MemTorch: An Open-source Simulation Framework for Memristive Deep Learning Systems

Corey Lammie, *Student Member, IEEE*, Wei Xiang, *Senior Member, IEEE*, Bernabé Linares-Barranco, *Fellow, IEEE*, and Mostafa Rahimi Azghadi, *Senior Member, IEEE*

***Abstract*—Memristive devices have shown great promise to facilitate the acceleration and improve the power efficiency of Deep Learning (DL) systems. Crossbar architectures constructed using memristive devices can be used to efficiently implement various in-memory computing operations, such as Multiply-Accumulate (MAC) and unrolled-convolutions, which are used extensively in Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNNs). Currently, there is a lack of a modernized, open source and general high-level simulation platform that can fully integrate any behavioral or experimental memristive device model and its putative non-idealities into crossbar architectures within DL systems. This paper presents such a framework, entitled *MemTorch*, which adopts a modernized software engineering methodology and integrates directly with the well-known *PyTorch* Machine Learning (ML) library. We fully detail the public release of *MemTorch* and its release management, and use it to perform novel simulations of memristive DL systems, which are trained and benchmarked using the CIFAR-10 dataset. Moreover, we present a case study, in which *MemTorch* is used to simulate a near-sensor in-memory computing system for seizure detection using Pt/Hf/Ti Resistive Random Access Memory (ReRAM) devices. Our open source *MemTorch* framework[1] can be used and expanded upon by circuit and system designers to conveniently perform customized large-scale memristive DL simulations taking into account various unavoidable device non-idealities, as a preliminary step before circuit-level realization.**

***Index Terms*—Memristors, ReRAM, Deep learning, Simulation framework**

I. Introduction

MEMRISTIVE crossbar architectures [1] have been used to reduce the time complexity of Vector-Matrix Multiplications (VMMs) used in DNNs from $\mathcal{O}(n^2)$ to $\mathcal{O}(n)$, and in extreme cases to $\mathcal{O}(1)$ [2], facilitating the acceleration and improving the power efficiency of DL systems. However, memristors are still considered an emerging technology, where their reliable manufacturing processes are yet to be achieved. As a result, DL architectures realized using memristor crossbars are putative to be prone to severe errors due to a number of device limitations including: finite discrete conductance states, device I/V non-linearity, failure, aging, cycle-to-cycle and device-to-device variability [3], [4]. Consequently, significant research efforts are being made to improve the reliability and robustness of memristive, or ReRAM crossbars, used to perform *in-situ* learning [5]–[7] and inference [2], [5], [8]–[11] in DL systems.

While various device-level behavioral memristor models have been proposed [12], they often fail to account for device limitations, which are commonly modeled using Simulation Program with Integrated Circuit Emphasis (SPICE) and compact Verilog-A models. Although SPICE and compact Verilog-A models of non-ideal memristors are plentiful [13]–[19], realizing and simulating DL systems using them is difficult and requires specialized low-level circuit simulation expertise and the usage of various tool-chains that are largely non-standardized, hence, are arduous to configure and use. Moreover, circuit-level simulation is usually performed using Central Processing Units (CPUs), making the simulation of large-scale memristive systems often time prohibitive. Simulations which are conducted using these tool-chains cannot easily be documented, containerized, and shared cross-platform, making them difficult to openly distribute, constraining emerging research efforts towards reproducible and transparent code.

Instead, a general cross-platform, heterogeneous, high-level, customizable and open-source simulation framework could be used to conveniently build and rapidly prototype tailored large-scale Memristive Deep Neural Networks (MDNNs) and Memristive Deep Learning Systems (MDLSs), as a preliminary step before circuit-level realization. Such a platform would:

1) Facilitate the cross-platform development and distribution of large-scale memristive systems;
2) Be fully open source and modular, extendable by all, well-documented, and community-driven;
3) Support heterogeneous platforms such as CPUs and Graphics Processing Units (GPUs);
4) Have a high-level Application Programming Interface (API), which is able to abstract performance-critical tasks described in various low-level languages.

In this paper, we present such a framework, entitled *MemTorch*, for deep memristive learning using crossbar architectures. *MemTorch* is an open-source [21] simulation framework that integrates directly with the open-source *PyTorch* ML library and adopts a modernized software engineering design methodology. To demonstrate *MemTorch*'s intuitive design, before proceeding further, we depict a typical use-case work flow in Fig. 1 that converts linear layers within a DNN to memristive-equivalent layers employing 1-Transistor 1-Resistor (1T1R) crossbars, and introduce various non-ideal device characteristics to the devices within them. These features offered by *MemTorch* enable us to perform simulations of

Corey Lammie, Wei Xiang and M. Rahimi Azghadi are with the College of Science and Engineering, James Cook University, Australia (e-mail: {corey.lammie, wei.xiang, mostafa.rahimiazghadi}@jcu.edu.au).

BernabÃl' Linares-Barranco is with the Institute of Microelectronics of Seville, IMSE-CNM, Parque TecnolÃşgico de la Cartuja, CSIC, University of Seville, Spain (e-mail: bernabe@imse.cnm.es).

[1]https://github.com/coreylammie/MemTorch.

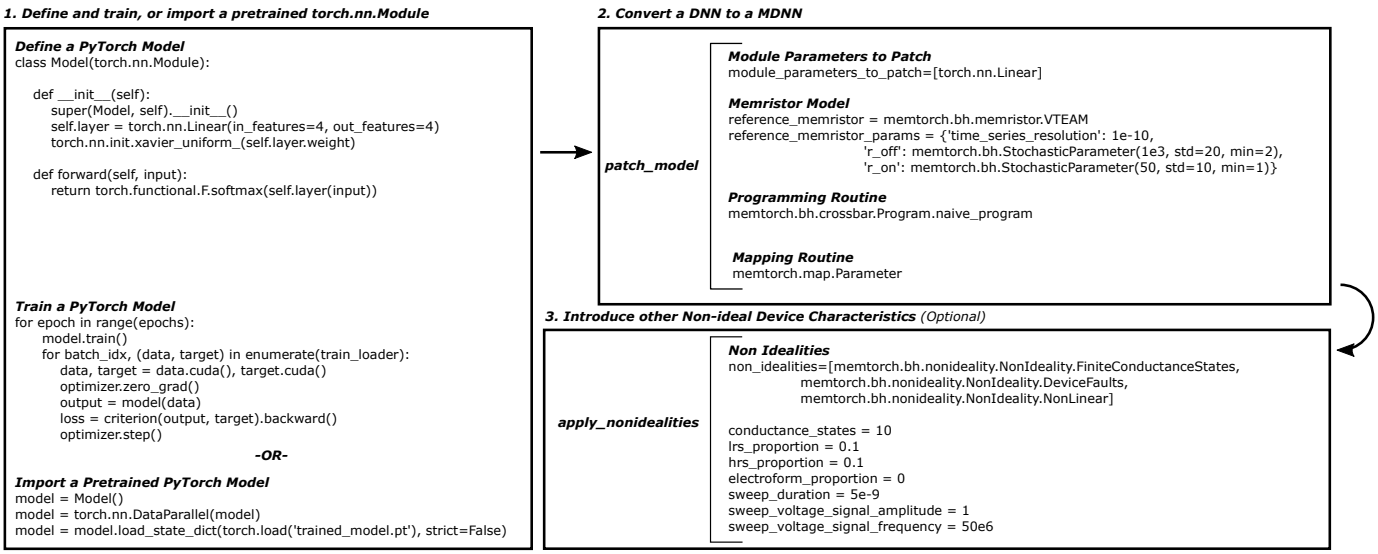

Fig. 1: A typical use-case workflow in *MemTorch*, used to convert *torch.nn.Linear* layers within a *torch.nn.Module* to memristive-equivalent layers employing 1T1R crossbars and a double-column parameter representation scheme. The VTEAM model was used to model memristive devices, which adopted the TEAM [20] model's parameters, with a linear dependence on w. R_{OFF} and R_{ON} were sampled from a normal distribution with $\bar{R}_{\text{OFF}} = 1000, \sigma = 20$, and $\bar{R}_{\text{ON}} = 50, \sigma = 10$. Three other non ideal device characteristics were also accounted for including a finite number (10) of discrete conductance states, device faults, and non-linear I/V device behavior. Here, *model.tune_()* can be called to tune memristive layers within the patched model using linear regression.

a deep memristive learning system considering non-idealities that can be customized for any device model.

The paper is structured as follows. Section II compares *MemTorch* to other related works. Section III presents a brief overview of the models, algorithms, and methods used within *MemTorch*. Section IV details the package structure of *MemTorch*, highlights its extendability, and discusses its release management and ongoing development. Section V details our current approach to modeling non-ideal device characteristics implemented within *MemTorch*. Section VI demonstrates the significance of *MemTorch* platform by using it to perform novel large-scale simulations taking into account customizable device non-idealities and presenting a case study. Section VII draws concluding remarks and sheds light on the future of *MemTorch*.

II. Related Work

We compare *MemTorch* to other memristor-based DNN frameworks and inference accelerators, which are software-based and do not rely on SPICE modeling. The comparison is shown in Table I.

Software-based frameworks and inference accelerators use a combination of programming languages such as C++, CUDA, MATLAB and Python to simulate the behavior of memristive devices during inference. RAPIDNN [22] tracts representative operands of a DNN model using clustering methods to optimize the model for in-memory processing and maps the extracted operands and their precomputed results into accelerator memory blocks. NSIM [23] proposes a hierarchical structure for memristor-based neuromorphic computing accelerators, with interfaces for customization. PUMA [9] uses general purpose execution units to enable the acceleration of a wide variety of ML inference workloads through a specialized Instruction Set Architecture (ISA) to retain the efficiency of in-memory computing and analog circuitry, without compromising programmability.

DL-RSIM [24] simulates the error rates of every sum-of-products computation in memristor-based accelerators, and injects the errors in targeted TensorFlow-based neural network models. PipeLayer [5] proposes highly parallel designs based on the notion of parallelism granularity and weight replication. Tiny but accurate [25] and An Ultra-Efficient Memristor-Based DNN Framework [26] propose memristor-based DNN frameworks that combine both structured weight pruning and quantization by incorporating the Alternating Direction Method of Multipliers (ADMM) algorithm for better pruning and quantization performance. Finally, the Non-ideal Resistive Synaptic Device Characteristic Simulation Framework [27] investigates the impact of non-ideal characteristics of resistive synaptic devices on MDNNs using the TensorFlow framework.

While all of the aforementioned frameworks support pre-trained DNN conversion, only DL-RSIM [24] and the Non-ideal Resistive Synaptic Device Characteristic Simulation Framework [27] support GPU-accelerated inference and parameter mapping. Furthermore, no current memristor-based DNN framework or inference accelerator supports CUDA or OpenCL kernel execution on heterogeneous platforms. Tiny but accurate [25] and the Ultra-Efficient Memristor-Based DNN Framework [26] are the only frameworks that are open-source, however, the source code for both frameworks is only available statically by Google Drive and Application Programming Interfaces (APIs) are not currently available.

TABLE I: Comparison of *MemTorch* to other memristor-based DNN simulation frameworks and inference accelerators.*Does not support GPU-accelerated inference and/or parameter mapping.†Models are shared using Google Drive without Application Programming Interfaces (APIs).

Simulation framework	Open-source	GPU	Pretrained DNN conversion	Programming language(s)
RAPIDNN [22]		✓*	✓	C++
MNSIM [23]			✓	Not Specified
PUMA [9]			✓	C++
DL-RSIM [24]		✓	✓	Python
PipeLayer [5]		✓*	✓	C++
Tiny but Accurate [25]	✓†		✓	MATLAB
Ultra-Efficient Memristor-Based DNN Framework [26]	✓†		✓	C++, MATLAB
Non-ideal Resistive Synaptic Device Characteristic Simulation Framework [27]		✓	✓	Python
MemTorch	✓	✓	✓	Python, C++, CUDA

MemTorch is, to the best of our knowledge, the only software-based memristor DNN simulation framework that:

1) Has a publicly accessible ReadTheDocs API that is automatically generated from source-code using docstrings;
2) Is fully open-source and distributed using the Python Package Index (PyPi) repository;
3) Adopts a highly-modular modernized software engineering methodology;
4) Natively supports CPU and GPUs during inference and tuning processes;
5) Can be extended, modified, and enhanced by all openly using *git*.

III. Preliminaries

This section reviews and presents the algorithms and models that are currently built into *MemTorch* and will be used to perform experimental simulations in Section VI.

A. Memristive Device Models

Within *MemTorch* we use two memristive device models for our simulations. These include the linear ion drift model by [28], and the VTEAM model by [16], which is a general model for voltage-controlled memristors that can be used to fit any experimental device data. These models are further explained below.

1) Linear ion drift model: An ideal linear ion drift model for a memristor is proposed in [28] and described by (1) – (3). Such a device has a physical width D, and contains two regions of width w and $D - w$, which are regions with high concentration of dopants and oxides, respectively. Here, μ_v is the ion mobility, R_{ON} is the resistance when $w(t) = D$, R_{OFF} is the resistance when $w(t) = 0$, $f(w)$ is a window function [29] that constraints $w \in [0, D]$, and p is a positive integer.

$$\frac{dw(t)}{dt} = \mu_v \frac{R_{\text{ON}}}{D} i(t) f(w), \tag{1}$$

$$v(t) = (R_{\text{ON}} \frac{w(t)}{D} + R_{\text{OFF}}\Big[1 - \frac{w(t)}{D}\Big])i(t), \tag{2}$$

$$f(w) = 1 - [\frac{2w}{D} - 1]^{2p}. \tag{3}$$

We use finite differences to obtain a numerical solution for the model's state variable w in all forthcoming simulations using: $R_{\text{ON}} = 1000\Omega$, $R_{\text{OFF}} = 2000\Omega$, $d = 10e^{-9}$m, $p = 2$, with $dt = 1e^{-3}$s.

2) VTEAM memristor model: The VTEAM model, presented in [16] and described by (4) – (7), is a general, simple and accurate model for voltage-controlled memristors. Modeled devices have a physical width w_{OFF}. R_{OFF} and R_{ON} are the maximum and minimum device resistances. $f_{\text{OFF}}(w)$ and $f_{\text{ON}}(w)$ are window functions, which constrain w to $\in [w_{\text{ON}}, w_{\text{OFF}}]$. Additionally, v_{OFF} and v_{ON} are two threshold voltages, while k_{OFF}, k_{ON}, α_{OFF} and α_{ON} are fitting parameters.

$$\frac{dw(t)}{dt} = \begin{cases} k_{\text{OFF}}(\frac{v(t)}{v_{\text{OFF}}} - 1)^{\alpha_{\text{OFF}}} f_{\text{OFF}}(w), 0 < v_{\text{OFF}} < v, \\ 0, v_{\text{ON}} < v < v_{\text{OFF}}, \\ k_{\text{ON}}(\frac{v(t)}{v_{\text{ON}}} - 1)^{\alpha_{\text{ON}}} f_{\text{ON}}(w), v < v_{\text{ON}} < 0. \end{cases} \tag{4}$$

A linear dependence of the resistance and w can be achieved, where the current-voltage relationship is described by (5).

$$v(t) = i(t)\Big[R_{\text{ON}} + \frac{R_{\text{OFF}} - R_{\text{ON}}}{w_{\text{OFF}} - w_{\text{ON}}}(w - w_{\text{ON}})\Big]. \tag{5}$$

Alternatively, an exponential dependence on w can be assumed, as in [30], described by (6) and (7), where λ is a fitting parameter determined by R_{OFF} and R_{ON}.

$$v(t) = i(t) R_{\text{ON}} e^{\frac{\lambda}{w_{\text{OFF}} - w_{\text{ON}}}(w - w_{\text{ON}})}, \tag{6}$$

$$e^{\lambda} = \frac{R_{\text{OFF}}}{R_{\text{ON}}}. \tag{7}$$

We use finite differences to obtain a numerical solution for the model's state variable in all forthcoming simulations using the TEAM [20] model's parameters in [16], with $dt = 1e^{-10}$s.

B. Window Functions

Within memristive device models, window functions are widely employed to restrict the changes of the internal state variables to specified intervals [31]. *MemTorch* currently natively supports the Biolek [29], Jogelkar [32], and Prodromakis [33] window functions, and can easily be extended to support others.

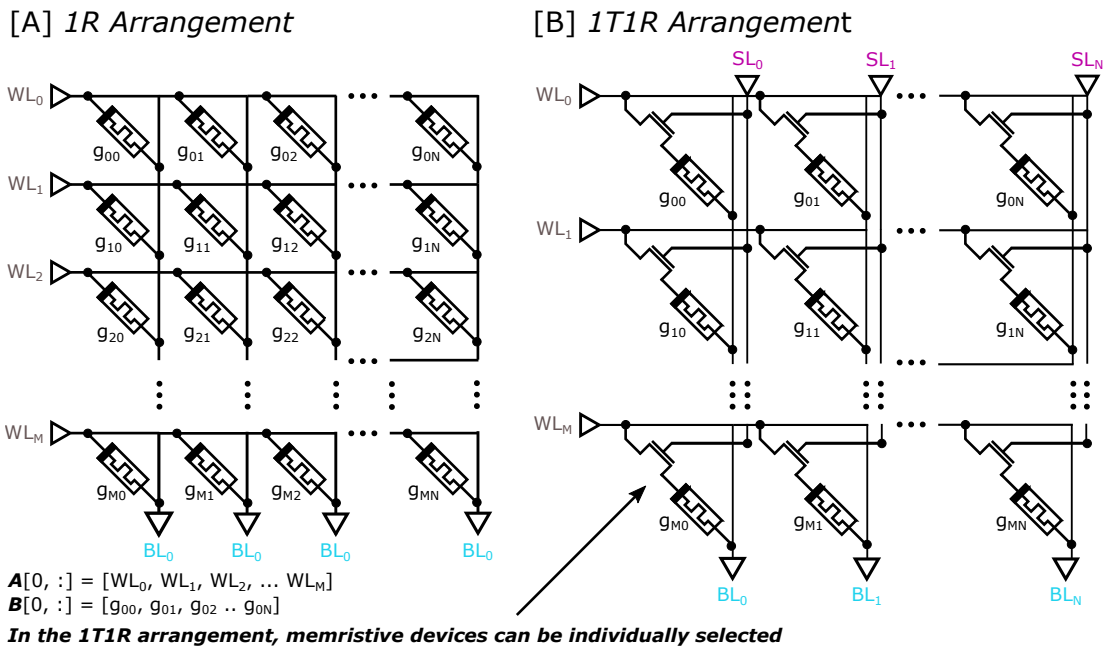

Fig. 2: Depiction of an $M \times N$ [A] 1R crossbar architecture and a [B] 1T1R crossbar architecture. Matrix-vector and matrix-matrix multiplication can be performed by encoding and presenting input vector or matrix $\boldsymbol{A}$ as voltage signals to each row of the crossbar's Word Lines (WLs). As shown in [A], assuming a linear I/V relationship, the total current in each column's Bit Line (BL) is linearly proportional with the sum of the multiplication of the WL voltages and conductance values in that column, i.e., $\boldsymbol{BL}[0,:] \propto \boldsymbol{A}[0,:] \times \boldsymbol{B}$. In the 1T1R arrangement [B], individual memristive devices can be selected using Select Lines (SLs).

C. Memristive Crossbar Architectures

Memristive devices can be arranged within crossbar architectures to perform VMMs, which are used extensively in forward and backward propagations within DNNs. There are two commonly used crossbar architecture configurations, namely 1-Transistor 1-Resistor (1T1R), and 1-Resistor (1R), which are both depicted in Fig. 2. In 1T1R arrangements, one transistor is used to select and control each memristive device, whereas in 1R arrangements, rows and columns of memristive devices are positioned perpendicular to each other, with memristive devices sandwiched in-between.

The product of a vector and a matrix or, in a more general form, two matrices, $\boldsymbol{A}$ of size $(M \times C)$ and $\boldsymbol{B}$ of size $(C \times N)$, can be computed using a crossbar-architecture, as illustrated in Fig. 2, where $\boldsymbol{A}$ represents input voltage signals and $\boldsymbol{B}$ is encoded within the crossbar as memristor conductances. As the output current of each column is linearly proportional to the elements of $\boldsymbol{AB}$, a linear constant, K, is used to correlate the current of columns in each crossbar accordingly. By separately presenting each row of $\boldsymbol{A}$ to the crossbar through Word Lines (WLs), all rows of $\boldsymbol{AB}$ can be computed.

As memristors cannot be programmed to have negative conductances, within MDNNs, weight matrices can either be represented using two crossbars per layer [34], as described by (8),

$$\boldsymbol{AB} = K \sum_{i=0}^{C} \boldsymbol{A}[i,:](g_{\text{pos}}[i,j] - g_{\text{neg}}[i,j]), \text{for } j = 0 \text{ to } N, \tag{8}$$

or using a singular crossbar per layer [35], [36] using complex weight mapping algorithms or current mirrors, as described by (9).

$$\boldsymbol{AB} = K \sum_{i=0}^{C} \boldsymbol{A}[i,:](g[i,j] - g_m), \text{for } j = 0 \text{ to } N. \tag{9}$$

For the single-column case, the current through g_m, used to mirror a constant current $-(\bar{R}_{\text{ON}} + \bar{R}_{\text{OFF}})/2$ to each crossbar, is copied to each column and subtracted from all memristor columns. Here, a constant current can easily be realized using a diode-connected NMOSFET by adjusting the NMOSFET channel width so that it has a passive resistance g_m. From here on, we refer to the weight matrix representation methodology adopted, that is, weather two crossbars are used per layer, one crossbar is used per layer, or another configuration is used to represent weight matrices, as the parameter representation scheme.

1) Memristor crossbar programming: The conductance of memristive devices can be altered between a low resistance state R_{ON} and a high resistance state R_{OFF}, by applying programming voltage pulses with different intervals and amplitudes. While individual devices within crossbars can be selected and programmed within 1T1R cells, in 1R arrangements, when a voltage is applied to a specific device, a non-zero voltage (usually half that of the nominal programming pulse amplitude) is applied to all other devices in the same row and column. Consequently, various multistage programming [37]–[40] and corrective methods [1], [2], [41], which can use analog voltage wave-forms, are often used to ensure

the difference between the programmed conductance states and the conductance states-to-program are within an acceptable tolerance.

2) Memristor crossbar tuning: The total current of each column in an ideal memristive crossbar is linearly proportional to the output elements of the VMM resultant matrix. Consequently, after each DNN layer's weights are programmed into a crossbar, linear regression can be used to correlate the output current of each column with any desired output to determine K for the crossbar, given a randomly generated input matrix that is sufficiently large. On account of device-device variations and device failures, further tuning is often required to recover accuracy loss and mitigate variances between intended and actual device conductance values. Tuning methods can either be used pre-programming [42], to improve robustness and reduce susceptibility to error, or post-programming by retraining device-specific conductance values [25].

3) Memristor crossbar weight mapping: Weights within unrolled convolutional layers [43] and linear layers can be mapped to equivalent conductance values using (10). When two crossbars are used to represent a layer's weights, for the crossbar representing positive weights, $\sigma(\boldsymbol{w}) = \boldsymbol{w}[\boldsymbol{w} \geqslant 0]$. For the crossbar representing negative weights, $\sigma(\boldsymbol{w}) = \boldsymbol{w}[\boldsymbol{w} \leqslant 0]$. When a single crossbar is used to represent a layer's weights, $\sigma(\boldsymbol{w}) = \boldsymbol{w} - g_m$.

$$\boldsymbol{g}[i,j] = \frac{(R_{\text{ON}} - R_{\text{OFF}})(\sigma(\boldsymbol{w})[i,j] - \mathbf{w}_{\text{min}})}{|\boldsymbol{w}|_{\text{max}} - \mathbf{w}_{\text{min}}} + R_{\text{OFF}}. \quad (10)$$

To reduce the inner weight gap in a given device, Algorithm 1, which was proposed by [8], can be used to exclude a small proportion, p_L, of weights with the absolute largest values to reduce inaccuracies of non-ideal memristive devices.

IV. MemTorch Package Structure

The *MemTorch* simulation framework is programmed in C++, CUDA and Python, with a Python interface. Performance critical tasks are performed using either C++ or CUDA, for CPU or GPU execution, respectively; otherwise Python is used. *MemTorch* relies heavily on the open source PyTorch [44] ML framework, which contains data structures named *tensors* for multi-dimensional vectors, for which mathematical operations are well-defined for using in both CPUs and GPUs. *MemTorch* uses the C++ and Python PyTorch APIs extensively to abstract low-level operations. Consequently, it supports native CPU and GPU operations. Table II depicts the current structure of the *MemTorch* simulation framework, which is publicly distributed using the Python Package Index (PyPI) repository. We detail the release management of *MemTorch* fully in Section IV-B

Algorithm 1 Memristor crossbar programming algorithm.

Input: Array containing all continuous weights in a given layer, $\mathbf{w}$, HRS/LRS ratio, p_L.

Output: Equivalent memristive crossbars conductance values, $\mathbf{g}$, indexed using i and j.

$\mathbf{w}$ = abs($\mathbf{w}$)
$\mathbf{w}$ = descending_order($\mathbf{w}$)
s = size($\mathbf{w}$)
index = int($p_L \cdot s$)
$\mathbf{w}_{\text{max}} = \mathbf{w}$[index]
$\mathbf{w}_{\text{min}} = \mathbf{w}_{\text{max}}/(\mathbf{R}_{\text{OFF}}/\mathbf{R}_{\text{ON}})$
$\mathbf{w} = \text{clip}(\mathbf{w}, \mathbf{w}_{\text{min}}, \mathbf{w}_{\text{max}})$
$\boldsymbol{g}[i,j] = \frac{(R_{\text{ON}} - R_{\text{OFF}}) \cdot (\sigma(\boldsymbol{w})[i,j] - \mathbf{w}_{\text{min}})}{|\boldsymbol{w}|_{\text{max}} - \mathbf{w}_{\text{min}}} + R_{\text{OFF}}$

A. MemTorch Sub-modules

MemTorch is made up of six distinct sub-modules: *MemTorch.bh*, *MemTorch.mn*, *MemTorch.cpp*, *MemTorch.cu*, *MemTorch.examples*, and *MemTorch.map*. General utility functions, such as data loaders or generic functions, are grouped within *MemTorch.utils*. Below sections provide an explanation for each of these sub-modules.

1) The MemTorch.bh sub-module: The *MemTorch.bh* sub-module encapsulates all crossbar models, crossbar programming methods, memristor models, memristor model window functions, models for all non-ideal device characteristics, and support for stochastic parameters. Currently supported crossbar and memristor models are presented in Section III. We fully detail our approach to modeling non-ideal device characteristics and stochastic parameters in Section V.

2) The MemTorch.mn sub-module: The *MemTorch.mn* sub-module mimics *torch.nn* and defines memristive *torch.nn.Module* children classes. *MemTorch.mn* currently extends *nn.Linear* and *nn.Conv2d*. *MemTorch.mn.Module.patch_model* can be used to either instantiate new layers, or to patch existing instances. *MemTorch.mn.Module.patch_model* iterates through and patches all named modules within classes extending from *torch.nn.Module* and adds a *self.tune_()* function to the class instance of the model that automatically patches each selected named module from the *module_parameter_patches* dictionary.

3) The MemTorch.cpp and MemTorch.cu sub-modules: The *MemTorch.cpp* sub-module encapsulates all Python-wrapped C++ extensions, whereas the *MemTorch.cu* sub-module encapsulates all Python-wrapped CUDA extensions. Currently, *MemTorch* uses one C++/CUDA quantization extension to represent the finite number of stable conductance states that non-ideal memristive devices have. Within *MemTorch*, execution of CUDA kernels on GPU(s) are massively parallelized using 128 CUDA threads and max(min($(n + 127)/128$, 4096), 1) CUDA blocks, where n is the number of elements of the tensor to quantize.

4) The MemTorch.examples sub-module: The *MemTorch.examples* sub-module encapsulates general-usage examples and scripts.

5) The MemTorch.map sub-module: The *MemTorch.map* sub-module encapsulates all mapping and tuning algorithms used when programming and tuning memristive crossbar arrays.

TABLE II: The current directory structure of the *MemTorch* package.[1]Denotes a Python class.[2]Denotes a collection of Python functions.[3]Denotes a CUDA header. [4]Denotes a collection of CUDA kernels.[5]Denotes a collection of C/C++ functions. [6]Denotes a Jupyter notebook.

***MemTorch* PyPI package directory structure**	**Description**
▾*MemTorch*	-
▾*MemTorch*.bh	Behavioral and experimental models
▾*MemTorch*.bh.crossbar	Crossbar models
MemTorch.bh.crossbar.Crossbar[1]	Class used to model memristor crossbar behavior
MemTorch.bh.crossbar.Program[2]	A collection of memristor crossbar programming methods
▾*MemTorch*.bh.memristor	Memristive device models
▾*MemTorch*.bh.memristor.Memristor[1]	Abstract class for memristive device models
MemTorch.bh.memristor.LinearIonDrift[1]	Ideal linear ion drift [28] memristor model
MemTorch.bh.memristor.VTEAM[1]	VTEAM [16] memristor model
▾*MemTorch*.bh.window	A collection of window functions
MemTorch.bh.window.Biolek[1]	The Biolek [29] window function
MemTorch.bh.window.Jogelkar[1]	The Jogelkar [32] window function
MemTorch.bh.window.Prodromakis[1]	The Prodromakis [33] window function
▾*MemTorch*.bh.nonideality	Device non-idealities
MemTorch.bh.nonideality.NonIdeality[1]	Enum class and method to apply device non-idealities
MemTorch.bh.nonideality.FiniteConductanceStates[2]	Finite conductance states non-ideality
MemTorch.bh.nonideality.DeviceFaults[2]	Device faults non-ideality
MemTorch.bh.StochasticParameter[2]	A collection of functions to create and interpret stochastic parameters
▾*MemTorch*.cpp	C++ extensions
MemTorch.cpp.quantize[1]	Quantization extension used to model a finite number of conductance states
MemTorch.cpp.quantize.quant[4]	C++ quantization kernel
▾*MemTorch*.cu	CUDA extensions
MemTorch.cu.quantize[1]	Quantization extension used to model a finite number of conductance states
MemTorch.cu.quantize.gpu[3]	CUDA header
MemTorch.cu.quantize.quant[4]	CUDA quantization kernel
MemTorch.cu.quantize.quant_cuda[5]	Python and C++ CUDA quantization kernel wrapper
▾*MemTorch*.mn	Torch.nn equivalent
MemTorch.mn.Module[2]	Functions to patch pre-trained DNNs
MemTorch.mn.Linear[1]	Memristive linear layer
MemTorch.mn.Conv2d[1]	Memristive conv2d layer
▾*MemTorch*.map	Crossbar mapping algorithms
MemTorch.map.Module[2]	Module crossbar mapping algorithms
MemTorch.map.Parameter[2]	Parameter crossbar mapping algorithms
▾*MemTorch*.examples	Usage examples
MemTorch.examples.GeneralUsage[6]	General usage examples
MemTorch.examples.Tutorial[6]	Jupyter notebook tutorial
MemTorch.examples.TutorialModel[2]	Network architecture used in the Jupyter notebook tutorial
MemTorch.utils[2]	Utility functions

B. Release Management

MemTorch is released using the *PyPi* repository, and can easily be installed using *pip*, Python's *de facto* package-management system, using *pip install MemTorch* (with CUDA support) or *pip install memtorch-cpu* (without CUDA support). As *MemTorch* is completely open-source, it can also be installed directly from its source code accessible at https://github.com/coreylammie/MemTorch. In addition to hosting all of *MemTorch*'s source code, its GitHub repository houses documentation, an API that is automatically generated from docstrings within source files using *ReadTheDocs*, a *gitter* chat-room for open discussion, and *Jupyter* notebooks containing examples and a tutorial[2] that demonstrates the usage of PyTorch and *MemTorch* to design, build, train, and test MDNNs using non-ideal memristive devices. To encourage collaboration and community engagement, current issues and feature requests are automatically tracked using *git*. The *Travis* Continuous Integration (CI) service is used to automatically build and test all pull and push requests using several unit tests. Incremental releases are also made using *git*, in which detailed release notes are made widely available. *MemTorch* is licensed under the GNU General Public License v3.0.

[2]https://github.com/coreylammie/MemTorch/blob/master/memtorch/examples/Tutorial.ipynb

V. Modeling Non-ideal Device Characteristics

Non-ideal device characteristics can either be encapsulated within device-specific memristive models, or introduced using the *MemTorch.bh.nonideality* sub-module. This sub-module can currently be used to introduce four non-ideal device characteristics to memristive device models: device-device variability, finite number of discrete conductance states, device failure, and non-linear I/V device characteristics. We leave native support for modeling other non-ideal device characteristics and disruptive current sneak paths using *MemTorch* to future extensions and/or branches. These or any other emerging or known non-idealities can also be openly added to *MemTorch* by the community. Three non-ideal device characteristics that are currently supported by MemTorch are shown in Fig. 3. Fig. 3[A] depicts typical non-linear I/V device characteristics using a set-reset curve and an inset hysteresis loop. Fig. 3[B] demonstrates gradual switching, which is used to achieve a

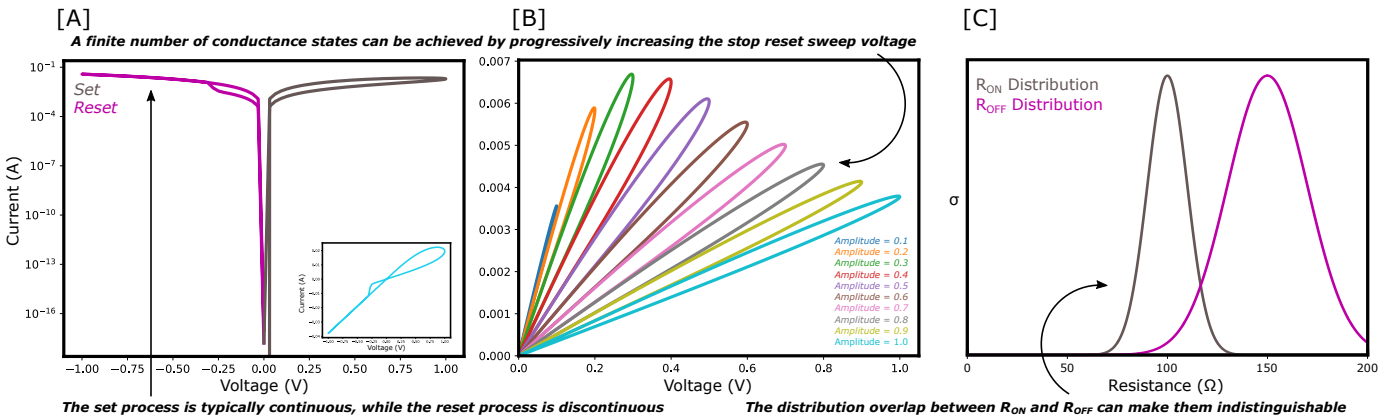

Fig. 3: Depiction of [A] device I/V characteristics, and [B] reset voltage double-sweeps demonstrating gradual switching from R_{ON} to R_{OFF}, which can be used to achieve 10 finite stable conductance states for the VTEAM model using the TEAM [20] model's parameters, with a linear dependence on w, achieved using sinusoidal signals with a fixed frequency of 50 MHz. [C] shows distributions of R_{ON} and R_{OFF}, which are caused by device-device variability, for a memristive device with $\bar{R}_{\text{ON}} = 100\Omega$ and $\bar{R}_{\text{OFF}} = 150\Omega$. In [C], overlapped regions are indistinguishable from each other.

finite number of stable conductance states, and Fig. 3[C] shows overlapping distributions of R_{ON} and R_{OFF}, which is caused by device-to-device variability.

A. Device-to-device Variability

Device-to-device variability is modeled stochastically using *MemTorch.bh.StochaticParameter*. Stochastic parameters are generated using the *MemTorch.bh.StochaticParameter.StochaticParameter()* function, which accepts an arbitrary number of keyword arguments, that are used to sample from a *torch.distributions* distribution each time a device model is instantiated. As an example, if the *torch.distributions.normal.Normal* normal distribution is used, ***kwargs*, that is used to pass a keyworded, variable-length argument dictionary to a function is required to include a mean value, minimum value, maximum value, and a standard deviation value. In Section VI, we use stochastic parameters to sample R_{ON} and R_{OFF} from a normal distribution with $\sigma R_{\text{ON}} = \sigma$ and $\sigma R_{\text{OFF}} = 2\sigma$. $\sigma R_{\text{OFF}} > \sigma R_{\text{ON}}$, as the variability of R_{OFF} has been demonstrated to be larger than R_{ON} [45]. As depicted in Fig. 3[C], device-device variability can cause the distribution of R_{ON} and R_{OFF} to overlap, resulting in R_{ON} and R_{OFF} occupying the same conductance regions.

B. Cycle-to-cycle Variability

Cycle-to-cycle (C2C) current variability [46] is modeled stochastically, similarly to device-to-device variability, using stochastic parameters for R_{ON} and R_{OFF}. *MemTorch.bh.nonideality.DeviceFaults.apply_cycle_variability()* is used to sample R_{ON} and R_{OFF} from a normal distribution with $\sigma R_{\text{ON}} = \sigma$ and $\sigma R_{\text{OFF}} = 2\sigma$ after each SET RESET cycle.

C. Finite Number of Discrete Conductance States

Realistic memristive devices are non-ideal and have a finite number of stable discrete electrically switchable conductance states, bounded by a low-conductance semiconducting state R_{OFF}, and a high-conductance metallic state, R_{ON} [47]. Previous works have investigated evenly spaced conductance or resistance states, and have demonstrated that, assuming they are relatively uniformly distributed, the spacing between states is not critical [8].

Therefore, deterministic discretization [48] can be used to represent a finite number of electrically switchable conductance states, as depicted in Fig. 3[B]. In order to efficiently quantize a tensor to a defined finite number of quantization states, in which each element can have a different range, CUDA kernels are used to perform a binary search on sorted tensors (generated using the *linspace* algorithm in C++) containing defined quantization states in $\mathcal{O}(n\log(n))$, where n is the number of quantized states.

PYBIND11_MODULE() is a function within the *pybind11* python library [49] that exposes C++ types in python to enable seamless operability between C++11, CUDA, and Python. This function is used within *MemTorch* to overload function pointers, so that the minimum and maximum arguments in *quantize()* can either be float values, or tensors of the same shape of the tensor to quantize, i.e., *quantization.quantize(torch.zeros(shape).uniform_(0, 1).cuda(), conductance_states, 0, 1)* and *quantization.quantize(torch.zeros(shape).uniform_(0, 1).cuda(), conductance_states, torch.zeros(shape).uniform_(0, 0.1).cuda(), torch.zeros(shape).uniform_(0.9, 1).cuda())* are both valid function calls.

D. Device Failure

Memristive devices are susceptible to failure, by either failing to eletroform at a pristine state, or becoming stuck at high or low resistance states [8]. *MemTorch* incorporates a specific function for accounting for device failure in simulating DL systems. Given a *nn.Module*, *MemTorch.bh.nonideality.DeviceFaults.apply_device_faults()* sets the conductance of a proportion of devices within

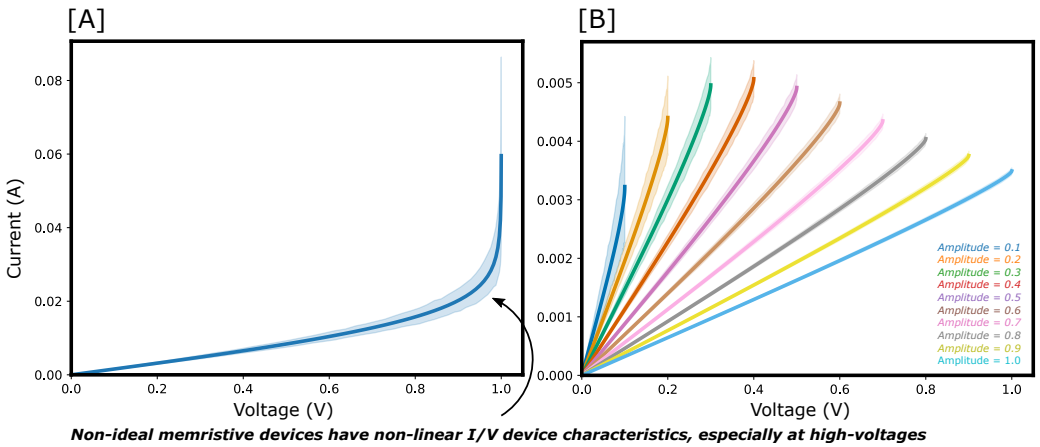

Fig. 4: Non-linear I/V characteristics for 100 devices (instances) of the VTEAM model using the TEAM [20] model's parameters, with a linear dependence on w, achieved using sinusoidal signals with a fixed frequency of 50 MHz. R_{ON} and R_{OFF} were stochastically sampled from a normal distribution with $\bar{x} = 50, \sigma = 25$, and $\bar{x} = 1000, \sigma = 50$, respectively. [A] depicts I/V characteristics for devices with an infinite number of discrete conductance states. [B] depicts I/V characteristics for devices with a finite number of discrete conductance states.

each crossbar to R_{ON} or R_{OFF}. It is assumed that the total proportion of devices set to R_{OFF} is equal to the proportion of devices that fail to eletroform at pristine states plus the proportion of devices stuck at a high resistance state. However, these proportions and the ratio of device failures can be manipulated as desired within *MemTorch*. Devices are chosen at random using *np.random.choice()*.

E. Non-linear I/V Characteristics

Non-ideal memristive devices have non-linear I/V device characteristics, especially at high voltages, which are difficult to accurately and efficiently model [8]. We demonstrate these characteristics using Fig. 4[A], by depicting the I/V curve of the VTEAM model between 0–1V using the TEAM [20] model's parameters. The *MemTorch.bh.nonideality.NonLinear.apply_non_linear()* function within *MemTorch* can be used to efficiently model non-linear device I/V characteristics during inference for devices with an infinite number of discrete conductance states, and for devices with a finite number of conductance states. For cases where devices are not simulated using their internal dynamics, it is assumed that the change in conductance during read cycles is negligible.

1) Devices with an infinite number of discrete conductance states: The *MemTorch.bh.nonideality.NonLinear.apply_non_linear()* function within *MemTorch* uses two methods to efficiently model non-linear device I/V characteristics for devices with an infinite number of discrete conductance states during inference:

1) During inference, each device is simulated for a single timestep, *device.time_series_resolution*, using *device.simulate()*.
2) Post weight mapping and programming, the I/V characteristics of each device are determined using a single reset voltage sweep. The I/V characteristics of each device are stored, and used as LUTs to compute device output currents during inference.

2) Devices with a finite number of discrete conductance states: The *MemTorch.bh.nonideality.NonLinear.apply_non_linear()* function within *MemTorch* effectively models non-linear I/V characteristics for devices with a finite number of discrete conductance states by determining the I/V characteristics of each device post weight mapping and programming during several single reset voltage sweeps. Fig. 4[B] depicts sweeps for 100 stochastic devices with 10 finite discrete conductance states. These are stored and used as LUTs to compute device output currents during inference, where each I/V curve corresponds to each finite discrete conductance state. In Fig. 4[B], the smallest voltage amplitude corresponds to the finite conductance state closest to R_{ON}, whereas the largest voltage amplitude corresponds to the finite conductance state closest to R_{OFF}.

VI. Simulation Results

In this section we use *MemTorch* to perform novel large-scale simulations, and present a case study that demonstrates *MemTorch*'s functionality to simulate a near-sensor in-memory computing system for seizure detection.

A. Novel Simulations

In this subsection we use *MemTorch* to perform novel large-scale simulations. For all of the forthcoming simulations, we followed the following training and test procedure. We first augmented a pretrained VGG-16 CNN trained using the CIFAR-10 training set. All convolutional and linear layers within the network were sequenced with either 2D- or 1D-batch-normalization layers to normalize outputs. The network was trained until improvement on the validation set was negligible (for 50 epochs) with a batch size of $\Im = 256$. The initial learning rate was $\eta = 1e-2$, which was decayed by an order of magnitude every 20 training epochs. Adam [50] was used to optimize network parameters and Cross Entropy

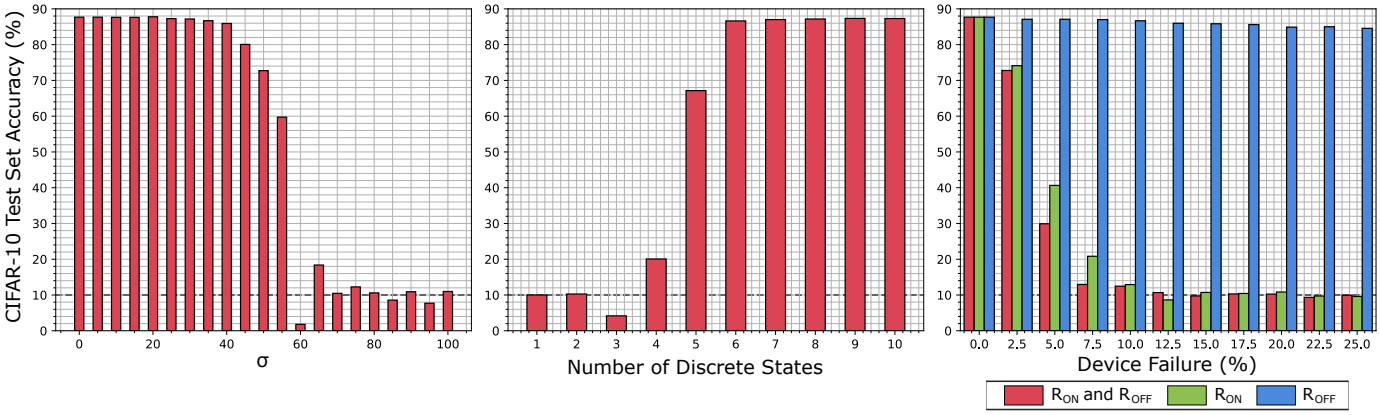

Fig. 5: Simulation results investigating the performance degradation of MDNNs benchmarked using the CIFAR-10 test set when three non-ideal device characteristics are separately accounted for. These include device-device variability (the left plot), where σ is a multiple of the standard deviation of the distributions used to sample R_{ON} and R_{OFF} (1 and 2, respectively), a finite number of conductance states (the middle plot), and device failure (the right plot), where devices become stuck at LRS and RHS states. The same percentage of failed devices become stuck at R_{ON} and R_{OFF} in the first series within the device failure subplot.

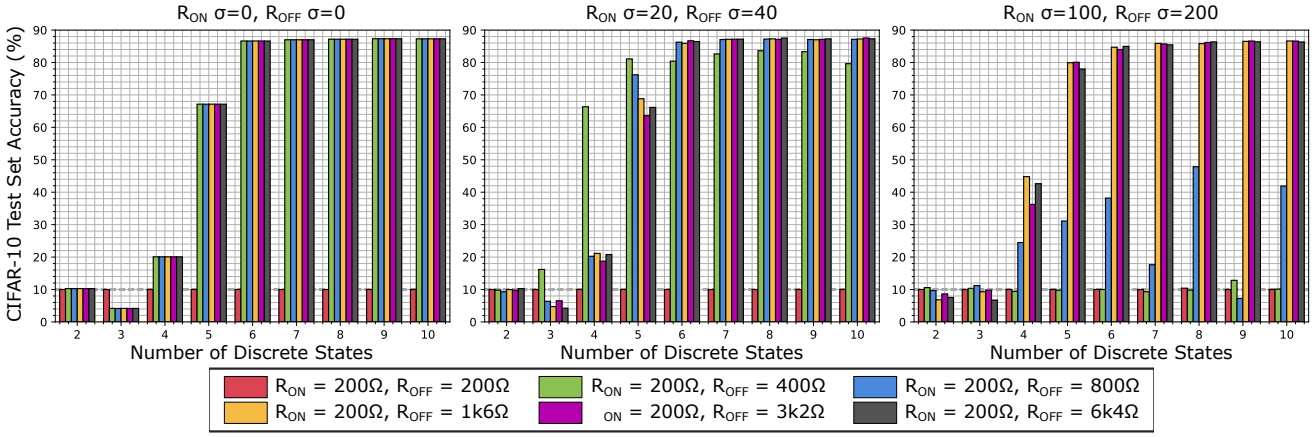

Fig. 6: Simulation results investigating the performance degradation of MDNNs benchmarked using the CIFAR-10 test set when two non-ideal device characteristics, i.e. device-device variability and a finite number of conductance states, are accounted for. All convolutional and linear layers were replaced with memristive-equivalent layers. MDNNs with different conductance ratios, $R_{\text{ON}}/R_{\text{OFF}}$, were simulated to generate each subplot.

(CE) [51] was used to determine network losses. The network achieved $> 90\%$ accuracy on the CIFAR-10 test set.

When implementing the MDNNs, each memristive layer's weights were mapped to a double column line crossbar architecture adopting a 1T1R arrangement. Linear regression was used to correlate the output current of each column and its corresponding output to determine K for each crossbar, given a randomly generated input matrix. For linear layers the random inputs had a size of (8 $\times$ in_features), while for convolutional layers the random inputs had a size of (8 $\times$ in_channels $\times$ 32 $\times$ 32). Unless otherwise stated, inputs to memristive layers were scaled from 0 to 1, to emulate voltage signals between 0V and 1V, which were applied to the word-lines of each memristive crossbar. All device models originated from the VTEAM model using the TEAM [20] model's parameters, with a linear dependence on w, and $\bar{R}_{\text{ON}}$ = 200Ω and $\bar{R}_{\text{OFF}}$ = 500Ω.

In Fig. 5, we investigate the performance degradation that is observed due to device-device variability, a finite number of conductance states, device failure, i.e. when devices fail to electroform and become stuck at low and high resistance states. The vertical dashed line in each plot (at 10% accuracy) indicates the nominal random guessing accuracy. As can be observed from Fig. 5, as the device-device variability increases, i.e. the variability of R_{ON} and R_{OFF}, the degradation in performance accelerates. Similarly, as the number of finite conductance states decreases, the degradation in performance increases. For both cases, the transition period between a notable accuracy ($\gg 10\%$) and the nominal random guessing accuracy is relatively small. For instance, when accounting for device-device variations the CIFAR-10 test set accuracy dropped from 85.92% to 1.8%, when σ was increased from 40 to 60. When modeling a finite number of conductance states the CIFAR-10 test set accuracy dropped from 86.63% to

Fig. 7: Simulation results investigating the performance degradation of MDNNs benchmarked using the CIFAR-10 test set when three non-ideal device characteristics are accounted for- device-device variability, a finite number of conductance states, and device failure, where devices become stuck at LRS and RHS states. MDNNs with different conductance ratios, $R_{\text{ON}}/R_{\text{OFF}}$, were simulated to generate each subplot within each row. In the first row of subplots, a proportion of devices are stuck at R_{ON}. In the second row of subplots, a proportion of devices are stuck at R_{OFF}. In the third row of subplots, a proportion of devices are stuck at R_{ON}. The same proportion of devices are stuck at R_{OFF}, i.e. for all simulations within the third row the total number of devices stuck are double of that in other rows.

4.17%, when the number of finite states was decreased from 6 to 3.

Looking at the device failure simulations, it is apparent that devices stuck at R_{ON} degrade the performance to a much larger extent than devices stuck at R_{OFF} within crossbars adopting a double-column parameter representation scheme. We believe the performance degradation observed when devices become stuck at R_{ON} is caused by network weights with small magnitudes (near R_{OFF}) being falsely presented with large magnitudes, i.e, R_{ON}, which significantly decreases accuracy. On the other hand, when many devices are stuck at R_{OFF}, they represent the many near-zero weights of the network and therefore, do not significantly decrease accuracy. This is especially the case when a large proportion of network weights are near-zero, which is often the case when ridge regression is used during training. When 25% of devices become stuck at R_{OFF}, the performance degradation was only 3.12%, whereas it was 78.09% when 25% of devices become stuck at R_{ON} and 77.74% when 25% of devices become stuck at R_{ON} and R_{OFF}.

In Fig. 6, we investigate the performance degradation that two non-ideal characteristics of memristive devices, i.e. a finite number of conductance states and device-device variability introduce to the VTEAM model. Moreover, we investigate the performance of MDNNs using memristive devices with different conductance ratios, $R_{\text{ON}}/R_{\text{OFF}}$, to emphasize the importance of modeling device-specific parameters. The number of finite conductance states, n, was varied between 2 – 10, the device-device variations of R_{ON} and R_{OFF}, σ and 2σ, were $\in [0, 20, 100]$, and the $R_{\text{ON}}/R_{\text{OFF}}$ ratio was $\in [1, 2, 4, 8, 16, 32]$.

As the figure depicts, as the variation of R_{ON} and R_{OFF}, σ, increases, a larger ratio of $R_{\text{ON}}/R_{\text{OFF}}$ is required to mitigate performance degradation. Moreover, memristive devices with more finite conductive states are significantly more resilient when device variation σ, is increased and the $R_{\text{ON}}/R_{\text{OFF}}$ ratio is decreased. When R_{ON} and R_{OFF} completely overlap, i.e. they are equal to 200Ω, for all cases, the nominal random guessing accuracy is only rarely marginally surpassed.

As can be seen from Figs. 5 and 6, the maximum test set accuracy achieved using the MDNNs is smaller than that of the original DNN prior-conversion. We largely attribute this to the crude tuning methodology employed using the k scale parameters for each layer. This degradation could be largely counteracted using more complex tuning methods such as [25], however, is beyond the scope of this work.

In Fig. 7, we investigate the performance degradation that three non-ideal characteristics of memristive devices introduce to the VTEAM model. These include device-device variability, a finite number of conductance states, and device failure, where devices become stuck at LRS and RHS states. For these simulations, the number of finite conductance states, n, was varied between 2 – 10, the device-device variation of R_{ON} and R_{OFF}, σ, was varied to $[0, 20, 100]$, while the proportion of devices that become stuck at R_{ON}, R_{OFF}, and R_{ON} and R_{OFF} were $\in [0\%, 5\%, 10\%, 15\%, 20\%, 25\%]$. In total, 486 separate simulations were performed to generate Fig. 7.

As can be seen from Fig. 7, as the device-device variability, σ, is increased, when a significant proportion of devices become stuck, a large degradation in performance is observed. Similarly to Fig. 5, networks with devices stuck at R_{OFF} are much more tolerant than those with devices stuck at either R_{ON} or R_{ON} and R_{OFF}. For such cases, the degradation is significant even when only a small proportion of devices become stuck. For $\sigma = 0$, when 5% of devices become stuck at R_{ON}, performance on the CIFAR-10 test set degrades by 41.93%.

We believe that the performance stability that is observed when devices become stuck at R_{OFF} is caused by device-level conductance states grouping together tightly when a large proportion of weights are near-zero. Consequently, we note that the susceptibility to device failure for devices that become stuck at different states is largely dependent on the weight distribution of the original DNN that is converted. This can be investigated in future works.

B. A MemTorch Case Study - Seizure Detection

In this subsection a case study is presented that details simulations of a near-sensor in-memory computing system adopting Pt/Hf/Ti ReRAM devices [52] for seizure detection [53] using a permutation-invariant MDNN, which is constructed by converting linear layers from a pre-trained DNN to memristive equivalent layers employing 1T1R crossbars. A double-column scheme is used to represent network weights within memristive crossbars. In addition to converting the developed DNN to an equivalent MDNN, we introduce non-ideal characteristics to devices within the MDNN and determine the performance degradation observed. The complete and detailed process and the source code of the network conversion and introducing the non-idealities are provided in a publicly accessible complementary Jupyter Notebook[3].

1) Seizure detection dataset: A seizure detection dataset [53] with 11,500 observations is adopted. For each observation, Electroencephalogram (EEG) signals were recorded from brain activity for a duration of 23.6 seconds, and were sampled 4097 times. Within this dataset, 2,300 observations correspond to brain activity recorded from patients that have experienced a seizure. We developed a custom-dataloader, described in the aforementioned Jupyter notebook, in which we used *sklearn.preprocessing.scale()* to normalize all samples so that they have zero mean and a unit standard deviation. No other pre-processing steps are performed.

2) Network architecture: A permutation-invariant DNN with three hidden layers of 200 neurons is used. All layers are sequenced with 1D batch normalization layers.

3) Training methodology: The permutation-invariant DNN is trained until the performance stagnates on the test set (for 50 epochs), with an initial learning rate, η, of 1e-1, which is decayed by one order of magnitude every 20 epochs. A batch size, $\Im$, of 1024 is used. Cross Entropy (CE) loss and the Adam optimizer are used in conjunction to optimize network parameters. Because the dataset adopted does not have clearly defined training and test sets, the performance is determined using 5-fold cross-validation. Moreover, as the classes are not

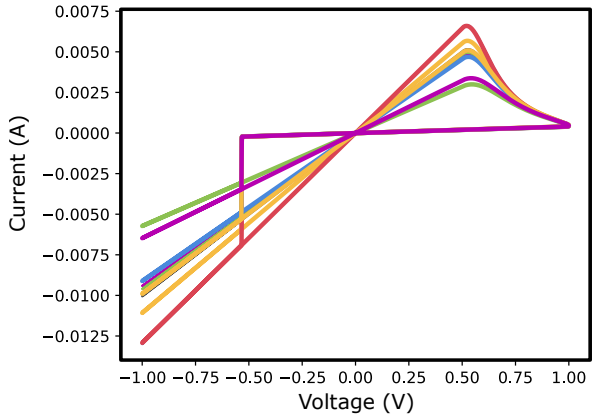

Fig. 8: I/V characteristics of a simulated VTEAM model with Pt/Hf/Ti ReRAM device parameters accounting for C2C current variability for 10 cycles. For each cycle, the current is measured when a voltage signal of $v(t) = sin(\pi t)$ for $t = 0$ to $t = 2$ is applied.

[3]https://github.com/coreylammie/MemTorch/blob/master/memtorch/examples/CaseStudy.ipynb

well-balanced, the F_1 score [54] is adopted in-place of the classification accuracy to determine performance on each of the test set folds. A baseline F_1 score of 0.6032 $\pm$ 0.1259 was recorded by determining the mean and the standard deviation of the F_1 score across all (5) folds of 100 randomly initialized networks where network parameters, $\boldsymbol{\theta}$, are sampled from uniform distributions between $-1/\sqrt{\text{size}(\boldsymbol{\theta})}$ and $1/\sqrt{\text{size}(\boldsymbol{\theta})}$. After training, an F_1 score of 0.9868 $\pm$ 0.0015 is reported for the trained DNN across all 5-folds.

4) Network conversion: A memristive device model is defined and characterized, which replaces all *torch.nn.Linear* layers within the DNN trained with equivalent 1T1R crossbar architectures adopting a double-column parameter representation scheme using our developed *memtorch.mn.Module.patch_model()* function. The converted MDNN model is then tuned using another developed function named *net.tune_()*.

During the conversion, a reference VTEAM model is instantiated using parameters from Pt/Hf/Ti ReRAM devices [52], with a linear dependence on w, to model all memristive devices within converted linear layers. In Fig. 8, C2C current variability is introduced using *MemTorch.bh.nonideality.DeviceFaults.apply_cycle_variability()*, and the *plot_hysteresis_loop()* function is used to verify the selected device's I/V characteristics. The *memtorch.bh.map.Parameter.naive_map()* function is used to convert the weights within all *torch.nn.Linear* layers to equivalent conductance values, to be programmed to the two memristive devices used to represent each weight (positive and negative) using (10).

We note that, a 1T1R arrangement is simulated here and device-level simulation of the programming routine is skipped, as each device can be individually selected. We note that if a 1R arrangement is selected for simulation, the programming routine cannot be skipped, because individual devices cannot be accurately programmed using a naive programming approach within a 1R arrangement. In which case, device-level simulation is performed for each device using *memtorch.bh.crossbar.gen_programming_signal()* and *memtorch.bh.memristor.Memristor.simulate()*, which use finite differences to model internal device dynamics.

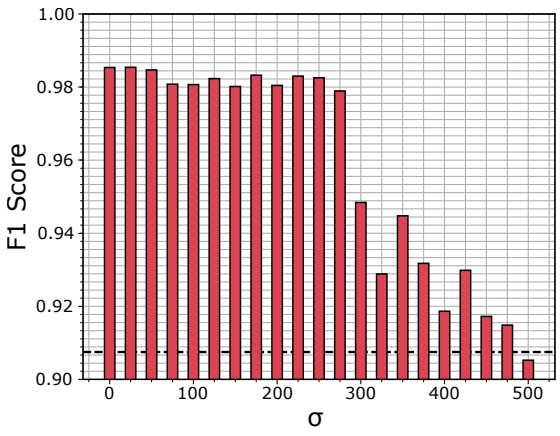

Fig. 9: Simulation results investigating the performance degradation of converted MDNNs utilizing simulated VTEAM model with Pt/Hf/Ti ReRAM device parameters when device-device variability is accounted for. σ is a multiple of the standard deviation of the distributions used to sample R_{ON} and R_{OFF} (1 and 2, respectively).

All patched *torch.nn.Linear* layers are tuned using linear regression. For this tuning, a randomly generated tensor of size (4098 $\times$ *self.in_features*) is propagated through each converted linear layer and each legacy (original DNN) layer, which is accessible using *layer.forward_legacy*. The *sklearn.linear_model.LinearRegression()* function is used to determine the coefficient and intercept of the linear relationship of each set of outputs. These values are used to define the *transform_output()* lamdba function that maps the output of each layer to their equivalent representations.

For further investigation, two baseline F_1 scores are calculated for MDNNs: 0.6060 $\pm$ 0.1304, and 0.9075 $\pm$ 0.0074. The first score shows the mean and the standard deviation of the F_1 score across all (5) folds of 100 randomly initialized MDNNs with Pt/Hf/Ti ReRAM devices. As can be seen, this is a similar F_1 score to the untrained randomized DNN discussed earlier. The second score, on the other hand, shows the mean and the standard deviation of the F_1 score across all (5) folds of 100 randomly initialized but tuned MDNNs with Pt/Hf/Ti ReRAM devices. Note that, for tuning, the legacy trained parameters of the original DNN are used. The conductance of each device in either case is randomly sampled from a uniform distribution between $1/R_{\text{OFF}}$ and $1/R_{\text{ON}}$. Unlike the two randomized MDNNs, the MDNN that its device conductances are converted from the trained DNN achieves an identical F_1 score to its DNN counterpart, i.e. 0.9868 $\pm$ 0.0015 across all 5-folds.

5) Device-to-device variability: In addition to network conversion, device-device variability is introduced using *memtorch.bh.StochasticParameter()*, by sampling R_{OFF} for each device from a normal distribution with $\bar{R}_{\text{OFF}}$ = 2k5Ω with standard deviation 2σ, and R_{ON} for each device from a normal distribution with $\bar{R}_{\text{ON}}$ = 100Ω with standard deviation σ. In Fig. 9, we progressively increase σ from 0 to 500. For $\sigma \gg 0$, R_{ON} and R_{OFF} are bounded to be positive. The vertical dashed line is used to indicate the baseline F_1 score of MDNNs, which are randomly initialized and tuned using the trained parameters of the original DNN, as previously determined in Section VI-B4, where it was also shown that the baseline F_1 score of MDNNs, which are randomly initialized but are not tuned is significantly smaller. It can be observed that as σ increases, the mean of the F_1 score across all folds decreases. The F_1 score decreases below the baseline F_1 score when $\sigma \geqslant$ 500.

6) Non-linear IV characteristics: Furthermore, non-linear IV characteristics are accounted for using *MemTorch.bh.nonideality.NonLinear.apply_non_linear()*, where *duration* = 2s, $voltage\_signal\_amplitude$ = 1V, and $voltage\_signal\_frequency$ = 0.5V. When simulating each device for a single-timestep, $1e^{-9}$, the score across all 5 folds decreases from 0.9868 $\pm$ 0.0015 to 0.9634 $\pm$ 0.0071. It is noted that the degradation observed here is largely due to the crude tuning methodology used, and that the disparity can

be decreased using pre- or more complex post-programming tuning methods.

VII. Conclusion

We presented an open-source simulation framework for deep memristive crossbar architectures entitled *MemTorch* that integrates directly with the open source PyTorch ML library and adopts a modernized software engineering approach. We compared *MemTorch* to similar works, detailed its package structure and release management, and performed experiments using it to demonstrate the convenience it provides in facilitating large-scale DNN to MDNN conversion and simulations. We also showed novel simulations investigating the performance degradation that several non-ideal memristive device characteristics introduce to large-scale DL systems. Furthermore, we presented a case-study to show that *MemTorch* can be used in designing any MDNN for any applications. We hope that *MemTorch* will be continuously used, expanded, and improved to advance the emerging field of memristive DL systems.

Acknowledgments

CL acknowledges the JCU DRTPS. MRA acknowledges a JCU Rising Star ECR Leadership Grant.

References

[1] M. Hu, H. Li, Y. Chen, Q. Wu, G. S. Rose, and R. W. Linderman, "Memristor Crossbar-Based Neuromorphic Computing System: A Case Study," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 10, pp. 1864–1878, Oct. 2014.

[2] C. Lammie, O. Krestinskaya, A. James, and M. R. Azghadi, "Variation-aware Binarized Memristive Networks," in *Proc. 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, Genoa, Italy, Nov. 2019, pp. 490–493.

[3] S. Mittal, "A Survey of ReRAM-Based Architectures for Processing-In-Memory and Neural Networks," *Machine Learning and Knowledge Extraction*, vol. 1, no. 1, pp. 75–114, 2018.

[4] G. C. Adam, A. Khiat, and T. Prodromakis, "Challenges Hindering Memristive Neuromorphic Hardware from Going Mainstream," *Nature communications*, vol. 9, no. 1, p. 5267, 2018.

[5] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning," in *Proc. IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Austin, TX, Feb. 2017, pp. 541–552.

[6] R. Hasan, T. M. Taha, and C. Yakopcic, "On-chip Training of Memristor Based Deep Neural Networks," in *Proc. International Joint Conference on Neural Networks (IJCNN)*, Anchorage, AK, May. 2017, pp. 3527–3534.

[7] O. Krestinskaya, K. N. Salama, and A. P. James, "Learning in Memristive Neural Network Architectures Using Analog Backpropagation Circuits," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 2, pp. 719–732, Feb. 2019.

[8] A. Mehonic, D. Joksas, W. H. Ng, M. Buckwell, and A. J. Kenyon, "Simulation of Inference Accuracy Using Realistic RRAM Devices," *Frontiers in Neuroscience*, vol. 13, p. 593, 2019.

[9] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W. Hwu, J. P. Strachan, K. Roy, and D. S. Milojicic, "PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference," *CoRR*, vol. abs/1901.10351, 2019. [Online]. Available: http://arxiv.org/abs/1901.10351

[10] H. Jeong and L. Shi, "Memristor devices for neural networks," *Journal of Physics D: Applied Physics*, vol. 52, no. 2, p. 023003, Oct. 2018. [Online]. Available: https://doi.org/10.1088%2F1361-6463%2Faae223

[11] H. Tsai, S. Ambrogio, P. Narayanan, R. M. Shelby, and G. W. Burr, "Recent Progress in Analog Memory-based Accelerators for Deep Learning," *Journal of Physics D: Applied Physics*, vol. 51, no. 28, p. 283001, Jun. 2018.

[12] D. Panda, P. P. Sahu, and T. Y. Tseng, "A Collective Study on Modeling and Simulation of Resistive Random Access Memory," *Nanoscale Research Letters*, vol. 13, no. 1, p. 8, Jan. 2018.

[13] I. Messaris, A. Serb, S. Stathopoulos, A. Khiat, S. Nikolaidis, and T. Prodromakis, "A Data-Driven Verilog-A ReRAM Model," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 12, pp. 3151–3162, Dec. 2018.

[14] X. Wang, B. Xu, and L. Chen, "Efficient Memristor Model Implementation for Simulation and Application," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 7, pp. 1226–1230, Jul. 2017.

[15] A. Ascoli, F. Corinto, and R. Tetzlaff, "Generalized Boundary Condition Memristor Model," *International Journal of Circuit Theory and Applications*, vol. 44, no. 1, pp. 60–84, 2016.

[16] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "VTEAM: A General Model for Voltage-Controlled Memristors," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 786–790, Aug. 2015.

[17] A. A. Emara, M. M. Aboudina, and H. A. H. Fahmy, "Corrected and Accurate Verilog-A for Linear Dopant Drift Model of Memristors," in *Proc. IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS)*, College Station, TX, Aug. 2014, pp. 499–502.

[18] S. Kvatinsky, K. Talisveyberg, D. Fliter, A. Kolodny, U. C. Weiser, and E. G. Friedman, "Models of Memristors for SPICE Simulations," in *Proc. IEEE 27th Convention of Electrical and Electronics Engineers in Israel*, Eilat, Israel, 2012, pp. 1–5.

[19] S. Kvatinsky, K. Talisveyberg, D. Fliter, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Verilog-A for Memristor Models," *CCIT Technical Report*, vol. 801, 2011.

[20] S. Kvatinsky, E. G. Friedman, A. Kolodny, and U. C. Weiser, "TEAM: ThrEshold Adaptive Memristor Model," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 60, no. 1, pp. 211–221, Jan. 2013.

[21] C. Lammie, W. Xiang, B. Linares-Barranco, and M. R. Azghadi, "coreylammie/MemTorch: Initial Release," 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3760696

[22] M. Imani, M. Samragh, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing, "RAPIDNN: In-Memory Deep Neural Network Acceleration Framework," *CoRR*, vol. abs/1806.05794, 2018. [Online]. Available: http://arxiv.org/abs/1806.05794

[23] L. Xia, B. Li, T. Tang, P. Gu, P. Chen, S. Yu, Y. Cao, Y. Wang, Y. Xie, and H. Yang, "MNSIM: Simulation Platform for Memristor-Based Neuromorphic Computing System," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 5, pp. 1009–1022, May. 2018.

[24] M. Lin, H. Cheng, W. Lin, T. Yang, I. Tseng, C. Yang, H. Hu, H. Chang, H. Li, and M. Chang, "DL-RSIM: A Simulation Framework to Enable Reliable ReRAM-based Accelerators for Deep Learning," in *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Diego, CA, Nov. 2018, pp. 1–8.

[25] X. Ma, G. Yuan, S. Lin, C. Ding, F. Yu, T. Liu, W. Wen, X. Chen, and Y. Wang, "Tiny but Accurate: A Pruned, Quantized and Optimized Memristor Crossbar Framework for Ultra Efficient DNN Implementation," *arXiv e-prints*, p. arXiv:1908.10017, Aug. 2019.

[26] G. Yuan, X. Ma, C. Ding, S. Lin, T. Zhang, Z. S. Jalali, Y. Zhao, L. Jiang, S. Soundarajan, and Y. Wang, "An Ultra-Efficient Memristor-Based DNN Framework with Structured Weight Pruning and Quantization Using ADMM," *arXiv e-prints*, p. arXiv:1908.11691, Aug. 2019.

[27] X. Sun and S. Yu, "Impact of Non-Ideal Characteristics of Resistive Synaptic Devices on Implementing Convolutional Neural Networks," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 3, pp. 570–579, 2019.

[28] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams, "The Missing Memristor Found," *Nature*, vol. 453, no. 7191, pp. 80–83, May 2008.

[29] Z. Biolek, D. Biolek, and V. BiolkovÃą, "Spice Model of Memristor With Nonlinear Dopant Drift," *Radioengineering*, pp. 210–214, 2009.

[30] M. D. Pickett, D. B. Strukov, J. L. Borghetti, J. J. Yang, G. S. Snider, D. R. Stewart, and R. S. Williams, "Switching Dynamics in Titanium Dioxide Memristive Devices," *Journal of Applied Physics*, vol. 106, no. 7, p. 074508, Oct. 2009.

[31] V. A. Slipko and Y. V. Pershin, "Importance of the Window Function Choice for the Predictive Modelling of Memristors," *CoRR*, vol. abs/1811.06649, 2018. [Online]. Available: http://arxiv.org/abs/1811.06649

[32] Y. N. Joglekar and S. J. Wolf, "The Elusive Memristor: Properties of Basic Electrical Circuits," *European Journal of Physics*, vol. 30, no. 4, pp. 661–675, May. 2009.

[33] T. Prodromakis, B. P. Peh, C. Papavassiliou, and C. Toumazou, "A Versatile Memristor Model With Nonlinear Dopant Kinetics," *IEEE Transactions on Electron Devices*, vol. 58, no. 9, pp. 3099–3105, Sep. 2011.

[34] F. Alibart, E. Zamanidoost, and D. B. Strukov, "Pattern Classification by Memristive Crossbar Circuits using ex situ and in situ Training," *Nature Communications*, vol. 4, no. 1, p. 2072, 2013.

[35] S. N. Truong and K.-S. Min, "New Memristor-based Crossbar Array Architecture with 50-% Area Reduction and 48-% Power Saving for Matrix-vector Multiplication of Analog Neuromorphic Computing," *Journal of semiconductor technology and science*, vol. 14, no. 3, pp. 356–363, 2014.

[36] J. Lee, J. K. Eshraghian, K. Cho, and K. Eshraghian, "Adaptive Precision CNN Accelerator Using Radix-X Parallel Connected Memristor Crossbars," *arXiv e-prints*, p. arXiv:1906.09395, Jun. 2019.

[37] I. E. Ebong and P. Mazumder, "Self-Controlled Writing and Erasing in a Memristor Crossbar Memory," *IEEE Transactions on Nanotechnology*, vol. 10, no. 6, pp. 1454–1463, Nov. 2011.

[38] M. S. Tarkov, "Mapping Neural Network Computations onto Memristor Crossbar," in *Proc. International Siberian Conference on Control and Communications (SIBCON)*, Omsk, Russia, May 2015, pp. 1–4.

[39] R. Hasan, C. Yakopcic, and T. M. Taha, "Ex-situ Training of Dense Memristor Crossbar for Neuromorphic Applications," in *Proc. IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, Beijing, China, 2015, pp. 75–81.

[40] K. Jo, C. Jung, K. Min, and S. Kang, "Self-Adaptive Write Circuit for Low-Power and Variation-Tolerant Memristors," *IEEE Transactions on Nanotechnology*, vol. 9, no. 6, pp. 675–678, Nov. 2010.

[41] B. Feinberg, S. Wang, and E. Ipek, "Making Memristive Neural Network Accelerators Reliable," in *Proc. IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Vienna, Austria, 2018, pp. 52–65.

[42] B. Zhang, N. Uysal, D. Fan, and R. Ewetz, "Handling Stuck-at-faults in Memristor Crossbar Arrays Using Matrix Transformations," in *Proc. 24th IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, ser. ASPDAC '19. New York, USA: ACM, 2019, pp. 438–443.

[43] K. Chellapilla, S. Puri, and P. Y. Simard, "High Performance Convolutional Neural Networks for Document Processing," 2006.

[44] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *NIPS Autodiff Workshop*, 2017.

[45] O. Krestinskaya, A. Irmanova, and A. P. James, "Memristive Non-Idealities: Is there any Practical Implications for Designing Neural Network Chips?" in *Proc. IEEE International Symposium on Circuits and Systems (ISCAS)*, Sapporo, Japan, May. 2019, pp. 1–5.

[46] E. Miranda, A. Mehonic, W. H. Ng, and A. J. Kenyon, "Simulation of Cycle-to-Cycle Instabilities in SiO $_x$ -Based ReRAM Devices Using a Self-Correlated Process With Long-Term Variation," *IEEE Electron Device Letters*, vol. 40, no. 1, pp. 28–31, 2019.

[47] W. Yi, S. E. Savel'ev, G. Medeiros-Ribeiro, F. Miao, M.-X. Zhang, J. J. Yang, A. M. Bratkovsky, and R. S. Williams, "Quantized Conductance Coincides with State Instability and Excess Noise in Tantalum Oxide Memristors," *Nature Communications*, vol. 7, no. 1, p. 11142, Apr. 2016. [Online]. Available: https://doi.org/10.1038/ncomms11142

[48] S. Yu, "Neuro-inspired Computing with Emerging Nonvolatile Memorys," *Proceedings of the IEEE*, vol. 106, no. 2, pp. 260–285, Feb. 2018.

[49] W. Jakob, J. Rhinelander, and D. Moldovan, "pybind11 âĂŤ Seamless Operability Between C++11 and Python," 2016, https://github.com/pybind/pybind11.

[50] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *Proc. 3rd International Conference on Learning Representations, ICLR*, San Diego, CA, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[51] Z. Zhang and M. R. Sabuncu, "Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels," *CoRR*, vol. abs/1805.07836, 2018. [Online]. Available: http://arxiv.org/abs/1805.07836

[52] E. Yalon, A. Gavrilov, S. Cohen, D. Mistele, B. Meyler, J. Salzman, and D. Ritter, "Resistive Switching in HfO_2 Probed by a MetalâĂŞInsulatorâĂŞSemiconductor Bipolar Transistor," *IEEE Electron Device Letters*, vol. 33, no. 1, pp. 11–13, 2012.

[53] R. G. Andrzejak, K. Lehnertz, F. Mormann, C. Rieke, P. David, and C. E. Elger, "Indications of Nonlinear Deterministic and Finite-dimensional Structures in Time Series of Brain Electrical Activity: Dependence on Recording Region and Brain State," *Physical review. E, Statistical, nonlinear, and soft matter physics*, vol. 64, p. 061907, Dec. 2001.

[54] M. Sokolova and G. Lapalme, "A Systematic Analysis of Performance Measures for Classification Tasks," *Information Processing & Management*, vol. 45, no. 4, pp. 427 – 437, 2009.

Corey Lammie (S'17) is currently pursuing a PhD in Computer Engineering at James Cook University (JCU), where he completed his undergraduate degrees in Electrical Engineering (Honours) and Information Technology in 2018. His main research interests include brain-inspired computing, and the simulation and hardware implementation of Spiking Neural Networks (SNNs) and Artificial Neural Networks (ANNs) using ReRAM devices and FPGAs. He has received several awards and fellowships including the intensely competitive 2020-2021 IBM international PhD Fellowship, a Domestic Prestige Research Training Program Scholarship, and the 2017 Engineers Australia CN Barton Medal awarded to the best undergraduate engineering thesis at JCU. Corey has served as a reviewer for several IEEE journals and conferences including IEEE Transactions on Circuits and Systems and the IEEE International Symposium on Circuits and Systems (ISCAS).

Wei Xiang (S'00-M'04-SM'10) is currently the Founding Chair and the Head of Discipline of Internet of Things Engineering with James Cook University, Cairns, QLD, Australia. Due to his instrumental leadership in establishing the AustraliaâĂŹs first accredited Internet of Things Engineering Degree Program, he was selected into Pearcy FoundationâĂŹs Hall of Fame in October 2018. He has authored or coauthored more than 250 peer-reviewed articles including 3 academic books and 130 journal articles. His research interests fall under the broad areas of communications and information theory, particularly the Internet of Things, and coding and signal processing for multimedia communications systems. He is an Elected Fellow of the IET in U.K. and Engineers Australia.

Bernabé Linares-Barranco (M'90-S'06-F'10) received the B. S. degree in electronic physics in June 1986 and the M. S. degree in microelectronics in September 1987, both from the University of Seville, Sevilla, Spain. From September 1988 until August 1991 he was a Graduate Student at the Dept. of Electrical Engineering of Texas A&M University. Since June 1991, he has been a Tenured Scientist at the "Instituto de MicroelectrÃşnica deSevilla", (IMSE-CNM-CSIC) Sevilla, Spain, In January 2003 he was promoted to Tenured Researcher and in January 2004 to Full Professor. Since February 2018, he is the Director of the "Insitituto de MicroelectrÃşnica de Sevilla". Over the past 20 years has been deeply involved with neuromorphic spiking circuits and systems, with strong emphasis on vision and exploiting nanoscale memristive devices for learning. He is an IEEE Fellow since January 2010.

Mostafa Rahimi Azghadi (S'07-M'14-SM'19) completed his PhD in 2014 in neuromorphic engineering and neural network learning in the University of Adelaide, achieving two doctoral research medals. He is currently a tenured Senior Lecturer at James Cook University, where he researches low-power and high-performance neuromorphic accelerators for neural-inspired and deep learning networks. Dr Rahimi has published near 60 articles attracting >1300 citations in his short academic career. He has received 20+ awards, scholarships, and fellowships including a 2015 South Australia Science Excellence award, a 2016 Endeavour Fellowship, a 2017 Queensland Young Tall Poppy Science Award, and a 2018 JCU Rising Star ECR Leadership Grant. Dr Rahimi serves as an associate editor of Frontiers in Neuromorphic Engineering and IEEE Access. He is a senior member of the IEEE and a TC member of Neural Systems and Applications of the IEEE Circuits and Systems Society.