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

Corey Lammie[a], Wei Xiang[b], Bernabé Linares-Barranco[c], Mostafa Rahimi Azghadi[a,*]

[a]*College of Science and Engineering, James Cook University, Australia*
[b]*School of Computing, Engineering and Mathematical Sciences, La Trobe University, Australia*
[c]*Institute of Microelectronics of Seville, IMSE-CNM, Parque Tecnológico de la Cartuja, CSIC, University of Seville, Spain*

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 these Resistive Random-Access Memory (RRAM) 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). However, memristive devices face concerns of aging and non-idealities, which limit the accuracy, reliability, and robustness of Memristive Deep Learning Systems (MDLSs), that should be considered prior to circuit-level realization. This Original Software Publication (OSP) presents *MemTorch*, an open-source[1] framework for customized large-scale memristive DL simulations, with a refined focus on the co-simulation of device non-idealities. MemTorch also facilitates co-modelling of key crossbar peripheral circuitry. MemTorch adopts a modernized software engineering methodology and integrates directly with the well-known PyTorch Machine Learning (ML) library.

Keywords: Memristors, RRAM, Non-Ideal Device Characteristics, Deep Learning, Simulation Framework

*Corresponding author
Email address: mostafa.rahimiazghadi@jcu.edu.au (Mostafa Rahimi Azghadi)
[1] https://github.com/coreylammie/MemTorch

1. 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 [3]. 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 [4, 5]. Consequently, significant research efforts are being made to improve the reliability and robustness of memristive, or RRAM crossbars, used to perform *in-situ* learning [6, 7, 8, 9] and inference [2, 6, 10, 11, 12, 13] in DL systems. A general cross-platform, heterogeneous, high-level, customizable and open-source simulation framework with a refined focus on the co-simulation of device non-idealities could be used to conveniently build, rapidly prototype, and investigate device non-idealities in customized large-scale Memristive Deep Neural Networks (MDNNs) and MDLSs. In this OSP, we present such a framework, entitled *MemTorch*, for deep memristive learning using crossbar architectures. MemTorch is an open-source [14] simulation framework that integrates directly with the open-source PyTorch ML library that:

1. Facilitates the cross-platform development and distribution of large-scale passive 0-Transistor 1-Resistor (0T1R) and active 1-Transistor 1-Resistor (1T1R) memristive deep learning systems;
2. Places a large emphasis on modeling non-ideal, but inevitable, device characteristics in arbitrary and customizable device models;
3. Supports heterogeneous platforms such as Central Processing Units (CPUs) and Graphics Processing Units (GPUs);
4. Has a high-level Application Programming Interface (API), which is able to abstract performance-critical tasks described in various low-level languages.

2. Related Work

We compare MemTorch to other memristor-based DNN frameworks and inference accelerators, which are software-based and do not rely on SPICE

Table 1: 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 [19]		✓*	✓	C++
MNSIM [20]			✓	Not Specified
PUMA [11]			✓	C++
DL-RSIM [21]		✓	✓	Python
PipeLayer [6]		✓*	✓	C++
Tiny but Accurate [22]	✓†		✓	MATLAB
Ultra-Efficient Memristor-Based DNN Framework [23]	✓†		✓	C++, MATLAB
Non-ideal Resistive Synaptic Device Characteristic Simulation Framework [24]		✓	✓	Python
Neurosim [25], NeuroSim+ [26], and DNN+NeuroSim [16, 17]	✓	✓	✓	C++, Python
IBM Analog Hardware Acceleration Kit [18]	✓	✓	✓	Python, C++, CUDA
MemTorch	✓	✓	✓	**Python, C++, CUDA**

modeling, in Table 1. More exhaustive comparisons are performed in [15]. Software-based frameworks and inference accelerators use a combination of programming languages to simulate the behavior of memristive devices. Among previous works, DNN+NeuroSim [16, 17] and the IBM Analog Hardware Acceleration Kit [18] are the most similar offerings, which integrate with both PyTorch and/or Tensorflow, and can be used to account for non-ideal device characteristics. However, they are largely concerned with algorithm-to-hardware mapping, and are designed to evaluate training and inference accuracy with hardware constraints. They are not designed to model any arbitrary device non-idealities for any behavioral device model. MemTorch, on the other hand, emphasizes the co-simulation of non-ideal device characteristics and generic behavioral device models with stochastic parameters for higher flexibility to simply account for process variance.

3. Software Framework

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 [27] ML framework, and uses the C++ and Python PyTorch APIs extensively to abstract low-level operations. Consequently, it supports native CPU and GPU operations.

3.1. Software Architecture

MemTorch is made up of seven distinct sub-modules. General utility functions, such as data loaders or generic functions, are grouped within `mem-`

`torch.utils`. The `memtorch.bh` sub-module encapsulates all crossbar models, crossbar mapping and programming methods, crossbar tile mapping and programming methods, memristor models, memristor model window functions, models for all non-ideal device characteristics, quantization methods, and methods to generate stochastic parameters. The `memtorch.mn` sub-module mimics `torch.nn` and defines equivalent memristive`torch.nn.Module` layers. `memtorch.mn` currently extends `torch.nn.Linear`, `torch.nn.Conv1d`, `torch.nn.Conv2d`, and `torch.nn.Conv3d`. `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_()` method, in addition to other helper methods, to the class instance of the model that automatically patches each selected named module.

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 C++ and CUDA bindings to perform inference for both active and passive modular tiled architectures, and to parallelize quantization operations. The use of bindings can be disabled, and legacy python methods (developed in previous versions of MemTorch) can be used instead using the `use_bindings` argument, when patching `torch.nn.Module` instances. `memtorch.examples` sub-module encapsulates general-usage examples and supporting scripts. The `memtorch.map` sub-module encapsulates all mapping and tuning algorithms used when programming and tuning memristive crossbar arrays. Finally, the `memtorch.submodule` sub-module encapsulates all external git sub-modules that MemTorch uses. We review and present the algorithms and models that are currently built into MemTorch in Appendix A, and our approach to modeling non-ideal device characteristics in Appendix B.

3.2. Software Functionalities

Complete examples demonstrating the functionality of MemTorch are publicly accessible[2]. ReadTheDocs[3] is used to explain all functionalities.

[2] `https://github.com/coreylammie/MemTorch/tree/master/memtorch/examples`
[3] `https://memtorch.readthedocs.io/en/latest/`

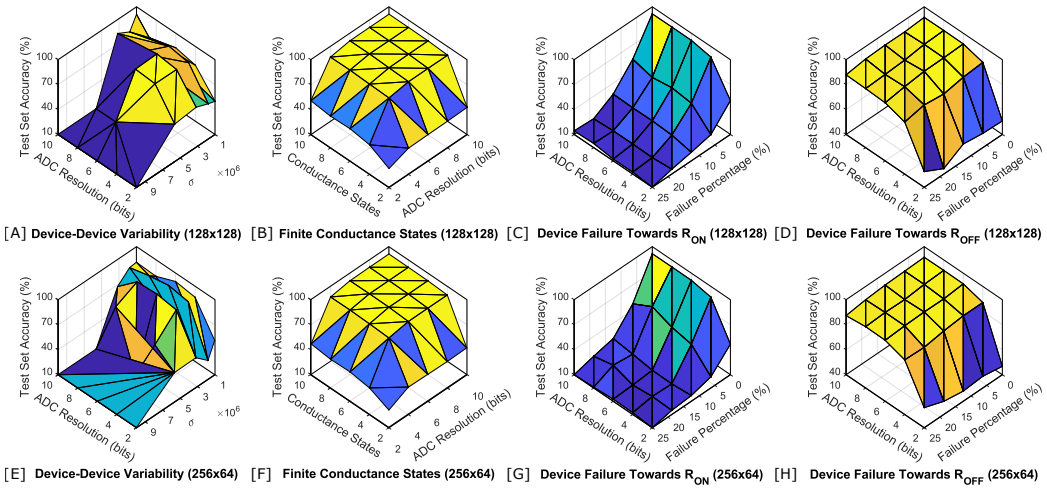

Figure 1: Simulation results when exemplar device non-idealities are considered for classifying CIFAR-10 dataset for two different modular crossbar tile sizes, 128x128 and 256x64. While MemTorch can be used to simulate both passive and active architectures, for demonstration purposes, in this figure, only active architectures are considered.

4. Implementation and Empirical Results

In Fig. 1, we present exemplar large-scale deep learning simulations to investigate the performance degradation due to device-device variability, finite number of conductance states, and device failure when two different modular crossbar sizes are considered. A separate case study is not presented, as we have previously used MemTorch in other works to perform hand gesture classification [28], epileptic seizure prediction [29], to develop an empirical metal-oxide device endurance and retention model [30], and to develop an extended Design Space Exploration (DSE) methodology for RRAM architectures [31]. Prior to simulation, all convolutional and linear layers from a pre-trained MobileNetV2 for CIFAR-10 were converted to memristive equivalent layers, as explained in detail in [4], and documented in Appendix C.

5. Illustrative Example

To demonstrate MemTorch's intuitive design, we depict a typical use-case work flow in Fig. 2. Here, `torch.nn.Linear` layers are converted to

[4] `https://github.com/coreylammie/MemTorch/tree/master/memtorch/examples/Exemplar_Simulations.ipynb`

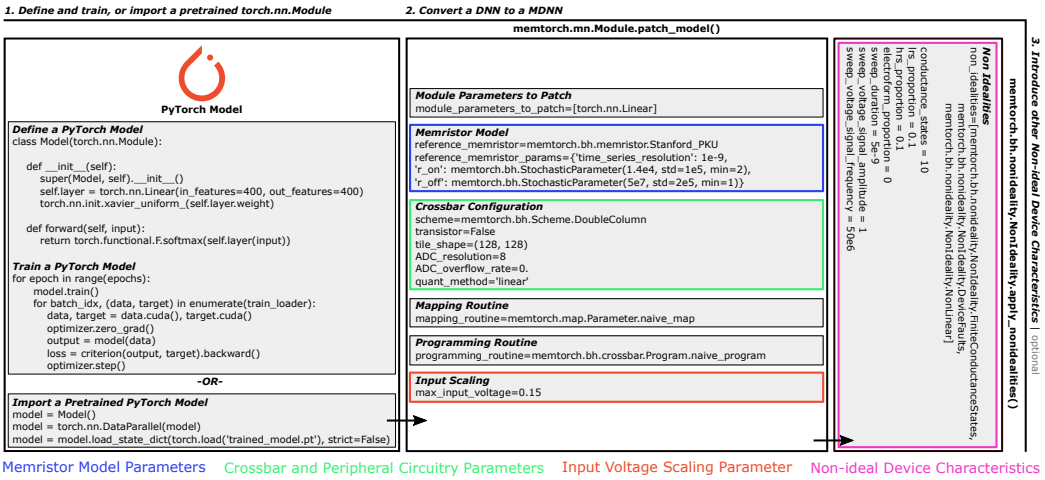

Figure 2: Illustration of a typical use-case workflow in MemTorch.

equivalent memristive layers constructed using modular crossbar tiles, that each contain (128×128) devices, which represent weights using a double-column parameter representation scheme. Inputs are scaled between $\pm 0.15V$, and 8-bit Analog to Digital Converters (ADCs) are used to read out column currents. The Stanford PKU RRAM model [32] is used to model TiN/Hf(Al)O/Hf/TiN devices from [33]. 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.

6. Conclusion

We presented an open-source simulation framework, entitled *MemTorch*, for large-scale deep memristive crossbar architectures. We showed that MemTorch is designed with a focus to integrate any desired behavioral or experimental device model, and introduce arbitrary device non-idealities, while co-simulating crossbar and peripheral circuitry. We compared MemTorch to similar works, detailed its package structure, and performed exemplar simulations to demonstrate its functionality. We hope that MemTorch will be adopted and expanded by the community to advance memristive deep learning research and development endeavours.

Acknowledgements

CL acknowledges the JCU DRTPS, CAS Society Pre-Doctoral Grant, and the IBM PhD Fellowship. MRA acknowledges a JCU Rising Star ECR Leadership Grant.

Appendix A. Algorithms and Models

This appendix reviews and presents the algorithms and models that are currently built into MemTorch.

Appendix A.1. Memristive Device Models

Within MemTorch, we use five base memristive device models that extend the `memtorch.bh.Memristor.Memristor` base class for our simulations. These include the linear ion drift model by [34]; the VTEAM model by [35], which is a general model for voltage-controlled memristors that can be used to fit a large range of experimental device data; the Stanford PKU RRAM model [32], which describes switching performance for bipolar metal oxide RRAM; and two versions of the data-driven Verilog-A RRAM model [36], which expresses device current-voltage characteristics and resistive switching rate as a function of the bias voltage and the initial resistive state of each device. For each base model, finite differences is used to obtain a numerical solution for each discretized time-step, dt. While only five base memristive models are currently supported natively, others, which can model the equivalent conductance of a memristive device for an arbitrary applied voltage signal, such as those modelling Phase Change Memory (PCM) or other device technology behavior, can easily be integrated modularly by extending `memtorch.bh.Memristor.Memristor`.

Appendix A.2. Window Functions

Within memristive device models, window functions are widely employed to restrict the changes of the internal state variables to specified intervals [37]. MemTorch currently natively supports the Biolek [38], Jogelkar [39], and Prodromakis [40] window functions, and can easily be extended to support others.

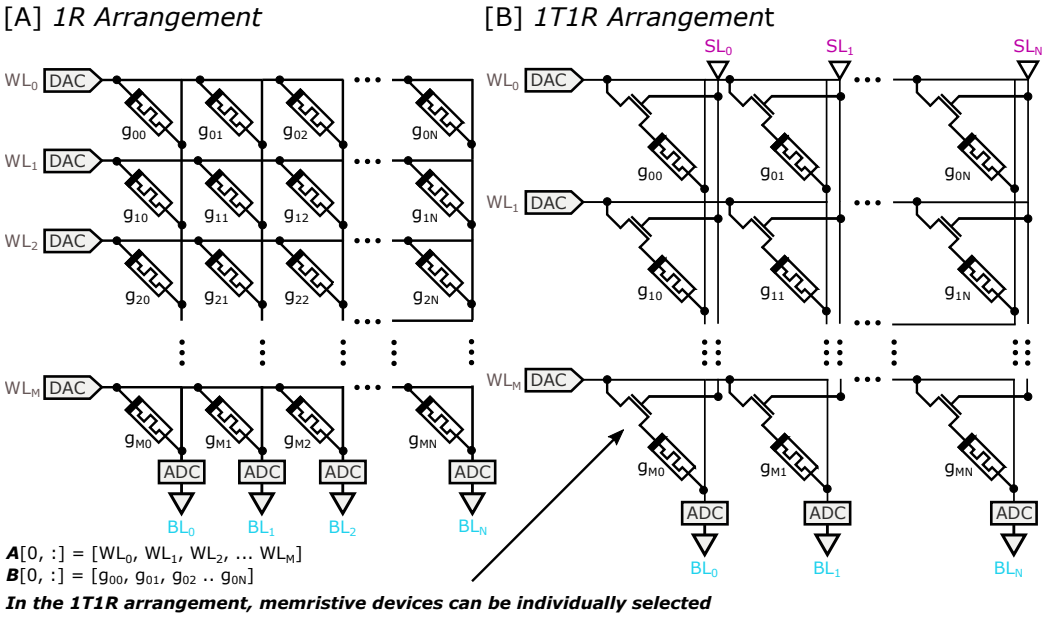

Figure A.3: Depiction of an $M \times N$ [A] 1R (0T1R) crossbar architecture and a [B] 1T1R crossbar architecture. Matrix-vector and matrix-matrix multiplication can be performed by encoding and presenting a scaled input vector or matrix $\boldsymbol{A}$ as voltage signals to each row of the crossbar's 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).

Appendix A.3. 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 or /0T1R), which are both depicted in Fig. A.3. 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. A.3, where $\boldsymbol{A}$ represents input voltage signals and $\boldsymbol{B}$ is encoded within the crossbar as memristor

conductances. Separate ADCs can be used to read out the current of each column in parallel, as depicted, or sample and hold circuits can be used in conjunction with a single ADC per crossbar, that can be used to read out the current of each column sequentially using Time-division Multiplexing (TDM). 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 ADC readout of each column accordingly. By separately presenting each row of $\boldsymbol{A}$ to the crossbar through WLs, all rows of $\boldsymbol{AB}$ can be computed.

Because memristors cannot be programmed to have negative conductances, within MDNNs, weight matrices can either be represented using two devices per weight [41], as described by (A.1),

$$\boldsymbol{AB} = K \sum_{i=0}^{C} \boldsymbol{A}[i,:](g_{\text{pos}}[i,j] - g_{\text{neg}}[i,j]), \text{for } j = 0 \text{ to } N, \tag{A.1}$$

or using a single device per weight [42, 43] using complex weight mapping algorithms or current mirrors, as described by (A.2),

$$\boldsymbol{AB} = K \sum_{i=0}^{C} \boldsymbol{A}[i,:](g[i,j] - g_m), \text{for } j = 0 \text{ to } N. \tag{A.2}$$

For the single-column case, the current through g_m, used to mirror a current $-2V/(\bar{R}_{\text{ON}} + \bar{R}_{\text{OFF}})$ to each crossbar, is copied to each column and subtracted from all memristor columns. This current can be realized using a diode-connected NMOSFET by adjusting the NMOSFET channel width so that it has a passive resistance g_m. From this stage forward, we refer to the weight matrix representation methodology adopted, that is, weather two devices are used to encode each weight, i.e, differential weight mapping, one device is used to encode each weight, or another configuration is used to encode weight matrices, as the parameter representation scheme.

Appendix A.3.1. Modular memristive crossbar tiles

Mapping complete unrolled neural network layers into large memristive crossbar architectures often results in poor performance. This is due to non-ideal device characteristics that introduce substantial current variability when accumulated currents from columns with a large number of devices are read out. When one or two large crossbars are used, for single-column and double-column parameter representation schemes, respectively, they cannot

easily be modularized because customized crossbar shapes are required to represent each individual layer. Instead of using large crossbars, modular crossbar tiles [44] can be used that map layers into multiple uniformly sized crossbars, commonly referred to in literature as *crossbar tiles*.

One large crossbar of size $(M \times N)$ can be mapped using $\text{ceil}(M/S_0) \times \text{ceil}(N/S_1)$ crossbar tiles, each with a size of $(S_0 \times S_1)$, where the total utilization, ρ, of all crossbar tiles can be determined using (A.3),

$$\rho = \frac{MN}{\text{ceil}(M/S_0)\text{ceil}(N/S_1)S_0S_1}. \tag{A.3}$$

Duplication of crossbar tiles and TDM can be used to regulate mapping to improve the array utilization and computation time by balancing latency among layers [45].

Appendix A.3.2. 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 [46, 47, 48, 49] and corrective methods [50, 2, 1], 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.

Appendix A.3.3. 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 vector. Consequently, after each DNN layer's weights are programmed into a crossbar or group of tiles, linear regression can be used to correlate the output current of each column with any desired output to determine K for the crossbar or group of tiles, 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

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.

$\mathbf{w} = \text{abs}(\mathbf{w})$

$\mathbf{w} = \text{descending\_order}(\mathbf{w})$

$s = \text{size}(\mathbf{w})$

$\text{index} = \text{int}(p_L \cdot s)$

$\mathbf{w}_{\max} = \mathbf{w}[\text{index}]$

$\mathbf{w}_{\min} = \mathbf{w}_{\max}/(\mathbf{R}_{\text{OFF}}/\mathbf{R}_{\text{ON}})$

$\mathbf{w} = \text{clip}(\mathbf{w}, \mathbf{w}_{\min}, \mathbf{w}_{\max})$

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

be used pre-programming [51], to improve robustness and reduce susceptibility to error, or post-programming by retraining device-specific conductance values [22].

Appendix A.3.4. Memristor crossbar weight mapping

Weights, denoted using $\boldsymbol{w}$, within unrolled convolutional layers [52] and linear layers can be mapped to equivalent conductance values, $\boldsymbol{g}$, using (A.4).

$$\boldsymbol{g}[i,j] = \frac{(g_{\text{ON}} - g_{\text{OFF}})(\sigma(\boldsymbol{w})[i,j] - \mathbf{w}_{\min})}{|\boldsymbol{w}|_{\max} - \mathbf{w}_{\min}} + g_{\text{OFF}}, \tag{A.4}$$

where $\mathbf{w}_{\min}$ represents the minimum weight value to encode, and $\mathbf{w}_{\max}$ represents the maximum weight value to encode. $g_{\text{ON}} = 1/R_{\text{ON}}$ and $g_{\text{OFF}} = 1/R_{\text{OFF}}$. When two crossbars are used to represent weight, crossbars containing positive components will have $\sigma(\boldsymbol{w}) = \boldsymbol{w}[\boldsymbol{w} \geqslant 0]$, while crossbars containing negative components will have $\sigma(\boldsymbol{w}) = \boldsymbol{w}[\boldsymbol{w} \leqslant 0]$. When a single crossbar is used to represent weights, $\sigma(\boldsymbol{w}) = \boldsymbol{w} - g_m$.

To reduce the inner weight gap in a given device, Algorithm 1 in [10] can be used to exclude a small proportion, p_L, of weights with the absolute largest values to reduce the variability effect of non-ideal memristive devices.

Appendix A.3.5. Passive memristor crossbar architectures

When modeling passive memristor crossbar architectures, source and line resistances should be accounted for. MemTorch utilizes a comprehensive crossbar array model with solutions for source and line resistances, as described in [53], to solve for node voltages $\boldsymbol{V}$, within passive crossbar architectures in each simulation time-step. Specifically, linear matrix algebra is used to solve (A.5)

$$\begin{bmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{C} & \boldsymbol{D} \end{bmatrix} \boldsymbol{V} = \boldsymbol{E}, \tag{A.5}$$

where for a crossbar of size $(M \times N)$, $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{D}$ are all $(MN \times MN)$ matrices, and $\boldsymbol{E}$ is a $(2MN \times 1)$ vector. All matrices and vectors in (A.5) are derived and defined in [53]. As the concatenated $\boldsymbol{ABCD}$ matrix is sparse, traditional linear matrix algebra methods cannot be easily used to solve (A.5), because they require a prohibitive amount of memory. Instead, sparse supernodal LU factorization with partial pivoting for general matrices [54] is used, as it is parallelizable, and has demonstrated the best empirical numerical performance, compared to related techniques, e.g., QR decomposition [55].

Appendix A.4. C++ and CUDA Bindings

Numerous performance critical operations, including tiled VMMs, linear matrix algebra, and quantization, are accelerated using C++ and CUDA bindings. `PYBIND11_MODULE()` is a method within the pybind11 [5] python library [56] that exposes C++ types in Python to enable seamless operability between C++11, CUDA, and Python. This library is used within MemTorch to overload method pointers and to expose C++ and CUDA functions to the developed Python API. The Eigen [57] C++ template library is used extensively to perform various linear algebra operations, as many Eigen functions can be compiled for use within CUDA kernels using `__device__ __host__` function type qualifiers.

Appendix B. Modeling Non-Ideal Device Characteristics

Non-ideal device characteristics can either be encapsulated within device-specific memristive models, or introduced to base (generic) models using

[5] `https://github.com/pybind/pybind11`

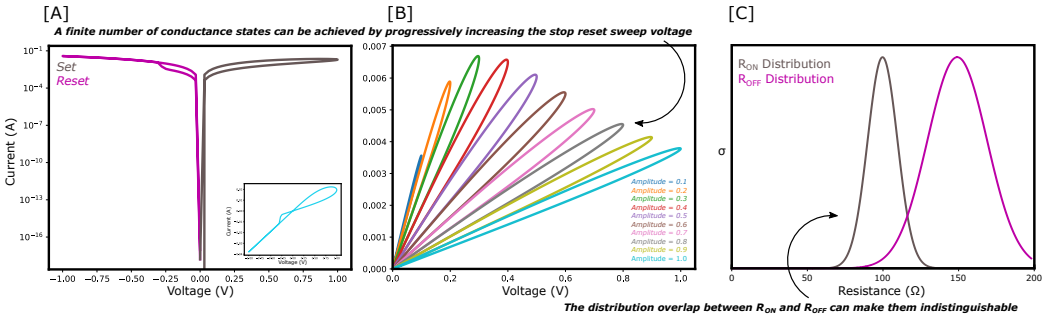

Figure B.4: Depiction of [A] device I/V characteristics, and [B] reset voltage double-sweeps demonstrating gradual switching from R_{ON} to R_{OFF}, which can be used to achieve 10 finite stable conductance states for the VTEAM model using the TEAM [58] model's parameters, with a linear dependence on 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.

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 to future releases. Three of the non-ideal device characteristics that are currently supported by MemTorch are shown in Fig. B.4. Fig. B.4[A] depicts typical non-linear I/V device characteristics using a set-reset curve and an inset hysteresis loop. Fig. B.4[B] demonstrates gradual switching, which is used to achieve a finite number of stable conductance states, and Fig. B.4[C] shows overlapping distributions of R_{ON} and R_{OFF}, which is caused by device-to-device variability.

Appendix B.1. 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()` method, which accepts an arbitrary number of keyword arguments, that are used to sample from a `torch.distributions` each time a device model is instantiated. To model device-device variability, 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 demon-

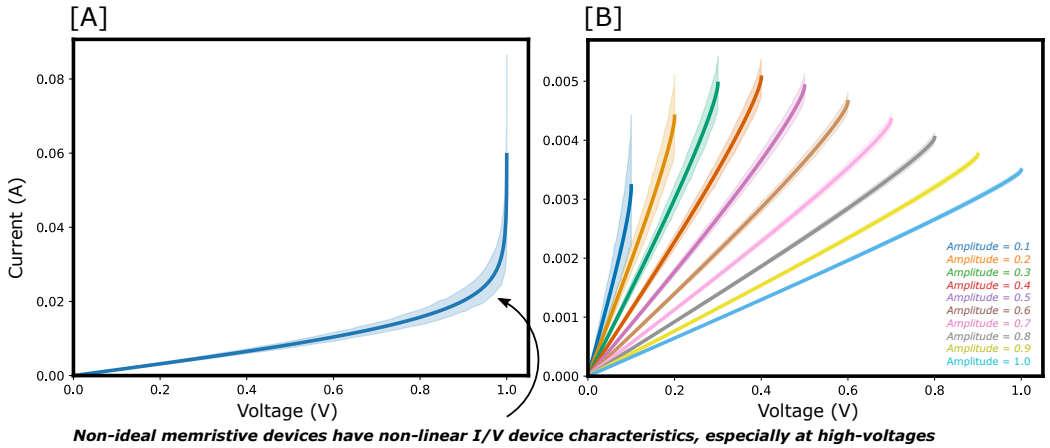

Figure B.5: Non-linear I/V characteristics for 100 devices (instances) of the VTEAM model using the TEAM [58] model's parameters, with a linear dependence on w, achieved using sinusoidal signals with a fixed frequency of 50 MHz. R_{ON} and R_{OFF} were stochastically sampled from a normal distribution 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.

strated to be larger than R_{ON} [59]. As depicted in Fig. B.4[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.

Appendix B.2. Cycle-to-cycle Variability

Cycle-to-cycle (C2C) variability [60] 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.

Appendix B.3. 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} [61]. 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 [10].

Therefore, deterministic discretization [62] can be used to represent a finite number of electrically switchable conductance states, as depicted in Fig. B.4[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.

Appendix B.4. 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 [10]. 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 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. Devices are chosen at random using `np.random.choice()`.

Appendix B.5. 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 [10]. We demonstrate these characteristics using Fig. B.5[A], by depicting the I/V curve of the VTEAM model between 0–1V using the TEAM [58] model's parameters. The `memtorch.bh.nonideality.NonLinear.apply_non_linear()` method 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.

Appendix B.5.1. Devices with an infinite number of discrete conductance states

The `memtorch.bh.nonideality.NonLinear.apply_non_linear()` method uses two methods to efficiently model non-linear device I/V charac-

teristics 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 Lookup Tables (LUTs) to compute device output currents during inference.

Appendix B.5.2. Devices with a finite number of discrete conductance states

The `memtorch.bh.nonideality.NonLinear.apply_non_linear()` method 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. B.5[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. B.5[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}.

Appendix C. Exemplar Simulation Details

For all simulations performed to obtain the results presented in Fig. 1, we followed the following training and test procedure. We first augmented a pretrained MobileNetV2 CNN trained using the CIFAR-10 training set. All convolutional and linear layers within the network were sequenced with batch-normalization layers with fixed affline parameters to normalize outputs. The network was trained until improvement on the validation set was negligible (for 100 epochs) with a batch size of $\Im = 256$. The initial learning rate was $\eta = 1e - 1$, which was decayed by an order of magnitude every 40 training epochs. Stochastic Gradient Descent (SGD) was used to optimize network parameters and Cross Entropy (CE) [63] 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 sampled from a uniform distribution between ± 1.0. 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 -0.3 to 0.3, to emulate voltage signals between ± 0.3V, which were applied to the word-lines of each memristive crossbar. All device models originated from the VTEAM model, with $\bar{R}_{\text{ON}} = 1.4\text{e}4\Omega$ and $\bar{R}_{\text{OFF}} = 5\text{e}7\Omega$, to model TiN/Hf(Al)O/Hf/TiN devices from [33].

Implementations are investigated using modular crossbar tiles of size 128×128 and 256×64, as these have previously been demonstrated to be effective in terms of utilization and power efficiency [45]. While power and latency balancing is beyond the scope of MemTorch 1.1.5, 256×64 tile size enables higher operation throughput and more analog operations per ADC compared to 128×128 tile size [45]. However, the area utilization may be lower for arrays with more than 64 columns considering the number of output channels.

References

[1] M. Hu, H. Li, Y. Chen, Q. Wu, G. S. Rose, R. W. Linderman, Memristor Crossbar-Based Neuromorphic Computing System: A Case Study, IEEE Transactions on Neural Networks and Learning Systems 25 (2014) 1864–1878.

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

[3] M. Rahimi Azghadi, Y. Chen, J. Eshraghian, J. Chen, C. Lin, A. Amirsoleimani, A. Mehonic, A. Kenyon, B. Fowler, J. Lee, Y. Chang, Complementary Metal-Oxide Semiconductor and Memristive Hardware for Neuromorphic Computing, Advanced Intelligent Systems 2 (2020) 1900189.

[4] S. Mittal, A Survey of ReRAM-Based Architectures for Processing-In-Memory and Neural Networks, Machine Learning and Knowledge Extraction 1 (2018) 75–114.

[5] G. C. Adam, A. Khiat, T. Prodromakis, Challenges Hindering Memristive Neuromorphic Hardware from Going Mainstream, Nature communications 9 (2018) 5267.

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

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

[8] C. Lammie, J. K. Eshraghian, W. D. Lu, M. R. Azghadi, Memristive Stochastic Computing for Deep Learning Parameter Optimization, IEEE Transactions on Circuits and Systems II: Express Briefs (2021).

[9] O. Krestinskaya, K. N. Salama, A. P. James, Learning in Memristive Neural Network Architectures Using Analog Backpropagation Circuits, IEEE Transactions on Circuits and Systems I: Regular Papers 66 (2019) 719–732.

[10] A. Mehonic, D. Joksas, W. H. Ng, M. Buckwell, A. J. Kenyon, Simulation of Inference Accuracy Using Realistic RRAM Devices, Frontiers in Neuroscience 13 (2019) 593.

[11] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W. Hwu, J. P. Strachan, K. Roy, D. S. Milojicic, PUMA: A Programmable Ultra-efficient Memristor-based Accelerator for Machine Learning Inference, CoRR abs/1901.10351 (2019).

[12] H. Jeong, L. Shi, Memristor devices for neural networks, Journal of Physics D: Applied Physics 52 (2018) 023003.

[13] H. Tsai, S. Ambrogio, P. Narayanan, R. M. Shelby, G. W. Burr, Recent Progress in Analog Memory-based Accelerators for Deep Learning, Journal of Physics D: Applied Physics 51 (2018) 283001.

[14] C. Lammie, W. Xiang, B. Linares-Barranco, M. R. Azghadi, coreylammie/MemTorch: Initial Release, 2020. URL: https://doi.org/10.5281/zenodo.3760696. doi:10.5281/zenodo.3760695.

[15] C. Lammie, W. Xiang, M. Rahimi Azghadi, Modeling and simulating in-memory memristive deep learning systems: An overview of current efforts, Array 13 (2022) 100116.

[16] X. Peng, S. Huang, Y. Luo, X. Sun, S. Yu, DNN+NeuroSim: An End-to-End Benchmarking Framework for Compute-in-Memory Accelerators with Versatile Device Technologies, in: IEEE International Electron Devices Meeting, 2019.

[17] X. Peng, S. Huang, H. Jiang, A. Lu, S. Yu, DNN+NeuroSim V2.0: An End-to-End Benchmarking Framework for Compute-in-Memory Accelerators for On-chip Training, 2020. arXiv:2003.06471.

[18] M. J. Rasch, D. Moreda, T. Gokmen, M. Le Gallo, F. Carta, C. Goldberg, K. El Maghraoui, A. Sebastian, V. Narayanan, A Flexible and Fast PyTorch Toolkit for Simulating Training and Inference on Analog Crossbar Arrays, in: Proceedings of the IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS), 2021. doi:10.1109/AICAS51828.2021.9458494.

[19] M. Imani, M. Samragh, Y. Kim, S. Gupta, F. Koushanfar, T. Rosing, RAPIDNN: In-Memory Deep Neural Network Acceleration Framework, CoRR abs/1806.05794 (2018).

[20] L. Xia, B. Li, T. Tang, P. Gu, P. Chen, S. Yu, Y. Cao, Y. Wang, Y. Xie, H. Yang, MNSIM: Simulation Platform for Memristor-Based Neuromorphic Computing System, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37 (2018) 1009–1022.

[21] M. Lin, H. Cheng, W. Lin, T. Yang, I. Tseng, C. Yang, H. Hu, H. Chang, H. Li, 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, 2018, pp. 1–8. doi:10.1145/3240765.3240800.

[22] X. Ma, G. Yuan, S. Lin, C. Ding, F. Yu, T. Liu, W. Wen, X. Chen, Y. Wang, Tiny but Accurate: A Pruned, Quantized and Optimized

Memristor Crossbar Framework for Ultra Efficient DNN Implementation, arXiv e-prints (2019) arXiv:1908.10017.

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

[24] X. Sun, 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 9 (2019) 570–579.

[25] P. Chen, X. Peng, S. Yu, NeuroSim: A Circuit-Level Macro Model for Benchmarking Neuro-Inspired Architectures in Online Learning, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37 (2018) 3067–3080.

[26] P. Chen, X. Peng, S. Yu, NeuroSim+: An integrated device-to-algorithm framework for benchmarking synaptic devices and array architectures, in: IEEE International Electron Devices Meeting, 2017, pp. 6.1.1–6.1.4. doi:10.1109/IEDM.2017.8268337.

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

[28] M. R. Azghadi, C. Lammie, J. K. Eshraghian, M. Payvand, E. Donati, B. Linares-Barranco, G. Indiveri, Hardware Implementation of Deep Network Accelerators Towards Healthcare and Biomedical Applications, IEEE Transactions on Biomedical Circuits and Systems 14 (2020) 1138–1159.

[29] C. Lammie, W. Xiang, and M. R. Azghadi, Towards Memristive Deep Learning Systems for Real-time Mobile Epileptic Seizure Prediction, in: Proc. IEEE International Symposium on Circuits and Systems (ISCAS), Daegu, South Koera., 2021.

[30] C. Lammie, M. R. Azghadi, D. Ielmini, Empirical Metal-Oxide RRAM Device Endurance and Retention Model For Deep Learning Simulations, Semiconductor Science and Technology (2021).

[31] C. Lammie, J. K. Eshraghian, C. Li, A. Amirsoleimani, R. Genov, W. D. Lu, M. R. Azghadi, Design Space Exploration of Dense and Sparse Mapping Schemes for RRAM Architectures (2022).

[32] Z. Jiang, S. Yu, Y. Wu, J. H. Engel, X. Guan, H. . P. Wong, Verilog-a compact model for oxide-based resistive random access memory (rram), in: International Conference on Simulation of Semiconductor Processes and Devices (SISPAD), Yokohama, Japan., 2014, pp. 41–44.

[33] A. Fantini, L. Goux, A. Redolfi, R. Degraeve, G. Kar, Y. Y. Chen, M. Jurczak, Lateral and Vertical Scaling Impact on Statistical Performances and Reliability of 10nm TiN/Hf(Al)O/Hf/TiN RRAM Devices, in: Symposium on VLSI Technology, 2014.

[34] D. B. Strukov, G. S. Snider, D. R. Stewart, R. S. Williams, The Missing Memristor Found, Nature 453 (2008) 80–83.

[35] S. Kvatinsky, M. Ramadan, E. G. Friedman, A. Kolodny, VTEAM: A General Model for Voltage-Controlled Memristors, IEEE Transactions on Circuits and Systems II: Express Briefs 62 (2015) 786–790.

[36] I. Messaris, A. Serb, S. Stathopoulos, A. Khiat, S. Nikolaidis, T. Prodromakis, A Data-Driven Verilog-A ReRAM Model, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37 (2018) 3151–3162.

[37] V. A. Slipko, Y. V. Pershin, Importance of the Window Function Choice for the Predictive Modelling of Memristors, CoRR abs/1811.06649 (2018).

[38] Z. Biolek, D. Biolek, V. Biolková, Spice Model of Memristor With Nonlinear Dopant Drift, Radioengineering (2009) 210–214.

[39] Y. N. Joglekar, S. J. Wolf, The Elusive Memristor: Properties of Basic Electrical Circuits, European Journal of Physics 30 (2009) 661–675.

[40] T. Prodromakis, B. P. Peh, C. Papavassiliou, C. Toumazou, A Versatile Memristor Model With Nonlinear Dopant Kinetics, IEEE Transactions on Electron Devices 58 (2011) 3099–3105.

[41] F. Alibart, E. Zamanidoost, D. B. Strukov, Pattern Classification by Memristive Crossbar Circuits using ex situ and in situ Training, Nature Communications 4 (2013) 2072.

[42] S. N. Truong, 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 14 (2014) 356–363.

[43] J. Lee, J. K. Eshraghian, K. Cho, K. Eshraghian, Adaptive Precision CNN Accelerator Using Radix-X Parallel Connected Memristor Crossbars, arXiv e-prints (2019) arXiv:1906.09395.

[44] D. J. Mountain, M. R. McLean, C. D. Krieger, Memristor Crossbar Tiles in a Flexible, General Purpose Neural Processor, IEEE Journal on Emerging and Selected Topics in Circuits and Systems 8 (2018) 137–145.

[45] Q. Wang, X. Wang, S. H. Lee, F. Meng, W. D. Lu, A Deep Neural Network Accelerator Based on Tiled RRAM Architecture, in: IEEE International Electron Devices Meeting (IEDM), San Francisco, CA., 2019, pp. 14.4.1–14.4.4.

[46] I. E. Ebong, P. Mazumder, Self-Controlled Writing and Erasing in a Memristor Crossbar Memory, IEEE Transactions on Nanotechnology 10 (2011) 1454–1463.

[47] M. S. Tarkov, Mapping Neural Network Computations onto Memristor Crossbar, in: Proc. International Siberian Conference on Control and Communications (SIBCON), Omsk, Russia, 2015, pp. 1–4. doi:10.1109/SIBCON.2015.7147235.

[48] R. Hasan, C. Yakopcic, 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. doi:10.1109/NANOARCH.2015.7180590.

[49] K. Jo, C. Jung, K. Min, S. Kang, Self-Adaptive Write Circuit for Low-Power and Variation-Tolerant Memristors, IEEE Transactions on Nanotechnology 9 (2010) 675–678.

[50] B. Feinberg, S. Wang, 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. doi:10.1109/HPCA.2018.00015.

[51] B. Zhang, N. Uysal, D. Fan, 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), ASPDAC '19, ACM, New York, USA, 2019, pp. 438–443. doi:10.1145/3287624.3287707.

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

[53] A. Chen, A comprehensive crossbar array model with solutions for line resistance and nonlinear device characteristics, IEEE Transactions on Electron Devices 60 (2013) 1318–1326.

[54] X. S. Li, M. Shao, A Supernodal Approach to Incomplete LU Factorization with Partial Pivoting, ACM Trans. Math. Softw. 37 (2011).

[55] P. Matstoms, Sparse QR Factorization in MATLAB, ACM Trans. Math. Softw. 20 (1994) 136–159.

[56] W. Jakob, J. Rhinelander, D. Moldovan, pybind11 — Seamless Operability Between C++11 and Python, 2016. Https://github.com/pybind/pybind11.

[57] G. Guennebaud, B. Jacob, et al., Eigen v3, http://eigen.tuxfamily.org, 2010.

[58] S. Kvatinsky, E. G. Friedman, A. Kolodny, U. C. Weiser, TEAM: ThrEshold Adaptive Memristor Model, IEEE Transactions on Circuits and Systems I: Regular Papers 60 (2013) 211–221.

[59] O. Krestinskaya, A. Irmanova, 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, 2019, pp. 1–5. doi:10.1109/ISCAS.2019.8702245.

[60] E. Miranda, A. Mehonic, W. H. Ng, 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 40 (2019) 28–31.

[61] W. Yi, S. E. Savel'ev, G. Medeiros-Ribeiro, F. Miao, M.-X. Zhang, J. J. Yang, A. M. Bratkovsky, R. S. Williams, Quantized Conductance Coincides with State Instability and Excess Noise in Tantalum Oxide Memristors, Nature Communications 7 (2016) 11142.

[62] S. Yu, Neuro-inspired Computing with Emerging Nonvolatile Memorys, Proceedings of the IEEE 106 (2018) 260–285.

[63] Z. Zhang, M. R. Sabuncu, Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels, CoRR abs/1805.07836 (2018).

Required Metadata

Current executable software version

Nr.	(executable) Software metadata description	Please fill in this column
S1	Current software version	1.1.5
S2	Permanent link to executables of this version	https://github.com/coreylammie/MemTorch/releases/tag/v1.1.5
S3	Legal Software License	GPLv3
S4	Computing platform/Operating System	Linux, OS X, Microsoft Windows
S5	Installation requirements & dependencies	A working Python interpreter (⩾3.6). If CUDA is True in `setup.py`, CUDA Toolkit (⩾10.1) and Microsoft Visual C++ Build Tools are required. If CUDA is False in `setup.py`, Microsoft Visual C++ Build Tools are required. All Python requirements are listed in `https://github.com/coreylammie/MemTorch/blob/master/requirements.txt`
S6	If available, link to user manual - if formally published include a reference to the publication in the reference list	`https://memtorch.readthedocs.io/en/latest/`
S7	Support email for questions	coreylammie@gmail.com, corey.lammie@jcu.edu.au

Table C.2: Software metadata (optional)

Current code version

Nr.	Code metadata description	Please fill in this column
C1	Current code version	1.1.5
C2	Permanent link to code/repository used of this code version	`https://github.com/coreylammie/MemTorch`
C3	Legal Code License	GPLv3
C4	Code versioning system used	git
C5	Software code languages, tools, and services used	Python, C++, CUDA.
C6	Compilation requirements, operating environments & dependencies	A working Python interpreter ($\geqslant$3.6). If CUDA is True in `setup.py`, CUDA Toolkit ($\geqslant$10.1) and Microsoft Visual C++ Build Tools are required. If CUDA is False in `setup.py`, Microsoft Visual C++ Build Tools are required. All Python requirements are listed in `https://github.com/coreylammie/MemTorch/blob/master/requirements.txt`
C7	If available Link to developer documentation/manual	`https://memtorch.readthedocs.io/en/latest/`
C8	Support email for questions	coreylammie@gmail.com, corey.lammie@jcu.edu.au

Table C.3: Code metadata (mandatory)