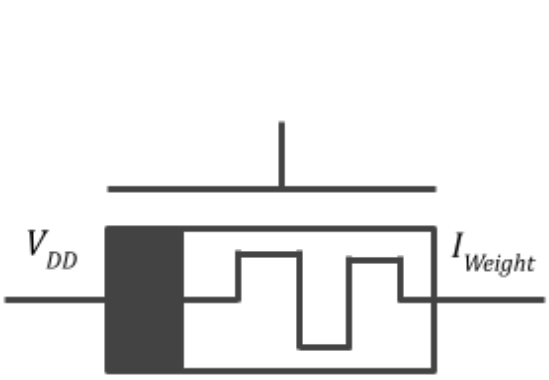
”High Resistance State (HRS)” will be used interchangeably with the term ”Low Conductance State (LCS)”, and likewise, the term ”Low Resistance State (LRS)” will be used interchangeably with ”High Conductance State”. Although not common nomenclature, referring to the states based on their conductance levels aligns more simply with the way the weights in neural networks have an upper and lower bound, can now be paired with HCS and LCS, respectively, to avoid confusion. Approximate computing utilizes the analog region between the two states, often a more volatile region than that of HRS and LRS. In SBRRAM, the number of possible states is not measured by actual intermediate divisions, but rather by the accuracy with which memory writes can occur. SBRRAM saw 7-bit write precision in 2012 [34], equivalent to under 1% write variability. Presently, there is a strong call for SBRRAM with 8-bit precision for use with convolutional filters, with most papers reporting that anything beyond 7-bit RRAM as unfeasible, even asserting a more reasonable range of 2 to 4 bit RRAM [35] being more commonplace.

Alternatively, Multi-State RRAM provides stability between HRS and LRS, and offers a number of intermediate resistance states (IRSs). MSRRAM with 16 states (4-bit) [36] has been reported.

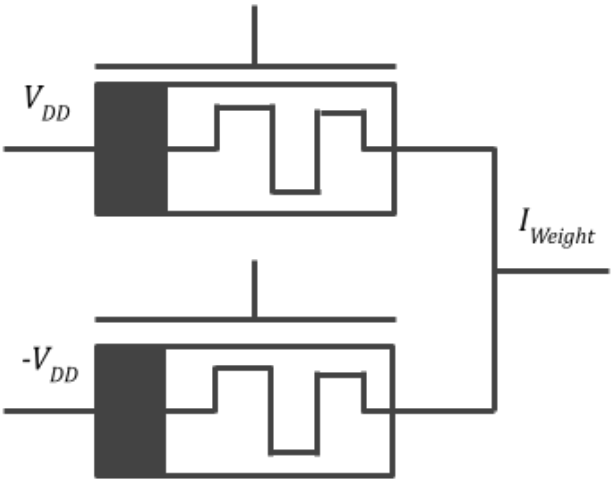
1.2.1.2 Synaptic Configurations The next relevant decision following that of state configuration when electing to utilize RRAM for hardware implementation of a machine learning algorithm is which cell configuration to use. The algorithms examined within the scope of this work rely on negative and positive weights, centered around zero.

One-transistor, one-RRAM (1T1R) cell synaptic configurations, as shown in Figure 6a, use only one RRAM cell to store the entirety of the weight. This can be accomplished in multiple ways, commonly defining a weight of 0 as the center state of the RRAM cell [38]. Two-transistor, two-RRAM (2T2R) cell configurations, as Figure 6b, rely on one RRAM cell to represent the weight below zero, and another to hold weights above zero [39]. At the cost of doubling area and power consumption on a synaptic level, either the precision or weight range is effectively doubled. It should also be noted that both Figure 6a and Figure 6b show gate terminals above the cells used

Approved for public release; distribution is unlimited.



(a) 1T1R



(b) 2T2R

Figure 6

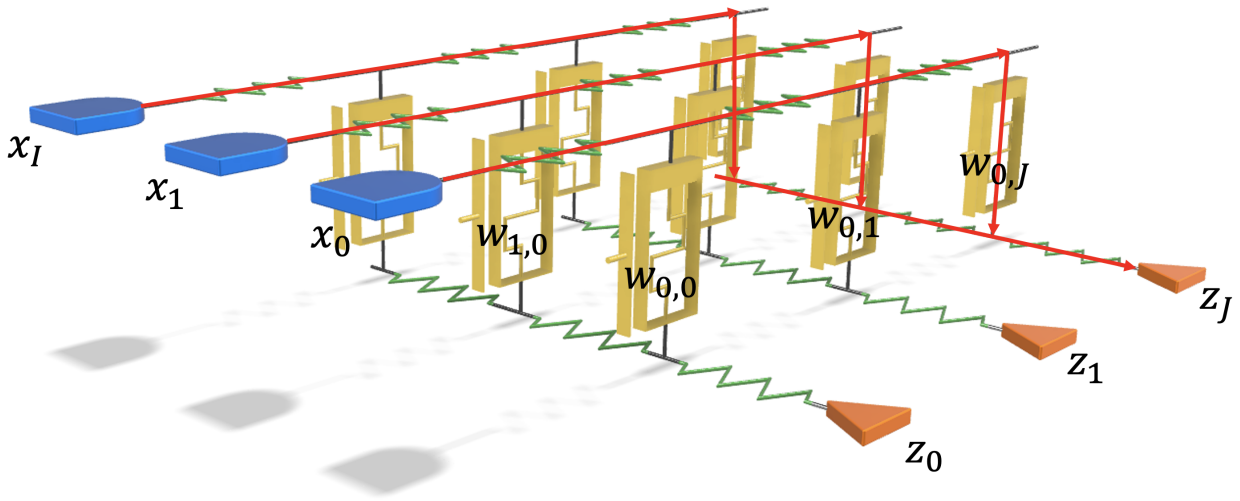
Figure 6a shows a 1T1R synaptic cell configuration. An incoming voltage is encoded as an output current, and later parts of the circuit determine the sign of the weight represented by the current. Figure 6b shows a 2T2R synaptic cell configuration, consisting of two memristors, one devoted to the positive part of the weight, and the other handling the negative portion of the weight. While there are a multitude of ways to implement this, at a high level, most follow the principles depicted in this diagram. Each cell in either subfigure shows a terminal on the side of the cell, representative of the gate terminal in Gated RRAM [37]. With non-gated RRAM, a transistor is used to select and write new values to the RRAM cells.

to write new values to the RRAM cells, representative of Gated RRAM [37]. However, in the case of standard RRAM cells, they likely rely on a transistor to select and write to each individual cell.

1.2.2 Crossbar Arrays

Many neuromorphic processors make use of memristive crossbar arrays to organize access to each individual cell [40]. A 3-D representation of a gated RRAM crossbar array can be found in Figure 7. This crossbar array is an implementation of a MAC operation from (Equation 1). Input voltages pass into the array via the three orange nodes on the left of Figure 7. They are then encoded as currents in the RRAM cells, at which point they are accumulated and then exit the memristive array through the blue nodes. This is representative of a 3 input, 2 output fully connected layer, and can be expanded and tiled as needed.

Approved for public release; distribution is unlimited.



$$y_J = x_0 \cdot w_{0,J} + x_1 \cdot w_{1,J} + \cdots + x_I \cdot w_{I,J}$$

Figure 7

3D representation of a fully connected RRAM crossbar array. Incoming voltages from the orange nodes are encoded as currents as they pass through the yellow RRAM cells and exit as MAC currents through the blue nodes. The gate terminal is characteristic of a novel device created by Dr. Jha's lab [41, 37]. Figure 7 shows an example calculation for the non-activated, non-biased output neuron J, using (Equation 1).

1.2.3 Non-Ideal Characteristics

With the motivation for developing neuromorphic machine learning processors now established, it is necessary to now examine the downsides. As many of the realized neuromorphic processors will report, RRAM contains nonidealities that cannot be ignored. Inherently, memristors are nonlinear devices, unlike normal resistors. Given that neural networks are fueled by their learning/optimization algorithms, a well written learning algorithm is fundamentally unlikely to be augmented by non-ideal behavior, unless done so by sheer chance. So, it is intuitive to assume that any nonideality too significant will impair the algorithm in almost any case. Therefore, sources normally attempt to discern not whether a given nonideality will enhance a learning algorithm, but rather to what degree can a machine learning algorithm tolerate this nonideality [42].

Approved for public release; distribution is unlimited.

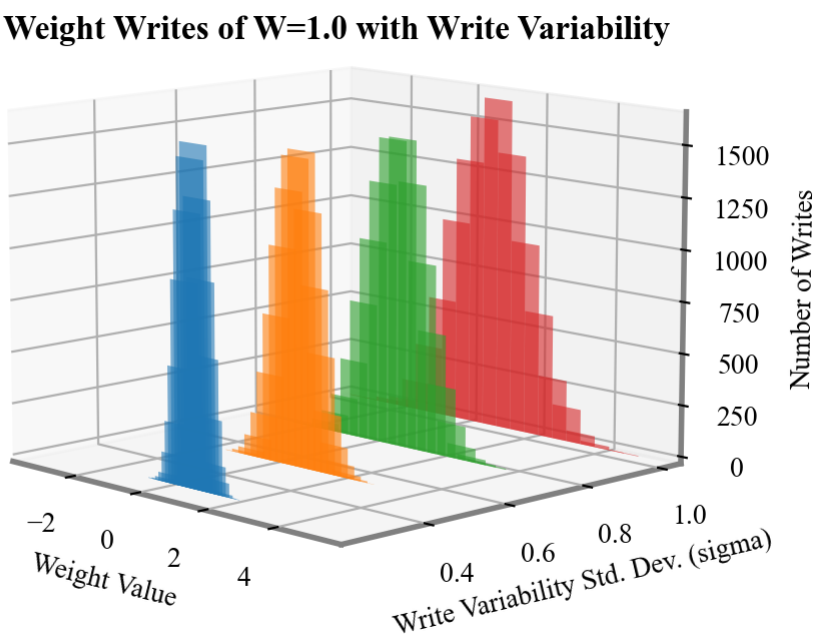


Figure 8

Example distribution of actual weight writes to different RRAM cells. Increased standard deviation (sigma) results in a wider normal distribution of weights deviating from the desired value of 1.0.

1.2.3.1 Write Variability In neuromorphic processors, the role of RRAM is most often to store a weight value, and aid in forward propagation when called upon. However, individual RRAM cells vary from device to device, resulting in, among many other things, write variability [43]. With RRAM cells holding weight values, this results in discrepancy between the actual written weight value and the requested weight value. Depending upon a multitude of factors, mainly material type and fabrication quality, RRAM can have a wider or narrower standard deviation of write variability [44], visualized in Figure 8.

1.2.3.2 Drift After writing to an RRAM cell, it can undergo a form of drift, where the value written to the cell slowly begins to move in a certain direction over time. This occurs in the more volatile, Short-Term Memory (STM) forms of RRAM, and is not usually as significant in MSRRAM. Experimentally collected by Brett Hochman, data from Dr. Rashmi Jha's lab in Figure 9 shows a SBRRAM cell drifting over time from its originally specified states, as it is

Approved for public release; distribution is unlimited.

influenced by the read voltage. Inherently, this behavior has a reasonable amount of stochasticity associated with it, as the resistance fluctuates substantially, notably during the $0.2V$ HRS trial in Figure 9a, and the $0.8V$ LRS trial in Figure 9b. Additionally, it is worth noting that the majority of drift occurs early on, and the cell appears to settle into its experimental state as time progresses, at which point it deviates less and less.

1.2.3.3 Time Decay Some types of RRAM also experience a form of time decay, not unlike that of a capacitor [45]. Data gathered by Aaron Ruen in Dr. Rashmi Jha's research lab, shown in Figure 10, illustrates how a written value to RRAM cells can slowly sink towards a higher resistance level, outputting less and less current over time.

1.2.3.4 Stuck-At-Fault RRAM cells experience failure from time to time, even with endurance cycles reaching into the trillions [46]. Similar in concept to other memory devices becoming Stuck-At-Fault (SAF), cells can become Stuck-At-State (SAS), and depending upon their synaptic cell configuration, this can manifest in a few different ways. 1T1R cells can either end up stuck at their HRS or LRS, to the point where they conduct freely or not at all. 2T2R cells can also become stuck at HRS or LRS, but in terms of neural network weights, this becomes the upper weight limit, 0, or the lower weight limit. Additionally, assuming that SAS RRAM cells have a 50% chance of residing at LRS, and a 50% chance of residing at HRS, this then indicates that 25% of cells would be stuck at the upper weight limit, 25% at the lower weight limit, and 50% at 0.

1.2.3.5 Random Telegraph Noise An obstacle with RRAM's scalability is the increase in Random Telegraph Noise (RTN) present as RRAM cells become smaller [47]. Researchers have provided methods for reducing the RTN when scaling RRAM [47], though it is still present to some degree. It can be modeled fairly similarly to write variability from Figure 8, following a normal distribution controlled by some standard deviation parameter σ. [48].

Approved for public release; distribution is unlimited.

1.2.3.6 Limited Precision

As discussed previously, MSRRAM features several states which can be used to store weight values, currently up to 4-bits. CPUs and GPUs rely on single, double, and half floating point precision, which, by comparison, is often substantially higher than the precision offered by RRAM today. While this downside may seem crippling, many machine learning algorithms can tolerate low precision, yielding opportunities for substantially lower power consumption and higher speed computations [49]. Therefore, limited precision may be a non-ideal characteristic, but it is more of a controlled sacrifice of resolution in favor of minimized energy usage.

1.2.4 Processor Architectures

The processor design process, enhanced with LowPy's various features, can be found in Figure 11. LowPy primarily serves as Keras [15] or any other machine learning library would, but augmented with support for non-ideal characteristics often found in various forms of neuromorphic volatile and non-volatile memory. It provides a chip designer with useful insights into how any machine learning model would behave when subjected to these non-ideal characteristics, during the initial design phase, as well as the simulation and qualification phases. These neuromorphic machine learning processors are classified based on their inferencing and training capabilities, as well as their ability to receive new weights into their memory.

1.2.4.1 Inferencing Only

The first processor configuration simulated in this work with LowPy is the inferencing only neuromorphic processor. Sometime after fabrication and before chip distribution, the memristive cells within the chip are assigned their weight values. For various reasons, ranging from simple design constraints to complex security requirements, the inferencing only processor cannot be written to again by the end user. Therefore, the weights are trained on an external processing unit, and then transferred to the inferencing only processor.

1.2.4.2 Online Learning

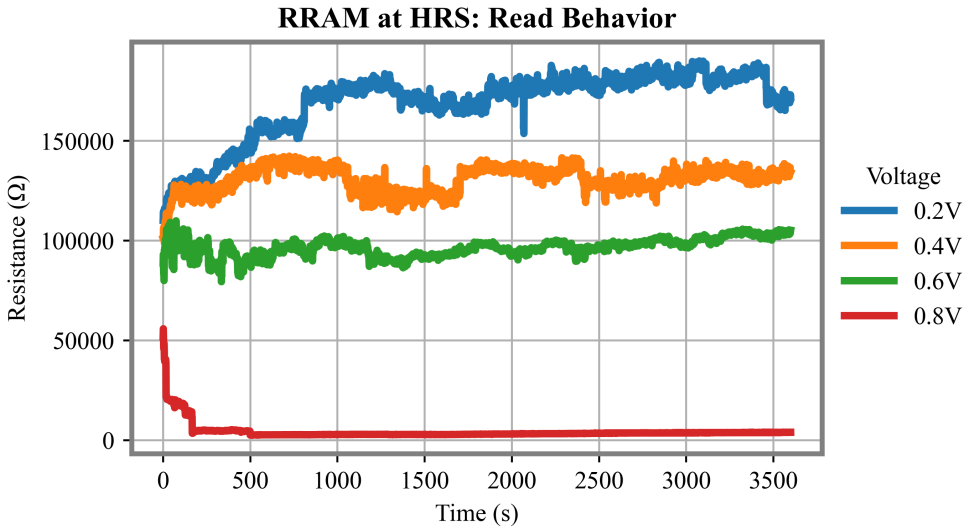
Neuromorphic processors capable of adjusting their weights and fitting to data are considered online learning processors. These processors might use STM

Approved for public release; distribution is unlimited.

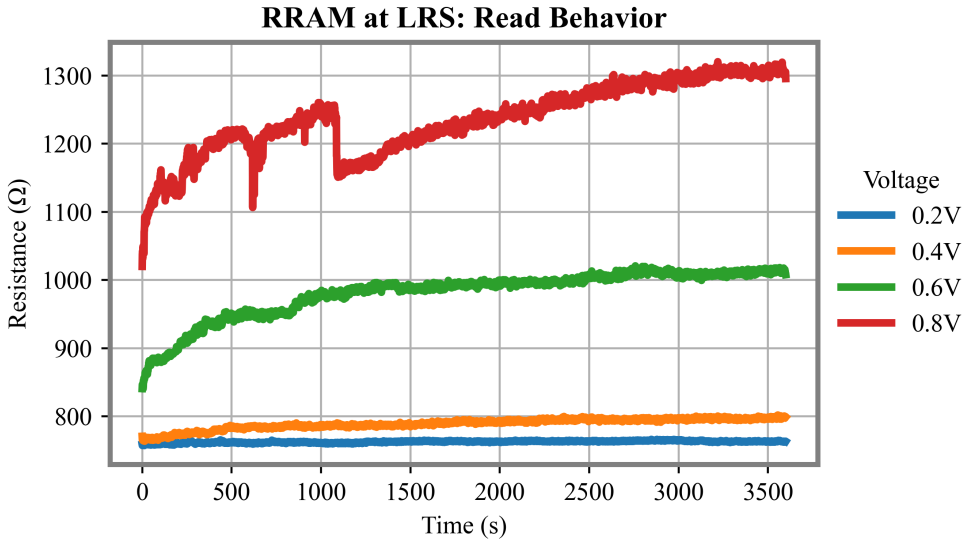
RRAM for training and LTM for inferencing [29], since training iterations are likely occurring frequently, and non-ideal characteristics likely do not have time to impact the weight values substantially.

1.2.4.3 Transfer Learning Processors which do not train on-chip, but instead depend on an external source to provide them with properly tuned weights are considered transfer learning processors. This is the same as an inferencing only chip, but now includes the circuitry and provides the user with the means to write new values to the memristive array. Not only are these employed in the neuromorphic field, but some of the larger datasets trained on a fleet of GPUs can have their weights exported, such that limited or no training is needed by another machine learning processor, be it a GPU, CPU, or neuromorphic.

Approved for public release; distribution is unlimited.



(a) HRS



(b) LRS

Figure 9

Figure 9a shows the resistance over time of RRAM cells set to their HRS state as read by different input voltages. Each trial was just under one hour in duration, and finds that the majority of drift occurs early on. Similarly, Figure 9b shows the resistance over time of RRAM cells set to their LRS state, and being read by different input voltages, with most of the drift occurring early on.

Approved for public release; distribution is unlimited.

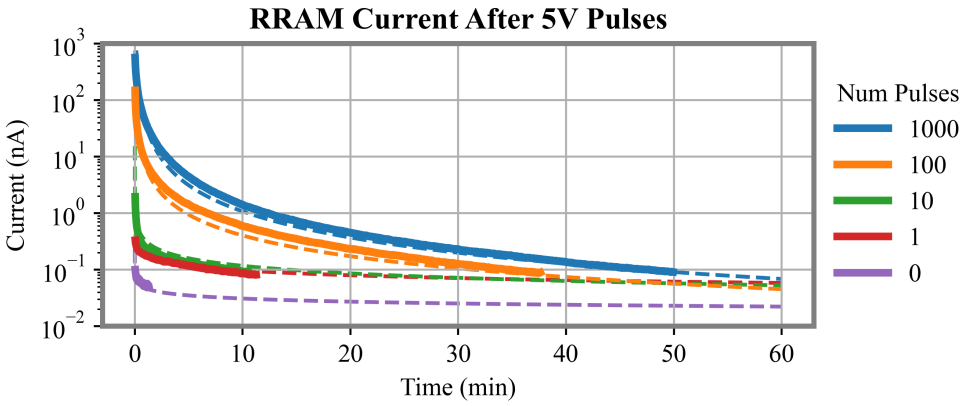


Figure 10

Experimentally gathered data showing the decay occurring within an MSRRAM cell over time. The cell is potentiated via a varied number of 5V pulses, and then read over time, showing the current output falling rapidly at first, and then more slowly as it approaches 0.1A, or its non-conducting, HRS.

Approved for public release; distribution is unlimited.

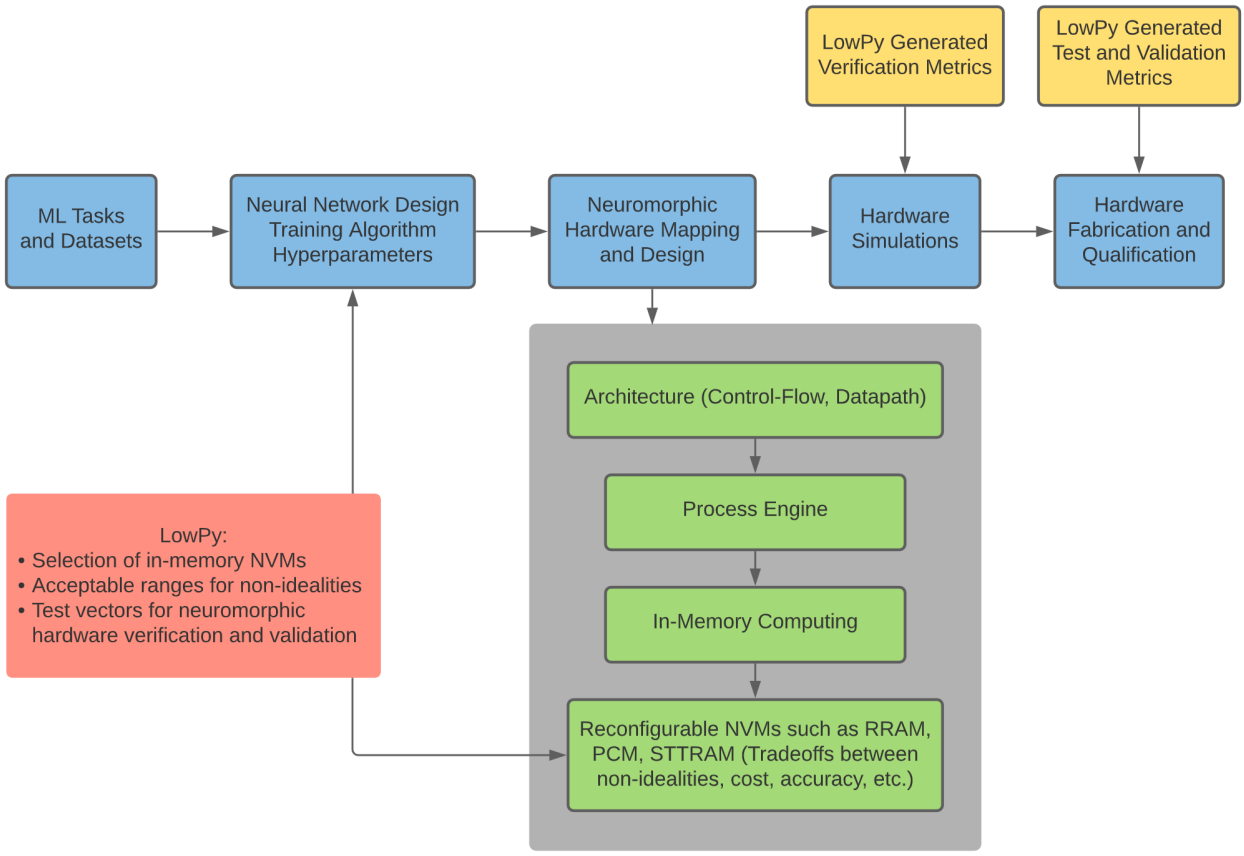


Figure 11

General neuromorphic processor design process, as augmented by LowPy's features. After determining what the machine learning processor will do, as well as the dataset to train on, the network design and hyperparameter tuning can be done with LowPy's insights taken into account. Within the neuromorphic hardware mapping and design, the reconfigurable nonvolatile memory can be chosen, based on the non-idealities accompanying each memory type. Hardware simulations can be augmented with the LowPy generated verification metrics, and then tested and validated during the fabrication and qualification phases.

Approved for public release; distribution is unlimited.

Chapter 2

LowPy

When realizing a machine learning algorithm in hardware, the design process behind neuromorphic, Application Specific Integrated Circuit (ASIC), RRAM-based processors can be quite involved. From a time and money invested standpoint, there is a lot to lose should an error be made during the investigation phase, which involves choosing a type of memory to store the weights, a type of algorithm to implement, and all of the hyperparameters that go into creating a neural network that provides sufficient results. LowPy aims to provide the user with an insight into what exactly these machine learning algorithms can tolerate, in terms of nonidealities accompanying the different memory devices available. Additionally, the specifications obtained from LowPy can be used for verification and validation of RRAM devices when designing a neuromorphic architecture.

2.1 Introduction

As mentioned previously, TensorFlow [13] provides optimized and well documented linear algebraic and data handling functions intended for use with machine learning. The Keras API [15] takes full advantage of the functionality provided by TensorFlow, and comes backed by all of the power with TensorFlow boiled down into simple, readable functions in the popular high level coding language Python.

Rather than recreate what has already been done, LowPy is designed to function as a wrapper around the Keras API, offering its end users lower level access into the objects contained within Keras for neuromorphic processor investigation.

2.1.1 Intended Audience

While LowPy contains functions predisposed for neuromorphic research and development, the functions are not designed in such a way that they are only useful for RRAM and no other form

Approved for public release; distribution is unlimited.

of memory. Rather, functions like `sas()` and `quantize()` can model SAS or mixed precision behavior for any form of memory that these attributes apply to, respectively.

2.2 Workflow

LowPy requires the use of Keras' [15] functional API, as opposed to sequential. As seen in Figure 12, the different training and testing phases are prepended and appended with `lowpy function vectors`. Regardless of weight initialization type, layer type, optimizer, loss function, or any other hyperparameters, LowPy simply watches the chosen weights of the machine learning network and alters them as the user specifies within these `lowpy function vectors`. For instance, the `noise()` function can be used to apply noise to the weights during the `pre train forward propagation` function vector, in order to simulate RTN affected weights before forward propagation. In order to prepare for the next weight read operation, the noise might be removed with `denoise()` during `post train forward propagation`, and then subsequently reapplied, among other nonidealities before gradients are calculated.

The advantage of the `lowpy function vector` approach is that the user can specify the order and timing of any non-ideal function they choose. Some RRAM architectures may rely on nonidealities between gradient calculation and application, while others may not. If no non-ideal functions are added to the `lowpy function vectors`, then the vector is skipped, and performance is not impacted.

2.2.1 Compatibility and Installation

LowPy has been tested on Windows 10, Mac OS Big Sur, and Ubuntu on Python3. To use with a GPU, TensorFlow [13] requires certain versions of Python3 and [14], and this was found to make a big difference with larger networks like the CNN and LSTM. It will be available as open source software, installable from the Python Package Index with `pip`.

```
pip install lowpy
# or
```


Approved for public release; distribution is unlimited.

```
pip3 install lowpy
```

2.2.2 Setup and Usage

As with a Keras model, begin with importing the required packages. TensorFlow [13] contains the Keras API [15], and NumPy [50] will be used to handle datasets and matrices.

```
import tensorflow as tf
import numpy as np
import lowpy as lp
# imported required packages
```

The next step is to import a dataset, and preprocess as needed.

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train = np.reshape(x_train, (-1, 784)) / 255
x_test  = np.reshape(x_test, (-1, 784)) / 255
# imported MNIST dataset and normalized digits
```

Models are defined using Keras' functional API. Adding "LowPy" to the layer name subjects it to the non-ideal characteristic functions. Without "LowPy" in the name, the layer will be overlooked when non-ideal functions are applied to the network.

```
inputs = tf.keras.Input(shape=(784,), name="digits")
hidden = tf.keras.layers.Dense(533, activation="sigmoid", name="LowPy:hidden")(inputs
)
outputs = tf.keras.layers.Dense(10, activation="softmax", name="LowPy:predictions")(
hidden)
# built a MLP using the Keras functional API
```

Choose any optimizer and loss function, and then compile with preferred metrics.

```
model = tf.keras.Model(inputs=inputs, outputs=outputs)
optimizer = tf.keras.optimizers.Adam()
loss_function = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer,loss_function,metrics=[tf.keras.metrics.
SparseCategoricalAccuracy()])
```


Approved for public release; distribution is unlimited.

```
# Created a model and compiled with a chosen optimizer, loss function, and metric
```

By default, all hyperparameters will be defined such that they do not interfere with the model. The currently supported hyperparameters are shown below.

```
history = lp.metrics()
simulator = lp.wrapper(
    history,
    variability_stdev=0.1,
    decay=1,
    precision=0,
    upper_bound=0.1,
    lower_bound=-0.1,
    percent_stuck_at_lower_bound=0.1,
    percent_stuck_at_zero=0.2,
    percent_stuck_at_upper_bound=0.1,
    rtn_stdev=0
)
```

To add non-ideal functions to the `lowpy function vectors` shown in Figure 12, define them in the simulation wrapper.

```
simulator.post_initialization = [
  simulator.initialization_variability,
  simulator.initialize_stuck_at_state_matrices
]
simulator.pre_train_forward_propagation = [
  simulator.apply_sas
]
simulator.post_gradient_application = [
  simulator.apply_write_variability
]
simulator.pre_evaluation = [
  simulator.apply_write_variability
]
```


Approved for public release; distribution is unlimited.

```
simulator.pre_test_forward_propagation = [
  simulator.apply_sas
]
```

Finish by wrapping the model with the LowPy wrapper.

```
simulator.wrap(model,optimizer,loss_function)
simulator.plot(variability_stdev)
```

After model compilation and wrapping, fit and evaluate the data, optionally, requiring the use of a GPU.

```
with tf.device('/GPU:0'):
    simulator.fit(x_test, y_test, epochs, x_train, y_train)
```

Running this script will run a MLP on the MNIST dataset with a 2T2R synaptic cell configuration, 40% of cells SAS, and write variability with a standard deviation of 0.1. A `for` loop after the `history=lp.metrics()` definition will allow the user to specify multiple variants of the network, and append them to the existing metrics.

2.2.3 Data Handling

All metrics tracked by LowPy are housed in the `lp.metrics()`. Currently, it tracks loss, accuracy, and optionally, weight updates for all cells for endurance related tracking. All metrics are output into their own CSV files, such that a script cancellation will not cause the loss of all progress, and so an end user may plot results with their own software and styles.

2.2.4 Plots

LowPy does include a plot function that creates plots in the format shown in this thesis. It is fairly likely that researchers will have their own plotting styles and software, so more emphasis was put on the CSV file organization than offering multiple styles and plotting methods.

Approved for public release; distribution is unlimited.

2.3 Supported Non-Ideal Functions

The non-ideal characteristics mentioned previously in the Neuromorphic Computing section are all included with LowPy. It is possible to create custom user defined functions, and simply use LowPy for access into the inner workings of Keras. However, if a non-ideal function does not exist, and it should be included, an issue can be raised on LowPy's GitHub page.

2.3.1 Write Variability

Noticeable discrepancy can be observed between the requested value when writing to a memristive device like RRAM, and the actual written value [43]. This nonideality is modeled easily using TensorFlow[13], and is shown in Algorithm 1.

Algorithm 1 writeVariability()

1: **for** `w` in `model.trainableWeights` **do**
2: `variedWeights` = tf.random.normal(`w`,$\sigma_{variability}$)
3: w.assign(`variedWeights`)
4: **end for**

The bulk of the work will be searching documentation to try and find a standard deviation value that matches the RRAM in terms of materials that is being implemented, since there are varying degrees of device variability. In simulation, it is tuned via $\sigma_{variability}$, affecting that standard deviation of device variability. This is illustrated in Figure 8, showing how the range of weights changes as $\sigma_{variability}$ increases.

2.3.2 Drift

Depnding on several factors, such as the type of memristor, the temperature, etc., device drift can manifest in a multitude of ways. For instance, the drift of SBRRAM could behave as seen in Fig. Figure 13. 1T1R cells may drift towards their HCS, or their LCS, shown in Figure 13

Approved for public release; distribution is unlimited.

by the upper or lower bound drift regions, respectively. If 2-state RRAM were used to store a neural network's weight, it may drift towards the upper and lower bounds, or towards zero. This is implemented in code in Algorithm 2.

Algorithm 2 drift()

1: **for** w in `model.trainableWeights` **do**
2: `driftWeights`$= w \cdot (1 - driftRateToZero)$
3: `driftWeights`$= w + tf.sign * ((upperBound - tf.abs(w)) * driftRateToBounds)$
4: `driftWeights`$= (w - upperBound) \cdot (upperBound - driftRateToUpper) + upperBound$
5: `driftWeights`$= (w + upperBound) * (1 - driftRateToLower) - upperBound$
6: w.assign(`driftWeights`)
7: **end for**

MSRRAM is signifcantly more complex, and may drift towards intermediate states, or experience a number of different behaviors.

2.3.3 Time Decay

Experimentally gathered data shown in Figure 10 illustrates the decay that occurs over time in the more volatile forms of SBRRAM. Simplistically, this is modeled in Algorithm 3.

Algorithm 3 decay()

1: **for** w in `model.trainableWeights` **do**
2: `decayedWeights` = tf.multiply(w,$decay$)
3: w.assign(`decayedWeights`)
4: **end for**

The thought behind this more simplistic approach of decay modeling is that the training time for a devoted neuromorphic processor should be fairly quick, and the high frequency of writes

Approved for public release; distribution is unlimited.

indicates that the devices likely won't have much time to decay. The decay constant in Algorithm 3 decays the weights towards 0 over the duration of training. As shown in Figure 14, the decay is altered such that a weight value of 1 would decay with varying rates over 1 epoch of training with MNIST [8].

2.3.4 Stuck-At-State

RRAM can be Stuck-At-State (SAS) in many different ways. MSRRAM may get stuck at intermediate states, short and conduct freely, or stay in their HRS. Algorithm 4 offers different options for both 1T1R [51] and 2T2R [52] synaptic configurations. In Algorithm 4, the matrices tracking which cells are Stuck At Upper Bound are in `SAUB`, and the cells Not Stuck At Upper Bound are in `NSAUB`. This process repeats for upper bound, as well as zero.

Algorithm 4 SAS()

1: **for** `w` in range(`model.trainableWeights`) **do**
2: $\texttt{NSALB} = tf.math.round((\texttt{SALB[w]} - 1) * -1)$
3: $\texttt{NSAZ} = tf.math.round((\texttt{SAZ[w]} - 1) * -1)$
4: $\texttt{NSAUB} = tf.math.round((\texttt{SAUB[w]} - 1) * -1)$
5: $\texttt{bounds} = (\texttt{lowerBound} * \texttt{SALB[w]}) + (\texttt{upperBound} \cdot \texttt{SAUB[w]})$
6: $\texttt{w}.assign(\texttt{bounds} + \texttt{weights[w]} \cdot \texttt{NSALB} \cdot \texttt{NSAZ} \cdot \texttt{NSAUB})$
7: **end for**

When these cells are stuck at the upper bound, lower bound, or zero, they can still be read from, and thus gradients will still be calculated. This function should be called before gradient calculation, and after gradient application to sufficiently model SAS behavior. There is another function, `initialize sas()` that allows LowPy to track which cells are SAS, so that the cells stuck at zero, or their bounds, can be remembered throughout training and/or testing. After initializing with this function, LowPy will only apply SAS behavior to the requested cells.

Approved for public release; distribution is unlimited.

2.3.5 *Random Telegraph Noise*

As with many CMOS devices, scalability with RRAM offers a host of issues, including increasing amounts of RTN [47]. In simulation, RTN is controlled by a standard deviation value, σ_{RTN},similar to write variability in Figure 8. The major difference between write variability and RTN is the impact of subsequent applications of noise. In simulation, write variability occurs immediately following a device write, while RTN occurs prior to a device read. For example, take a weight of $w = 0.5$. Application of RTN may raise the weight to $w = 0.51$. With the actual written weight still at $w = 0.5$, this original weight value should be retained, and then RTN can be removed before the next read operation. Algorithm 5 shows RTN applicaiton, and Algorithm 6 shows removal.

Algorithm 5 applyRTN()

1: `preRTN` $=$ `model.trainableWeights`
2: **for** `w` in `model.trainableWeights` **do**
3: `noisyWeights` $= tf.random.normal(w.shape, mean = w, stddev = sigmaRTN)$
4: `w`$.assign($`noisyWeights`$)$
5: **end for**

Algorithm 6 removeRTN()

1: **for** `w` in `model.trainableWeights` **do**
2: `w`$.assign(preRTN)$
3: **end for**

2.3.6 *Limited Precision*

Multi-state RRAM [36] offers a designer lower precision memory storage in favor of faster computation and substantially lower power consumption. Referred to as mixed precision computing [49], significant power is saved during operations like MAC in (Equation 1), which is visualized in Figure 7 as a RRAM crossbar array. In concept, this is no different than switching from a

Approved for public release; distribution is unlimited.

variable type like `double` to `float` to `half`. Algorithm 7 demonstrates how this is implemented in LowPy.

Algorithm 7 precision()

1: **for** `w` in `model.trainableWeights` **do**
2: $\texttt{one} = \texttt{w} + abs(\texttt{lowerBound})$
3: $\texttt{two} = \texttt{one} \cdot \texttt{numStates}/\texttt{range}$
4: $\texttt{three} = tf.clip\_by\_value(\texttt{two}, clip\_value\_min = 0, clip\_value\_max = \texttt{numStates})$
5: $\texttt{four} = tf.round(\texttt{three})$
6: $\texttt{five} = \texttt{four}/(\texttt{numStates}/\texttt{range})$
7: $\texttt{six} = \texttt{five} - abs(\texttt{lowerBound})$
8: $\texttt{w}.assign(\texttt{six})$
9: **end for**

This is a parallelized implementation of hardware quantization, in software. The weights are multiplied by the specified number of states and divided by the range between the upper and lower bounds. After clipping the weights between the upper and lower bounds, the weights are rounded to their new states. Finally, the weights are normalized, and shifted back down to their original range. This function allows a user to specify a number of states, and requires an upper and lower bound. These upper and lower bound values can be considered a hyperparameter on their own, since they cause clipping of weights, and the closer they are together, the higher the resolution of the weights.

Approved for public release; distribution is unlimited.

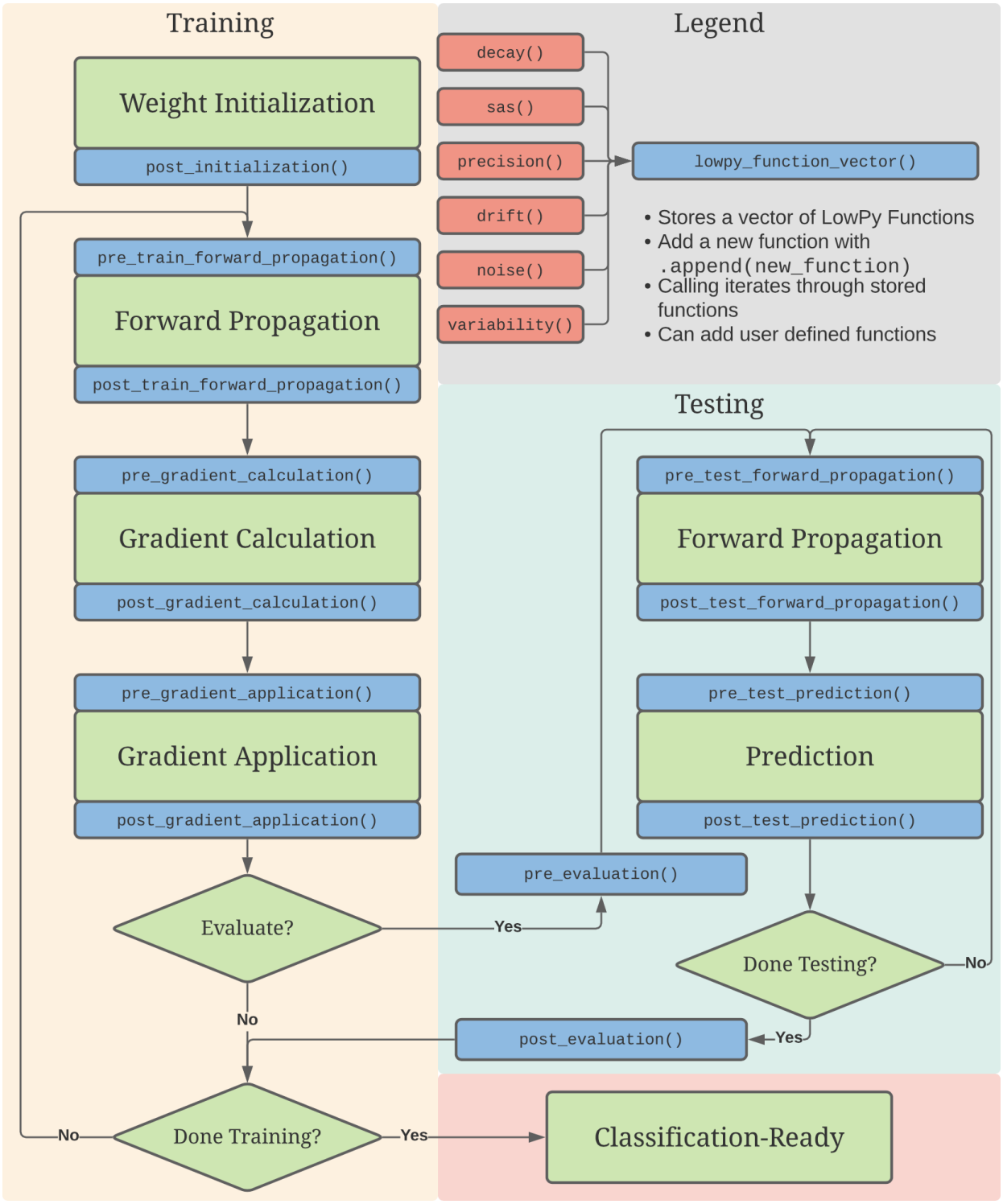


Figure 12

Design flow of LowPy.

47
Approved for public release; distribution is unlimited.

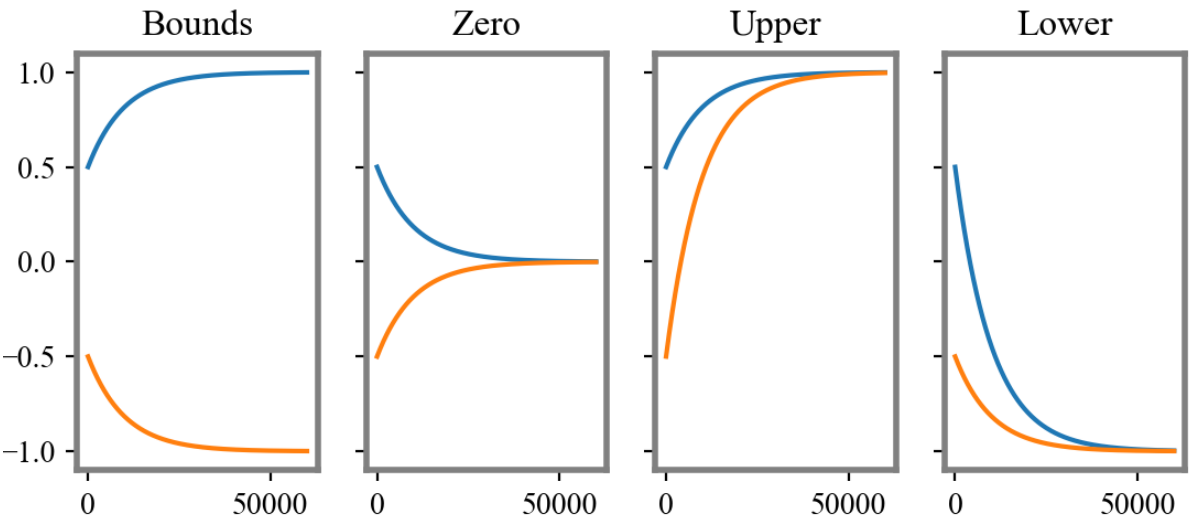


Figure 13

Various forms of device drift found particularly in memristive devices. Drifting towards bounds involves the weights moving towards the HRS and LRS of RRAM. Zero is somewhat ”opposite”, where it drifts towards zero. Upper bound drift involves all weights drifting towards the state representative of the upper bound, with drift to lower bound being the opposite.

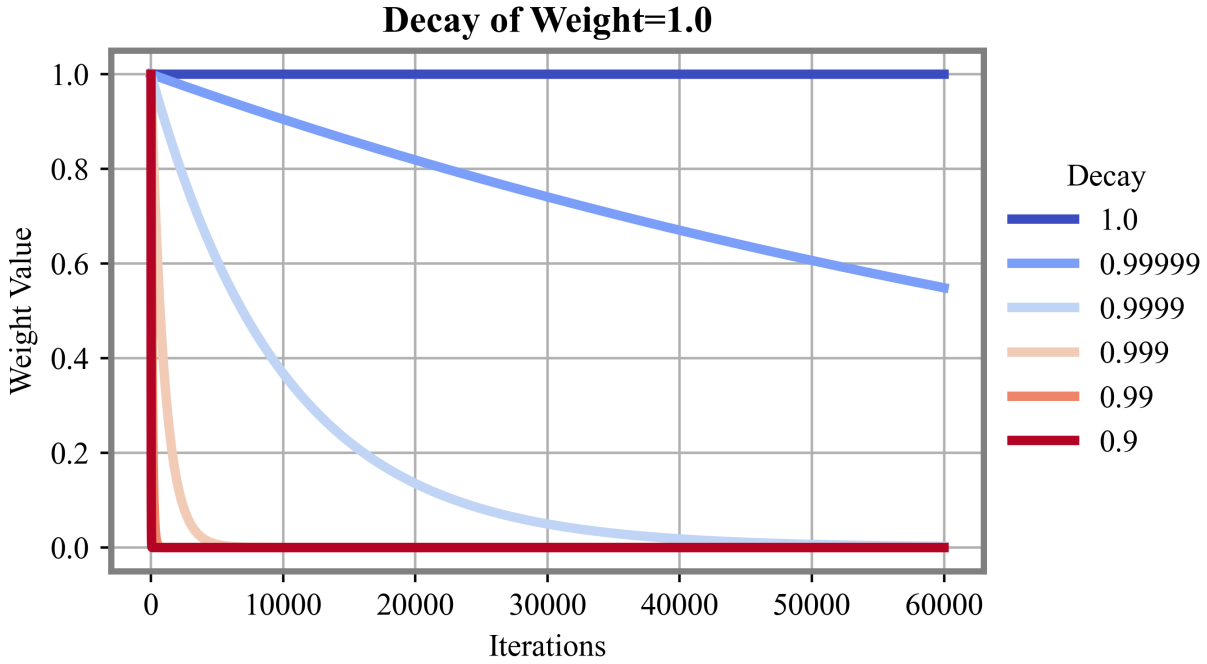


Figure 14

Illustration of memristive device time decay over 60,000 iterations with a varied decay constant. 1.0 = no decay.

Approved for public release; distribution is unlimited.

Chapter 3

Inferencing-Only Neuromorphic Processor Simulation

These next three chapters demonstrate the simulation capabilities of LowPy in terms of possible combinations of processor, network, and nonideality types. As mentioned earlier, four different networks were chosen to offer a reasonable coverage for various network architectures and dataset types, also determined based on what is popularly implemented in current neuromorphic research.

3.1 Simulation Hardware

To provide information on the expected performance times of LowPy, a few different PC configurations were used to enhance simulation throughput, and verify that LowPy code worked on the three most common operating systems:

- Windows 10 PC with Intel®Core™ i5 4690k 3.50GHz CPU, 16GB RAM, Nvidia RTX 3090 24GB GPU
- Mac OS Big Sur with Intel®Core™ i9 8-Core CPU, 16GB RAM, GPU not used
- Linux PC with Intel®Core™ i9-9900K 3.60GHz CPU, 16GB RAM, Nvidia RTX 2080 Ti 11GB GPU

The Nvidia GPUs were significantly faster than all CPUs, and the Linux PC's CPU was fastest for all other networks. The Macbook's CPU was faster than the Windows PC's, possibly due to the extra cores available, or better turbo boosting up to 4.9 GHz. Simulation times varied significantly depending on which nonidealities being simulated, but always ended up taking longer than the unmodified Keras layers. However, the SLP usually took about 20 minutes to 1 hour, MLP taking between 30 minutes and 2 hours, the CNN taking up to 3 hours, and the LSTM sometimes taking

49
Approved for public release; distribution is unlimited.

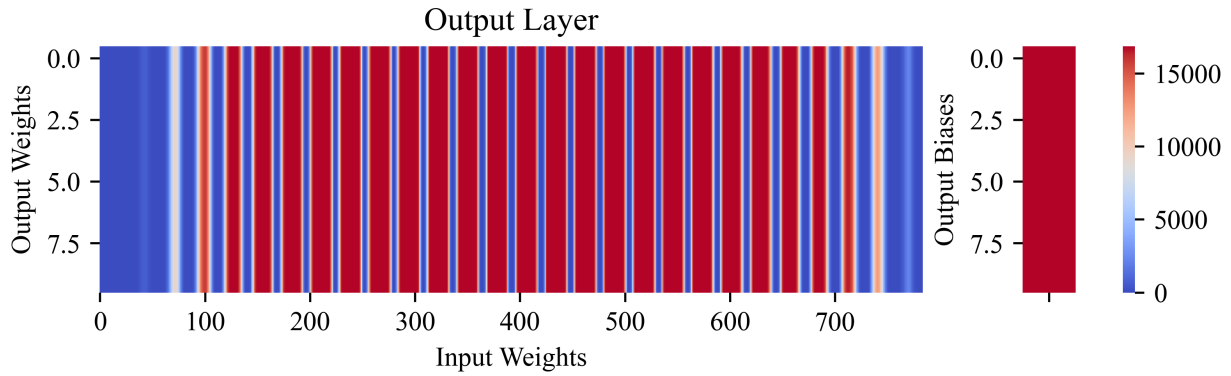


Figure 15

Weight update visualization of an SLP trained on MNIST. Red corresponds to the upper limit of over 15,000 updates over the course of training, while blue signifies 0 updates.

up to 7 hours to simulate. However, the 2080 Ti was able to handle up to 2 LSTMs at a time, and probably could run more instances with more RAM, or better memory utilization from LowPy.

3.2 Single Layer Perceptron

The SLP, trained on MNIST [8], was trained for 10 epochs, where the 60,000 inputs provided were split 90% for training, 10% for testing. There are 7,840 trainable parameters. The Keras [15] model is shown below:

```
inputs = tf.keras.Input(shape=(784,), name="digits")
outputs = tf.keras.layers.Dense(10, name="LowPy-predictions")(inputs)
```

It was subjected to six different nonidealities.

The number of weight updates that occurred over 10 epochs of MNIST [8] training in batches of 32 inputs are visualized in Figure 15. Noteably, the cells which received the most updates were likely connected to neurons in the center inputs. This makes sense, since as seen in Figure 1, the data from the digits is largely centralized in each picture. Some perimeter cells never received an update, while other central cells received an update with each iteration.

Approved for public release; distribution is unlimited.

3.3 Multi-Layer Perceptron

The MLP, trained on MNIST [8], was also trained for 10 epochs, where the 60,000 inputs provided were split 90% for training, 10% for testing. The hidden layer contains $784 \cdot 533 + 533 = 418,405$ trainable weights, while the output layer contains $533 * 10 + 10 = 5340$ trainable weights. The Keras [15] model is shown below.

```
inputs  = tf.keras.Input(shape=(784,), name="digits")
d1      = tf.keras.layers.Dense(533, activation="sigmoid", name="lowpy-hidden")(
inputs)
outputs = tf.keras.layers.Dense(10, activation="softmax", name="lowpy-predictions")(
d1)
```

The number of weight updates that occurred over 10 epochs of MNIST [8] training in batches of 32 inputs are visualized in Figure 16. In contrast to the SLP, the output layer was almost entirely updated with each iteration, while the hidden layer was not updated as frequently. Similarly to the SLP, the biases saw a substantial amount of updates, and the hidden layer shows the same lack of perimeter cell updates that the SLP outer layer does in Figure 15.

3.4 Convolutional Neural Network

The CNN, trained on MNIST [8], was also trained for 10 epochs, where the 60,000 inputs provided were split 90% for training, 10% for testing. The number of weight updates that occurred over 10 epochs of MNIST [8] training in batches of 32 inputs are visualized in Figure 17. Interestingly, the filters are adapted fairly regularly in both convolutional layers to better classify the data. It should be noted that the filters are flattened in Figure 17, so visualization is more complicated. The output layer sees a more sporadic update of weights, where there is not any easily discernible pattern occurring, unlike the SLP and MLP. There are $3 * 3 * 32 = 288$ weights in the first convolutional layer, $288 * 64 = 18,432$ weights in the second convolutional layer, and $1843 * 10 = 18,430$ weights in the final output layer. In total, this sums to approximately $33,000$ trainable parameters. The Keras [15] model is shown below:

Approved for public release; distribution is unlimited.

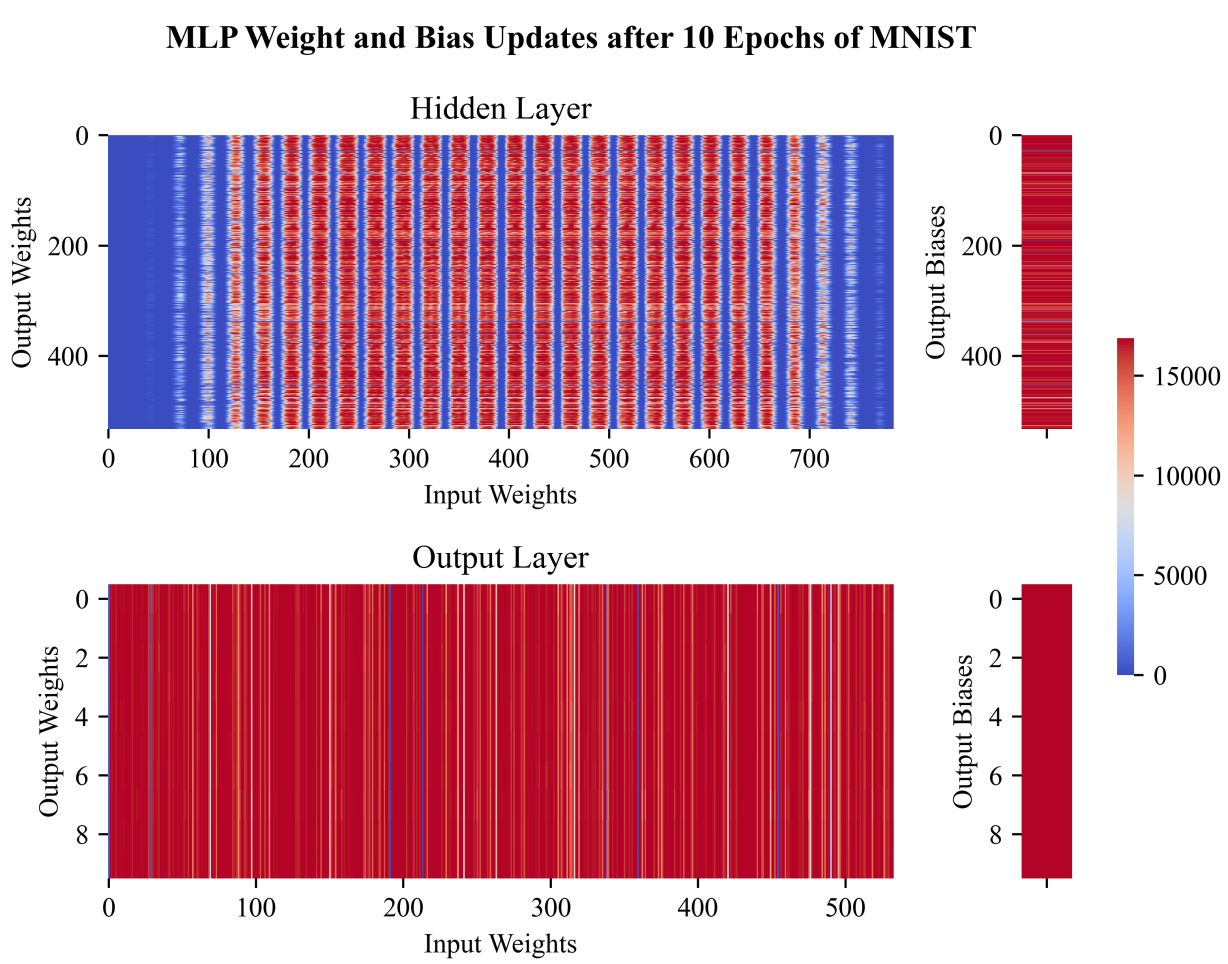


Figure 16

Weight update visualization of an MLP trained on MNIST. Red corresponds to the upper limit of over 15,000 updates over the course of training, while blue signifies 0 updates.

```
inputs = tf.keras.Input(shape=(28,28,1), name="digits")
c1 = tf.keras.layers.Conv2D(32, kernel_size=(3, 3), activation="relu")(inputs)
p1 = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(c1)
c2 = tf.keras.layers.Conv2D(64, kernel_size=(3, 3), activation="relu")(p1)
p2 = tf.keras.layers.MaxPooling2D(pool_size=(2, 2))(c2)
f1 = tf.keras.layers.Flatten()(p2)
d1 = tf.keras.layers.Dropout(0.5)(f1)
outputs = tf.keras.layers.Dense(10, activation="softmax", name="lowpy-output")(d1)
```


Approved for public release; distribution is unlimited.

3.5 Long Short-Term Memory Network

The LSTM, trained on IMDB [12], was trained for 5 epochs, where the 25,000 inputs provided first reduced to 2,500, and then split 90% for training, 10% for testing. The Keras [15] model is shown below:

```
inputs = tf.keras.Input(shape=(None,), dtype="int32")
emb = tf.keras.layers.Embedding(max_features, 128)(inputs)
bid1 = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True),
name="LowPy-bidirectional1")(emb)
bid2 = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64),name="LowPy-
bidirectional2")(bid1)
outputs = tf.keras.layers.Dense(2, activation="sigmoid",name="LowPy-output")(bid2)
```

The reasoning behind the reduction in this test is that tracking every weight update is very computationally expensive, and even with this reduction, the LSTM still took a full hour on an NVIDIA RTX 3090 GPU (though it may have been bottlenecked slightly by an Intel i5 4690K CPU).

The number of weight updates that occurred over 5 epochs of IMDB [12] training in batches of 16 inputs are visualized in Figure 18. There were over 170,000 trainable parameters in this 2-layer bidirectional LSTM. While the MLP had substantially more trainable paremeters, both the CNN and LSTM took much longer to process, due to the more complex operations involved in forward and backpropagating these networks. Almost every weight was updated during the LSTM training process, meaning that an RRAM realization of this particular network may wear out more of its cells sooner than other network types.

3.6 Write Variability

The inferencing processor was subjected to the write variability nonideality outlined in Algorithm 1. Since this was inferencing only, it was applied during the $pre\_evaluation()$ phase for all four networks, trained under the same configuration as mentioned earlier in this chapter. Results are shown in Table 1. Write variability, as controlled by $\sigma_{variability}$, appears to have no impact on

Approved for public release; distribution is unlimited.

Table 1

Impacts of Write Variability on Inferencing-Only Processor

$\sigma_{variability}$	SLP	MLP	CNN	LSTM
0	92.8%	98.0%	98.9%	78.2%
1e-05	92.8%	98.0%	98.9%	78.2%
0.0001	92.8%	98.0%	98.9%	78.2%
0.001	92.8%	98.0%	98.9%	78.2%
0.01	92.7%	97.9%	98.9%	78.0%
0.1	89.0%	97.1%	97.6%	67.1%
1	35.0%	11.8%	28.9%	50.2%

Table 2

Impacts of Drift to Bounds on Inferencing-Only Processor

Drift Rate to Bounds	SLP	MLP	CNN	LSTM
0	92.5%	98.2%	98.7%	76.2%
1e-06	92.5%	98.2%	98.6%	76.2%
1e-05	92.5%	98.2%	98.6%	76.4%
0.0001	92.0%	97.8%	98.4%	76.0%
0.001	74.7%	80.2%	97.3%	68.6%
0.01	69.7%	74.2%	97.1%	60.5%
0.1	69.2%	73.6%	97.1%	59.7%

any network until $\sigma_{variability} \geq 0.01$. After that, any increase in write variability scrambles the weights to the point where performance suffers, across our tested networks and datasets.

3.7 Drift

The inferencing processor was subjected to the drift to bounds nonideality outlined in Algorithm 2. Since this processor is inferencing only, it was applied during the $pre\_inference()$ phase for all four networks, trained under the same configuration as mentioned earlier in this chapter. Results are shown in Table 2. Drift to bounds, as controlled by $drift\_rate\_to\_bounds$, has little

Approved for public release; distribution is unlimited.

Table 3

Impacts of Decay on Inferencing-Only Processor

decay	SLP	MLP	CNN	LSTM
1.0	92.6%	97.6%	98.9%	79.5%
0.99999	92.6%	97.6%	98.9%	79.3%
0.9999	92.6%	96.2%	98.9%	77.8%
0.999	92.6%	27.7%	98.9%	71.0%
0.99	80.9%	11.9%	27.5%	55.0%
0.9	16.6%	10.1%	11.5%	52.5%

impact until ≥ 0.0001. Since this processor only receives weights once, the weights will eventually drift to the bounds, and drop performance to the point which the processor is unusable. And, it is likely that the cells will drift outside of the testing phase over time. Therefore, it may also be useful to monitor tests completed after the weights have drifted a certain percent towards their bounds, or another configuration.

3.8 Time Decay

The inferencing processor was subjected to the decay nonideality outlined in Algorithm 3. Since this processor is inferencing only, it was applied during the $pre\_inference()$ phase for all four networks, trained under the same configuration as mentioned earlier in this chapter. Results are shown in Table 3. Decay, as controlled by $decay$, has its first detrimental performance impact at $decay = 0.9999$, or 99.99% weight value retained per iteration. However, the MLP seemed to suffer the greatest. This is likely due to the hidden layer weights having a tighter standard deviation around zero than other layers, and being scrambled sooner than other layer types. Aside from this observation, similar conclusions can be made, where ensuring that this nonideality lie below a certain threshold will have less performance impact on the network. Similar to drift, this will affect the network performance even when the processor is not being used, so decay based RRAM is not likely to be suitable in an inferencing processor for most applications.

Approved for public release; distribution is unlimited.

Table 4

Impacts of 2T2R Stuck at State on Inferencing-Only Processor

% Cells SAS	SLP	MLP	CNN	LSTM
0%	92.6%	97.9%	98.6%	80.2%
10%	91.9%	97.3%	98.4%	77.1%
20%	89.7%	96.8%	98.2%	72.5%
30%	88.1%	95.7%	97.4%	74.0%
40%	86.2%	91.8%	96.0%	67.8%
50%	81.2%	76.5%	93.1%	70.5%

3.9 Stuck At State

The inferencing processor was subjected to the Stuck At State (SAS) nonideality outlined in Algorithm 4. Since this processor is inferencing only, it was applied during the $pre\_evaluation()$ phase for all four networks, trained under the same configuration as mentioned earlier in this chapter. Results are shown in Table 4. 25% of stuck cells were set to be stuck at the lower bound weight of -0.1, 25% of stuck cells were set to be stuck at the upper bound weight of 0.1, and the remaining 50% of stuck cells were stuck at 0. The SAS simulations show some interesting results, particularly with the LSTM. Given that the LSTM simulations were very light (meaning, they did not train or test on a particularly large set of inputs, and that it was a rather small network), it may be difficult to make generalizations sturdy enough to support behavioral prediction for other forms of LSTMs on other datasets. However, it was still interesting to note that accuracy did not decrease perfectly linearly throughout our tests - 40% of cells SAS yielded less accuracy than 50% of SAS cells. Perhaps this could be due to some of the LSTM weights being more integral to higher performance, and selecting more of those weights during the 40% trial than the 50% trial. With the other networks, it behaves perfectly linearly, dropping as more cells become SAS. However, it becomes apparent across the board that these RRAM processors are fairly tolerant to a high number of cells being SAS.

Approved for public release; distribution is unlimited.

Table 5

Impacts of RTN on Inferencing-Only Neuromorphic Processor

σ_{RTN}	SLP	MLP	CNN	LSTM
0.0	92.4%	98.0%	98.9%	75.7%
0.0001	92.5%	98.0%	98.8%	75.8%
0.001	91.4%	97.8%	97.8%	76.5%
0.01	53.1%	23.2%	32.6%	59.8%
0.1	20.5%	12.4%	9.3%	51.5%
1.0	14.0%	10.9%	6.8%	50.0%

3.10 Random Telegraph Noise

The inferencing processor was subjected to the Random Telegraph Noise (RTN) nonideality outlined in Algorithm 5. Since this processor is inferencing only, it was applied during the $pre\_inference()$ phase and removed during the $post\_inference()$ phase for all four networks, trained under the same configuration as mentioned earlier in this chapter. Results are shown in Table 5. As seen in the LSTM accuracy of Table 5, it is possible for a nonideality to have a positive impact on the performance of a neural network. While a consistent, positive increase in accuracy should not be ignored, it may speak to a poor choice of optimization, loss function, or some other architectural decision or hyperparameter choice - since nonidealities have the affect of altering weights from their intended values. However, it is encouraging to see that even with noise disturbing weight reads of up to 20% standard deviation from their written values, that the network can still perform nearly ideally, and in the case of this particular LSTM, better than without RTN.

3.11 Limited Precision

The inferencing processor was subjected to the limited precision nonideality outlined in Algorithm 7. Since this processor is inferencing only, it was applied during the $pre\_evaluation()$ phase for all four networks, trained under the same configuration as mentioned earlier in this chapter. Results are shown in Table 6. Results for the SLP, MLP, and LSTM in Table 6 begin with

Approved for public release; distribution is unlimited.

Table 6

Impacts of Limited Precision on Inferencing-Only Neuromorphic Processor

Num States	SLP	MLP	CNN	LSTM
1024	75.4%	78.3%	98.6%	69.0%
512	75.4%	78.2%	98.6%	69.0%
256	75.4%	78.2%	98.6%	69.0%
128	75.4%	78.2%	98.6%	69.0%
64	75.4%	78.3%	98.6%	69.0%
32	75.3%	78.3%	98.6%	69.1%
16	75.2%	78.3%	98.6%	68.6%
8	75.5%	78.2%	98.6%	68.7%
4	75.6%	76.5%	98.5%	69.1%
2	75.8%	75.9%	98.5%	69.4%

a lower starting accuracy at 1024, or 10 bits of precision, than the previous trials did. This is likely due to weight clipping. In order to simulate a set number of states, a range of bounds must be set. Meaning, an upper and lower weight must be determined, such that the resolution, or number of intermediate states, between these bounds can be set. Because of this, the bound values were chosen to be $0.1 \leq W \geq -0.1$. The dilemma faced when choosing your upper and lower bounds comes when balancing weight resolution with number of weights clipped. Any weight update that attempts to set the value above the upper bound will be clipped to the upper bound of 0.1, while the same case applies for weights falling below -0.1. These weight bounds should likely be tuned as a hyperparameter when deciding on your network architecture and dataset, and these values of 0.1 and -0.1 seemed to offer a good balance between resolution and clipping for most cases. However, for the SLP, MLP, and LSTM, it would appear that many of the weights were clipped to the bounds when they were written to the inferencing processor, meaning that it would be beneficial in practice to expand the range between the upper and lower bounds. However, this data is still meaningful, as it shows that a decrease in precision still has detrimental impacts on accuracy. The CNN appears not to have suffered much clipping, and benefits from the resolution of the weights offered by these somewhat restrictive bound choices.

Approved for public release; distribution is unlimited.

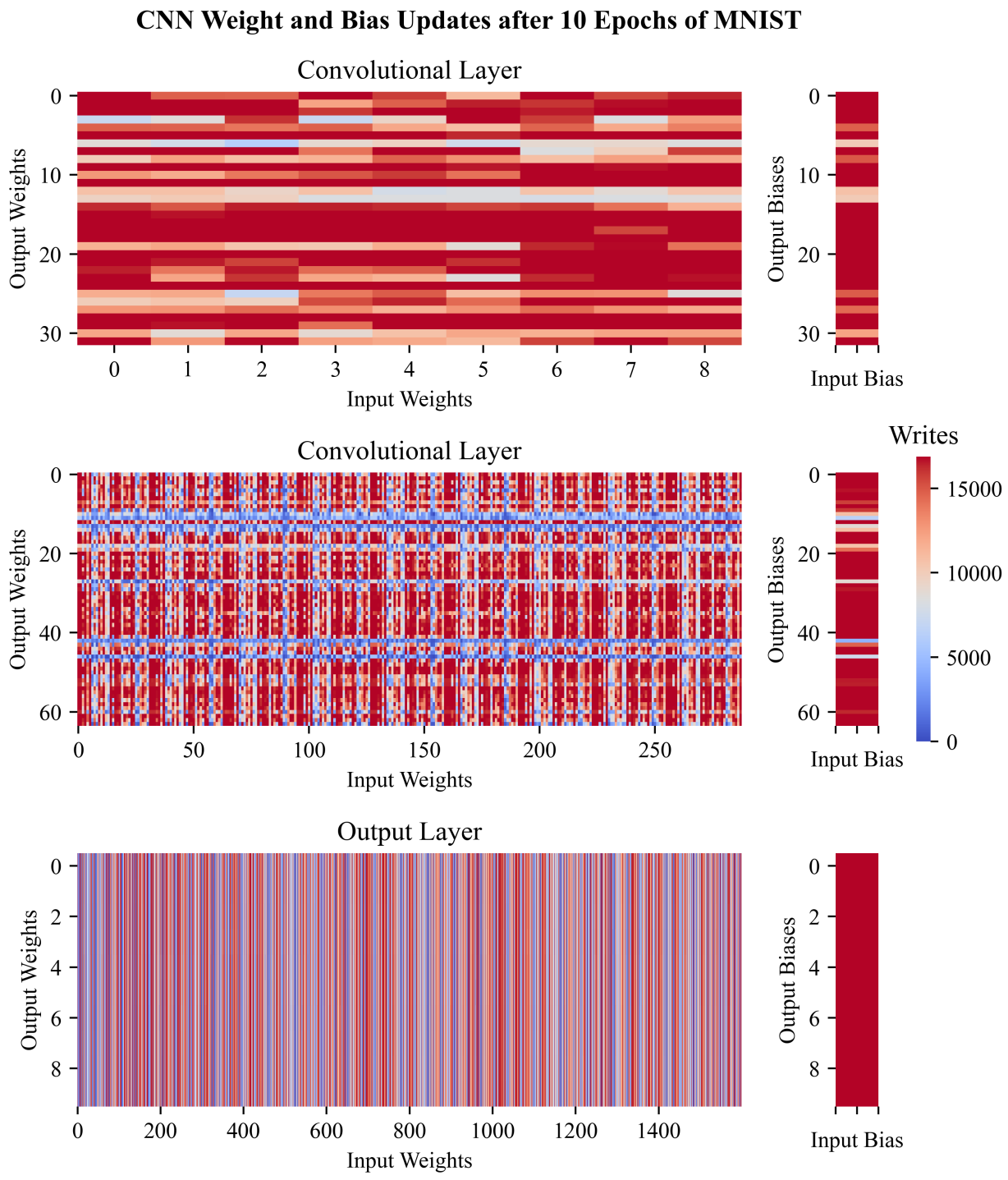


Figure 17

Weight update visualization of an CNN trained on MNIST. Red corresponds to the upper limit of over 15,000 updates over the course of training, while blue signifies 0 updates.

Approved for public release; distribution is unlimited.

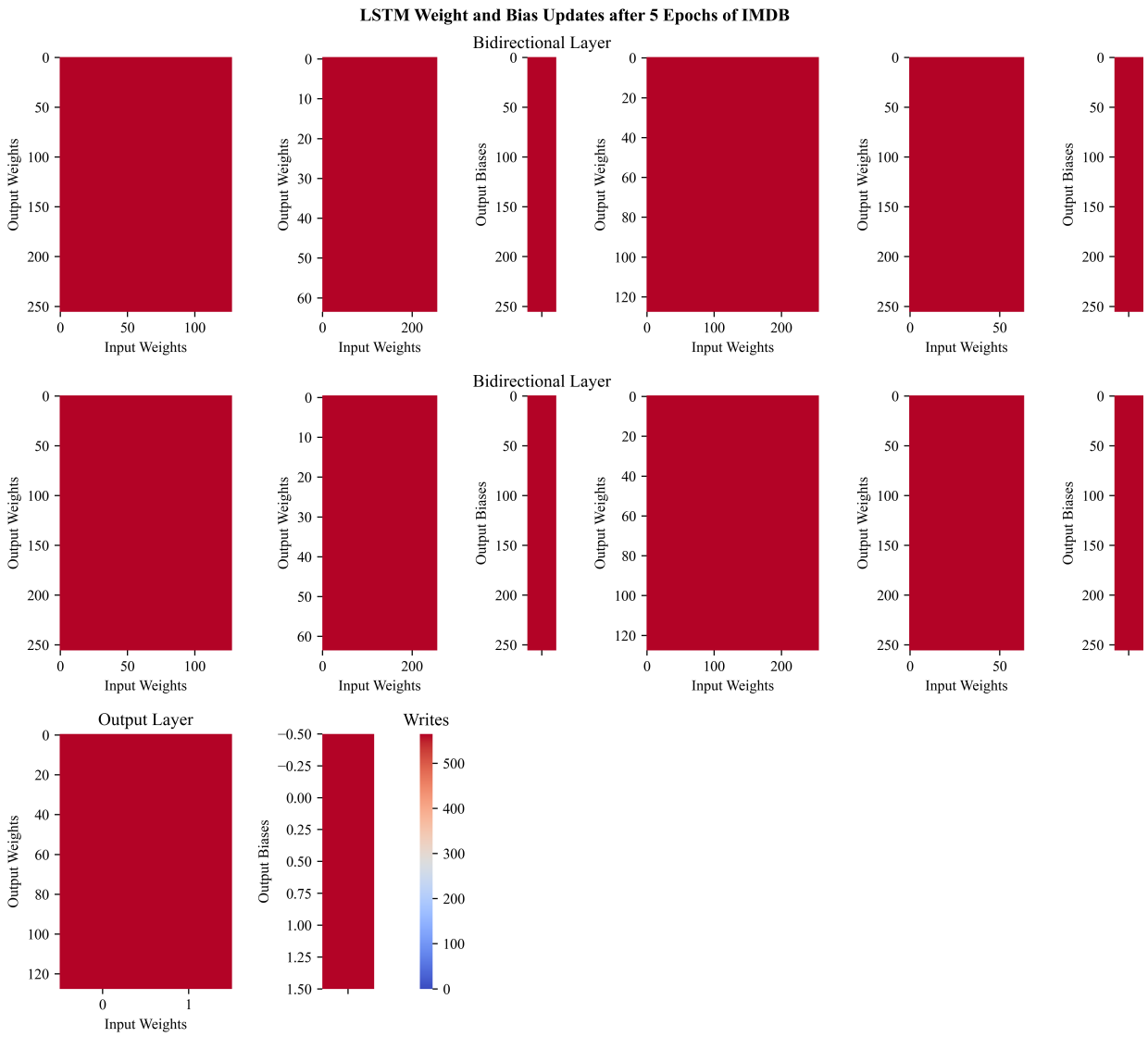


Figure 18

Weight update visualization of an LSTM trained on IMDB. Red corresponds to the upper limit of over 15,000 updates over the course of training, while blue signifies 0 updates.

60
Approved for public release; distribution is unlimited.

Chapter 4

Online Learning Neuromorphic Processor Simulation

Simulations in this work will be split into two chapters: Online Learning, and Transfer Learning processors. As mentioned previously, the online learning neuromorphic processor is capable of updating its own weights to fit the input data, as well as inferencing on it, while the transfer learning processors rely on external sources to train their weights.

Typically with online learning RRAM processors, the weights are stored in RRAM cells that are responsible for both training and inferencing. Therefore, any nonidealities occurring during testing will not be removed when resuming training.

4.1 Write Variability

$\sigma_{variability}$ values were spaced logarithmically from 0 to 1. Write variability is applied at two sections of Figure 12:

```
simulator.post_initialization = [
  simulator.initialization_variability
]
simulator.post_gradient_application = [
  simulator.write_variability
]
```

This configuration works regardless of synaptic configuration, since device variability is normally distributed around the requested weight value for both 1T1R and 2T2R. Results are shown in Figure 19. The SLP in Figure 19a is unaffected by write variability until $\sigma_{variability} = 0.01$. Anything above that causes substantial performance issues. Similar results are found with the MLP in Figure 19b, the CNN in Figure 19c, and the LSTM in Figure 19d. Notably, the LSTM experienced overtraining during $\sigma_{variability} = 0.01$, likely due to exploding gradients, among other factors.

61
Approved for public release; distribution is unlimited.

4.2 Drift

Similar to write variability, drift values were spaced logarithmically from 0 to 0.1. Drift is applied at two sections of Figure 12, occurring prior to any read operation:

```
simulator.pre_train_forward_propagation = [
  simulator.apply_decay
]
simulator.pre_test_forward_propagation = [
  simulator.apply_decay
]
```

The logarithmically varied drift coefficients represent the percentage of drift per iteration, as applied to every weight, shown in Algorithm 2, illustrating that there are several drift configurations, depending on synaptic connection style, as well as the type of RRAM selected. The chosen configuration for simulation in this work was drift towards bounds, which can be characteristic of a 2T2R configuration.

Results are shown in Figure 20. The SLP performed better than the MLP in terms of tolerance to drift, with all networks beginning to suffer performance degradation ≥ 0.001, or 0.1% drift per iteration. The drift to bounds was tolerated reasonably well by the CNN, since the convolutional filters were exempt from this non-ideality. Results for the LSTM were more similar to the SLP and MLP, in that anything ≥ 0.001 was fairly detrimental to performance.

4.3 Time Decay

Similarly to write variability, decay values were spaced logarithmically, this time from 0 down to 0.9. Decay is applied during two sections of Figure 12, which is when the values are read from the inferencing processor:

```
simulator.pre_train_forward_propagation = [
  simulator.apply_decay
]
```


Approved for public release; distribution is unlimited.

```
simulator.pre_test_forward_propagation = [
  simulator.apply_decay
]
```

In this case, time decay was simulated for a 2T2R synaptic connection, since it decays towards 0, implemented in Algorithm 3. A 1T1R synaptic connection might decay towards LCS, which would likely be the lower bound. Results are shown in Figure 21. All networks struggled with decay ≥ 0.99. Most notably, the SLP performed much better than the MLP. This is likely due to the tighter range of the weights in the hidden layer, losing their optimized values well before the outer layer might. Given that this is an ideal CNN with non-ideal classification layers since decaying filters would yield useless results, it makes sense that it did a much better job converging than the MLP. The LSTM was the first network to show signs of struggle, with the third trial of $decay = 0.9999$ reducing the accuracy by a few percent.

4.4 Stuck-At-State Modeling

Over time, memristive cells can fail, sometimes becoming SAS, as mentioned previously. This affects all read operations, since the cells can no longer be written to, and in simulation, will be applied during two sections of Figure 12. The matrices which track which cells are stuck at certain states must also be initialized:

```
simulator.post_initialization = [
    simulator.initialize_sas_matrices
]
simulator.pre_train_forward_propagation = [
  simulator.apply_sas
]
simulator.pre_test_forward_propagation = [
  simulator.apply_sas
]
```

There are many forms of SAS that RRAM synapses can take. MSRRAM may get stuck at some

Approved for public release; distribution is unlimited.

intermediate state, short and conduct freely, or break and stop conducting altogether. Therefore, the most common forms of SAS were included in Algorithm 4, containing different choices for 1T1R [51] and 2T2R [52] synaptic cell configurations. As long as the cells are set to their stuck values in simulation during any read operation, then it can easily be simulated at a high level.

Results are shown in Figure 22. All networks classified well, with both training and testing bits SAS. In these simulations, the percentage of total SAS cells is varied from 0% to 50%. This is a 2T2R configuration, and as such, 50% of the cells are stuck at 0, while the other 50% reside at the bounds. The SLP shows some signs of performance loss as more and more cells become SAS, but still achieves over 91.5% accuracy. All other networks perform mostly unfazed, likely due to the similarities between SAS and machine learning dropout.

4.5 Random Telegraph Noise

σ_{rtn} values were spaced logarithmically from 0 to 1, similarly to write variability. RTN is applied prior to read operations, and subsequently removed, shown in Figure 12:

```
simulator.pre_train_forward_propagation = [
  simulator.apply_rtn
]
simulator.post_gradient_calculation = [
  simulator.remove_rtn
]
simulator.pre_test_forward_propagation = [
  simulator.apply_rtn
]
simulator.post_inference = [
  simulator.remove_rtn
]
```

This configuration works regardless of synaptic configuration, since device variability is normally distributed around the requested weight value for both 1T1R and 2T2R. Results are shown in Figure 23. The impact of noise does not become apparent until a $\sigma_{RTN} \geq 0.01$ has been reached.

Approved for public release; distribution is unlimited.

This applied to all networks, and seemed to affect the CNN the most. The CNN and LSTM were unable to train with too much noise present, while the others appeared to train somewhat normally, given enough iterations.

4.6 Limited Precision

The *numStates* value was spaced in terms of bits, 0 to 10 bits. This shows a good spread of the lowest tolerable precision, and the possible achievable performance as RRAM advances, and new developments present higher precision and write accuracy. Limited precision is applied prior to read operations, shown in Figure 12:

```
simulator.pre_train_forward_propagation = [
  simulator.precision
]
simulator.pre_test_forward_propagation = [
  simulator.precision
]
```

This configuration works regardless of synaptic configuration, since the number of states can be controlled by the user. RRAM cells with 4 bits of precision may have 16 states, but two of them set up in a 2T2R configuration may offer 32 states, 31 states, or a different value, depending on many factors. Results are shown in Figure 24. Notably, the networks still performed with reasonable accuracy when restricted to only 16 states for training and inferencing. The CNN, however, was able to achieved almost 90% accuracy for 1, 2, and 3 bits, which shows that the bulk of the work is done in data preparation by the convolutional filters. Therefore, RRAM may serve as a good low power classification layer succeeding a convolutional layer. The LSTM's highest resolution of 1024 states fell to 60% accuracy at the second epoch. This would seem to be a fluke, but a similar drop occurred for 256 states, meaning there could be something more going on than just unlucky gradient descent.

65
Approved for public release; distribution is unlimited.

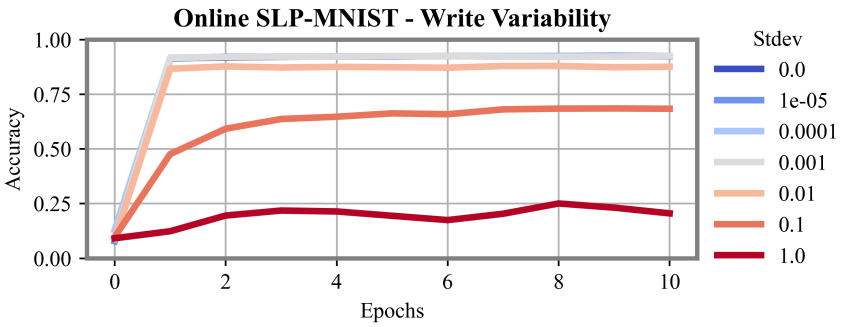
4.7 Discussion

In terms of an online training processor, performance is likely to suffer more due to non-idealities compared to that of an inferencing processor. This is due to the nonidealities having a heavy influence over not only the evaluation phase, but also the testing phase.

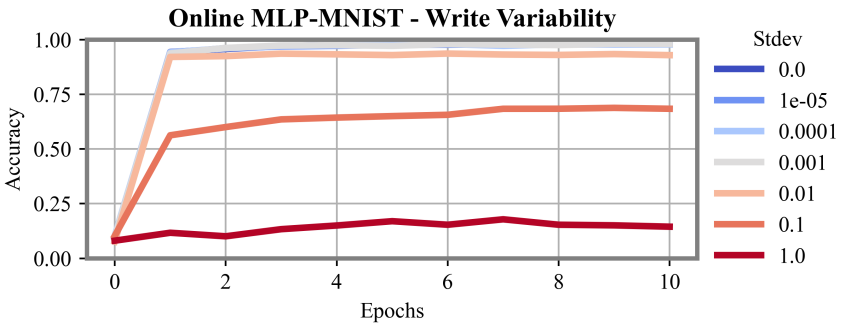
While it is difficult to make broad generalizations for these four different networks across six different nonidealities, it seems that there is an intuitive principle that the further a non-ideal characteristic alienates an algorithm from its original behavior, the more performance tends to suffer. This unfortunately indicates that there is little chance that a non-ideal behavior that has any substantial impact on weight value is unlikely to aid a network in its convergence.

If not boosting the accuracy of a network, then the best case scenario is that the nonideality has no noticeable impact on accuracy. This was found under certain thresholds for all non-ideal functions, with Figure 24c coming to mind in terms of significant tolerance to a non-ideal behavior. All networks seemed to tolerate 2T2R SAS behavior with little issue, shown in Figure 22. However, most other results simply show that minimizing these non-idealities is likely a requirement for online learning neuromorphic processors.

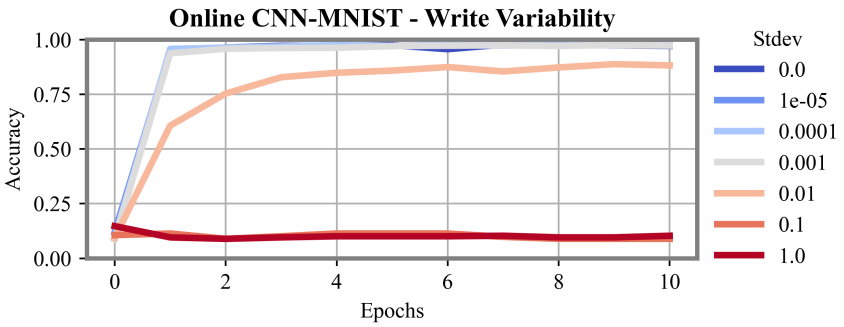
Approved for public release; distribution is unlimited.



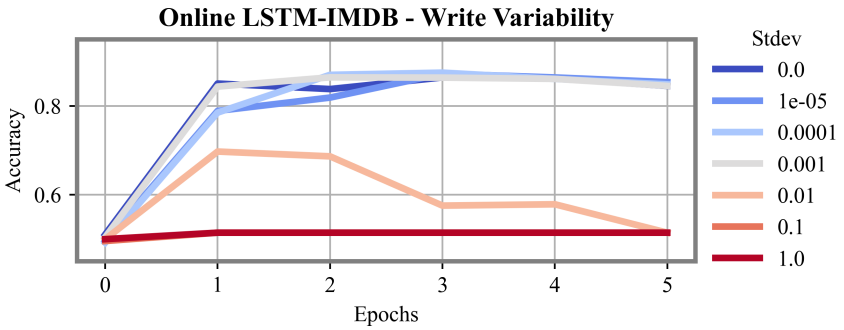
(a) SLP



(b) MLP



(c) CNN

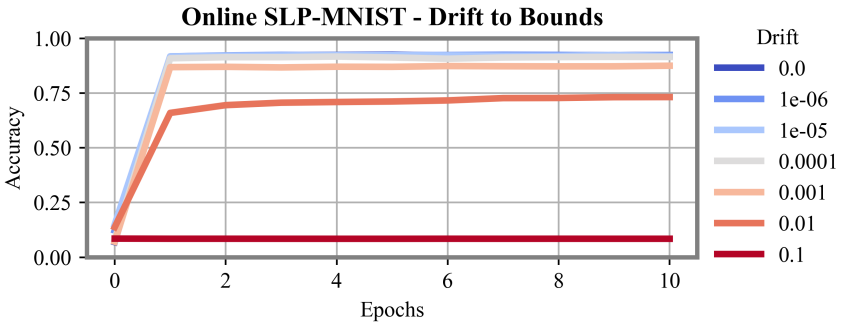


(d) LSTM

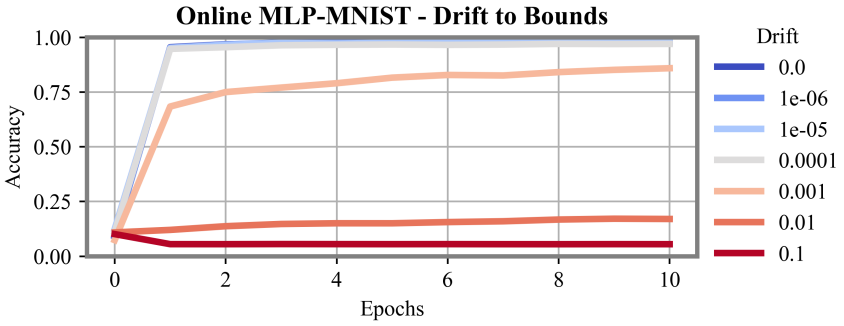
Figure 19

Results of online training neuromorphic processor subjected to write variability.

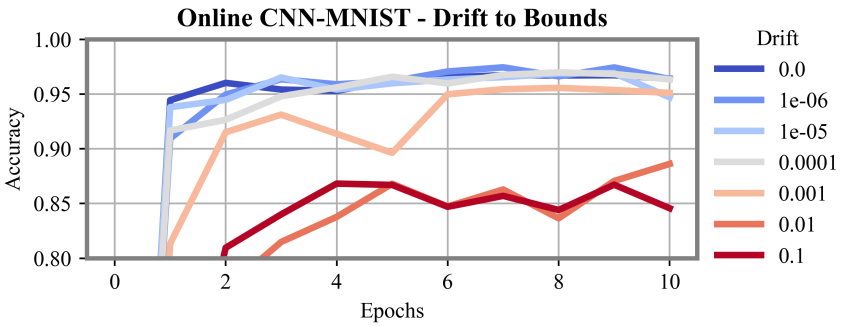
67
Approved for public release; distribution is unlimited.



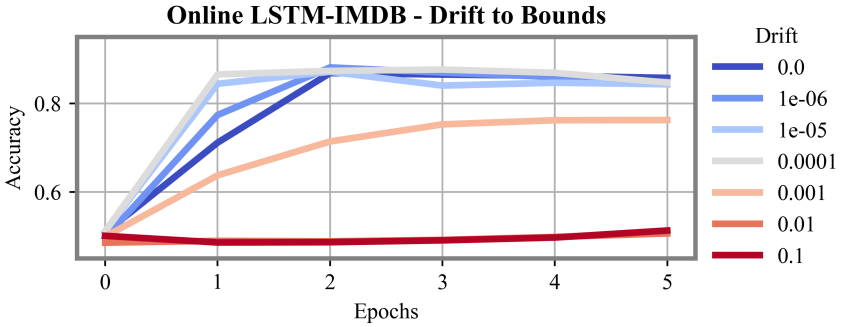
(a) SLP



(b) MLP



(c) CNN

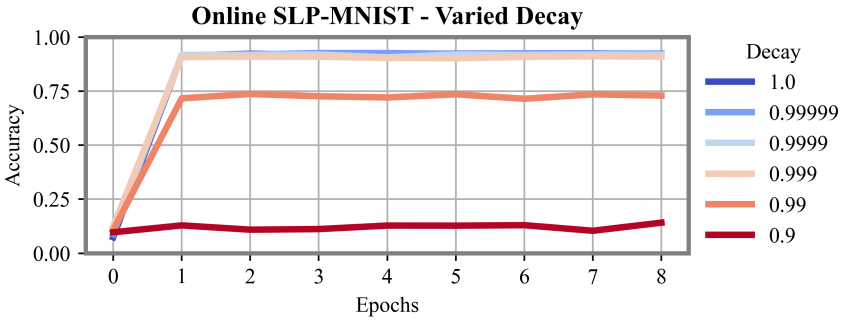


(d) LSTM

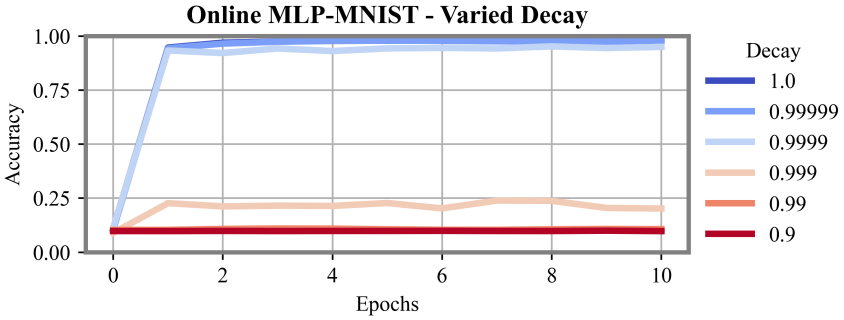
Figure 20

Results of online training neuromorphic processor subjected to drift towards its bounds.

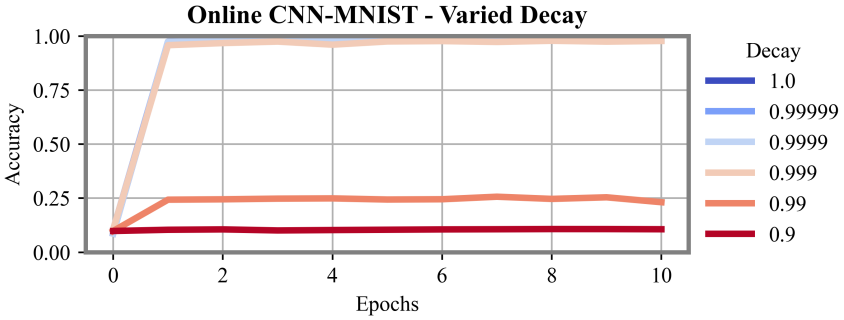
68
Approved for public release; distribution is unlimited.



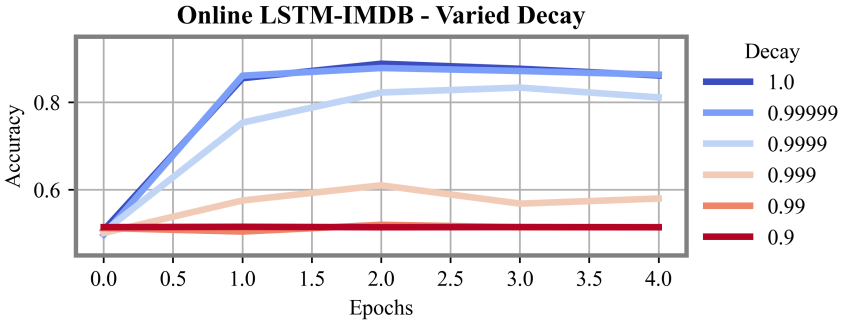
(a) SLP



(b) MLP



(c) CNN

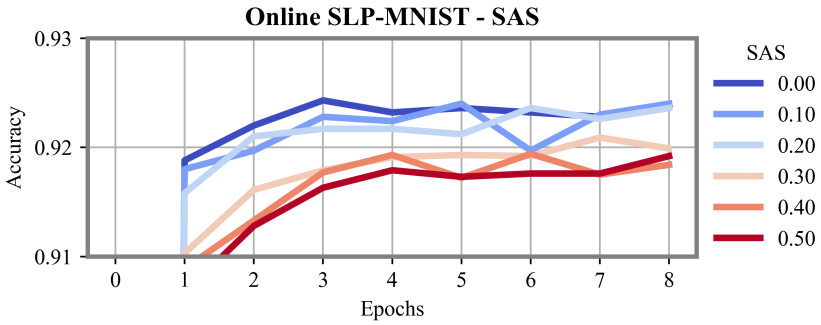


(d) LSTM

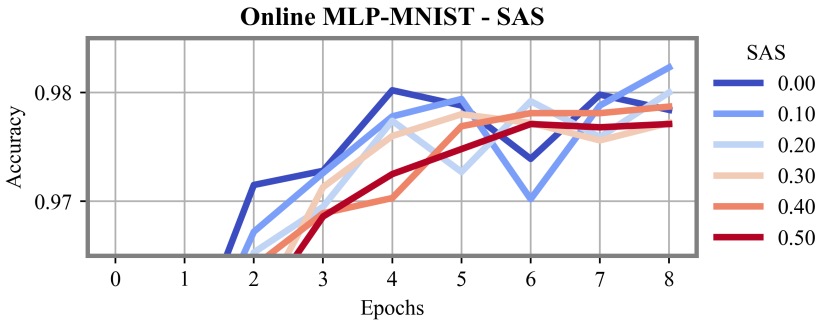
Figure 21

Results of online training neuromorphic processor subjected to time decay.

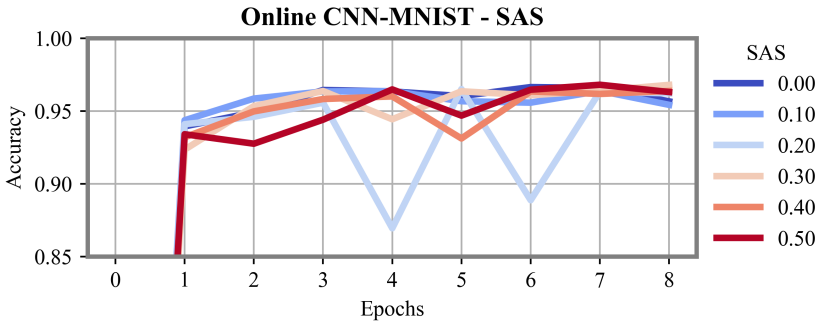
69
Approved for public release; distribution is unlimited.



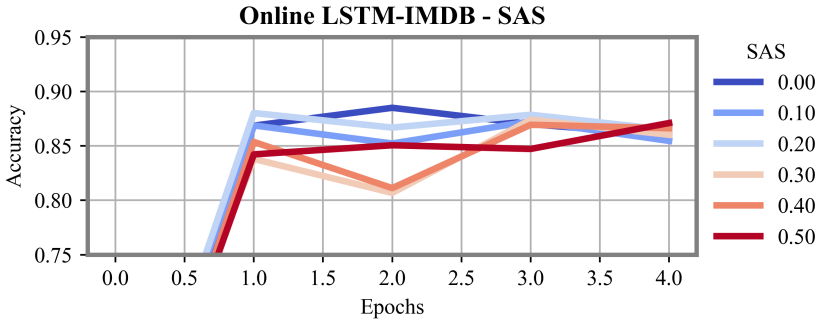
(a) SLP



(b) MLP



(c) CNN

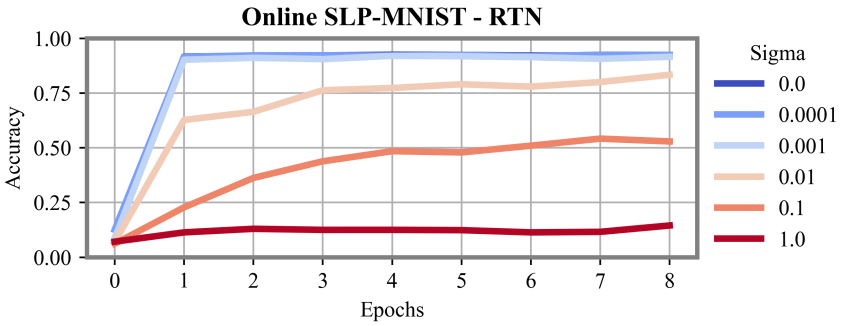


(d) LSTM

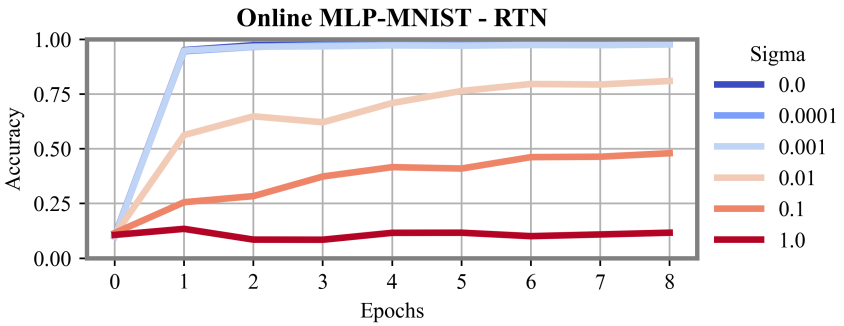
Figure 22

Results of online training neuromorphic processor subjected to 2T2R SAS.

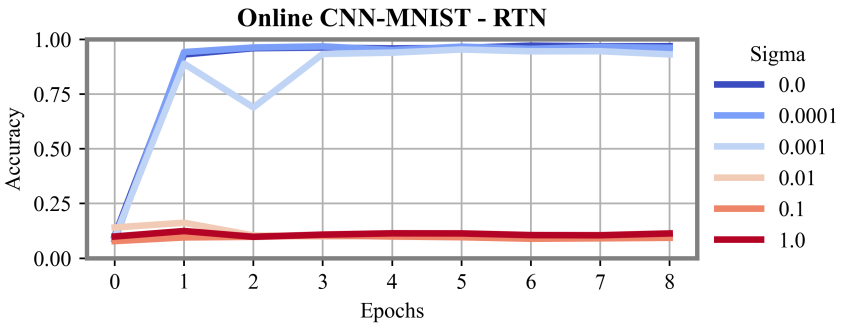
Approved for public release; distribution is unlimited.



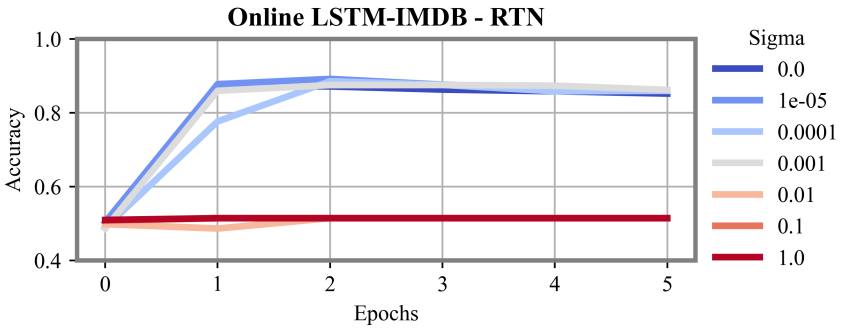
(a) SLP



(b) MLP



(c) CNN

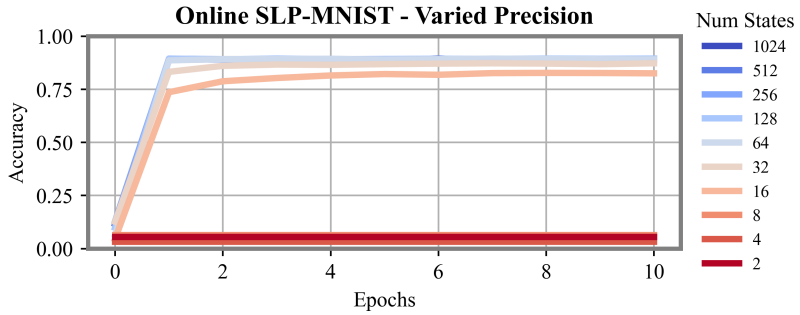


(d) LSTM

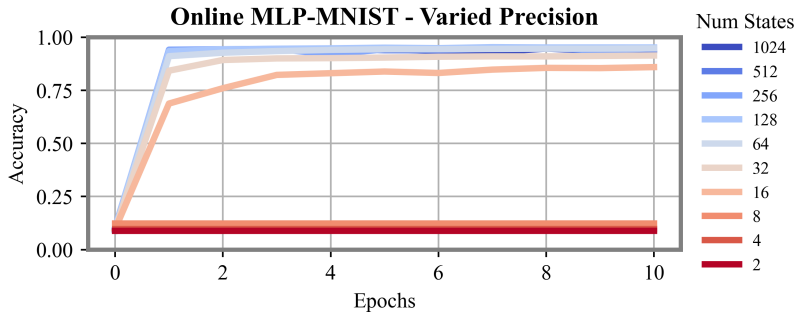
Figure 23

Results of online learning neuromorphic processor subjected to RTN.

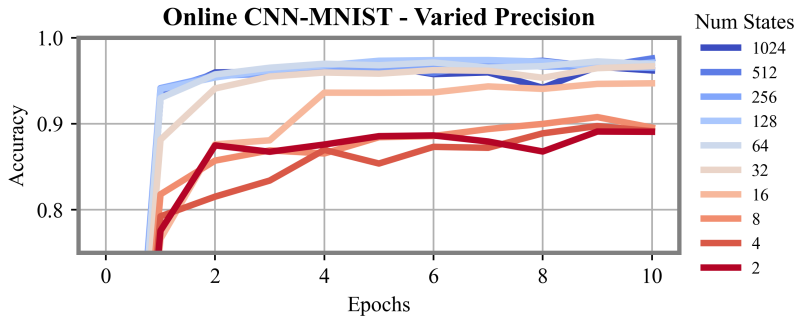
71
Approved for public release; distribution is unlimited.



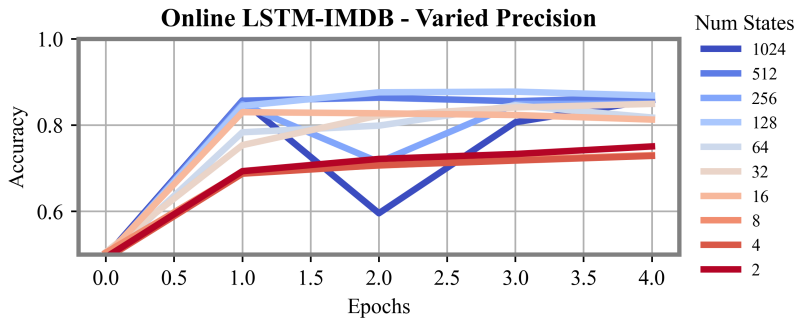
(a) SLP



(b) MLP



(c) CNN



(d) LSTM

Figure 24

Results of online training neuromorphic processor subjected to limited precision.

Approved for public release; distribution is unlimited.

Chapter 5

Transfer Learning Neuromorphic Processor Simulation

Transfer learning is not unique to neuromorphic processors - as previously mentioned, it is used even with GPUs to save on training time with a head start contained in the initialized weights [4]. Transfer learning processors, also referred to as inferencing processors, do not contain the circuitry necessary to train their weights internally, and rely on some external source to train their weights for them, such as a GPU or CPU.

Normally, the goal of machine learning is to maximize accuracy. What if the training chip's peak accuracy does not necessarily align with our non-ideal inferencing chip's peak accuracy? For instance, due to something like write variability or noise, the peak accuracy for the inferencing chip may not be the same peak accuracy as that of a GPU. Therefore, it is important to determine if there will be feedback from the inferencing chip to choose the highest accuracy possible, or if the trainer will converge with the highest accuracy it can attain, and not worry about the inferencing processor.

This greatly affects simulations in LowPy. For example, time decay could be present on the inferencing processor, but does not afflict the GPU training chip. After each epoch of training, the weights from the GPU are then loaded onto the RRAM cells. The RRAM cells are evaluated, and their accuracy is recorded. What happens to the weights on the RRAM cells then? They've been decaying during testing, so should they be ignored, or written back to the trainer GPU, in hopes that the GPU will take the decayed weights and make them more accurate? In this work, it was decided that the values would be loaded back onto the training chip after testing.

5.1 Write Variability

As with the online learning processor write variability simulations, $\sigma_{variability}$ values were spaced logarithmically from 0 to 1. However, write variability is only applied during one section

Approved for public release; distribution is unlimited.

Figure 12, which is when the values are written to the inferencing processor:

```
simulator.pre_evaluation = [
  simulator.write_variability
]
```

As with before, this configuration works regardless of synaptic configuration, since device variability is normally distributed around the requested weight value for both 1T1R and 2T2R. Results are shown in Figure 25. The SLP in Figure 25a is unaffected by write variability until $\sigma_{variability} = 0.01$. Anything above that causes substantial performance issues. Similar results are found with the MLP in Figure 25b, the CNN in Figure 25c, and the LSTM in Figure 25d. Notably, the LSTM experienced overtraining during $\sigma_{variability} = 0.01$, likely due to exploding gradients.

5.2 Drift

As with the online learning processor drift to bounds, drift values were spaced logarithmically from 0 to 0.1. Drift is now applied at one section of Figure 12, occurring prior to any inference operation:

```
simulator.pre_test_forward_propagation = [
  simulator.apply_decay
]
```

The logarithmically varied drift coefficients represent the percentage of drift per iteration, as applied to every weight, shown in Algorithm 2, illustrating that there are several drift configurations, depending on synaptic connection style, as well as the type of RRAM selected. The chosen configuration for simulation in this work was drift towards bounds, which can be characteristic of a 2T2R configuration.

Results are shown in Figure 26. As is to be expected, drift in an inferencing processor is much less detrimental than drift in an online processor, since the training phase is unaltered. None of the networks failed to converge, and the SLP in Figure 26a, along with the MLP in Figure 26b both saw 10% drift to bounds classify with higher performance than 1% drift to bounds,

Approved for public release; distribution is unlimited.

interestingly enough. Perhaps this change in drift was due to a weight initialization predisposed to produce a higher accuracy, or a more nuanced reason. Since gradient descent can easily get trapped in local minima during the optimization phase, drift may have caused the gradient descent process to avoid being trapped in a local minima altogether. The LSTM was fairly tolerant to drift in particular, though seemed to be limited by the overfitting which occurred around the third epoch onward.

5.3 Time Decay

As with the online learning processor, decay values were spaced logarithmically, this time from 0 down to 0.9. Decay is now applied during one section of Figure 12, which is prior to every inference:

```
simulator.pre_test_forward_propagation = [
  simulator.apply_decay
]
```

As with before, time decay was simulated for a 2T2R synaptic connection, since it decays towards 0, implemented in Algorithm 3. Results are shown in Figure 27. As with before, networks struggled with decay ≥ 0.99. And again, the SLP performed much better than the MLP. All networks appear to have performed with fewer sharp jumps and falls in accuracy as well.

5.4 Stuck-At-State Modeling

Cells again were simulated in a 2T2R configuration, varied from 0% SAS to 50%. From Figure 12, it is only necessary to apply SAS prior to inferencing. The matrices which track which cells are stuck at certain states must also be initialized:

```
simulator.post_initialization = [
    simulator.initialize_sas_matrices
]
simulator.pre_train_forward_propagation = [
```


Approved for public release; distribution is unlimited.

```
  simulator.apply_sas
]
simulator.pre_test_forward_propagation = [
  simulator.apply_sas
]
```

Results are shown in Figure 28. All networks classified well, even better than the online learning counterpart. All networks handled even 50% cells SAS with ease, with the SLP dropping the most accuracy of 2.5%.

5.5 Random Telegraph Noise

σ_{rtn} values were again spaced logarithmically from 0 to 1. RTN is applied prior to inferencing operations, and subsequently removed, shown in Figure 12:

```
simulator.pre_test_forward_propagation = [
  simulator.apply_rtn
]
simulator.post_inference = [
  simulator.remove_rtn
]
```

This configuration works regardless of synaptic configuration, since device variability is normally distributed around the requested weight value for both 1T1R and 2T2R. Results are shown in Figure 29. As with the online processor, the impact of noise does not become apparent until a $\sigma_{RTN} \geq 0.01$ has been reached. This applied to all networks, and seemed to affect the CNN the most. The CNN and LSTM were again unable to train with too much noise present, while the others appeared to train somewhat normally, given enough iterations. Since noise is occurring constantly, it makes sense that noiseless training would not perform much differently than noiseless inferencing.

Approved for public release; distribution is unlimited.

5.6 Mixed Precision

The *numStates* value was again spaced in terms of bits, 0 to 10. Limited precision is applied prior to evaluation operations, shown in Figure 12. It is not applied prior to every inference, since the weight does not need to change with each test iteration:

```
simulator.pre_evaluation = [
  simulator.precision
]
```

This configuration works regardless of synaptic configuration, since the number of states can be controlled by the user. RRAM cells with 4 bits of precision may have 16 states, but two of them set up in a 2T2R configuration may offer 32 states, 31 states, or a different value, depending on many factors. Results are shown in Figure 30. The transfer learning processor showed a great deal of higher tolerance relative to the online training processor simulations. The only accuracy loss occurred during the SLP at 2 states, and all others performed as well as is reasonably expected with no nonidealities present.

5.7 Discussion

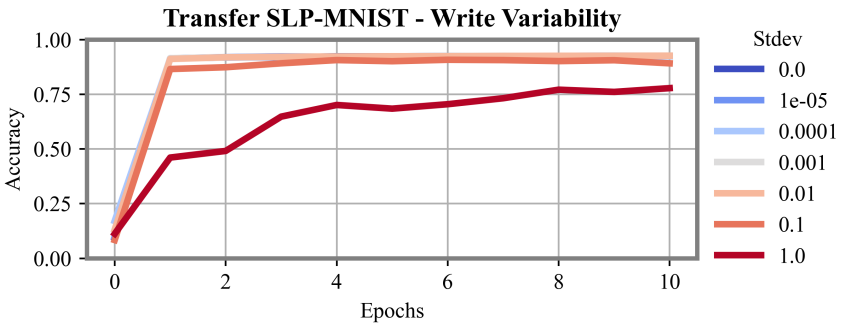
Not surprisingly, the transfer learning processor performed much better than the online learning processor. It is important to keep in mind that the transfer processor was not only having its weights written to, but was sending them back out to the trainer with the nonidealities applied. This will not always be the case, it is likely much more convenient to simply write to the memristive crossbar array without reading the values back out. Not only that, but the method of transfer learning simulated here may even show worse accuracies than if the weights were not sent back to the external trainer, but there are likely exceptions to that statement.

While notable performance improvements were observed when switching to the transfer learning processor such as the limited precision, the results were overall very similar. The same thresholds from the online processor were likely where performance loss began to occur for the

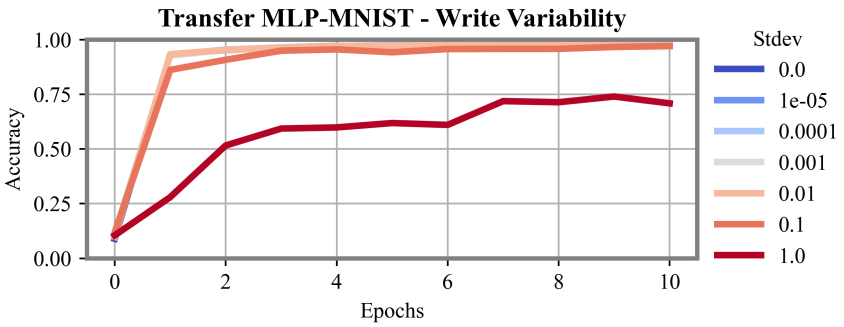
Approved for public release; distribution is unlimited.

transfer learning processor as well, though some nonidealities like write variability and precision loss did have an improved tolerance with the transfer learning processor, since the iterative training process was unaltered.

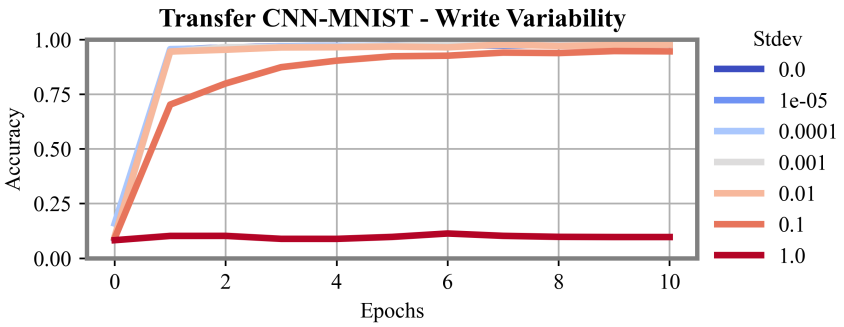
Approved for public release; distribution is unlimited.



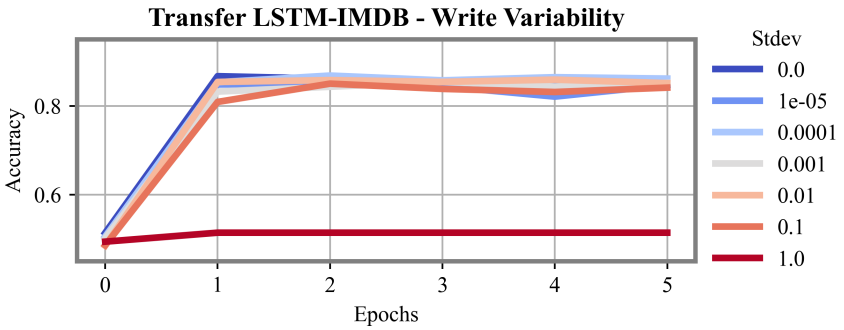
(a) SLP



(b) MLP



(c) CNN

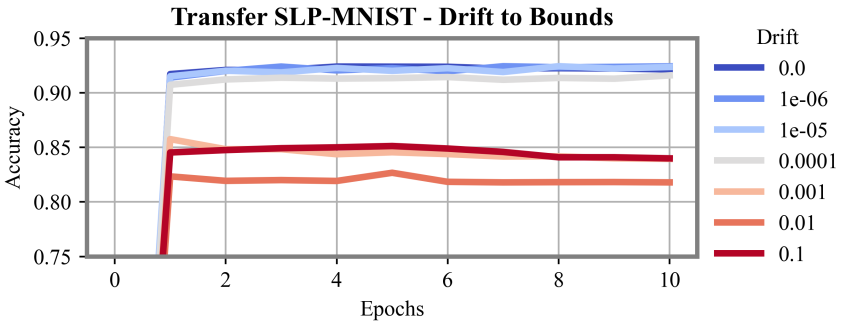


(d) LSTM

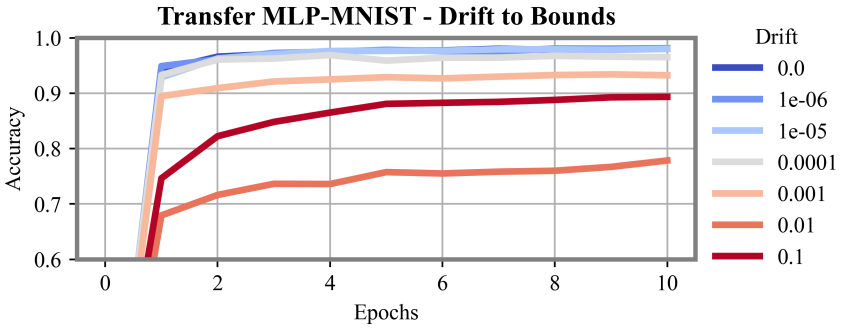
Figure 25

Results of transfer learning neuromorphic processor subjected to write variability.

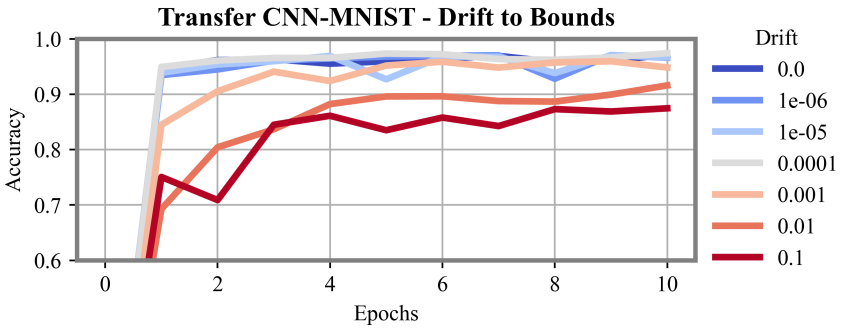
Approved for public release; distribution is unlimited.



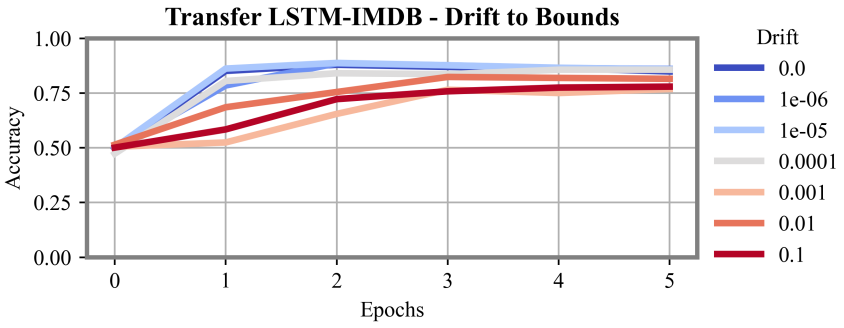
(a) SLP



(b) MLP



(c) CNN

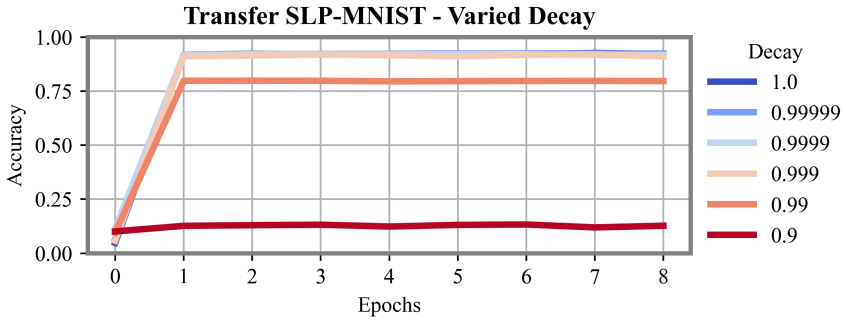


(d) LSTM

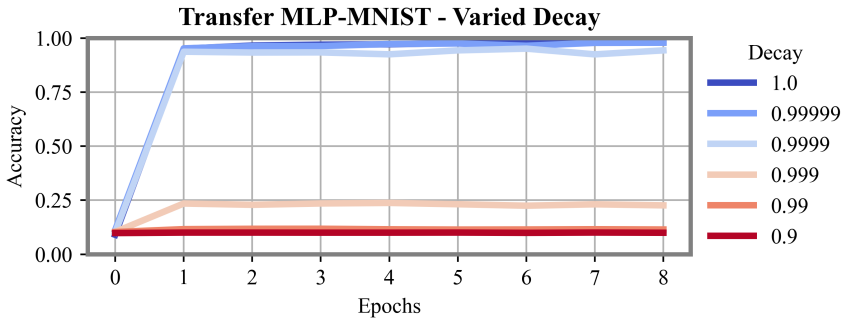
Figure 26

Results of transfer learning neuromorphic processor subjected to drift towards its bounds.

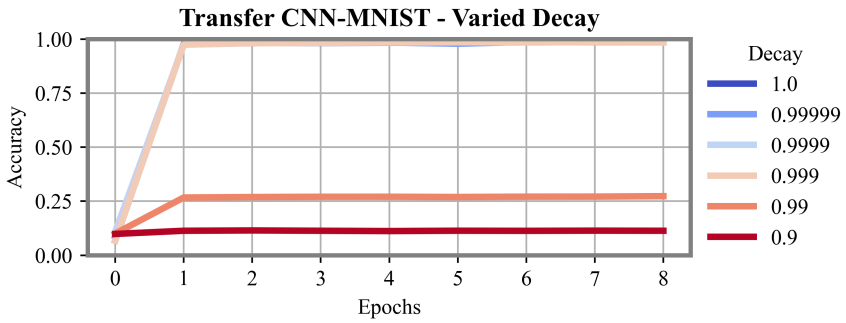
Approved for public release; distribution is unlimited.



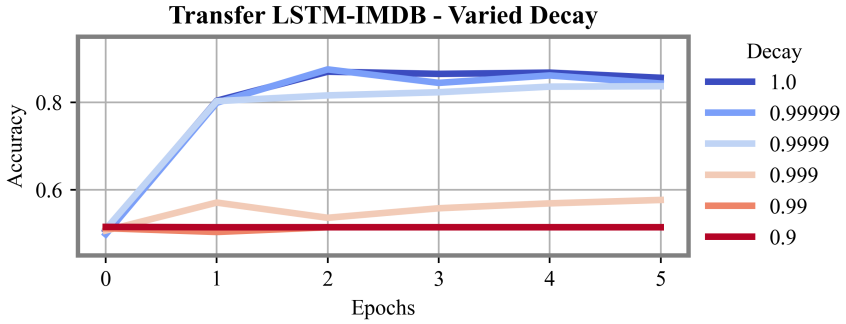
(a) SLP



(b) MLP



(c) CNN

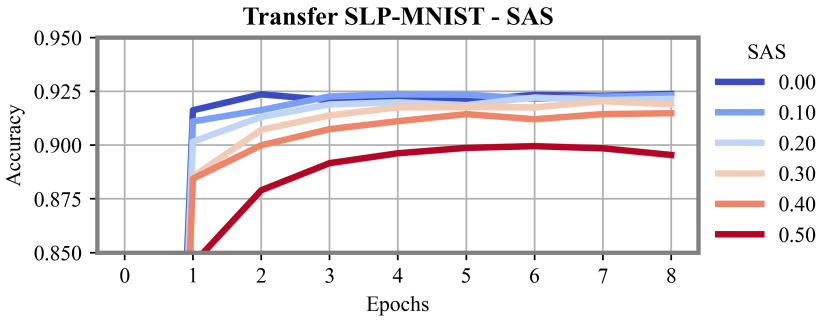


(d) LSTM

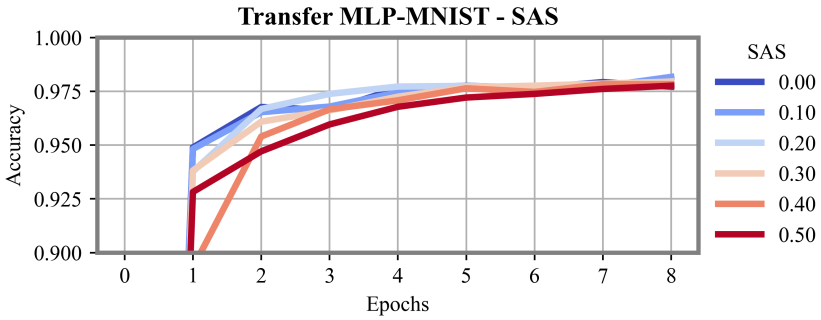
Figure 27

Results of transfer learning neuromorphic processor subjected to time decay.

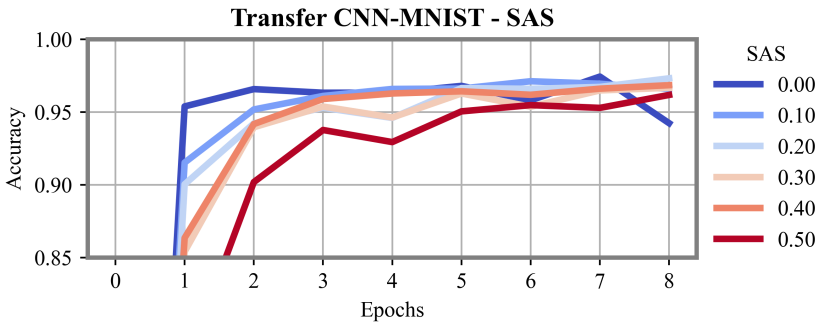
Approved for public release; distribution is unlimited.



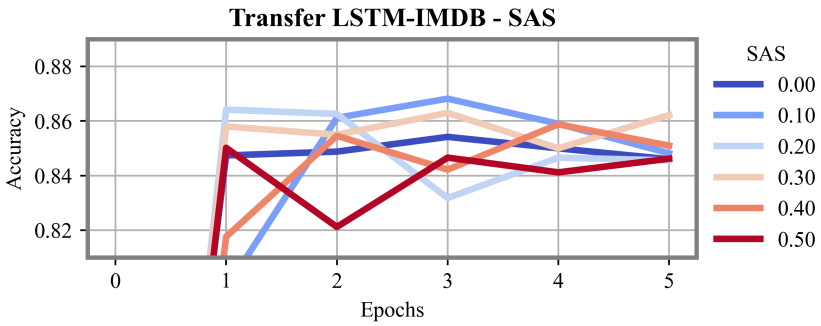
(a) SLP



(b) MLP



(c) CNN

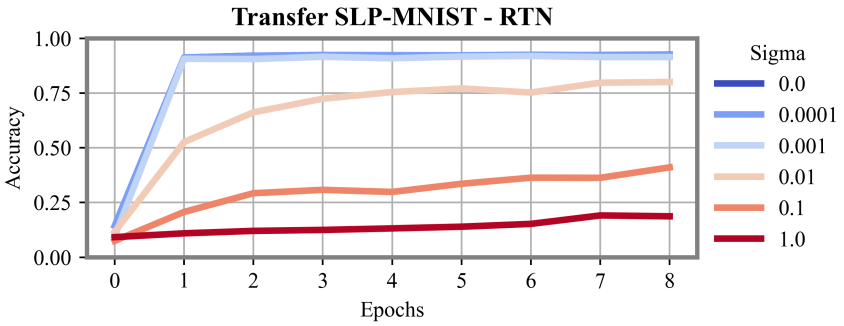


(d) LSTM

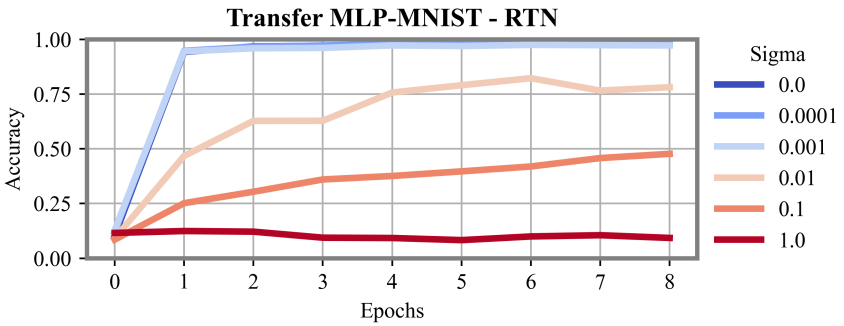
Figure 28

Results of transfer learning neuromorphic processor subjected to 2T2R SAS.

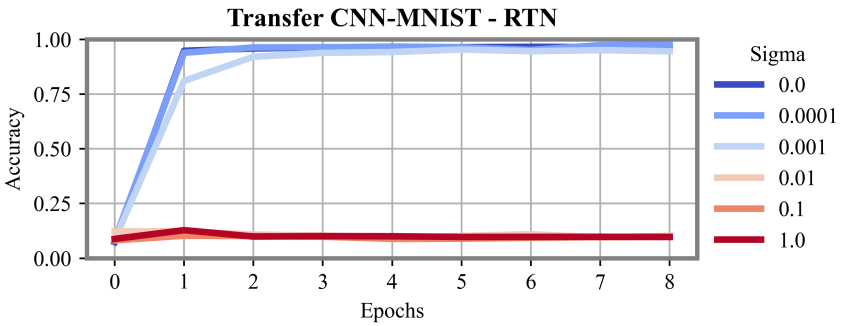
Approved for public release; distribution is unlimited.



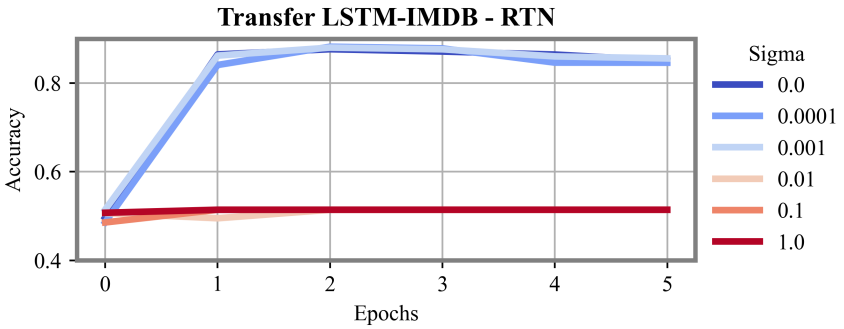
(a) SLP



(b) MLP



(c) CNN

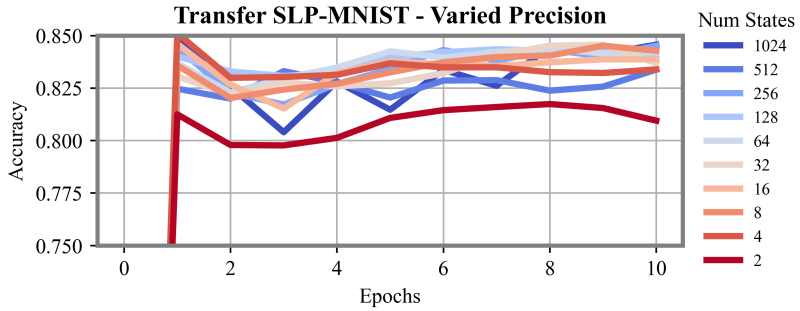


(d) LSTM

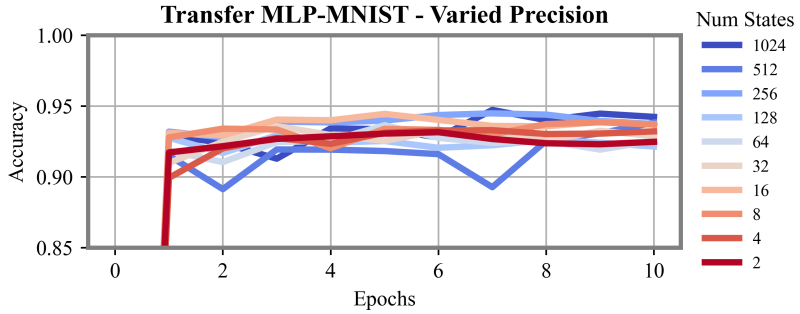
Figure 29

Results of transfer learning neuromorphic processor subjected to RTN.

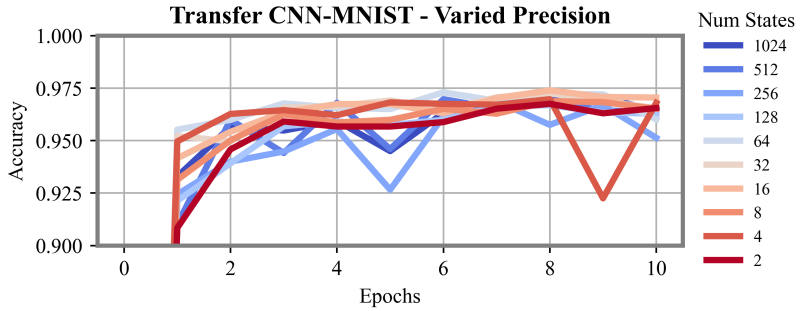
83
Approved for public release; distribution is unlimited.



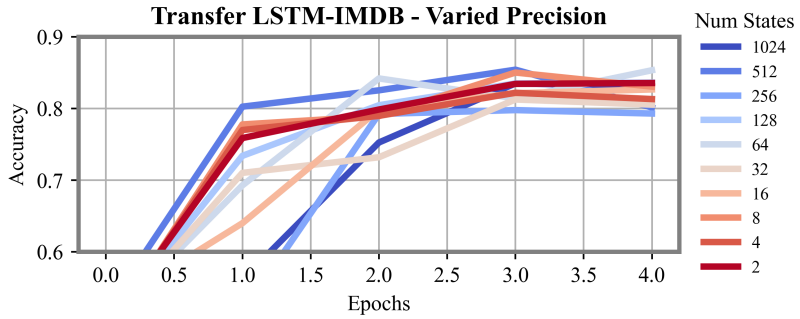
(a) SLP



(b) MLP



(c) CNN



(d) LSTM

Figure 30

Results of transfer learning neuromorphic processor subjected to limited precision.

Approved for public release; distribution is unlimited.

Chapter 6

Conclusion

With such an involved design process for neuromorphic processors, the quick, high-level insights that LowPy offers can, at the very least, offer the user a starting point for where the severity of the memristive device's various nonidealities should be. At most, it gives a precise, quantifiable value for what can be tolerated your specific combination of algorithm, optimizer, loss function, and all other hyperparameters in a given machine learning network, regarding the most commonly observed memristive nonidealities, allowing for verification and validation of RRAM devices when designing a neuromorphic architecture based on the specifications obtained from LowPy.

Regarding overall generalizations to be made from the simulations presented for the inferencing only, online learning, and transfer learning processors, keeping the nonidealities below the limits listed in Table 7. These generalizations should serve as a starting point for any processor style, but is largely dependent upon a number of individual factors, such as the chosen optimizer, loss function, hyperparameters, and dataset.

The structuring of LowPy to wrap around Keras [15], as well as its reliance on the paralellizeable TensorFlow [13] functions, offers the user powerful computations through high level function calls, even usable on Cuda [14] GPUs. This provides the familiarity of Keras, and keeps machine learning simulations in line with current research and industry usage.

The event based design shown in Figure 12 allows the user to choose when nonidealities are applied to any layer with the string "LowPy" in its layer `name`. Not only that, it allows the user to apply a nonideality during one event, and then subsequently remove it, reverting their weights to the previous state. This implementation choice offers flexibility so that LowPy can be used for a wider range of applications.

Approved for public release; distribution is unlimited.

Table 7

Thresholds where Non-Ideal Memristive Properties Begin to Reduce Neural Network Accuracy

Nonideality	Inferencing Only	Online Learning	Transfer Learning
$\sigma_{variability}$	≥ 0.01	≥ 0.01	≥ 0.1
% drift to bounds per iteration	$\geq 0.01\%$	$\geq 0.1\%$	$\geq 0.01\%$
% decay per iteration	$\geq 0.01\%$	$\geq 0.01\%$	$\geq 0.01\%$
% SAS	$\geq 10\%$	$\geq 20\%$	$\geq 40\%$
σ_{RTN}	≥ 0.001	≥ 0.001	≥ 0.01
Number of states	≤ 32	≤ 32	≤ 2

6.1 Limitations

While LowPy benefits from wrapping around an existing machine learning API, it is only allowed access to whatever the API provides. For instance, if the multiplication circuit within a neuromorphic ASIC has nonidealities present, then the otherwise convenient Keras [15] layers, optimizers, and loss functions will need to be redesigned by the end user from scratch, at which point the convenience of LowPy is lost.

Along with that, there is a balance to be struck between being too RRAM-specific, and being too generalized such that no one can make use of it. If all properties replicate RRAM's decay perfectly, it likely no longer represents the way in which other memory devices decay.

6.2 Future Work

The specificity problem mentioned previously can be solved with the implementation of multiple functions for RRAM, general devices, and any other specific devices that can benefit from LowPy. There is plenty more tailoring to be done to the provided algorithms, and more nonidealities to be modeled. Given that the goal is not to replace the precise nature of SPICE and other lower level simulators, the amount of features added should be tempered, should they ever cause unreasonable training times. However, topics such as high level power consumption

Approved for public release; distribution is unlimited.

estimations could still be possible, since LowPy tracks cell updates and other network properties. Overall, there is still more to be done for LowPy to meet its end goal of providing a user with a familiar, powerful python library capable of simulating any nonideality found particularly in memristive devices.

87
Approved for public release; distribution is unlimited.

References

[1] Z. M. Research, *Machine learning market - by service: Global industry perspective, comprehensive analysis, and forecast, 2020-2027*. [Online]. Available: https://www.zionmarketresearch.com/report/machine-learning-market.

[2] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 620–629. DOI: 10.1109/HPCA.2018.00059.

[3] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, "Survey and benchmarking of machine learning accelerators," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–9. DOI: 10.1109/HPEC.2019.8916327.

[4] P. Budzianowski and I. Vulic, "Hello, it's GPT-2 - how can I help you? towards the use of pretrained language models for task-oriented dialogue systems," *CoRR*, vol. abs/1907.05774, 2019. arXiv: 1907.05774. [Online]. Available: http://arxiv.org/abs/1907.05774.

[5] A. Fayyazi, M. Ansari, M. Kamal, A. Afzali-Kusha, and M. Pedram, "An ultra low-power memristive neuromorphic circuit for internet of things smart sensors," *IEEE Internet of Things Journal*, vol. 5, no. 2, pp. 1011–1022, 2018. DOI: 10.1109/JIOT.2018.2799948.

[6] F. C. Bauer, D. R. Muir, and G. Indiveri, "Real-time ultra-low power ecg anomaly detection using an event-driven neuromorphic processor," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 6, pp. 1575–1582, 2019. DOI: 10.1109/TBCAS.2019.2953001.

[7] M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018. DOI: 10.1109/MM.2018.112130359.

[8] Y. LeCun *et al.*, *The mnist database of handwritten digits*, http://yann.lecun.com/exdb/mnist/, 2015.

[9] H. H. See, B. Lim, S. Li, H. Yao, W. Cheng, H. Soh, and B. C. K. Tee, *St-mnist – the spiking tactile mnist neuromorphic dataset*, 2020. arXiv: 2005.04319 [cs.NE].

Approved for public release; distribution is unlimited.

[10] S. Liu, K. Li, Y. Sun, X. Zhu, Z. Li, B. Song, H. Liu, and Q. Li, "A taox-based electronic synapse with high precision for neuromorphic computing," *IEEE Access*, vol. 7, pp. 184 700–184 706, 2019. DOI: 10.1109/ACCESS.2019.2961166.

[11] T. J. Bailey, A. J. Ford, S. Barve, J. Wells, and R. Jha, "Development of a short-term to long-term supervised spiking neural network processor," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 11, pp. 2410–2423, 2020. DOI: 10.1109/TVLSI.2020.3013810.

[12] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, "Learning word vectors for sentiment analysis," in *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, Portland, Oregon, USA: Association for Computational Linguistics, Jun. 2011, pp. 142–150. [Online]. Available: http://www.aclweb.org/anthology/P11-1015.

[13] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *Tensorflow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: https://www.tensorflow.org/.

[14] NVIDIA, P. Vingelmann, and F. H. Fitzek, *Cuda, release: 10.2.89*, 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit.

[15] F. Chollet *et al.*, *Keras*, https://keras.io, 2015.

[16] F. Rosenblatt, "Perceptron simulation experiments," *Proceedings of the IRE*, vol. 48, no. 3, pp. 301–309, 1960. DOI: 10.1109/JRPROC.1960.287598.

[17] F. Rosenblatt, *Frank rosenblatt: Principles of neurodynamics: Perceptrons ...* Mar. 1961. [Online]. Available: https://apps.dtic.mil/dtic/tr/fulltext/u2/256582.pdf.

[18] F. Cummins, "Learning to forget: Continual prediction with lstm," English, *IET Conference Proceedings*, 850–855(5), Jan. 1999. [Online]. Available: https://digital-library.theiet.org/content/conferences/10.1049/cp_19991218.

[19] A. Sidorov, *Transitioning entirely to neural machine translation*, Jun. 2018. [Online]. Available: https://engineering.fb.com/2017/08/03/ml-applications/transitioning-entirely-to-neural-machine-translation/.

Approved for public release; distribution is unlimited.

[20] H. Robbins and S. Monro, "A Stochastic Approximation Method," *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951. DOI: 10.1214/aoms/1177729586. [Online]. Available: https://doi.org/10.1214/aoms/1177729586.

[21] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017. arXiv: 1412.6980 [cs.LG].

[22] E. Z. Farsa, A. Ahmadi, M. A. Maleki, M. Gholami, and H. N. Rad, "A low-cost high-speed neuromorphic hardware based on spiking neural network," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 66, no. 9, pp. 1582–1586, 2019. DOI: 10.1109/TCSII.2019.2890846.

[23] E. Donati, M. Payvand, N. Risi, R. Krause, and G. Indiveri, "Discrimination of emg signals using a neuromorphic implementation of a spiking neural network," *IEEE Transactions on Biomedical Circuits and Systems*, vol. 13, no. 5, pp. 795–803, 2019. DOI: 10.1109/TBCAS.2019.2925454.

[24] P. U. Diehl, G. Zarrella, A. Cassidy, B. U. Pedroni, and E. Neftci, "Conversion of artificial recurrent neural networks to spiking neural networks for low-power neuromorphic hardware," in *2016 IEEE International Conference on Rebooting Computing (ICRC)*, 2016, pp. 1–8. DOI: 10.1109/ICRC.2016.7738691.

[25] D. Ielmini and S. Ambrogio, "Neuromorphic computing with resistive switching memory devices," *Advances in Non-Volatile Memory and Storage Technology*, pp. 603–631, Jun. 2018. DOI: 10.1016/b978-0-08-102584-0.00017-6. [Online]. Available: https://www.nature.com/articles/s41928-018-0092-2.

[26] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, 1971. DOI: 10.1109/TCT.1971.1083337.

[27] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, May 2008. DOI: https://doi.org/10.1038/nature06932. [Online]. Available: https://www.nature.com/articles/nature06932.

[28] A. Valentian, F. Rummens, E. Vianello, T. Mesquida, C. L. de Boissac, O. Bichler, and C. Reita, "Fully integrated spiking neural network with analog neurons and rram synapses," in *2019 IEEE International Electron Devices Meeting (IEDM)*, 2019, pp. 14.3.1–14.3.4. DOI: 10.1109/IEDM19573.2019.8993431.

[29] T. J. Bailey, A. J. Ford, S. Barve, J. Wells, and R. Jha, "Development of a short-term to long-term supervised spiking neural network processor," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 11, pp. 2410–2423, 2020. DOI: 10.1109/TVLSI.2020.3013810.

Approved for public release; distribution is unlimited.

[30] H. Wu, P. Yao, B. Gao, W. Wu, Q. Zhang, W. Zhang, N. Deng, D. Wu, H. .-. P. Wong, S. Yu, and H. Qian, "Device and circuit optimization of rram for neuromorphic computing," in *2017 IEEE International Electron Devices Meeting (IEDM)*, 2017, pp. 11.5.1–11.5.4. DOI: 10.1109/IEDM.2017.8268372.

[31] X. Wang, Q. Wang, F. Meng, S. H. Lee, and W. D. Lu, "Deep neural network mapping and performance analysis on tiled rram architecture," in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2020, pp. 141–144. DOI: 10.1109/AICAS48895.2020.9073942.

[32] J. Han, H. Liu, M. Wang, Z. Li, and Y. Zhang, "Era-lstm: An efficient reram-based architecture for long short-term memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1328–1342, 2020. DOI: 10.1109/TPDS.2019.2962806.

[33] B. Li, P. Gu, Y. Shan, Y. Wang, Y. Chen, and H. Yang, "Rram-based analog approximate computing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 12, pp. 1905–1917, 2015. DOI: 10.1109/TCAD.2015.2445741.

[34] F. Alibart, L. Gao, B. D. Hoskins, and D. B. Strukov, "High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm," *Nanotechnology*, vol. 23, no. 7, p. 075 201, Jan. 2012. DOI: 10.1088/0957-4484/23/7/075201. [Online]. Available: https://doi.org/10.1088/0957-4484/23/7/075201.

[35] W. Cao, L. Ke, A. Chakrabarti, and X. Zhang, "Neural network-inspired analog-to-digital conversion to achieve super-resolution with low-precision rram devices," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–7. DOI: 10.1109/ICCAD45719.2019.8942099.

[36] P. Stoliar, P. Levy, M. J. Sánchez, A. G. Leyva, C. A. Albornoz, F. Gomez-Marlasca, A. Zanini, C. T. Salazar, N. Ghenzi, and M. J. Rozenberg, *Non-volatile multilevel resistive switching memory cell: A transition metal oxide-based circuit*, 2013. arXiv: 1310.3613 [cond-mat.other].

[37] A. Jones and R. Jha, "A compact gated-synapse model for neuromorphic circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2020. DOI: 10.1109/TCAD.2020.3028534.

[38] S. Ambrogio, S. Balatti, V. Milo, R. Carboni, Z. Wang, A. Calderoni, N. Ramaswamy, and D. Ielmini, "Novel rram-enabled 1t1r synapse capable of low-power stdp via burst-mode communication and real-time unsupervised machine learning," in *2016 IEEE Symposium on VLSI Technology*, 2016, pp. 1–2. DOI: 10.1109/VLSIT.2016.7573432.

[39] X. Sun, X. Peng, P. Chen, R. Liu, J. Seo, and S. Yu, "Fully parallel rram synaptic array for implementing binary neural network with 1 to -1 weights and 1 to 0 neurons," in *2018 23rd*

Approved for public release; distribution is unlimited.

Asia and South Pacific Design Automation Conference (ASP-DAC), 2018, pp. 574–579. DOI: 10.1109/ASPDAC.2018.8297384.

[40] Q. Xia and J. J. Yang, "Memristive crossbar arrays for brain-inspired computing," *Nature Materials*, vol. 18, no. 4, pp. 309–323, 2019. DOI: 10.1038/s41563-019-0291-x. [Online]. Available: https://www.nature.com/articles/s41563-019-0291-x.

[41] A. Jones, A. Rush, C. Merkel, E. Herrmann, A. P. Jacob, C. Thiem, and R. Jha, "A neuromorphic slam architecture using gated-memristive synapses," *Neurocomputing*, vol. 381, pp. 89–104, 2020, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2019.09.098. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231219314894.

[42] A. J. Ford and R. Jha, "Memristive device variability performance impact on neuromorphic machine learning hardware," in *2020 11th International Green and Sustainable Computing Workshops (IGSC)*, 2020, pp. 1–7. DOI: 10.1109/IGSC51522.2020.9291114.

[43] A. Chen and M. Lin, "Variability of resistive switching memories and its impact on crossbar array performance," in *2011 International Reliability Physics Symposium*, 2011, MY.7.1–MY.7.4. DOI: 10.1109/IRPS.2011.5784590.

[44] V. Parmar and M. Suri, "Exploiting variability in resistive memory devices for cognitive systems," in *Advances in Neuromorphic Hardware Exploiting Emerging Nanoscale Devices*, M. Suri, Ed. New Delhi: Springer India, 2017, pp. 175–195, ISBN: 978-81-322-3703-7. DOI: 10.1007/978-81-322-3703-7_9. [Online]. Available: https://doi.org/10.1007/978-81-322-3703-7_9.

[45] Q. Wang and D. He, "Time-decay memristive behavior and diffusive dynamics in one forget process operated by a 3d vertical pt/ta2o5-x/w device," *Scientific Reports*, vol. 7, no. 1, 2017. DOI: 10.1038/s41598-017-00985-0. [Online]. Available: https://www.nature.com/articles/s41598-017-00985-0#citeas.

[46] M.-J. Lee, "A fast, high-endurance and scalable non-volatile memory device made from asymmetric ta2o5-x/tao2-x bilayer structures," *Nature Materials*, Jul. 2011. DOI: https://doi.org/10.1038/nmat3070. [Online]. Available: https://www.nature.com/articles/nmat3070.

[47] D. Veksler, G. Bersuker, L. Vandelli, A. Padovani, L. Larcher, A. Muraviev, B. Chakrabarti, E. Vogel, D. C. Gilmer, and P. D. Kirsch, "Random telegraph noise (rtn) in scaled rram devices," in *2013 IEEE International Reliability Physics Symposium (IRPS)*, 2013, MY.10.1–MY.10.4. DOI: 10.1109/IRPS.2013.6532101.

[48] S. Choi, Y. Yang, and W. Lu, "Random telegraph noise and resistance switching analysis of oxide based resistive memory," *Nanoscale*, vol. 6, pp. 400–404, 1 2014. DOI: 10.1039/C3NR05016E. [Online]. Available: http://dx.doi.org/10.1039/C3NR05016E.

Approved for public release; distribution is unlimited.

[49] M. Le Gallo, "Mixed-precision in-memory computing," *Nature Electronics*, Apr. 2018. [Online]. Available: https://doi.org/10.1038/s41928-018-0054-8.

[50] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. DOI: 10.1038/s41586-020-2649-2. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2.

[51] S. Ambrogio, S. Balatti, V. Milo, R. Carboni, Z. Wang, A. Calderoni, N. Ramaswamy, and D. Ielmini, "Novel rram-enabled 1t1r synapse capable of low-power stdp via burst-mode communication and real-time unsupervised machine learning," in *2016 IEEE Symposium on VLSI Technology*, 2016, pp. 1–2. DOI: 10.1109/VLSIT.2016.7573432.

[52] Z. Zhou, P. Huang, Y. C. Xiang, W. S. Shen, Y. D. Zhao, Y. L. Feng, B. Gao, H. Q. Wu, H. Qian, L. F. Liu, X. Zhang, X. Y. Liu, and J. F. Kang, "A new hardware implementation approach of bnns based on nonlinear 2t2r synaptic cell," in *2018 IEEE International Electron Devices Meeting (IEDM)*, 2018, pp. 20.7.1–20.7.4. DOI: 10.1109/IEDM.2018.8614642.

Approved for public release; distribution is unlimited.