APL Machine Learning

TUTORIAL | NOVEMBER 28 2023

Using the IBM analog in-memory hardware acceleration kit for neural network training and inference

Manuel Le Gallo ; Corey Lammie ; Julian Büchel ; Fabio Carta ; Omobayode Fagbohungbe ; Charles Mackin ; Hsinyu Tsai ; Vijay Narayanan ; Abu Sebastian ; Kaoutar El Maghraoui ; Malte J. Rasch

Check for updates

APL Mach. Learn. 1, 041102 (2023)
https://doi.org/10.1063/5.0168089

View Online
Export Citation

02 July 2024 08:09:40

APL Energy
Latest Articles Online!
Read Now
AIP Publishing

AIP Publishing

APL Machine Learning TUTORIAL pubs.aip.org/aip/aml

Using the IBM analog in-memory hardware acceleration kit for neural network training and inference

Cite as: APL Mach. Learn. 1, 041102 (2023); doi: 10.1063/5.0168089
Submitted: 17 July 2023 • Accepted: 23 October 2023 •
Published Online: 28 November 2023

View Online Export Citation CrossMark

Manuel Le Gallo,[1] Corey Lammie,[1] Julian Büchel,[1] Fabio Carta,[2] Omobayode Fagbohungbe,[2] Charles Mackin,[3] Hsinyu Tsai,[3] Vijay Narayanan,[2] Abu Sebastian,[1] Kaoutar El Maghraoui,[2,a)] and Malte J. Rasch[2,a)]

AFFILIATIONS

[1] IBM Research Europe, 8803 Rüschlikon, Switzerland
[2] IBM Research - Yorktown Heights, Yorktown Heights, New York 10598, USA
[3] IBM Research - Almaden, San Jose, California 95120, USA

[a)] Authors to whom correspondence should be addressed: kelmaghr@us.ibm.com and malte.rasch@ibm.com

ABSTRACT

Analog In-Memory Computing (AIMC) is a promising approach to reduce the latency and energy consumption of Deep Neural Network (DNN) inference and training. However, the noisy and non-linear device characteristics and the non-ideal peripheral circuitry in AIMC chips require adapting DNNs to be deployed on such hardware to achieve equivalent accuracy to digital computing. In this Tutorial, we provide a deep dive into how such adaptations can be achieved and evaluated using the recently released IBM Analog Hardware Acceleration Kit (AIHWKit), freely available at https://github.com/IBM/aihwkit. AIHWKit is a Python library that simulates inference and training of DNNs using AIMC. We present an in-depth description of the AIHWKit design, functionality, and best practices to properly perform inference and training. We also present an overview of the Analog AI Cloud Composer, a platform that provides the benefits of using the AIHWKit simulation in a fully managed cloud setting along with physical AIMC hardware access, freely available at https://aihw-composer.draco.res.ibm.com. Finally, we show examples of how users can expand and customize AIHWKit for their own needs. This Tutorial is accompanied by comprehensive Jupyter Notebook code examples that can be run using AIHWKit, which can be downloaded from https://github.com/IBM/aihwkit/tree/master/notebooks/tutorial.

© 2023 Author(s). All article content, except where otherwise noted, is licensed under a Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/). https://doi.org/10.1063/5.0168089

02 July 2024 08:09:40

I. INTRODUCTION

Despite providing remarkable breakthroughs in various domains, Deep Neural Networks (DNNs) have been accompanied by a dramatic and growing increase in computational demands for training and inference. With the slowing down of Moore's law and the ending of Dennard scaling, power consumption becomes a key design constraint. Thus, energy-efficient implementations on emerging specialized hardware that leverages approximate and in-memory computing techniques have become essential for AI systems. This has been accompanied by a rise in dedicated AI hardware accelerators and an increased interest in AI processors that are efficient, fast, or both when carrying out AI tasks. In addition to traditional digital accelerators, including the Google Tensor Processing Unit, Amazon Inferentia, and IBM Artificial Intelligence Unit,[1] accelerators based on Analog In-Memory Computing (AIMC) using Non-Volatile Memory (NVM) are being actively researched.[2–4] AIMC accelerators that are based on resistive memory device technologies, such as Phase Change Memory (PCM),[5–8] Resistive Random Access Memory (ReRAM),[9–12] and Magnetic

© Author(s) 2023

Random Access Memory (MRAM),[13] have shown great promise in accelerating and reducing the power consumption of deep learning systems. By leveraging the physical properties of such memory devices, computations are performed at the same place where the data are stored, which could considerably improve the run-time and power consumption of today's digital computing technology.[14] In an AIMC chip, spatially instantiated synaptic weights are encoded in the tunable analog conductance of these devices arranged in crossbar arrays. Matrix–Vector Multiplications (MVMs) are among the most ubiquitous operations in deep learning and can be performed directly using the network weights stored on the chip.[15] In addition, weight updates for DNN training can be performed in-place by tuning the device conductance with suitable programming pulses.[16,17]

However, despite prolonged ongoing efforts, analog resistive memory devices suffer from various nonidealities, such as device-to-device and cycle-to-cycle variations. These inherent characteristics limit their accuracy and reliability for use in practical deep learning workloads.[18–20] Therefore, many large-scale simulations encompassing device and circuit nonidealities have been performed to quantify their impact on DNN accuracy for training and inference.[21–28] Although some of these studies have been realized on circuit-level simulators (e.g., SPICE), the size and complexity of deep learning workloads motivated the adoption of an alternative approach of using customized simulation frameworks/toolkits, which are integrated into modern deep learning frameworks, including PyTorch and TensorFlow. In contrast to SPICE-based simulation, which is cycle-accurate, this new alternate approach provides an interface between accurate mathematical models of non-ideal device characteristics and peripheral circuitry and high-level deep learning frameworks. This methodology enables seamless integration between modern DNN frameworks and the noisy physical characteristics of AIMC hardware by modeling the physical properties of AIMC and taking them into account for the training and inference of state-of-the-art DNN models. It is within this scope that we have recently open-sourced the IBM Analog Hardware Acceleration Kit (AIHWKit), a simulation toolkit that focuses on the algorithmic and functional levels as opposed to hardware and circuit design levels.[29] The aim of this toolkit is to provide a complete software package to estimate the accuracy of DNNs mapped to AIMC hardware for the advancement of algorithmic analog deep learning.

TABLE I. Comparison of AIHWKit with different related open-source AIMC simulation frameworks/toolkits. Traditional SPICE-based simulators are not compared.

Framework		NeuroSim and derivatives[31–35]	XB-SIM[36]	MemTorch[37,38]	IBM analog hardware acceleration kit[29]	CrossSim[39]
Year		2017	2019	2020	2021	2022
Prog. language(s)		Python, C, C++	Python, C++, CUDA	Python, C, C++, CUDA	Python, C++, CUDA	Python, CuPy
ML library	PyTorch	✓		✓	✓	
	TensorFlow	✓				✓
Supported network types	Dense (MLP)[a]	✓	✓	✓	✓	✓
	Convolutional	✓	✓	✓	✓	✓
	Recurrent			✓	✓	
	Transformer				✓	
On-chip inference	Accuracy est.	✓	✓	✓	✓	✓
	HW-calib. noise	✓			✓	✓
	HWA training[b]	✓			✓	
	Performance est.	✓	✓			
On-chip training	Digital gradient	✓			✓	✓
	In-memory grad.				✓	
	Performance est.	✓				
Unit testing				✓	✓	
Package index(s)				PyPi	PyPi, CF[c]	
Actively maintained[d]		✓			✓	✓

[a]Multi-layer perceptron.
[b]Hardware-aware training.
[c]Conda–Forge.
[d]As per the current date of publication.

02 July 2024 08:09:40

© Author(s) 2023

In Table I, we compare the key features of AIHWKit to those of related open-source AIMC simulation toolkits. Traditional simulators, i.e., SPICE-based simulators, are not compared. We refer the reader to Ref. 26 for a more comprehensive overview. As listed in Table I, only three out of the listed five toolkits are actively maintained: NeuroSim, AIHWKit, and CrossSim. The toolkits are compared against five key dimensions: Machine Learning (ML) libraries, supported network types, on-chip inference capabilities, on-chip training, and on-chip inference. Despite its current lack of support for performance estimation, AIHWKit is the only actively maintained tool that supports all the features listed and fully embraces modernized software engineering practices. In addition to being available on popular package indices (PyPi and conda–forge[30]), AIHWKit uses automated continuous integration and continuous development services (CI/CD) (e.g., Travis) to execute unit tests and to build and deploy standardized packaged releases.

It is noted that a large number of AIMC simulation frameworks have been developed. However, most of them remain closed-source or have been solely used for standalone research projects. Hence, they have not attracted significant attention from the broader research community. Consequently, they have been omitted from our comparative study. While many of these toolkits are complementary in nature, such as those listed in Table I, it is clear that the lack of standardization and excessive tool fragmentation are still prevalent when it comes to AIMC simulation and software toolkits.

The rest of this paper is organized as follows: In Sec. II, AIMC concepts are introduced to familiarize the reader with the kinds of research problems that can be tackled with AIHWKit. In Sec. III, a comprehensive overview of the AIHWKit design is provided, along with a detailed description of each simulated AIMC nonideality. Then, in Secs. IV and V, in-depth step-by-step descriptions on how to perform inference and training with AIHWKit are provided. We explain standard practices to faithfully capture hardware aspects as well as algorithmic techniques to improve accuracy. In Sec. VI, we present the Analog AI Cloud Composer, which leverages the AIHWKit simulation platform to allow a seamless, no-code interactive cloud-hosted experience and provide physical AIMC hardware access. In Sec. VII, we provide three concrete examples of customization of AIHWKit that the user could implement to fit their own research needs. Finally, Sec. VIII provides an outlook on possible future research directions and additions for AIHWKit.

II. AIMC CONCEPTS

A. Detailed introduction to AIMC

By exploiting the physical attributes of memory devices and their array-level organization, it is possible to perform specific computational tasks in the memory itself without the need to shuttle data between the memory and the processing units. The AIMC computational paradigm is paving the way for a range of applications, including scientific computing and deep learning.[2] Memory devices exhibiting two or more stable states can perform in-memory arithmetic operations, such as MVMs. For example, to perform the matrix–vector multiplication $W\mathbf{x} = \mathbf{y}$, the elements of matrix W, i.e., w_{ij}, can be mapped linearly to the conductance values of memory-based unit-cells organized in a crossbar configuration. The values of the input vector $\mathbf{x}$ can be mapped linearly to the amplitudes (durations) of read voltages, applied to the crossbar along the rows, or Word-Lines (WLs). The resulting current (charge) measured along the columns of the array, or Bit-Lines (BLs), will be proportional to the result of the computation, $\mathbf{y}$. Another attribute exploited for computation is accumulative behavior, whereby the device conductance progressively increases or decreases with the successive application of programming pulses. This enables the tuning of the synaptic weights of a neural network during training.

As shown in Fig. 1(a), an AIMC chip would ideally comprise a network of AIMC cores, each of which would perform a MVM primitive along with some light digital post-processing operations. Each AIMC core comprises a crossbar array of memory-based unit-cells along with bit-line drivers, analog-to-digital converters (ADCs), custom digital compute units to post-process the raw ADC outputs, local controllers, transceivers, and receivers. Core-to-core communication can be realized using a flexible on-chip network, akin to those used in traditional digital DNN accelerators. To realize a complete AIMC accelerator for DNN workloads, AIMC cores that each perform weight-stationary and energy-efficient MVM operations at $\mathcal{O}(1)$ time complexity can be combined with special-function Digital Processing Units (DPUs) to implement auxiliary DNN operations, such as activation functions and self-attention computation. Such an architecture is projected to provide highly competitive throughput while offering 40×–140× higher energy efficiency than an NVIDIA A100 graphics processing unit (GPU).[14] Therefore, there is a strong premise for AIMC to enable highly efficient execution of DNN workloads.

There are many promising candidates for the memory element in AIMC, including PCM, ReRAM, Electrochemical Random Access Memory (EcRAM), complementary metal-oxide semiconductor (CMOS) capacitive cells, Flash memory, MRAM, ferroelectric memory such as ferroelectric field-effect transistor (FeFET) or ferroelectric tunnel junction (FTJ), and photonic memory. The list shown in Fig. 1(b) is not a complete list of possible memory elements but provides examples of how analog resistance levels are achieved with various materials and circuit implementations. All devices described in Fig. 1(b) have hardware-calibrated models implemented in AIHWKit to simulate training and/or inference (see Secs. IV and V). In PCM, data are stored by using the electrical resistance contrast between a high-conductive crystalline phase and a low-conductive amorphous phase of a phase-change material. The phase-change material resistance can be modulated by creating amorphous regions of varying sizes through the application of electrical current pulses. ReRAM switches between high and low conductance states based on the formation and dissolution of a filament in a non-conductive dielectric material. Intermediate conductance is achieved either by modulating the width of the filament or by modulating the composition of the conductive path. EcRAM modulates the conductance between the source and drain terminals using the gate reservoir voltage that drives ions into the channel. Finally, CMOS-based capacitive cells can also be used as memory elements for analog computing, as long as leakage is controlled and the compute and read operations can be completed quickly.

Clearly, at the time of writing, there is still no "optimal" AIMC device technology, as each one of the current available technologies has its strengths and weaknesses, as illustrated in Fig. 1(b). For instance, PCM devices are arguably considered the most mature among resistive memory types; however, they suffer from temporal conductance drift, and the uni-polar/asymmetric switching

02 July 2024 08:09:40

© Author(s) 2023

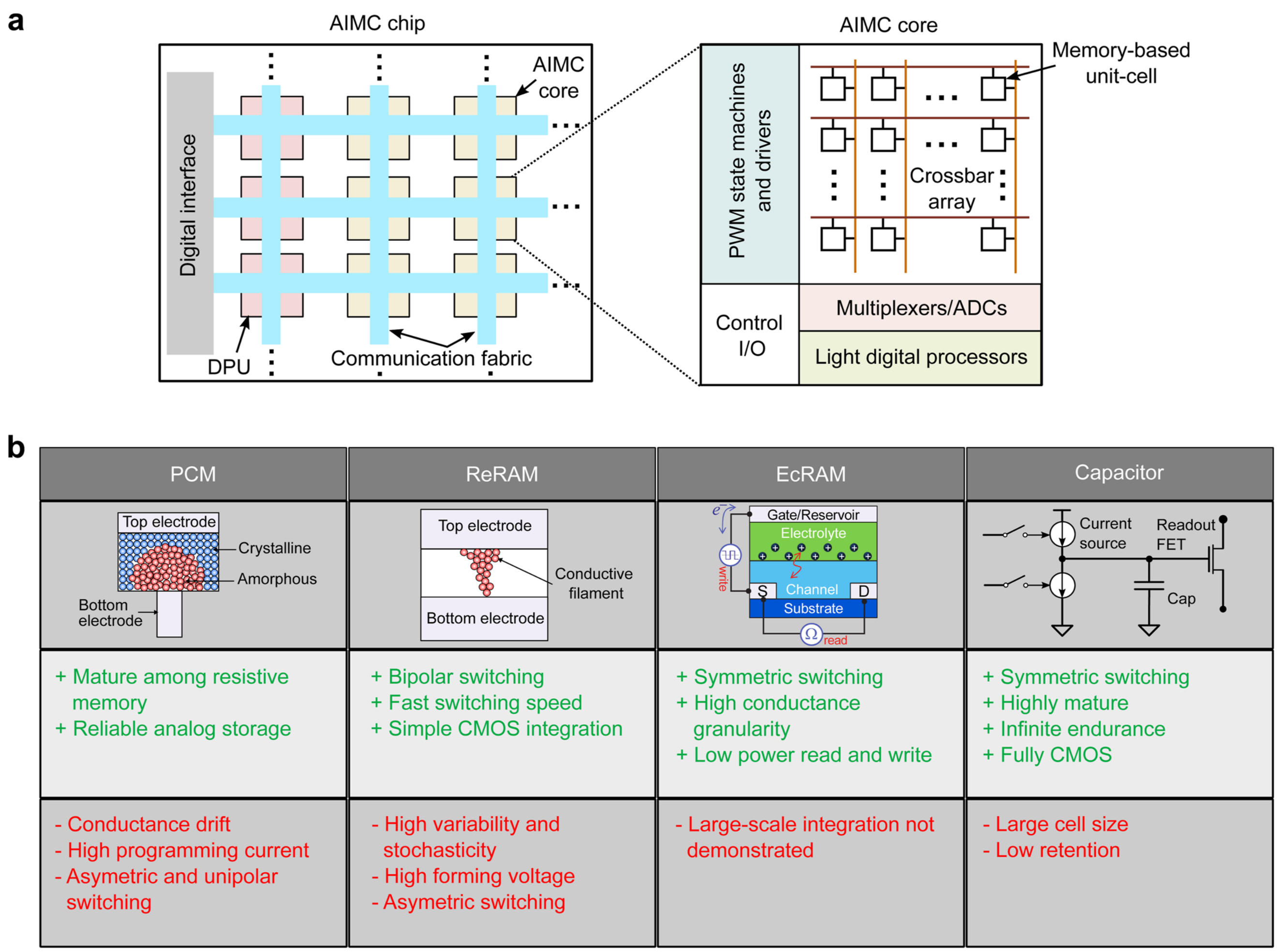

FIG. 1. (a), Illustration of a potential AIMC chip. (b) AIMC devices implemented in AIHWKit and their properties.

behavior leads to several complications for training. This is one of the key motivations behind building a simulator such as AIHWKit: to allow the exploration of the impact of various devices with their multitude of characteristics on the performance of AI models.

B. How to perform DNN training and inference with AIMC

A neural network layer can be implemented on (at least) one crossbar array of an AIMC core, in which the weights of that layer are stored in the charge or conductance state of the memory devices at the crosspoints [see Fig. 2(a)]. Because the state of a memory device can encode only a positive quantity, usually at least two devices in a differential configuration are used per weight: one to represent a positive synaptic weight component and the other to represent a negative weight component. The propagation of data through the layer is performed in a single step by inputting the data into the crossbar rows and deciphering the results in the columns. The results are then passed through the neuronal activation function and input to the next layer. The neuronal activation function is typically implemented at the crossbar periphery using analog or digital circuits. Because every layer of the network is stored physically on different arrays, each array needs to communicate at least with the array(s) storing the next layer for feed-forward networks, such as multi-layer perceptrons (MLPs) or convolutional neural networks (CNNs). For recurrent neural networks (RNNs), the output of an array needs to communicate with its input.

The efficient matrix multiplication realized via AIMC is very attractive for inference-only applications, where data are propagated through the network on offline, pre-trained weights. In this scenario, the weights are typically trained using conventional GPU-based hardware and then are subsequently programmed into the AIMC chip, which performs inference. However, because of device and circuit level nonidealities in the AIMC chip, custom techniques must be included in the training algorithm to mitigate their effect on network accuracy [so-called hardware-aware (HWA) training]. For inference tasks, device nonidealities that affect network

02 July 2024 08:09:40

© Author(s) 2023

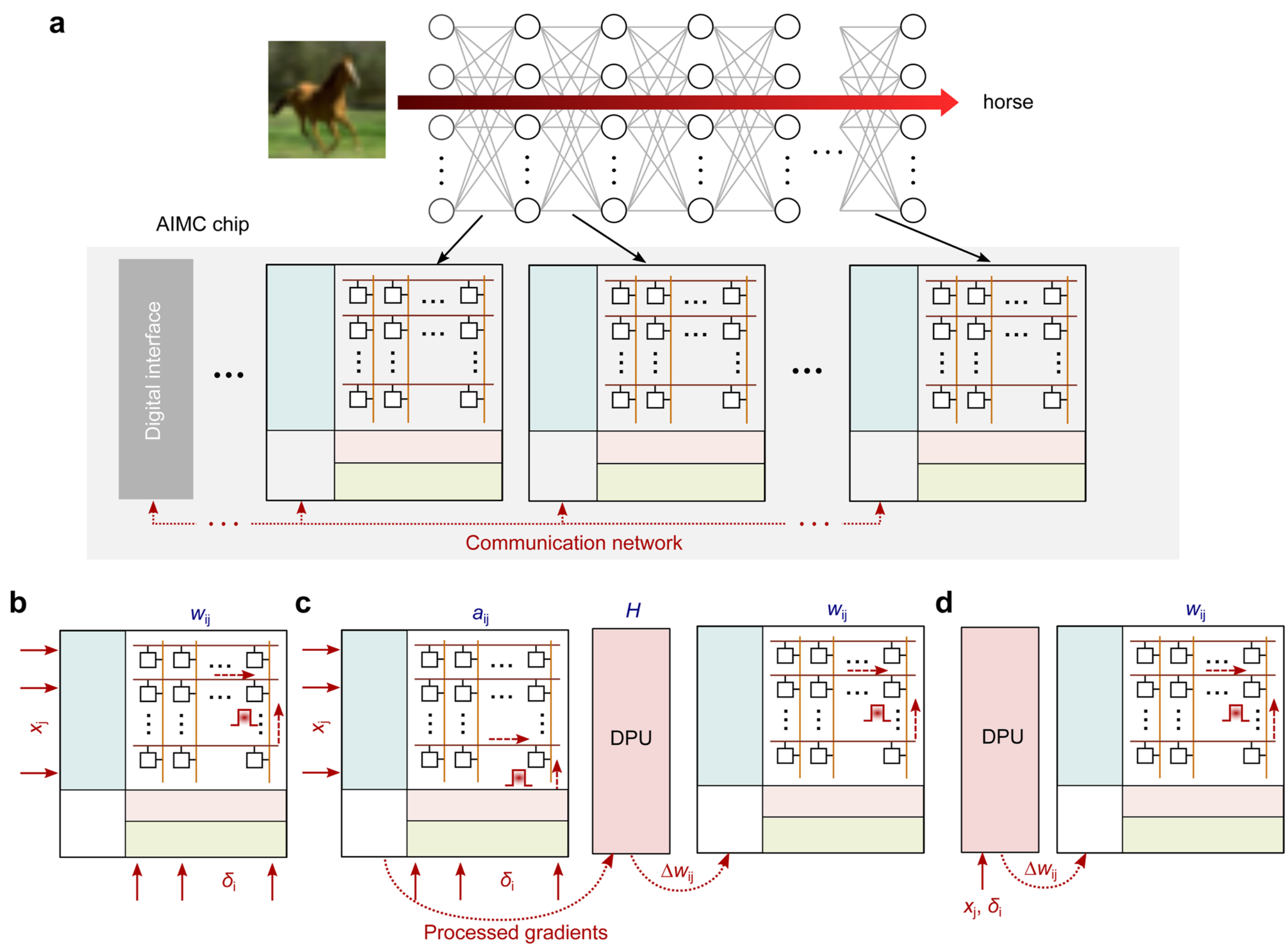

FIG. 2. (a) Mapping of a neural network to an AIMC chip. (b) Implementation of an in-memory SGD weight update. (c) Implementation of the TTv2 weight update. (d) Implementation of the mixed-precision weight update.

accuracy include conductance drift, programming errors, read noise, and stuck on/off devices. Circuit nonidealities, including finite resolution of digital-to-analog converters (DACs) and ADCs, parasitic voltage drops on the devices during readout when a high current is flowing through the crossbar wires (IR-drop), noise from the peripheral circuits at the crossbar output (e.g., amplifiers), and parasitic currents from sneak-paths during readout, will also negatively impact the accuracy.

AIMC can also be used in the context of neural network training with backpropagation. This training involves three stages: forward propagation of labeled data through the network, backward propagation of the error gradients from output to input of the network, and weight update based on the computed gradients with respect to the weights of each layer. This procedure is repeated over a large dataset of labeled examples for multiple epochs until satisfactory performance is reached by the network. When performing training of a neural network mapped on AIMC cores, forward propagation is performed in the same way as inference, as described above. The only difference is that all the activations x_j of each layer have to be stored locally in the periphery. Next, backward propagation is performed by inputting the error gradient δ_i from the subsequent layer onto the columns of the current layer and deciphering the result from the rows. The resulting sum $\sum_i \delta_i w_{ij}$ needs to be multiplied by the derivative of the neuron non-linear function, which is computed externally, to obtain the error gradient of the current layer. Finally, weight updates are implemented based on the outer product of activations and error gradients $\Delta w_{ij} \propto \delta_i x_j$ of each layer.

The weight update is performed in-memory by applying suitable electrical pulses to the devices, which will increase or decrease their conductance in proportion to the desired weight update. There are multiple approaches to perform the weight update with AIMC. Each approach has its advantages and drawbacks. One approach is to perform a parallel weight update by sending deterministic or stochastic overlapping pulses from the rows and columns simultaneously to implement an approximate outer product and program the devices at the same time [Fig. 2(b)].[16,17,40] This method, which we term in-memory stochastic gradient descent (in-memory SGD),

02 July 2024 08:09:40

© Author(s) 2023

has the advantage of performing a fully parallel analog weight update on the crossbar array at $O(1)$ time complexity and, therefore, is highly efficient in terms of speed. However, it requires stringent specifications on the conductance update granularity (minimum increase/decrease of device conductance with a single pulse), asymmetry (difference in device response when increasing or decreasing conductance), and linearity (dependence of conductance update on the device conductance state) to obtain accurate training, and high device endurance is critical. To mitigate some of these issues, the Tiki-Taka (TT) training algorithm was proposed,[41,42] which significantly relaxes the device conductance update requirements. Here, two matrices are encoded in AIMC cores, A and W. W encodes the network weights, whereas A computes and accumulates the weight gradient information. A is updated via parallel weight updates as described for in-memory SGD. After a certain number of updates on A, W is updated based on reading the gradient information from A via parallel weight updates. In the second version of Tiki-Taka (TTv2),[42,43] an additional matrix H, implemented in the digital domain, is used. H implements a low pass filter while transferring the gradient information processed by A to W, which further improves the robustness of non-ideal conductance updates. This low pass filter reduces the gradient noise and averages the gradient information over more inputs before updating the weights. A schematic implementation of the TTv2 weight update is shown in Fig. 2(c). Finally, a third approach is to perform so-called mixed-precision deep learning by computing the weight updates on a separate digital processor and accumulating them in a high-precision digital memory [Fig. 2(d)].[44] When the accumulated weight updates reach a threshold, the corresponding devices get updated through single-shot programming pulses. This approach is much less sensitive to nonidealities such as limited device granularity because the gradient is not computed using AIMC but instead in high-precision floating point (FP). It is also more flexible since the more complex learning rules can readily be implemented digitally. Moreover, the digital computation and accumulation of weight updates significantly relax the requirements on device endurance. However, the cost of the digital computations is significant [$\mathcal{O}(n^2)$ for a $n \times n$ weight matrix] and, thus, limits the speed of the AIMC training, even though forward and backward passes are fast ($\mathcal{O}(1)$). In contrast, for the in-memory SGD and Tiki-taka learning rules, the number of additional digital operations is linear to the size of the input vector ($\mathcal{O}(n)$) and often executed only periodically, so that the update is done much faster than for mixed-precision. All three methods presented here, as well as continuously improved versions, are implemented in AIHWKit, and Sec. V describes how to configure them for testing on different AIMC device models.

III. AIHWKIT DESIGN

As laid out in Sec. II, AIMC can accelerate certain parts of typical DNN (and other computing) workloads. Dense MVMs are particularly favorable for AIMC when the matrix elements are stationary and stored in (analog) memory. However, today's DNNs are often heterogeneous and include a variety of layers, such as nonlinear activation functions or attention mechanisms, that cannot be efficiently computed in-memory. The AIHWKit, which primarily focuses on functional verification of AIMC, is thus designed to handle both digital as well as AIMC components within the same DNN compute graph.

A. Simulator code-design overview

Since the AIHWKit is based on the ML framework PyTorch, the user can rely on the vast library of digital FP layers and functions for defining common DNNs. Only some layers of the DNN that are supposed to run on AIMC will use the simulation AIMC capabilities of the AIHWKit. The overall design is depicted in Fig. 3. The DNN is conveniently defined in standard PyTorch syntax using, e.g., `Linear` and `Conv2d` layer modules for fully connected and convolutional layers, respectively. If one decides to simulate such a layer on AIMC, AIHWKit provides corresponding layer modules, such as `AnalogLinear` and `AnalogConv2d`, respectively, that simulate the underlying matrix–vector products with customizable AIMC nonidealities. In such a way, the impact of AIMC nonidealities on the function of the DNN (e.g., prediction accuracy) can be measured. The analog layers available are listed in Table II. As illustrated further in Fig. 3, each analog layer module consists of one or multiple *analog tiles* that are meant to be a single physical crossbar core with an immediate periphery. Analog weights are assumed to be stationary once initialized. For instance, a large linear layer could be made up of multiple 512×512 crossbar arrays, where multiple non-ideal MVMs need to be performed and concatenated. The partial sum of the individual outputs is assumed to be computed with FP accuracy. In this case, the *tile module* would consist of multiple *analog tiles* with additional digital summations. Each analog tile in AIHWKit itself consists of a physical (simulated) memristive crossbar (of class `SimulatorTile`), as well as immediate periphery such as ADCs or error dynamic corrective steps such as noise or bound management.[45] Depending on the hardware customization, it can also hold an affine transform (digital output scales and biases, global or column-wise), which is known to greatly improve the mapping of weights to conductances and is needed for converting ADC-tics to meaningful quantities for the subsequent layers of the DNN (see also Ref. 22).

In AIHWKit, nonidealities, material response characterization, and general hardware configurations of each analog tile can be specified by a *RPUConfig*. The *RPUConfig* is, in principle, unique per analog tile; however, in common use cases, one assumes the same *RPUConfig* for each analog tile on the chip. We will explain how to configure the AIMC hardware using the *RPUConfig* in detail in Sec. III C. Internally, each analog tile will call the low-level `SimulatorTile` class to actually perform the non-ideal computations requested by the *RPUConfig*. As indicated in Fig. 3, a number of optimized core routines are available that simulate the AIMC MVMs. In particular for analog training, when the MVM as well as the outer-product update are both done in-memory, the C++/CUDA library (*RPUCuda*) is used through Python bindings to increase the simulation performance.

B. Model conversion and analog optimizers

As described in Sec. III A, typical PyTorch syntax is used to define the DNN to be simulated. This has the advantage that the vast amount of pre-coded and DNNs available for download from the ML community are readily usable for AIHWKit. However, layers within the DNNs that should run using AIMC need to

02 July 2024 08:09:40

© Author(s) 2023

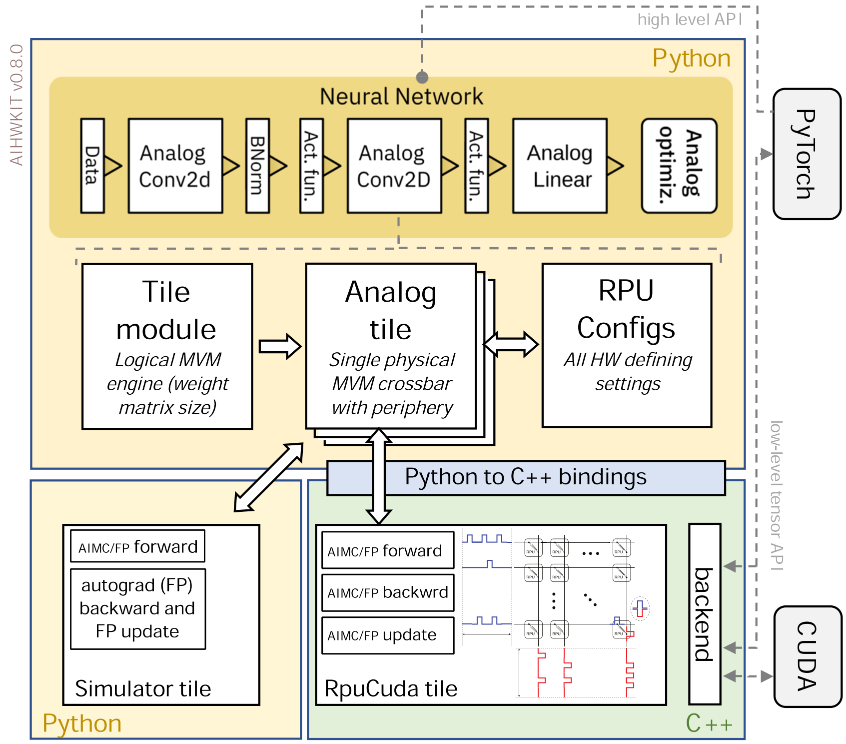

FIG. 3. Design of the AIHWKit. A DNN is defined with typical PyTorch commands, except for layers that are to be performed in AIMC. We provide analog layers to implement convolution layers, linear layers, etc. (see Table II). Each of these analog layer modules contains (at least) one analog tile module that encapsulates the analog computations as well as the concatenation of logical tile arrays. Each analog tile module consists of one or multiple *analog tiles*. These analog tiles encapsulate the NVM crossbar operations together with immediate peripheral compute (such as ADC and DAC, affine output scaling, and bias). Each analog tile can be configured in a broad way using a *RPUConfig*. The *RPUConfig* determines in a highly customizable way how the non-ideal AIMC forward, backward, and update behavior is actually implemented and what peripheral aspects and device materials are used in the AIMC hardware of investigation.

02 July 2024 08:09:40

TABLE II. Analog layer modules. In addition, the toolkit provides `mapped` versions that enforce the mapping of large weight matrices onto multiple physical tiles.

Module name	Torch equivalent	Functionality
`AnalogLinear`	`Linear`	Linear layer with bias
`AnalogConv1d`	`Conv1d`	1-dim convolution
`AnalogConv2d`	`Conv2d`	2-dim convolution
`AnalogConv3d`	`Conv3d`	3-dim convolution
`AnalogRNN`	`RNN`	Recurrent layer(s) with configurable cell
`AnalogLSTM`	`LSTM`	Uni/bi-directional LSTM layers

be replaced by their "analog" counterparts (see Table II). To ease the conversion of pre-coded (and possibly pre-trained) DNNs to AIHWKit, convenient conversion tools are provided that automatically replace PyTorch layers, such as `Linear`, with their counterparts, e.g., `AnalogLinear`. Thus, e.g., a call

```
from aihwkit.nn.conversions import convert_to_analog
analog_model = convert_to_analog(model, rpu_config)
```

would convert all applicable layers of the FP DNN to an *analog model* featuring AIMC layers, where all analog tiles instantiated are configured using the same hardware configuration, *RPUConfig*, and the FP weights. Note that here we always assume that enough analog tile resources are available on the chip to store the requested weight matrices of the DNN. Furthermore, weights are initialized perfectly without any programming noise, which is appropriate for untrained DNNs as the initial setting is random anyway. However, if weights are pre-trained, extra steps are necessary to actually program the weights into the conductances of the crossbars so that they show a realistic deviation from the targets as expected for the material choice. We will describe this process in detail in Sec. IV, where we also describe how inference is performed on this *analog model* and

© Author(s) 2023

TABLE III. Examples of different *config* classes (the suffix `RPUConfig` is omitted). Note that we make a distinction between chips that are only designed for inference (defined by *configs* having `Inference` in their name) and chips that support in-memory training (all other `RPUConfig` types). In the case of inference-only chips, only the forward pass is done with analog nonidealities, and tools are available to add phenomenological programming noise and drift during the evaluation (see Sec. IV). Training with such a configuration means "hardware-aware training," where a more robust FP model is trained, e.g., with noise injection to be programmed on the analog inference chip during the evaluation phase. On the other hand, in the case of in-memory training, the backward pass is non-ideal as well, and the weight update is defined by the material properties of the device model, as pulses will be used to incrementally update the device in-memory using the corresponding gradients. Thus, in this case, fully analog in-memory training is performed (see Sec. V for more details).

`RPUConfig` name	Algorithm	Forward	Backward	Update
Inference	AIMC inference/SGD	AIMC	FP	FP
TorchInference	AIMC inference/SGD	AIMC	FP	FP using PYTORCH[46] *autograd*
Single	In-memory SGD[16]	AIMC	AIMC	Stoch. pulsed in-memory update ($\overset{\cdots}{\longrightarrow} \check{W}$)
UnitCell	Specialized SGD	AIMC	AIMC	Using multiple devices (crossbars), based on the compound, see Table IX

how one could potentially retrain the model with noise injection for increased AIMC robustness.

The AIHWKit provides *analog optimizers*, such as `AnalogSGD`, that make PYTORCH aware of the analog layers so that the correct (custom) forward, backward, and update passes (and potential post-update steps) are performed, as requested in the *RPUConfig*. Before going into detail on training and inference, we first introduce the extensive hardware customization possibilities using the *RPUConfig* in the next Secs. III C and III D.

C. Tile-level *RPUConfig* specifies all analog hardware settings

The *RPUConfig* is a Python data class that has a number of fields and sub-structures that allow the specification of hardware properties, such as the amount and type of nonidealities, in the AIMC MVMs. On a higher level, AIHWKit provides a number of basic *RPUConfig* classes that are used to distinguish fundamentally different hardware designs. In particular, it distinguishes between in-memory analog training and chips that do not support training capabilities and instead are used for inference only. Inference-only configurations are based on the `InferenceRPUConfig` class, whereas in-memory training settings are either derived from the `SingleRPUConfig` or `UnitCellRPUConfig` classes (see Table III for an overview of different *RPUConfig* types). Note that the main difference between in-memory training and inference-only chips is how the backward and update nonidealities are defined. While for inference-only chips, they are simply done in FP (possibly implementing hardware-aware training, see Sec. IV), whereas in the case of in-memory training configurations, a plethora of device-material settings and parameters define specialized AIMC Stochastic Gradient Descent (SGD) algorithms.

02 July 2024 08:09:40

TABLE IV. Typical fields of the *RPUConfig* data class and their functionality. Note that not all fields are available for each of the *RPUConfig* types (see Table III). There are more fields available not mentioned here that are specific to `InferenceRPUConfig` (such as `noise_model` and `drift_compensation`), which specify hardware-aware training and evaluation options for inference-only chip designs (see Sec. IV for a detailed description).

RPUConfig field	Parameter class	Functionality
`tile_class`	...	Specifies the class used for the analog tile (e.g., `AnalogTile`)
`tile_array_class`	...	Logical array class used if requested (typical `TileModuleArray`)
`device`	`PulsedDevice` / `UnitCell`	Specifies the material device properties for in-memory update (e.g., ReRAM-like device-to-device variation during pulsed update)
`forward`	`IOParameters`	Specify the AIMC MVM nonidealities during the forward pass (e.g., IR drop strength)
`backward`	`IOParameters`	Specify the AIMC MVM nonidealities during the backward pass (transposed MVM)
`update`	`UpdateParameters`	Specify the pulsing properties during the update (e.g., pulse train length)
`mapping`	`MappingParameter`	Architectural and peripheral settings (e.g., maximal tile size, whether to use digital affine scales and biases)
`pre_post`	`PrePostProcessingParameter`	Pre-post processing (e.g., input range learning)

© Author(s) 2023

Given that the *RPUConfig* mainly specifies the hardware settings, in general, all its properties are assumed to be constant and non-changeable after the analog model is constructed using a particular *RPUConfig*. However, in some cases, one wants to experiment with one hardware setting, e.g., during training, while changing some hardware settings during inference, which would mean changing some properties of the *RPUConfig* after model creation. While this cannot be conducted directly by modifying the *RPUConfig* fields of the constructed model, it can still be conducted indirectly by exporting and importing its state, as long as the class of the *RPUConfig* does not change. In more detail, it can be achieved by constructing a second model `analog_model_new` using a new and modified *RPUConfig* `rpu_config_new` and loading the state dictionary from the first model `analog_model` without loading the *RPUConfig* from the state dictionary by using the `load_rpu_config` flag. For example:

```
analog_model = convert_to_analog(model, rpu_config)
# [..] e.g. train analog_model here. Then construct new model:
analog_model_new = convert_to_analog(model, rpu_config_new)
analog_model_new.load_state_dict(
    analog_model.state_dict(), load_rpu_config=False)
```

Now the new model, `analog_model_new,` has the same parameters as `analog_model` but a modified *RPUConfig*. Any further evaluation or training will thus be based on the new hardware configuration.

In Table IV, typical sub-fields of a *RPUConfig* are listed. Note that there are other fields that define additional input processing (`pre_post`) or the weight-to-tile mapping (`mapping`) properties. All nonidealities of the AIMC MVM itself are defined in the `forward` and `backward` fields, respectively, as described in Sec. III D.

TABLE V. `IOParameters` class customizes the MVM AIMC nonidealities. Here, a selection of commonly used settings is summarized. Note that the nonidealities can be selected independently for a "normal" MVM during the forward pass and the transposed MVM, which is used during the backward propagation in the case of in-memory training (see *RPUConfig* field in Table IV).

Class field	Typical value	Functionality
`is_perfect`	False	Debug switch for removing all nonideality settings
`mv_type`	*OnePass*	Select the type of analog mat-vec computation. For instance, whether only one pass is performed, so that negative and positive currents are added in analog, or multiple passes, where positive and negative inputs are given sequentially in two passes
`noise_management`	*AbsMax*	Type of noise management,[16] which is a dynamic input scaling per input vector (dynamic quantization)
`bound_management`	*None*	Type of output bound management. When set to *Iterative*, each MVM is "speculatively" computed, which means that it is dynamically recomputed with reduced inputs only if the output is hit. Note that this incurs a run-time penalty in practice
`inp_bound`	1.0	Input bounds and ranges for the digital-to-analog converter (DAC). The MVM computation is typically normalized to a fixed −1 to 1 input range
`ir_drop`	1.0	Scale of IR drop along the inputs (rows of the weight matrix)
`w_noise`	0.01	Scale of output referred MVM-to-MVM weight read noise
`w_noise_type`	*AddConstant*	Type of the weight noise, for instance, adds a constant Gaussian to each weight element
`inp_noise`	0.0	Standard deviation of Gaussian (additive) input noise (after applying the DAC quantization)
`inp_res`	254	Resolution (or quantization steps) for the full input (signed) range of the DAC
`inp_sto_round`	False	Whether to enable stochastic rounding of the DAC
`out_bound`	10.0	Output range for an analog-to-digital converter (ADC) in normalized units. Typically, maximal weight and input are normalized to 1, so that 10 means outputs are clipped at a current generated from 10 max inputs with all max weights
`out_noise`	0.04	Standard deviation of Gaussian output noise
`out_nonlinearity`	0.0	S-shaped non-linearity applied to the analog output (with possible output-to-output variation)
`out_res`	254	Number of discretization steps for ADC or resolution in the full (signed) output range
`out_sto_round`	False	Whether to enable stochastic rounding of the ADC

02 July 2024 08:09:40

© Author(s) 2023

D. Configurable MVM nonidealities

As mentioned above, MVMs implemented on AIMCs are nonideal. This is due to a number of device and circuit nonidealities, including but not limited to device-to-device and cycle-to-cycle conductance variations, output noise, weight read noise, IR drop, and quantization noise. The `forward` field of *RPUConfig* handles attributes related to how each AIMC MVM is to be performed in the forward pass (during inference as well as during training), and the `backward` field handles all attributes related to a possibly non-ideal backward pass during backpropagation. It is noted that all `forward` or `backward` attributes *do not change* the underlying weights (conductances) from one MVM to the next. Instead, *reversible* noise is added as requested, and for some nonidealities, such as IR-drop, the expected MVM output is modified in-place.

Long-term effects, such as diffusion processes, are not considered by default at the level of the duration of processing a single mini-batch. Instead, diffusion or decay processes can be applied only after processing a mini-batch. The user has the responsibility to ensure that this approximation of the long-term effects is reasonable for the hardware and materials under investigation. Other long-term weight-related effects, including programming noise, retention, $1/f$ noise, and drift, can be specified using specialized *RPUConfig* fields related to inference (e.g., `noise_model`; see Sec. IV for details).

Mathematically, the generally simulated AIMC forward and backward passes can be expressed as

$$y_i = \alpha_i^{\text{out}} f_{\text{adc}}\left(\sum_j (\breve{w}_{ij} + \sigma_{\text{w}}\xi_{ij})(f_{\text{dac}}(x_j) + \sigma_{\text{inp}}\xi_j) + \sigma_{\text{out}}\xi_i\right) + \beta_i, \quad (1)$$

where f_{adc} and f_{dac} model are the (possible non-linear) analog-to-digital and digital-to-analog processes (together with dynamic scaling and range clipping), and the ξ are the Gaussian noise. In general, it is assumed to have an analog part of the weight $\breve{W}$, the *analog weight*, that is stored in physical units. The ADC counts (that have arbitrary units) are then converted back to the correct FP range by a digital out-scaling factor(s) α_i^{out} that could either be set to be column-wise (i.e., depending on i) or tile-wise. The bias β_i could be digital or analog as well. Mathematically, because of the output scales, the actual weight W is given by a combination of the analog weight $\breve{W}$ and the output scales, $W \approx \alpha^{\text{out}}\breve{W}$. Since the physical units of $\breve{W}$ and y_i are, therefore, arbitrary (they can be incorporated in α^{out}), we define the analog weight as well as the input voltage in normalized units (maximally 1) for simplicity and define all MVM nonideality parameters with respect to these normalized units.

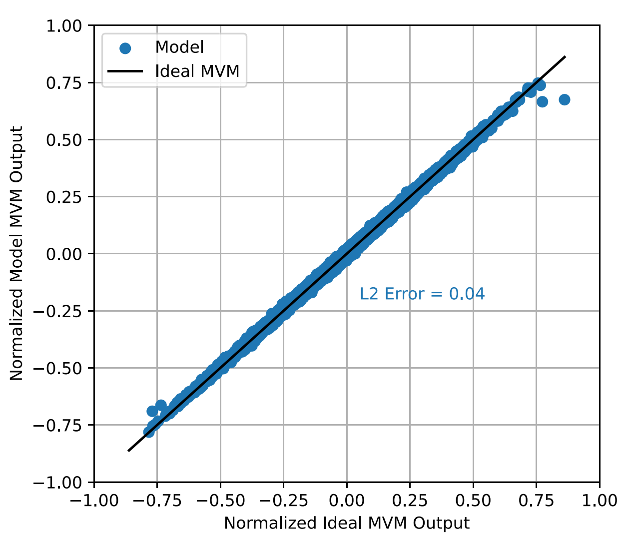

FIG. 4. Non-ideal MVMs from a 512 × 512 analog tile simulated using the AIHWKit with commonly used settings, as listed in Table V, when programming noise is not applied. Inputs are sampled from a sparse uniform distribution with a sparsity of 50%, and weights are sampled from a clipped Gaussian distribution with a standard deviation of 0.246. Output values are normalized using `out_bound`, so clipping happens at different normalized output values.

A non-ideal MVM performed with AIHWKit using the typical value settings shown in Table V is depicted in Fig. 4. In general, *RPUConfig* fields can be specified by either passing the keyword values to the `IOParameter` class or by simply modifying the attributes of the class. For instance, to set the `out_noise` parameter of the forward and backward passes, one can write

```
from aihwkit.simulator.configs import SingleRPUConfig, IOParameters
# choice 1
rpu_config_1 = SingleRPUConfig(
                  forward=IOParameters(out_noise=0.1),
                  backward=IOParameters(out_noise=0.1)
              )
# choice 2
rpu_config_2 = SingleRPUConfig()
rpu_config_2.forward.out_noise = 0.1
rpu_config_2.backward.out_noise = 0.1
```

Note that while for inference-only chips the forward pass matters (see Sec. IV), for in-memory training both forward pass and backward MVM nonidealities are set separately. Most *RPUConfig* related classes and enumerators can be imported from `aihwkit.simulator.configs`. Consequently, we will omit the import statements below. In Secs. III D 1–III D 7, we give an overview of the different configurable MVM nonidealities that can be simulated using the AIHWKit.

1. AIMC network weight encoding

When performing MVMs, the conductance of NVM elements is usually linearly mapped to a range of weight values, and it is assumed that a typical pulse-width modulation of the voltage input[5,6] can be approximated by a time average (so that x corresponds to the mean voltage given). Multiple-passes per MVM (for example, applying positive and negative inputs in two separate phases) can be simulated. However, the toolkit currently does not natively support a bit-wise "digital" mapping of weights, where only 1 and 0 states are (approximately) represented by conductances, and multiple devices are used with different significances to approximate a digital MVM.[19] However, it could be readily implemented by defining a new analog tile module that consists of multiple analog tiles representing different significances and summing over the individual outputs. In Sec. VII, we give some examples of how to customize analog tile modules.

Before going into more detail about describing the simulated AIMC nonidealities, there are a number of configurations that define how to map the FP weights W to the analog weights $\breve{W}$

02 July 2024 08:09:40

© Author(s) 2023

and the output scales α^{out} [see Eq. (1)]. These are governed by the `MappingParameter` in the `mapping` field of the *RPUConfig*.

a. Analog tile size and bias. The `max_in_size` and `max_out_size` properties set the (maximal) tile size in the input and output dimensions. If for a given layer the weight matrix is larger than this maximal size, multiple analog tiles will be used to represent the full weight matrix, where the outputs of each tile are assumed to be added up in FP precision (after ADC conversion) and concatenated and split as demanded. Note that currently there is no accuracy effect from limiting the output size, as simulations are all independent of columns. Thus, to increase simulation speed, it is advisable in most cases to set `max_out_size` to 0 to turn off the splitting. However, the input size is crucial for some nonidealities (such as IR drops or ADC saturation) and, thus, should be set as required by the hardware design.

The bias of the analog layer can either be encoded in the analog tile (as an additional column) or assumed to be digital (selected with `digital_bias`).

b. Initial weight mapping. The property `weight_scaling_omega` specifies how initially [when (re)setting the weights of an analog layer or using `analog_tile.set_weights()`] weights W are distributed among the analog weights $\breve{W}$ and the output scale(s) α^{out}. The value specifies the analog weight value $\breve{w}^*$ that is used for the absolute max of $w_{\max} \equiv \max_{ij}|w_{ij}|$. Thus, if `weight_scaling_omega` equals ω (and `weight_scaling_columnwise = False`), then $\alpha^{\text{out}} \leftarrow w_{\max}/\omega$ and $\breve{W} \leftarrow \omega W/w_{\max}$. Typically, $\omega = 1$ for inference or somewhat smaller for training (see Ref. 47 for details). This initial weight mapping can also be done per column (thus computing the maximum per column and having individual output scales per column) when setting `weight_scaling_columnwise`.

Note that for the special case $\omega = 0$, the initial weight mapping is turned off, that is, $\alpha^{\text{out}} = 1$. In this case, the user has to make sure that MVM nonideality values are correctly specified and weights are not too large to invalidate range assumptions. It is advisable to always map the weights correctly to avoid these complications. Moreover, the AIHWKit supports learning the digital output scales during training, either as tile-wise or column-wise scales (`out_scaling_columnwise`), which is enabled with `learn_out_scaling`.

2. Output noise

When an analog MVM is performed, weight-independent noise from the peripheral circuits at the crossbar output is introduced, from sources such as operational transconductance amplifiers used in ADCs. This is referred to as *output noise*, which is called σ_{out} in Eq. (1). In the AIHWKit, output noise is assumed to be additive Gaussian, i.e., it is sampled from a normal distribution centered around zero. The standard deviation of the output noise σ_{out} can be specified with `out_noise` (see Table V).

3. Short-term weight noise

In addition to output noise, when performing MVMs, weight-dependent noise, referred to as short-term weight noise, can be applied. In Eq. (1), this noise corresponds to σ_{w}. This Gaussian noise of zero mean thus models variations in the weights that occur every time an MVM is performed, such as short-term read fluctuations. For efficiency of implementation, this noise is applied to the output y_i and, therefore, does not modify the actual weight matrix from one mini-batch to the next. In principle, the σ_{w} could be a function of actual conductances and inputs. The AIHWKit so far supports three different types of short-term weight noise, which are listed and described in Table VI. The weight noise type is specified by `w_noise_type` and its standard deviation or scale by `w_noise` (see Table V).

4. Input and output quantization

In conventional AIMC systems, for each crossbar, analog-to-digital and digital-to-analog conversions are required to convert the WL inputs and BL outputs using DACs and ADCs, respectively. Due to practical constraints, these conversions are performed with reduced precision and, thus, introduce input and output quantization noise. In the AIHWKit, both input and output quantization are modeled using the following assumptions: values are bounded between a fixed range, i.e., a minimum and maximum value, and

TABLE VI. Types of short-term weight noise set using the `w_noise_type` property.

Type	Description
`NONE`	Do not apply short-term weight noise
`ADDITIVE_CONSTANT`	Apply constant additive noise with a standard deviation given by `w_noise`. Note that the weight noise is applied directly to the mapped weights [they can be accessed with `get_weights(apply_weight_scaling = False)`], which are typically in the range $-1, \ldots, 1$
`PCM_READ`	Apply output-referred PCM-like short-term read noise that scales with the amount of current generated for each output line and thus scales with both conductance values and input strength. In this case, `w_noise` specifies the scale, for which a value of 0.0175 has been found to capture PCM device measurements (for details, see Ref. 22, section "Short-term PCM read noise")

02 July 2024 08:09:40

© Author(s) 2023

$2^{n_{\mathrm{bit}}} - 1$ quantization states are linearly spaced between (inclusive of) these values. Optionally, one can also add input and output noise to model conversion inaccuracies and S-shaped output non-linearity to model non-linear ADC saturation.

Generally, the input (DAC) and output (ADC) quantizations are modeled as uniform quantizations between symmetric bounds around zero. In more detail, it is

$$\mathrm{quant}_b^r(z) \equiv \mathrm{clip}_{-b}^b\left(2br\,\mathrm{round}\left(\frac{z}{2br}\right)\right), \tag{2}$$

where the resolution r controls the number of bins in the range $-b, \ldots, b$. The distance between adjacent bins is $2br$. The input and output resolution can be specified using the `inp_res` and `out_res` properties of the `IOParameters`, respectively, and the bounds with `inp_bound` and `out_bound` (see Table V).

The resolution can either be set as the number of discrete values using an integer value or as the distance relative to a range of 1 between each discrete value (the resolution) using a floating point value. Assume that the bound is set to $b = 1$ and the resolution to $r = 1/2$. This would result in a partition of three bins, namely $-1 \le x < -\frac{1}{2}$, $-\frac{1}{2} \le x < \frac{1}{2}$, and $\frac{1}{2} \le x \le 1$ (where the value x is clipped at the bounds). This would need at least two bits to code in digital (one of the $2^2 = 4$ values is discarded). Thus, in general, to set a bit resolution of, e.g., $n_{\mathrm{bit}} = 8$, the resolution parameters need to be set to either $(2^{n_{\mathrm{bit}}} - 2)$ or $1.0/(2^{n_{\mathrm{bit}}} - 2)$. If this is set to -1, quantization noise is not modeled; however, the clipping bound is still applied. Stochastic rounding[48] can be modeled by enabling the boolean `inp_sto_round` and `out_sto_round` properties.

Input and output bounds, i.e., the clipping bounds/ranges for ADCs and DACs [see Eq. (2)], can be specified using the `inp_bound` and `out_bound` properties, respectively (see Table V). The input bound corresponds to the maximum (read) voltage amplitude/duration for a given WL input. Typically, we assume that the `inp_bound` is set to 1.0 so that the voltage is given in normalized units and maximally 1. To convert the actual input range into these normalized units, an additional scalar factor is used, which can also be learned (see Sec. IV B 4) or dynamically set (see Sec. III D 6 for details).

The output bound is a design choice referring to the maximally accumulated currents before the ADC saturates. Typically, we assume that weights are given in normalized units as well and clipped at maximal 1 [which needs to be ensured by enabling remapping or clipping in the case of HWA training (Sec. IV) or is set as a material device property during training (Sec. V)]. Thus, if `out_bound` is set to 10 (the default), the ADC will saturate when more than ten inputs are maximally on (1) while all weights are set to the maximal conductance (1). In other words, the output bound can be interpreted as corresponding to the maximum number of devices in a given column that can be at a maximum conductive state when all corresponding WL inputs are at a maximum, i.e., 1.0, and all other WL inputs are disabled before hard ADC saturation occurs.

5. IR drop

Ideally, for each crossbar, the voltage along each BL can be assumed to be constant. In a real crossbar, however, finite wire resistance causes current and voltage drops between adjacent rows and columns. This phenomenon is commonly referred to as IR drop[49] and can be accurately modeled using a number of non-linear differential equations. In the AIHWKit, to keep the simulation-time reasonable when modeling IR drop, a number of approximations are made. First, IR drop is modeled independently for each BL, as column-to-column differences are implicitly corrected (to first order) when programming weights with an iterative-based programming scheme. Second, only the average integration current is considered. Finally, the solution is approximated with a quadratic equation. We refer to Ref. 22 for more details. The scale of IR drop `ir_drop` and the physical ratio of wire conductance from one cell to the next to the physical max conductance `ir_drop_g_ratio` can be set as part of the `IOParameters` (see Table V). The latter default value is computed with 5 μS maximal conductance and 0.35 Ω wire resistance, i.e., $(1/0.35/5 \times 10^{-6}) = 571\,428.57$. Note that the approximations made here to obtain a fast implementation do not allow an arbitrary setting of this parameter. The approximations only hold when the order of magnitude of this default value is not changed.

6. Noise and bound management

To avoid the operation of peripheral circuitry in non-linear regimes and to improve signal quality, noise and bound management can be employed.[45] Noise management is used to dynamically re-scale inputs using a linear factor, α, prior to digital-to-analog conversion to match the (fixed) input range, and bound management is used to dynamically avoid or minimize the amount of output clipping (e.g., by dynamically recomputing with down-scaled inputs when outputs were clipped). Note that while these dynamic techniques often improve accuracy, they also may implicate higher chip complexity to implement additional (FP) operations needed, which typically translate to higher run-time, energy, or performance costs (not captured with AIHWKit). Thus, the user needs to carefully adjust these settings as appropriate for the hardware under consideration. In any case, the AIHWKit can readily be used to quantify the impact on accuracy when enabling such dynamic compensation methods for a given AI workload.

Different types of noise and bound management strategies are available (see documentation for `NoiseManagementType` and `BoundManagementType`). By default, the following bound and noise management strategy types are used:

```
rpu_config.forward.bound_management = BoundManagementType.ITERATIVE
rpu_config.forward.noise_management = NoiseManagementType.ABS_MAX
```

The noise management type `NoiseManagementType.ABS_MAX` sets initially $\alpha \equiv \max_i |x_i|$ and, thus, divides the input by the absolute maximum, e.g., $\mathbf{x}/\alpha$, before reaching the DAC and then re-scales the output of the ADC with α again. For `BoundManagementType.ITERATIVE`, the MVM is recomputed iteratively with setting $\alpha \leftarrow \alpha/2$ until the output bounds are not clipped anymore. `max_bm_factor` sets the maximal bound management factor (if this factor is reached, the iterative process is stopped), and `max_bm_res` sets the maximum effective resolution number of the inputs. It is noted that, for inference, noise/bound management is typically disabled/not used, as it requires additional computational resources to be implemented in hardware and is not supported in typical AIMC inference chips.

02 July 2024 08:09:40

© Author(s) 2023

7. Other MVM nonidealities

In addition to the aforementioned nonidealities, the AIHWKit can be used to simulate many other MVM nonidealities, including but not limited to voltage offset variation, device polarity read dependence, output asymmetry, and S-shaped non-linearity. We refer the reader to the API documentation of the `IOParameters` for a comprehensive list of parameters and values, which have not been explicitly described in this section.

IV. ANALOG IN-MEMORY DNN INFERENCE

As previously mentioned, the AIHWKit can be used to accurately model AIMC MVMs and, by extension, DNN inference by simulating a large variety of device and circuit-nonidealities. In this section, we introduce additional nonidealities used to model DNN inference. In addition, techniques for training for inference, also referred to as HWA training, will be discussed. We also describe how externally trained models can be imported into the AIHWKit to perform inference evaluation simulations and discuss best practices for inference evaluation.

We assume that the reader is familiar with the AIHWKit high-level design (Sec. III) and how to configure the hardware characteristics using the *RPUConfig* (Sec. III C). In particular, here we discuss the situation of investigating a chip that is designed for AIMC inference only, so that the *RPUConfig* is derived from the `InferenceRPUConfig` class (see Table III). We will discuss the additional *RPUConfig* fields available for this case.

A. Noise models for inference

In Sec. III, configurable MVM nonidealities are described, which can be used for modeling both DNN on-chip inference and training. In Secs. IV A 1 and IV A 2, we introduce additional deviations and long-term effects on the weights, which are specific to DNN inference.

When evaluating a given analog model for inference accuracy, prior to the inference evaluations, programming noise, as well as long-term effects up to a time t_{inf} (such as drift and accumulated read noise, see below), need to be applied. In AIHWKit, this is done with special methods:

```
analog_model.program_analog_weights()
analog_model.drift_analog_weights(t_inf)
```

Thereafter, the test set can be evaluated with the correctly applied long-term effects on the model. In the following, we describe in more detail what noise and compensations are applied during these calls.

1. Phenomenological weight noise models

During inference, weight programming error, conductance drift, and read noise are modeled using phenomenological noise models. Some of these models, such as the `PCMLikeNoiseModel`[50] and `ReRamWan2022NoiseModel`,[9] are hardware-calibrated. The PCM model is calibrated using a large number of device measurements, as depicted in Fig. 5. The phenomenological noise model to use can be specified using the `noise_model` field of the *RPUConfig*, as follows:

02 July 2024 08:09:40

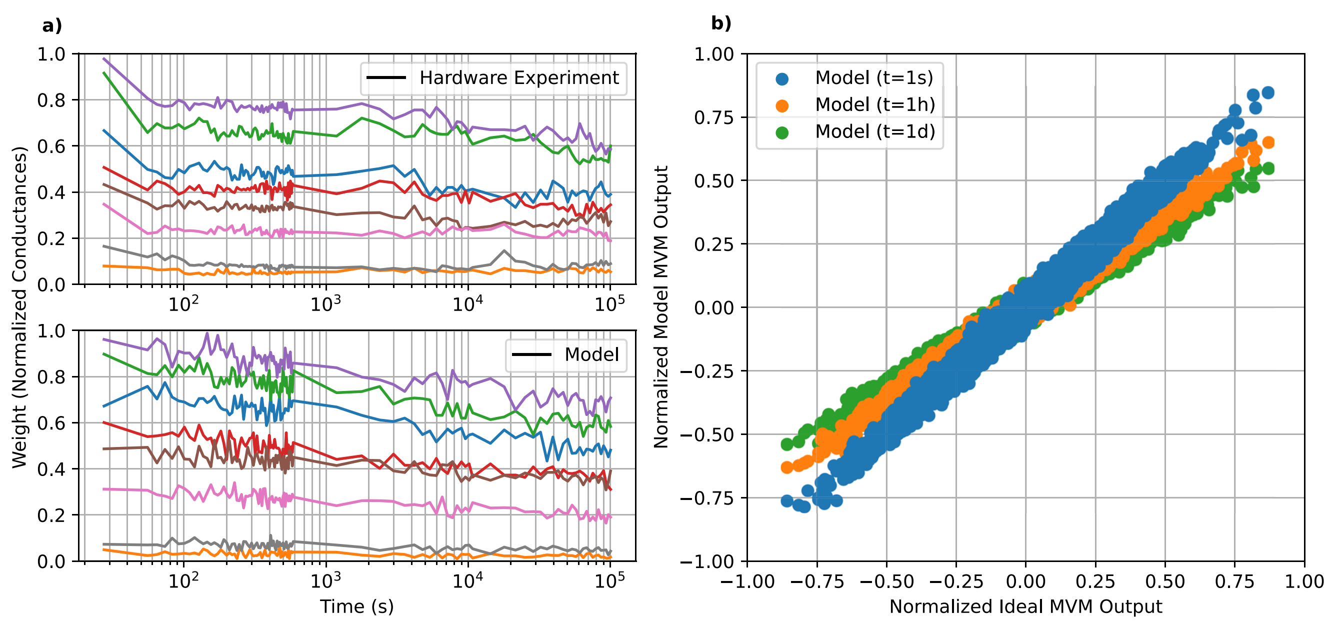

FIG. 5. (a) Experimentally (hardware) obtained temporal evolution of PCM conductance[21] compared to that simulated by the AIHWKit `PCMLikeNoise` statistical noise model. Note that it is assumed all weights are programmed at the same time in the simulation, whereas in the experiment, devices converged at different iterations of programming. (b) Non-ideal MVMs from a 512 × 512 analog tile simulated using the AIHWKit with commonly used settings, as listed in Table V, and the `PCMLikeNoise` statistical noise model. Inputs are sampled from a sparse uniform distribution with a sparsity of 50%, and weights are sampled from a clipped Gaussian distribution with a standard deviation of 0.246. For $t = 1$ s, the reported L_2 error of the MVM is 13%.

© Author(s) 2023

```
from aihwkit.inference import PCMLikeNoiseModel
rpu_config.noise_model = PCMLikeNoiseModel()
```

Note that most inference-only related classes and tools can be imported from `aihwkit.inference`.

a. Weight programming error. When programming real NVM devices, the programmed conductances, g_{ij}^{P}, differ from the desired target values, $\hat{g}_{ij}$, due to many underlying mechanisms, including but not limited to cycle-to-cycle and device-to-device variability, WL and BL voltage mismatches, device-level voltage asymmetries,[51] and temporal drift. While many of these mechanisms can be emulated for a given programming scheme to infer the weight programming error, it is much more computationally efficient to compute the programming error using an arbitrary function, $g_{ij}^{\mathrm{P}} = f(\hat{g}_{ij})$, which is defined for each device model and programming scheme. It is typically assumed that the weight error can be modeled using a normal distribution centered around $\hat{g}_{ij}$, where the standard deviation, σ, is dependent on $\hat{g}_{ij}$, as follows:

$$g_{ij}^{\mathrm{P}} = \mathcal{N}(\hat{g}_{ij}, \sigma(\hat{g}_{ij})). \tag{3}$$

In the AIHWKit, the `apply_programming_noise_to_conductance(g_target)` method of the noise model (base class) is used to apply the weight programming error. For more details and how to customize the noise model, see Sec. VII.

b. Conductance drift. Many types of NVM devices, most prominently PCM, exhibit temporal evolution of the conductance values, referred to as the conductance drift. This poses challenges for maintaining synaptic weights reliably.[52] Conductance drift is most commonly modeled using the following equation:

$$g_{\mathrm{drift}}(t) = g^{\mathrm{P}}(t/t_0)^{-\nu}, \tag{4}$$

where t_0 is the time at which the programmed conductance g^{P} is measured and ν is the drift exponent. In practice, conductance drift is highly stochastic because ν depends on the programmed conductance state and varies across devices. In the AIHWKit, the `apply_drift_noise_to_conductance(g_prog, nu_drift, t_inference)` method of the noise model (base class) is used to apply the conductance drift noise.

c. Low-frequency read noise. When devices are read, after the conductances have been programmed, there will be instantaneous fluctuations in the hardware conductances due to the intrinsic noise from the NVM devices. Many NVM devices exhibit $1/f$ noise and random telegraph noise characteristics, which alter the effective conductance values used for computation. This noise is referred to as read noise because it occurs when the devices are read after they have been programmed. Note that here we refer to longer-term and lasting effects on the conductances after programming, such as low-frequency $1/f$ fluctuations (typically much slower than processing a single mini-batch) as opposed to weight read fluctuations on the time-scale of a single MVM. Therefore, this read noise is resampled only once at every inference time t_{inf}. Short-term read fluctuations that are resampled every MVM can instead be set using the `IOParameters` as listed in Table V.

The low-frequency read noise is typically modeled using a normal distribution centered around zero with a standard deviation of σ_{nG} dependent on the time elapsed since the programming, i.e., $\mathcal{N}(0, \sigma_{nG}(t))$.[50] The conductance of a device as a function of time, accounting for both conductance drift and read noise, can be modeled using the following equation:

$$g(t) = g_{\mathrm{drift}}(t) + \mathcal{N}(0, \sigma_{nG}(t)). \tag{5}$$

In the AIHWKit, the `apply_noise(weights, t_inference)` method of the noise model (base class) is used to apply both conductance drift and read noise.

2. Drift compensation

Various methods can be employed to mitigate the effect of conductance drift during inference.[53] In the AIHWKit, such techniques are referred to as *drift compensation techniques*. As proposed in Ref. 54, a single scaling factor, γ, can be applied to the output of an entire crossbar (after the ADC) in order to compensate for a global conductance shift. In the AIHWKit, to compute the correct value for a time t_{inf} after the conductance programming (at $t_{\mathrm{inf}} = 0$), first a measure for the strength of a reference output using MVMs right after programming is stored in s_0. When compensating after a time t_{inf}, the same MVMs are computed with the drifted weights to get another output strength s_t. The compensation factor is then set to $\gamma \equiv s_0/s_t$. For the global drift compensation (`GlobalDriftCompensation`), the output strength is computed as the mean absolute y_i values resulting from giving all one-hot vectors as input. However, other strength measures can be implemented by customizing the drift compensation, as explained in Sec. VII.

The drift compensation type can be specified using the `drift_compensation` field of the *RPUConfig*:

```
from aihwkit.inference import GlobalDriftCompensation
rpu_config.drift_compensation = GlobalDriftCompensation()
```

B. Hardware-aware training for inference

HWA training, a popular alternative to on-chip training, can also be used to train networks for deployment on AIMC hardware. Unlike on-chip training, HWA training is solely performed in software and does not require detailed behavioral or physical device models. Instead, additional operations, such as weight noise injection, are added during forward and backward propagation passes, and standard SGD methods are used. These are added to increase the model's robustness[21,22,55–59] and can be specified using different *RPUConfig* parameters (as part of the `InferenceRPUConfig` class), which are discussed in Secs. IV B 1–IV B 4.

1. AIMC forward pass during HWA training

It is common for HWA training to assume a perfect backward pass, with nonidealities only added during the forward pass, which is the default behavior of `InferenceRPUConfig`. MVM nonidealities added to the `forward` field (see Table V) of the class are applied when the model is in `train()` mode and `eval()` mode. One can configure additional noise sources that are only present when the model is in `train()` mode (see Secs. IV B 2–IV B 4 for details). While `InferenceRPUConfig` uses a C++/CUDA backend, `TorchInferenceRPUConfig` is purely based on PyTorch, making debugging easier as one is able to step through every part of the forward pass. Switching to the PyTorch based tile is as simple as exchanging `InferenceRPUConfig` with `TorchInferenceRPUConfig` (see Table III).

02 July 2024 08:09:40

© Author(s) 2023

TABLE VII. Types of weight modifiers. Some experimental weight modifier types, including `WeightModifierType.DOREFA`, are not listed. Parameters are grouped in the class `WeightModifierParameter` and accessible in the `modifier` attribute of the *RPUConfig*.

Type	Description
NONE	No weight modifier is applied
DISCRETIZE	Weights are discretized (quantized) according to the resolution specified by `res`. If `sto_round` is enabled, stochastic rounding is performed
MULT_NORMAL	Multiplicative Gaussian noise is added to all weights with a standard deviation of `std_dev`
ADD_NORMAL	Additive Gaussian noise is added to all weights with a standard deviation of `std_dev`
POLY	Noise is added to all weights from a normal distribution with a standard deviation of $\sigma_{\text{wnoise}}(c_0 + c_1\|w_{ij}\|/\omega + c_N\|w_{ij}\|^N/\omega^N)$, where ω is either the actual absolute max weight (if `rel_to_actual_wmax` is set) or the value `assumed_wmax`. σ_{wnoise} is set using the `std_dev` parameter. The coefficients $c_0, \ldots, c_N$ are set using the `coeffs` parameter
PROG_NOISE	Identical to POLY except that a positive or negative weight will remain positive or negative, respectively, after the noise is applied to simulate the situation of programming the weight to two separate conductances depending on the sign. If weights change sign after applying noise, the absolute value with the preserved sign is taken

2. Weight modifier parameter

Weight modifier parameters (`WeightModifierParameter`), set using the special field `modifier` of the *RPUConfig*, are used to specify different attributes about the injected weight noise during HWA training, such as the noise type and amplitude. In Table VII, a description of each weight modifier parameter type is provided. When a weight modifier type other than COPY is used, unless otherwise specified, for the duration of a mini-batch, each weight will be modified during both forward and backward propagation cycles. Drop connect,[57,60,61] which is used to set weights to zero with a given probability during training, can be used with any other modifier type in combination. As an example, additive Gaussian noise with a standard deviation of 0.1 can be applied, in addition to drop connect with a drop connect probability of 0.05, as follows:

```
from aihwkit.simulator.configs import WeightModifierType
rpu_config.modifier.type = WeightModifierType.ADD_NORMAL
rpu_config.modifier.std_dev = 0.1
rpu_config.modifier.pdrop = 0.05
```

For relatively small networks and datasets, we found that increasing the number of times we draw samples from our weight distribution improves the robustness of the programming noise of our model. This can be achieved by adding noise drawn from the distribution specified by `WeightModifierType` for every sample in the batch. Concretely, for inputs of shape `[batch_size,d_in]` and a layer weight of shape `[d_in,d_out]`, instead of applying noise to the weights once, yielding again a matrix of shape `[d_in,d_out]`, we add noise for every sample in the batch, yielding a weight matrix of shape `[batch_size,d_in,d_out]`. This feature can be turned on by setting `rpu_config.modifier.per_batch_sample` to `True`. Note that this feature is only available for the PyTorch-based analog tile implementation, which can be selected by using `TorchInferenceRPUConfig` as the `rpu_config` class.

3. Weight clipping and remapping parameter

Weight clipping and remapping ensure that the weight is correctly mapped to (normalized) conductances in the range and, thus, should always be applied during HWA training (at least `fixed_value` clipping to 1) to avoid unrealistic weight ranges that are not in line with the assumptions when specifying the other MVM nonidealities (such as ADC range, etc., see Table V). Note that the weight range here refers to the *analog weight* $\breve{W}$. The actual FP weight is given by the $\breve{W}$ times the (digital) output scaling parameters (see Sec. III D 1 for details).

Weight clipping parameters (`WeightClipParameter`), set using the special field `clip` of the *RPUConfig*, are used to specify different attributes that control how weights are clipped during HWA training. In Table VIII, different weight clipping technique types are listed. Weight remapping parameters (`WeightRemapParameter`), set using the special field `remap` of the *RPUConfig*, are used to specify different attributes that control how weights are re-mapped to analog weights $\breve{W}$ and the output scales α^{out} during HWA training using the assumption of having digital output scales that can represent part of the full weight together with the value represented in the conductances (see Sec. III D 1). The `remapped_wmax` parameter specifies the assumed maximum analog weight value. This is typically set to 1.0. In Table VIII, different weight remapping parameters are listed. As an example, weight clipping using `LAYER_GAUSSIAN` at two times the standard deviation of the weight distribution and weight remapping (in `CHANNELWISE_SYMMETRIC` mode) can be enabled as follows:

```
from aihwkit.simulator.configs import WeightClipType, WeightRemapType
rpu_config.clip.type = WeightClipType.LAYER_GAUSSIAN
rpu_config.clip.sigma = 2.0
rpu_config.clip.fixed_value = 1.0
rpu_config.mapping.type = WeightRemapType.CHANNELWISE_SYMMETRIC
```

02 July 2024 08:09:40

© Author(s) 2023

TABLE VIII. Types of weight clipping and remapping techniques.

Type	Description
NONE	Clipping/remapping behavior is disabled
	Weight clipping
FIXED_VALUE	Weights are clipped to fixed value give, symmetrical around zero, specified by `rpu_config.clip.fixed_value`
LAYER_GAUSSIAN	Calculates the second moment of the whole weight matrix and clips at σ times the result symmetrically around zero. σ is specified using the `rpu_config.clip.sigma` parameter
	Weight remapping
LAYERWISE_SYMMETRIC	Remap according to the absolute max of the full weight matrix
CHANNELWISE_SYMMETRIC	Remap each column (output channel) in respect to the absolute max

Note that mapped weights in the analog representation should always be smaller than the assumed maximal value (typically 1) to ensure that this clipping at a fixed value can be used in combination.

4. Setting and learning the input ranges

As previously described, inputs are first clipped to a fixed range before being presented to each crossbar. The input range for each crossbar can either be learned during training, dynamically computed during inference, or fixed (set manually).

In the AIHWKit, pre-post processing parameters, specified using `PrePostProcessingParameter`, can be used to augment digital input and output processing steps. Currently, input range learning is the only natively supported processing step. Input range learning can be used to find the optimal input range for each crossbar during HWA training. For initialization, one can use the first `init_from_data` input batches for calculating a moving average of the `init_std_alpha`$^{\text{th}}$ standard deviation of the input distribution. After a certain number of batches have been presented, learning takes over. This is done by calculating the gradient of the input range to be proportional to the amount of clipping caused by the current input range and the gradient of the crossbar inputs. This typically widens the input range so that no clipping occurs; however, a tight input range is often more favorable since it reduces quantization error and boosts the overall signal strength, which is important if the hardware suffers from output noise. How much the input range is tightened at every backward pass can be controlled via the `decay` attribute, which adds `input_range`*`decay` to the gradient if not more than some percentage of the inputs are clipping. This percentage can be controlled via `input_min_percentage`. As an example, using a value of 0.95 as `input_min_percentage` will only lead to a tightening of the input range if less than 5% of the inputs have been clipped using the current input range. The input range can also be loosened up if the outputs are clipping at the ADC. This can be turned on by setting `manage_output_clipping = True`. Again, for `output_min_percentage = 0.95`, the input range is not loosened if less than 5% of the outputs are clipping. It should be noted that this feature is currently not supported in the torch-based tile. By default, the gradient of the input range (before decaying) is scaled by the current input range. To turn this feature off, set `gradient_relative = False`. For an example of how to use input range learning, see notebook `hw_aware_training.ipynb`.[62]

If learning the input ranges is not desired, but HWA training with DACs and ADCs is, then a second option is to use the `NoiseManagementType` to dynamically scale the inputs during HWA training and inference such that each input covers the full input range. However, note that this is typically not supported by most AIMC hardware due to the high computational overhead involved in implementing this dynamic range computation. For more details, refer to Sec. III D 6.

To simplify the HWA training, one might eventually want to train without DACs and ADCs altogether, in which case one can simply enable a perfect forward pass by setting `forward.is_perfect = True` in the *RPUConfig*. In this case, one has to calibrate them post-training before deploying them on hardware. Setting the input ranges post-training typically involves calibration using a subset of the training data. During the calibration phase, the model is in evaluation mode, which means that layers such as `torch.nn.Dropout` operates in inference mode, and any distortions such as output noise, weight noise, or input quantization are turned off. The activations from every crossbar are then cached until no more inputs are provided. To avoid exhausting the memory, one can set an upper limit on the number of activation samples cached at every crossbar. In order to prevent sampling of activations that are not representative of the true distributions, new samples are randomly mixed into the cache, which is then trimmed to the maximum number of samples. After the sampling phase, the `input_range` field of every crossbar is populated with a certain quantile of the recorded samples. This ensures that outliers are not mapped to the full range, causing an overall weak signal for the intermediate values. This model is demonstrated in notebook `post_training_input_range_calibration.ipynb`.[63]

For large models, caching even a couple of hundred activation samples per crossbar might already be too memory intensive. For this reason, a moving average of the quantile can be computed.

02 July 2024 08:09:40

© Author(s) 2023

This drastically reduces the memory footprint since no caching is required, but it still enables input range calibration on large amounts of data. However, the moving average is, of course, an approximation to the true quantile, which might lead to worse performance.

5. Importing externally trained models

Externally trained models can be imported to the AIHWKit, either to be retrained using HWA training for inference or direct inference evaluation. Currently, the AIHWKit natively supports the conversion of PYTORCH models, so models trained using other Machine Learning (ML) frameworks first require conversion to a PYTORCH-based model. External libraries, such as those listed here,[64] can be used to convert trained models from many popular libraries to PYTORCH-based models.

All linear (dense) and convolutional layers of an arbitrary PYTORCH-based model can be automatically converted to analog equivalent layers using the `aihwkit.nn.conversion.convert_to_analog(module, rpu_config)` methods, where the AIMC hardware properties (including tile size, etc.) are defined in `rpu_config`. Other layers, namely Long Short-Term Memory (LSTM) cells, require manual in-place conversion.

It should be noted that most imported models do not have pre-calibrated input ranges, which is why, most of the time, one needs to calibrate them after loading the model. For information on how to do that, see Sec. IV B 4.

6. Hardware-aware training example

For HWA training, one typically starts off with a model that was pre-trained without any AIMC nonidealities or techniques, such as weight clipping or noise injection. If there is a need for training from scratch, the user can either define the network in PYTORCH and then convert it using `convert_to_analog` or directly substitute the individual layers with their analog counterparts in the model definition. Notebook `hw_aware_training.ipynb`[62] demonstrates this workflow with a ResNet-32 trained on the Cifar-10[65] dataset. We start off by pre-training the model to the baseline accuracy, which in this case hovers around 94%. For the HWA training, we first generate an *RPUConfig* that is then used when converting the model to analog. For training an analog network, one has to use an `AnalogOptimizer`, which adds specific logic to be executed after parameter updates. In this case, we use the simple `AnalogSGD`; however, more complex algorithms can be used by mixing `AnalogOptimizerMixin` into the PYTORCH-based optimizer class (see `AnalogAdam` for an example). It should be noted that for HWA training, the learning rate might need to be reduced. By how much depends on the network, but reducing it by roughly one order of magnitude is a good starting point. Apart from that, we are able to use the same training code to do HWA training on the converted analog model since all HWA training parameters are automatically applied as defined in the *RPUConfig*. After HWA training, we perform inference using the model, which is now in `eval()` mode (see Sec. IV C for more information).

C. Inference accuracy evaluation

For a given *RPUConfig,* during inference evaluation of the analog model, the parameter setting specific to the HWA training, such as the specified weight noise modifier type, i.e., `rpu_config.modifier.type`, is not used (unless `modifier.enable_during_test` is explicitly set to `True` for debugging purposes). The MVM nonidealities, as specified by the `forward` field of the *RPUConfig* (see Table V), are, however, *always* applied (during HWA training as well as inference evaluation), since they define the AIMC properties rather than any extra regularization techniques for HWA training. As described in more detail in Sec. IV A, programming noise can be applied by calling either the `analog_model.program_analog_weights()` or `analog_model.drift_analog_weights(t_inference)` methods, which both inject programming noise using the `rpu_config.noise_model`. For the latter, in addition to programming noise, the current reference weights (i.e., the conductance state of all devices) are drifted for `t_inference` seconds.

1. Multiple models and evaluation instances

As AIMC hardware is inherently stochastic, a single evaluation instance is typically not representative of the behavior of the modeled hardware over multiple evaluation instances. Consequently, multiple evaluation instances should be used to evaluate both the mean and variance (typically the standard deviation) of the metrics being evaluated. Moreover, as many analog NVM devices, such as PCM, are susceptible to temporal conductance drift and the behavior of analog In-Memory Computing (IMC) hardware can evolve over time, performance-based metrics for analog IMC hardware are typically reported for a specific length of time with respect to a reference point-in-time. This is typically defined as the point-in-time when all devices have been programmed. Ideally, multiple model (random initialization) instances should also be used.

2. Inference evaluation example

Notebook `hw_aware_training.ipynb`[62] provides an inference configuration example that uses the `PCMLikeNoiseModel` during inference. The mean and standard deviation of the test set are reported for different logarithmically spaced time steps, from `t_inference` = 60.0 s up to one year ($365 \times 24 \times 60 \times 60$ s). For each point in time, the mean and standard deviation of the test set's accuracy are reported across five evaluation instances. Note that we kept the number of repetitions low for this example. In practice, one should repeat the same measurements at least ten times (we typically use 25). The soundness of the experiments can be even further improved, if computational resources allow, by training the same network multiple times and reporting the performance metrics averaged across the different model instances.

V. ANALOG IN-MEMORY DNN TRAINING

While using AIMC chips dedicated to inference is a common application for in-memory acceleration, the training of today's ever-increasing DNNs would greatly benefit from hardware acceleration as well. For that purpose, analog in-memory training algorithms have been developed (as introduced in Sec. II). From an algorithmic as well as chip architecture perspective, analog in-memory training is far more challenging than solely AIMC inference. In particular, for in-memory SGD training, the backward pass as well as the incremental update are done in-memory and, thus, subject to additional noise sources and nonidealities. For the development of robust AIMC training algorithms, it is thus especially important to have

02 July 2024 08:09:40

© Author(s) 2023

TABLE IX. Compounds are derived from `UnitCell` and used to define a specialized updated behavior of the `UnitCellRPUConfig` (e.g., set to the `device` field). To indicate a weight matrix W thought of stored on an analog crossbar, we here write $\breve{W}$. To indicate a pulsed outer product update (according to Ref. 16), we write $\overset{\cdots}{\longrightarrow}$. Slow (row-wise) read and pulsed update are indicated with $\dot{\longrightarrow}$, and a column-wise read (that is, an AIMC MVM forward pass with one-hot inputs and addition to a digital matrix) is indicated with $\rightarrow$. Note that each of the compounds has a number of configuration settings for exploring the hyperparameters of the optimizers.

Compounds	Algorithm	Update
Vector	In-memory SGD	$\overset{\cdots}{\longrightarrow} \breve{W}$ w/multiple devices per crosspoint
MixedPrecision	Mixed-precision[44]	Digital rank-update onto χ, (row-wise) pulsed transfer $\chi \dot{\longrightarrow} \breve{W}$
Transfer	Tiki-taka[41]	$\overset{\cdots}{\longrightarrow} \breve{A}$, slow (row-wise) transfer $\breve{A} \dot{\longrightarrow} \breve{W}$
BufferedTransfer	TTv2[42]	$\overset{\cdots}{\longrightarrow} \breve{A} \rightarrow H \dot{\longrightarrow} \breve{W}$, with digital matrix H
ChoppedTransfer	Chopped-TTv2[43]	$\underset{+-}{\overset{\cdots}{\rightarrow}} \breve{A}$ with chopper, $\breve{A} \rightarrow H \dot{\longrightarrow} \breve{W}$
DynamicTransfer	AGAD[43]	$\underset{+-}{\overset{\cdots}{\rightarrow}} \breve{A} \rightarrow H$ with dynamic offset correction, $H \dot{\longrightarrow} \breve{W}$

good estimates of attainable accuracy assuming a particular device material, as well as be able to determine the limits of device material properties that still guarantee convergence of the training algorithm.

The AIHWKit provides a particularly rich set of tools for the testing and development of AIMC training algorithms. Out-of-the box, it provides naïve in-memory SGD using stochastic pulse trains,[16] as well as improved in-memory training algorithms, such as mixed-precision,[44] Tiki-taka I and II,[41,42] as well as the newest state-of-the-art algorithmic developments, namely Chopped-TTv2 (c-TTv2) and Analog Gradient Accumulation with Dynamic reference (AGAD)[43] (see Table IX).

A. Configuration of material properties for in-memory analog training

For in-memory training, apart from the actual AIMC training algorithm, the device material response properties are important. The fully in-memory training algorithm will typically use stochastic pulse trains and cross-point pulse coincidences to implement the outer product,[16] or might use an intermediate digital computation before updating the analog weights matrix with incremental pulses.[44] In AIHWKit, each single incremental pulse update is explicitly modeled according to a device response model. AIHWKit

02 July 2024 08:09:40

TABLE X. Selection of (predefined) device models and configurations (the `Device` and `DevicePreset` name suffixes are omitted here). For AIMC training, the update behavior (the weight change in response to a pair of coincident voltage pulses from BL and WL) is governed by the `device` field of the *RPUConfig*. AIHWKit provides numerous functional device models as well as presets, where the parameters of the functional device models are set according to measurements. All devices additionally implement device-to-device variations, where each device in the array will be set to slightly varying parameters (typically drawn from a Gaussian around the mean with user-defined variance). For instance, the actual update δ is typically computed as $\delta_{ij} + \sigma\xi$, where σ is the pulse-to-pulse standard deviation ($\xi \in \mathcal{N}(0,1)$) and $\delta_{ij} = \delta w_{\min} + \sigma_{\text{d-to-d}}\xi$ is set at the device array construction time to model device-to-device variations (indices and details are omitted in the simplified equations above). The device is modeled in normalized weight units, assuming a linear mapping of weights to conductances. For more complete equations and details, see the API documentation.

Device config	Simplified mathematical model	Functionality
Constant-Step	$w \leftarrow \text{clip}(w \pm \delta)$	Update independent of current weight (conductance)
Linear-Step	$w \leftarrow \text{clip}(w \pm \delta(1 - \gamma w))$	Gradual saturation toward weight bounds with clipping
SoftBounds	$w \leftarrow w \pm \delta\left(1 - \frac{w}{b_\pm}\right)$	Gradual saturation toward the bounds
Pow-Step	$w \leftarrow w \pm \delta\left(\frac{b_+ - w}{b_+ - b_-}\right)^\gamma$	Power dependency on weight
Exp-Step	$w \leftarrow w \pm \delta\left(1 - c_0 e^{-c_1 w}\right)$	Exponential dependency with current weight with parameters c_0 and c_1
Piecewise-Step	$w \leftarrow w \pm \delta((1-q)v_k + q\, v_{k+1})$	User-defined nodes v_k with linear interpolation, $w \in [v_k, v_{k+1}]$, $q = \frac{w - v_k}{v_{k+1} - v_k}$
ReRamES	Based on *Exp-Step*	Preset setting for ReRAM[66]
ReRamArrayOM	Based on *SoftBoundsReference*	Preset setting from optimized material ReRAM arrays[67]
ReRamArrayHf2O	Based on *SoftBoundsReference*	Preset setting from HfO2 ReRAM arrays[67]
Capacitor	Based on *Linear-Step*	Preset setting for CMOS[68]
EcRam	Based on *Linear-Step*	Preset setting for ECRAM[69]
EcRamMO	Based on *Linear-Step*	Preset setting for single metal-oxide ECRAM[70]
GokmenVlasov	Based on *Constant-Step*	Device setting used by Gokmen and Vlasov[16]
PCM	Based on *Exp-Step* and *OneSided*	PCM preset device pair with one-sided update (and occasional reset)

© Author(s) 2023

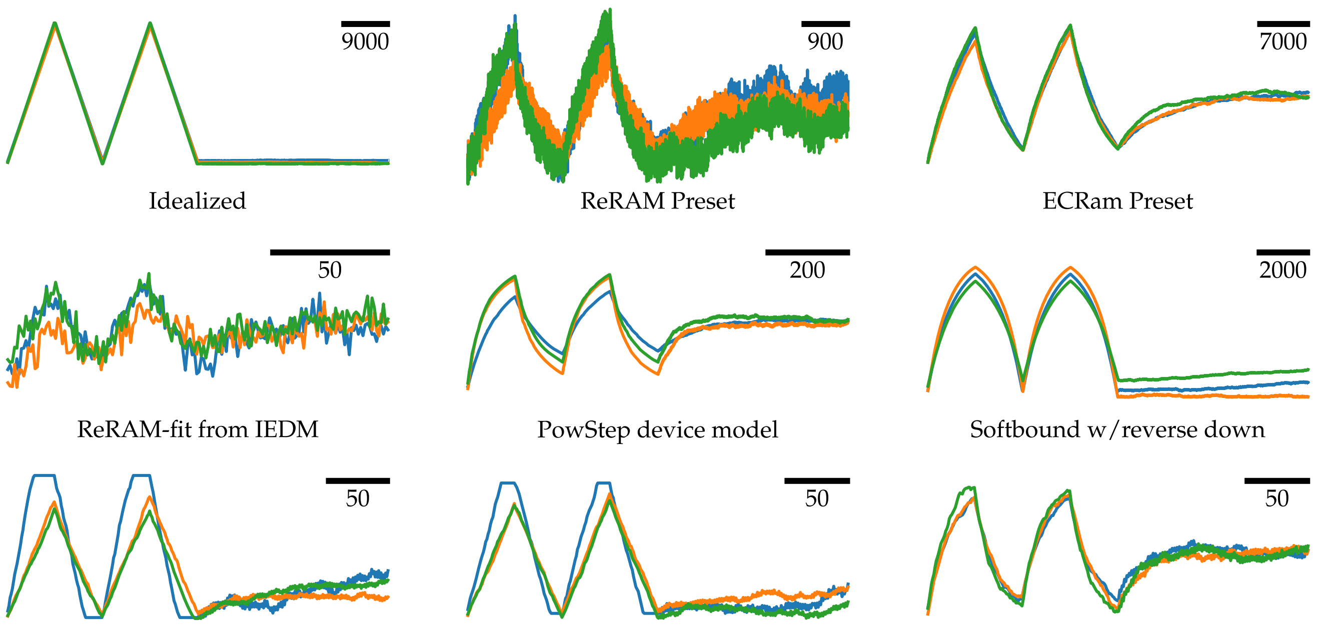

FIG. 6. Example conductance responses to a series of up, down, and up-down pulses for different device configurations as listed in Table X. Note that various asymmetric shapes and device-to-device variations (different colors) can be set by the user. Presets that fit measurements are available, and a fitting tool for new device measurements is provided as well. The bar shows the x-axis scale in the number of pulses given.

02 July 2024 08:09:40

provides highly optimized and self-tuned GPU routines to enable larger-scale AIMC in-memory training simulations at this level of detail. This is different from the approach for inference in Sec. IV, where statistical weight programming noise models are used instead.

Material response properties for in-memory training are captured in functional device models, such as the soft-bounds model that has been used to model conductance responses to voltage pulses for ReRAM devices.[66] AIHWKit also provides other models and data-calibrated preset settings that are (partly) summarized in Table X. See also Fig. 6 for an illustration.

When setting up a *RPUConfig* to specify an in-memory training simulation, the *device configurations* listed in Table X can be assigned to the `device` field or as part of a *device compound*. If one wants to define a plain in-memory SGD using stochastic

TABLE XI. Selection of the possible configuration of the update pulse behavior. The most often used parameter is the desired bit length (the maximal number of pulses per update), which effectively determines the maximal change in conductance (weight). Since each pulse given to the device is of equal minimal amplitude (with possible variations determined by the device model setting), the maximal amount that can be written onto the device is the number of pulses given per update times the average change in conductance per pulse. Thus device update will clip at some point if the SGD demands a too large gradient update. Small update values (smaller than the minimal update) are effectively implemented by stochastic pulsing probabilities smaller than one. To determine the probability the average expected minimal conductance response at the (logical) zero point is used and expected to be known (`dw_min` in many device models). See Ref. 16 for details.

Update field	Default value	Functionality
`desired_bl`	31	Desired length of the pulse trains. In the case of using the update BL management, it is the maximal pulse train length
`pulse_type`	*StochasticCompressed*	Pulse types used when computing the outer product. Can be stochastic or implicitly deterministic
`update_bl_management`	True	Dynamic selection of the length of the pulse train as described in Refs. 43 and 45
`update_management`	True	Scaling of the update pulse probability to load-balance the word and bit-lines. See Ref. 45 for details
`x/d_res_implicit`	0.0	Resolution (i.e., bin width) of each quantization step for activation x or the error d, respectively, in the case of *DeterministicImplicit* pulse trains

© Author(s) 2023

pulsing,[16] the device configuration is directly applied to the `device` field, and additional properties for the update behavior, such as pulsing schemes and corrective methods, are set in the `update` field (see Table XI).

However, the `device` field can also be a *device compound*, in which case multiple crossbars (or parts of the more complicated unit cell at each crosspoint) are simulated according to the definition of the training algorithm [compare to Figs. 2(b)–2(d)]. The specialized AIMC update algorithms supported are listed in Table IX.

In Sec. V B, we give an example of how to fit device material measurements to one of the device models in AIHWKit, use this configuration to train a DNN with one of the specialized AIMC training algorithms, and evaluate the impact of some of the device properties on the achievable accuracy.

B. Analog in-memory training: From device measurements to DNN accuracy

Notebook `analog_training.ipynb`[71] provides an example of how the AIHWKit can be used to evaluate the performance of newly characterized devices in the context of analog IMC. The notebook starts by introducing the *RPUConfig*, which is used to define many of the hardware aspects of the analog tile used in the simulation. Properties such as the tile size (which is the number of devices used in rows and columns), the number of bits used by the ADC/DAC converters, and others, as well as the specialized update algorithm and material properties, can all be defined within the *RPUConfig*.

A typical scenario in the development of new device materials includes iterations of device fabrication, characterization, and evaluation of their performance for the application under study, in this

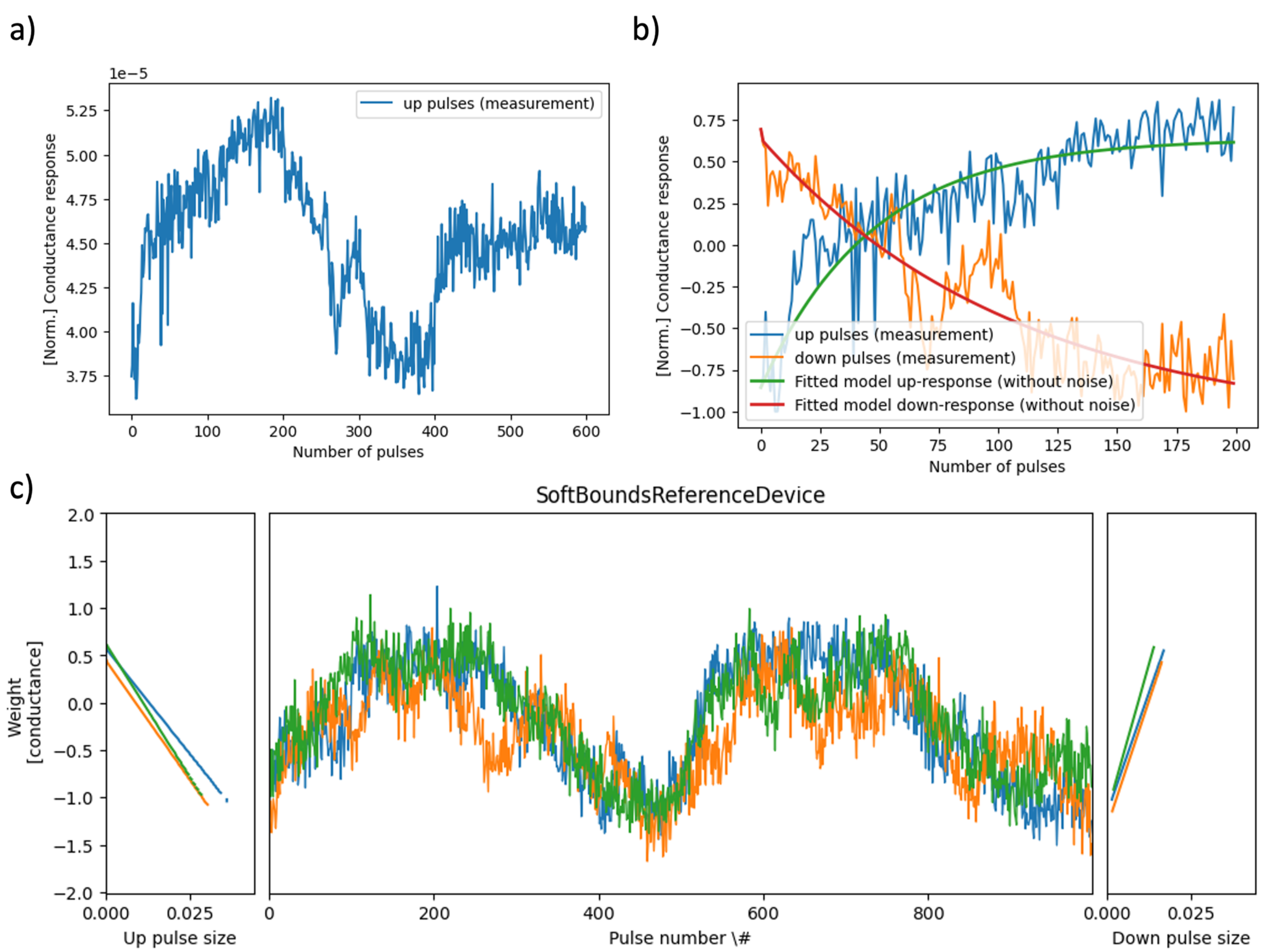

FIG. 7. (a) Response curve of a ReRAM memristive element. 200 pulses to increase the conductance, followed by 200 pulses to decreases it, give the maximum and minimum conductance that the device can reach. The following 1 increase and 1 decrease pulses are repeated 100 times to find the symmetry point of the device. (b) The response curve and the fitted model. (c) Modeled response curve with noise and device-to-device variation.

02 July 2024 08:09:40

© Author(s) 2023

case, full-scale DNN training. In particular, one wants to understand how the fabricated device performs when used in an AIMC accelerator. To illustrate the steps involved, we show how device measurements can be fitted to one of the AIHWKit device models and then show how its impact on training accuracy can be evaluated.

1. Fit device measurements to a device model provided by AIHWKIT

In Fig. 7(a), a typical conductance response of an ReRAM device to voltage pulse trains is shown. Here, to span the full conductance range of the device, a sequence of 200 electrical pulses is given, which incrementally increases the device conductance, followed by 200 pulses that decrease the device's conductance. This 200 up/200 down sequence is then followed by a 1 up/1 down pulse sequence, which moves the conductance of the device to its symmetry point.[41,72]

In the AIHWKit, there are many different device models that can be used to represent the electrical response of different devices. ReRAM electrical response is well represented through the `SoftBoundsReferenceDevice` model (see Table X), where the conductance response is gradually saturating to some level and the symmetry point can be implicitly controlled by a tunable reference device. The device models typically have many parameters that can be fitted to the measured device characteristics. Among others, `w_max` and `w_min` represent the maximum and minimum analog weight[73] value that the device can represent, respectively (in normalized conductance units of the analog weights); `dw_min` represents the mean of the distribution of the weight change that the device can achieve at symmetry point with a standard deviation of `dw_min_std`; and `dw_min_dtod` specifies the device-to-device variation of the `dw_min` parameter so that different devices can have a slightly different response curve.

The AIHWKit provides a fitting utility `fit_measurements` that can be used to extract most of the needed parameters from device measurements. The notebook shows how the fitting utility can be used to automatically fit the device model to the measured response curve shown in Fig. 7(b). Since in this example the model is extracted from a single device, a device-to-device variation of 10% is assumed. Figure 7(c) shows the fitted device response curve when noise and device-to-device variation are applied to simulate the real device characteristics. After fitting the device model, the device configuration can now be used to customize the *RPUConfig* used in the AIMC training.

2. How to specify the RPUConfig for in-memory training

Assuming the fitted device configuration is now given as `device_config_fit` (see notebook for an example), we can now build the *RPUConfig* to describe the hardware and algorithmic choices of the DNN in-memory training simulation. As described in Sec. III C, the *RPUConfig* defines many more aspects of the analog tile hardware than just the device material behavior; for instance, nonidealities in the forward and backward passes as well as the digital periphery choices (see Table IV). As illustrated in more detail in the notebook, we can build the following *RPUConfig* for SGD in-memory training, where we set some (here arbitrary selected) non-default parameters:

```
from aihwkit.simulator.configs import (
    SingleRPUConfig, UpdateParameters, IOParameters
)
rpu_config = SingleRPUConfig(
    device=device_config_fit,
    forward=IOParameters(out_noise=0.1),
    backward=IOParameters(out_noise=0.1),
    update=UpdateParameters(desired_bl=10),
)
```

In this training example, in-memory training with stochastic pulses is used to train the network. Before training the DNN with this *RPUConfig* and device setting, a DNN needs to be constructed and converted to *analog tiles* that are roughly equivalent to AIMC crossbars.

3. Construct the desired DNN and convert to analog

While the AIHWKit provides analog layers to directly build up an analog DNN (see Table II), it is often more convenient to automatically convert a native PYTORCH model into an analog model using the provided conversion utilities. As we show in the notebook, the native PYTORCH DNN, a three layer fully connected network defined using the standard PYTORCH syntax, is converted to an analog model by the `convert_to_analog` utility. This utility translates the layers with parameters (i.e., the three fully connected layers in this case) to be simulated with AIMC tiles, whereas other layers, such as the Sigmoid and Softmax activation functions, are kept and, thus, assumed to be processed in digital at full precision. In the AIHWKit, it is generally assumed that analog signals are converted back to digital numbers after each tile operation so that activation functions and other layers can be computed in FP. Because AIHWKit is a functional simulator that aims to compute attainable accuracy with configurable AIMC nonidealities and is not concerned with performance or latency estimation, the digital layers simply use native PYTORCH code, assuming floating-point precision.

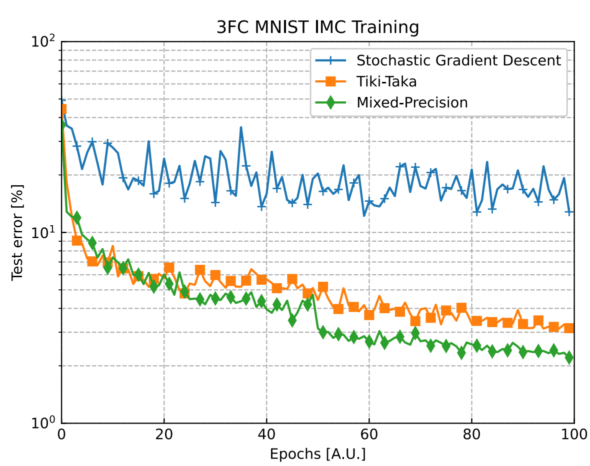

FIG. 8. Accuracy achieved by the different algorithms after ten epochs of training. The Tiki-Taka (TT) and Mixed-Precision algorithms, being specifically designed around IMC, clearly outperform more standard SGD algorithms.

02 July 2024 08:09:40

© Author(s) 2023

4. Train the analog model and inspect the results

The constructed network is trained on the MNIST dataset for 100 epochs with a batch size of 10 and a learning rate starting at 0.1, which is further decayed at the 50th and 80th epochs by a factor of 10 each time. Figure 8 shows the performance achieved during training and compares the performance achieved by different analog in-memory training algorithms. The naive SGD performs quite poorly, which is mainly due to the limited number of states and the asymmetry in up vs down response, as standard SGD requires very symmetrical update characteristics (see Ref. 16 or Ref. 47 for device specifications of SGD). Therefore, this example shows the need for innovation not only at the device level to limit device nonidealities and obtain a better response curve but also at the algorithmic level to relax some of the requirements on the AIMC device.

5. Selecting different in-memory training optimizers

In the above, we only used the standard in-memory SGD training using stochastic pulse trains. The choice of other update algorithms is done by configuring the `device` field with the appropriate compound (see Table IX). To make the building of the *RPUConfig* more convenient, the `build_config` tool exists:

```
from aihwkit.simulator.configs import build_config
algorithm = 'ttv2'  # one of tiki-taka, ttv2, c-ttv2, mp, sgd, agad
rpu_config = build_config(algorithm, device=device_config_fit)
```

The notebook shows some more algorithmic choices. In this regard, the Tiki-Taka (TT) algorithm[45] is specifically designed for training neural networks with non-ideal devices. Both SGD and TT use error backpropagation to train the network; however, the TT algorithm replaces each weight matrix W with two matrices, referred to as $\breve{A}$ and $\breve{W}$. The gradients are accumulated directly onto $\breve{A}$ (using pulse coincidence) for a certain number of updates before being transferred to $\breve{W}$. Figure 8 shows the improved performance that the TT algorithm achieves. In contrast to the TT algorithm, the Mixed-Precision (MP) optimizer[44] uses digital computing for the update of the gradient accumulator matrix instead of gradient accumulation in-memory. The accumulated gradient matrix M is kept, computed with floating-point digital precision, and then used to update the (analog) weight matrix. Given that gradients are accumulated flawlessly (but without the benefits of in-memory acceleration of the update pass), we expect that the accuracy will improve for MP.

C. Optimize hyperparameters of the analog optimizer

As is common in SGD training, algorithmic hyperparameters such as the learning rate need to be tuned for a given AI workload. Similarly, the specialized analog optimizers come with a number of additional hyperparameters that often need to be tuned to obtain the best training result for a given device material configuration. Examples of such algorithmic hyperparameters, e.g., the c-TTv2 algorithm,[43] are specified in the "compound"-level of the device field (see also Table IX). For instance, the parameters `auto_granularity` and `in_chop_prob` that govern the (inverse of the) learning rate onto the $\breve{W}$ matrix and the chopper probability, respectively, can be set with

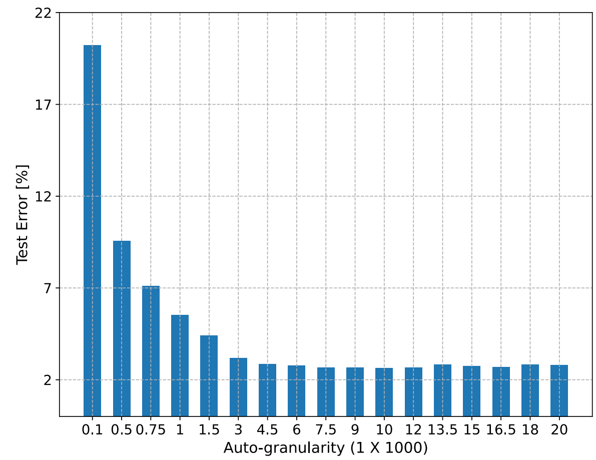

FIG. 9. Validation test error (100% - accuracy) achieved for different `auto_granularity` values when using the Chopped-TTv2 (c-TTv2) in-memory training algorithm. Each data point is generated using a new *RPUConfig* setting with an adjusted `auto_granularity` value, and using this new *RPUConfig* to train the model on the train set is then tested on the separated validation set. The model used for this experiment is the model defined in Sec. V B 3, and the base *RPUConfig* is the `ReRamArrayOMPresetDevice` with a `dw_min_factor` of 1.0 defined in Sec. V D 1.

```
rpu_config = build_config("c-ttv2", device=device_config_fit)
rpu_config.device.auto_granularity = 2.0
rpu_config.device.in_chop_prob = 0.1
```

Note that in this case, the `device` is of the `UnitCell` type (as detailed in Table IX). While default hyperparameter values are set to result in reasonable training behavior, depending on the optimizer, other hyperparameters might need to be tuned, such as the learning rate onto the gradient accumulation matrix (`fast_lr`) or the rate of transfer reads. See Ref. 43 for a more detailed discussion of the TT optimizers and their variants.

In practice, these hyperparameters need to be optimized on a separate validation dataset to obtain optimal AIMC training results for a given device model. Here, we show the effect of `auto_granularity` on the model inference error (see Fig. 9). Note that the validation test error reduces with larger `auto_granularity` values, which essentially increases the amount of noise averaging on the digital hidden matrix of the c-TTv2 algorithm used here.

D. Device parameter variation to obtain device specifications

Apart from directly evaluating the impact of a measured device on DNN accuracy, one is often interested in the impact of certain selected device properties on accuracy. That could be used, in particular, for material innovations and process developments, but also for algorithmic improvements. For instance, what is the largest device-to-device variation one can accommodate for successful training with a given analog optimizer? If known, useful design targets can be set for device material improvements. In the AIHWKit, all device properties can be easily varied so that such targets can be conveniently obtained, as illustrated in this section.

02 July 2024 08:09:40

© Author(s) 2023

Although many device models are available in AIHWKit (see Table X and Fig. 6), which have a variety of parameters, some properties are common, such as `dw_min,` which governs the amount of conductance change induced by a single pulse. A typical characteristic of the device is the (average) number of states, which is defined by dividing the average conductance range by `dw_min`. In most device models, the number of states and conductance ranges are, thus, configured by the parameters `dw_min`, `w_max`, and `w_min` and their standard deviations (across pulses and devices). Here, we take a closer look at the effect of some of these common material specifications and the algorithmic hyperparameters on the attainable accuracy of the trained model in comparison to standard SGD floating point training. As a small example of DNN, we again use the 3-layered fully connected (3FC) model defined and trained in Sec. V B 3 on the MNIST dataset.

1. Setting the RPUConfig with custom device parameters

The basic *RPUConfig* used in this section is based on the material specification obtained from ReRAM arrays.[67] This device is already available in the preset library of the AIHWKit and is named `ReRamArrayOMPresetDevice`. Here, we investigate the impact of different values of various parameters describing the mean response and variability, such as `dw_min`, `dw_min_std`, `write_noise`, `w_max`, `w_max_std`, `w_min`, `w_min_std,` etc.

We first train the 3FC model with the given device material configuration using the specialized analog in-memory c-TTv2 algorithm (see also Table IX). The impact of changes in the device-to-device variation, `dw_min`, which determines the number of states in the device, is investigated. To achieve this, a multiplicative factor is introduced for each parameter of interest. For example, a `dw_min` factor is introduced to change the `dw_min` value. This could be achieved as follows:

```
from copy import deepcopy
from aihwkit.simulator.configs import build_config
from aihwkit.simulator.presets import ReRamArrayIEDM2022PresetDevice
def get_rpu_config(device_config, dw_min_factor = 1.0):
    device = deepcopy(device_config)
    device.dw_min *= dw_min_factor
    return build_config("c-ttv2", device=device)
# example of increasing the number of states by 2
rpu_config = get_rpu_config(ReRamIEDM2022PresetDevice(), 0.5)
```

The resulting *RPUConfig* due to a change in a parameter factor is then used to train the model until convergence, and the evaluation accuracy on the test set is stored. Similar functions can be defined for other variations (see notebook[71] for more examples).

2. Impact of number of states of the device material

The number of states of the device material has a large impact on the in-memory training quality, as shown in the following: The number of states of the devices is here defined by the ratio of the average conductance range to the expected response magnitude at (algorithmically) zero. Note that the device conductance in the AIHWKit is usually modeled in dimensionless parameters, assuming that the conductance is directly proportional to the analog weight value. All parameters of the AIMC MVM are related to this normalized unit. Thus, typically, the `w_min` and `w_max` values of the device are fixed to -1 and 1, respectively. Digital output scales can be used to set the analog weights initially, as described in Sec. III D 1. For training, we use default torch weight initialization ranges and set the weight scaling omega to 0.3 so that the analog weights are guaranteed to be filled with uniform numbers from $-0.3, \ldots, 0.3$ (independent of the layer size[47]), and then fix the output scales during training. This is achieved by setting

```
rpu_config.mapping.weight_scaling_omega = 0.3
rpu_config.mapping.learn_out_scaling = False
rpu_config.mapping.weight_scaling_columnwise = False
```

02 July 2024 08:09:40

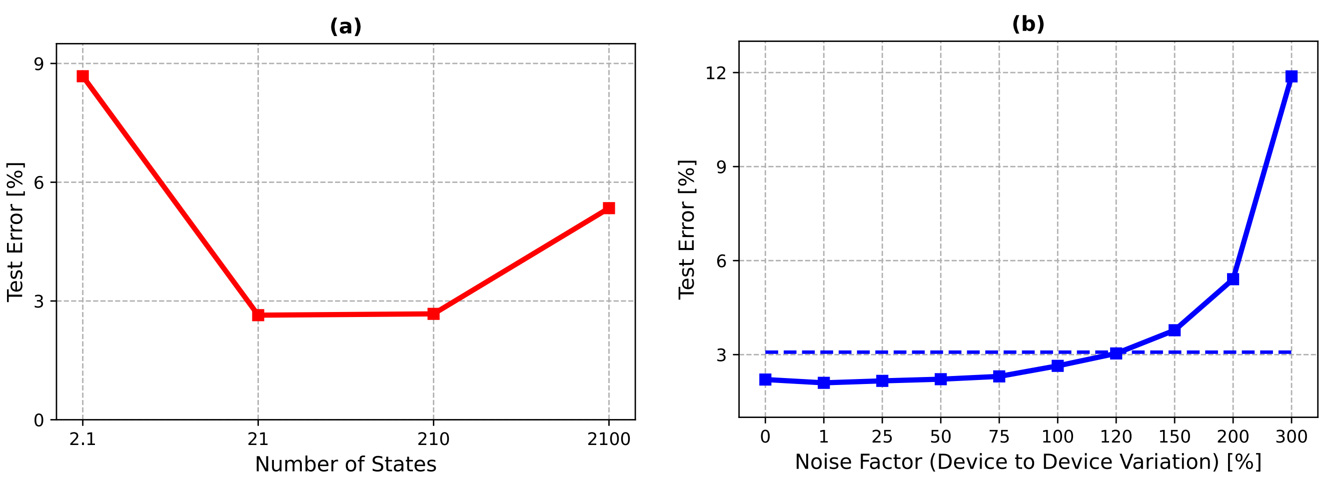

FIG. 10. (a): Test error achieved by different `dw_min` factors using the c-TTv2 algorithm (here plotted as the number of states, which is defined as the weight range divided by `dw_min`). Each data point is generated using a new *RPUConfig* obtained by varying the `dw_min` factor only and using the new *RPUConfig* to train the model to obtain the test error. (b): Test error achieved by the different device-to-device variation noise factor using the c-TTv2 algorithm. Each data point is the test error of a separate training run with modified *RPUConfig* obtained by varying the device-to-device variation noise factors. The dashed line indicates 99% of the accuracy achieved with no variation noise. This shows that about 120% of the noise (with respect to the noise as measured for the device data at 100%) is tolerable without a significant accuracy drop.

© Author(s) 2023

When we now set the dw_min parameter, the value is in normalized conductance units, ranging from −1 to 1, so that the number of states is simply 2 divided by the value of dw_min.

Figure 10(a) shows how the model test error varies with changes in the dw_min factor. It shows there is a range of dw_min values that improve the attainable accuracy, which thus means that one should optimize the device materials to match the requirements. Note, however, that too low dw_min factor values (a higher number of states) also negatively impact the accuracy.

3. Impact of device-to-device variations on accuracy

Similarly, the impact of device-to-device variations can be estimated. For that, a similar parameter factor, called the noise factor and measured in percentages, is introduced as a multiplicative factor to control write noise and the various standard deviations in the material specification in relation to the baseline. A noise factor value of zero means that there is no device-to-device variation.

Figure 10(b) shows how the device-to-device variation noise factor influences the inference performance. The validation error generally increases when increasing the noise factor, suggesting that variations negatively impact the in-memory training. However, the rate of change is very small for noise factors between 1% and 70%, compared to the rate of change when the noise factor becomes greater than about 70%. Hence, reducing the noise variation to 70% of the baseline might significantly improve the in-memory training performance and, thus, could be a helpful target for the next device material designs.

VI. ANALOG AI CLOUD COMPOSER

In the following, we describe the Analog AI Cloud Composer (AACC) platform, a cloud offering that provides the benefits of using the AIHWKit simulation platform in a fully managed cloud setting. The Analog Composer is introducing for the first time *Analog AI as a service* or, in short, *AAaaS*. The cloud composer can be freely accessed at https://aihw-composer.draco.res.ibm.com.

We first describe the architecture of the cloud composer and then the various services it provides including inference, training, and hardware access. We then present future features and directions.

A. Composer design and architecture

The AAICC is a novel approach to AIMC that leverages the AIHWKit simulation platform to allow a seamless, no-code interactive cloud experience. With access to the open-source AIHWKit libraries and an easy-to-use interface, it provides a platform for researchers, hardware-engineers, developers, and enthusiasts to explore, experiment, simulate, and create Analog AI neural networks and tune various analog devices to create accurate and sustainable AI models. This platform also serves as an educational tool to democratize IMC and introduce its key concepts.

The AAICC adopts a modern distributed architecture based on IBM Cloud services and guidelines. The user input is limited to data (not code) with strong control and validations during the lifecycle of the application and the input data. The design maintains a separation of concerns and responsibilities between the various components. Tracking, monitoring, and auditing services are enforced to ensure security and compliance according to IBM Cloud security standards.

The architecture of the AAICC can be divided into five main components, as illustrated in Fig. 11:

a. *The Front-end Client Component*: This component provides an entry point for clients to interact with the composer application. Two scenarios are supported. The user can interact with the composer through a web application or through the command-line interface. Through this component, the user defines a training or inference experiment that can run on the AIHWKit simulator.

b. *The API*: The API component is an HTTP microservice that provides the endpoints that are used by the web application and

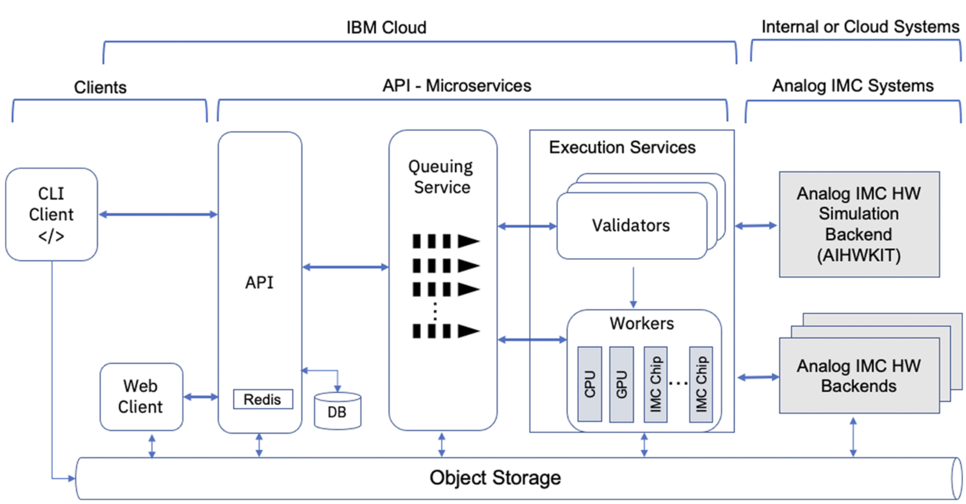

FIG. 11. Analog AI Cloud Composer (AAICC) architecture.

02 July 2024 08:09:40

© Author(s) 2023

the backend Python libraries. The API provides user authentication, database access, setup of the queuing system, job process flow setup, and collection of various statistics.

c. *The Backend Execution Services*: These services are responsible for executing all the training and inference jobs that are submitted by end users. There are two sub-components in the execution services: the validator and the workers. The validator service ensures that all training and inference jobs that are submitted are composed correctly and adhere to the specifications of the AIHWKit for defining the neural network, the various training or inference parameters, and the supported hardware configurations. For example, it validates that the analog layers and the *RPUConfig* are correctly defined. The workers are responsible for executing the submitted experiments and sending the results to the front end component for web rendering. Various worker instances are implemented, depending on the backend infrastructure that will be used to execute the experiment. We have implemented three workers. The GPU worker provides GPU acceleration for running the AIHWKit training or inference experiments. The central processing unit (CPU) worker will run the submitted experiments on a CPU backend server. The IMC hardware worker will run the experiments on supported IMC chips. The design is flexible as it allows us to plugin more workers as more IMC chips are implemented or different backend implementations are added. The infrastructure is also based on Kubernetes, which allows automatic scaling of the resources depending on the load the application receives from end-users.

d. *The Queuing Services*: This component provides the asynchronous-based communication backbone between the various components of the composer application. It implements various message queues for the CPU, GPU, and any future IMC hardware backends. The CPU and GPU queues are used to route jobs to the AIHWKit backend simulation library and receive notifications of the results when the jobs are done. The IMC hardware queue(s) are used to route the jobs to analog IMC chips that will be supported on the platform. In addition, we have a validator queue that serves the communication between the validator and the execution workers.

e. *The Backend Analog IMC Systems*: This component provides access to the AIHWKit for simulating training or inference on a variety of AIMC hardware options. Real AIMC chips will also be used to run inference or training on actual hardware (see Sec. VI D).

B. Analog AI training service

The AAICC offers two key services: in-memory training (as explained in Sec. V) and inference (as explained in Sec. IV), as shown in Fig. 12. In what follows, we explain how these services can be used to configure, launch, and perform experiments using the AIHWKit. Most of the experiments are based on templates that users can choose from and customize further.

The training user experience in the AAICC offers the end user the choice to start from an existing template or build an analog neural network from scratch. Each template provides a neural network

02 July 2024 08:09:40

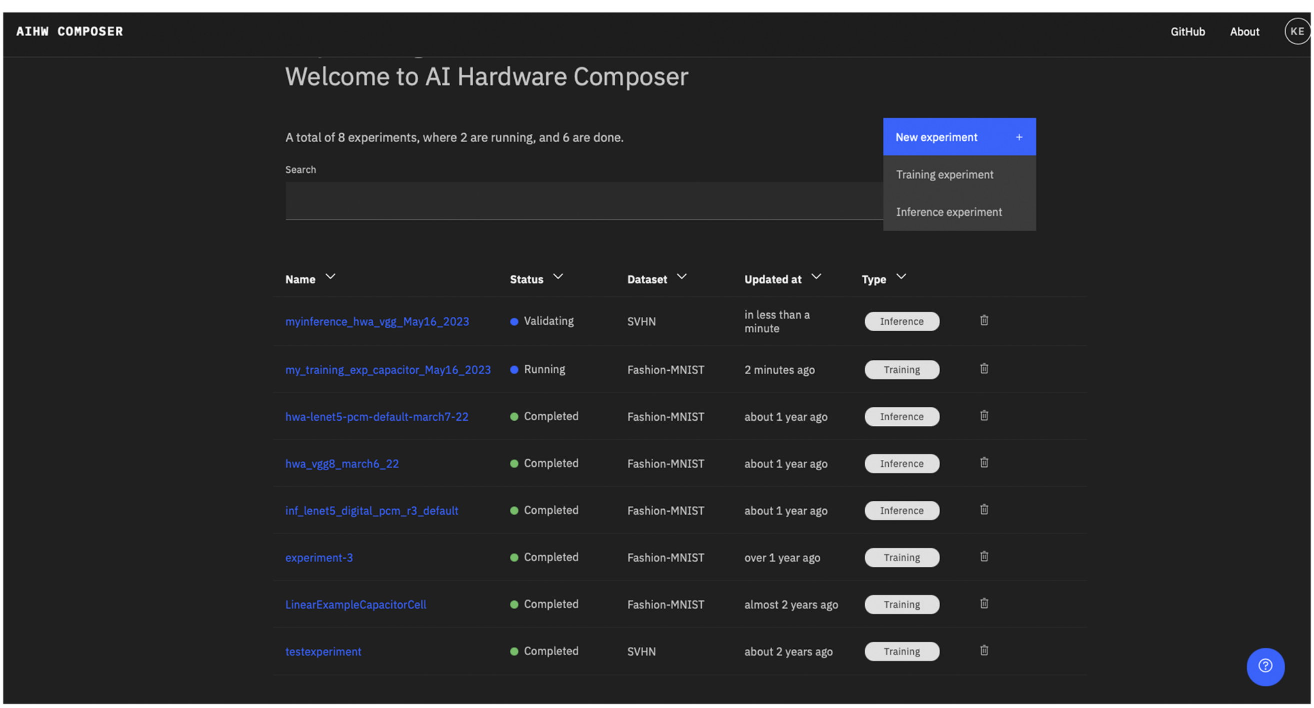

FIG. 12. Analog AI Cloud Composer (AAICC) experiments menu.

© Author(s) 2023

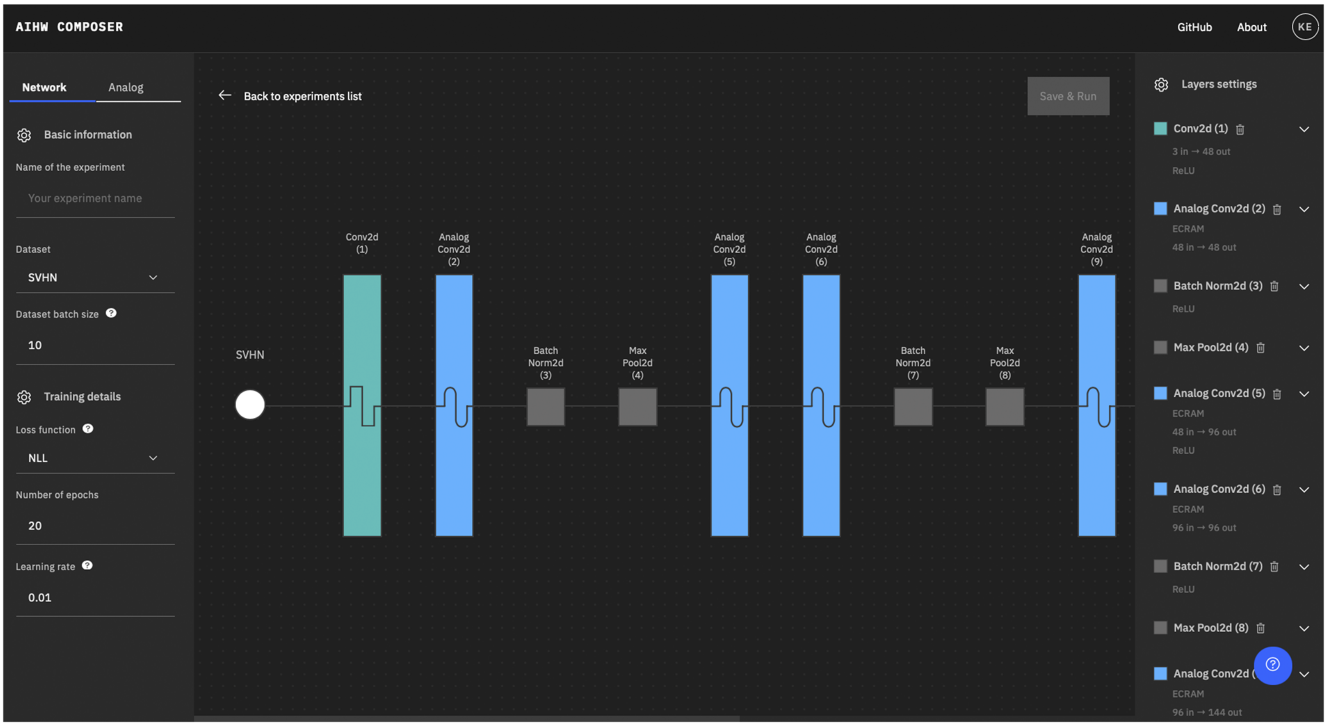

FIG. 13. Analog AI Cloud Composer (AAICC) training user interface.

architecture translated into analog layers or a mix of analog and digital layers, a dataset, an optimizer choice, various training parameters, and an analog preset choice. We currently support templates that use the VGG8, 3FC, and LeNet DNNs for image classification tasks using various device materials and optimizer settings. This list can be easily extended as we support more neural networks and datasets.

The AIHWKit includes built-in analog presets that implement different types of devices that could be used to implement AIMC

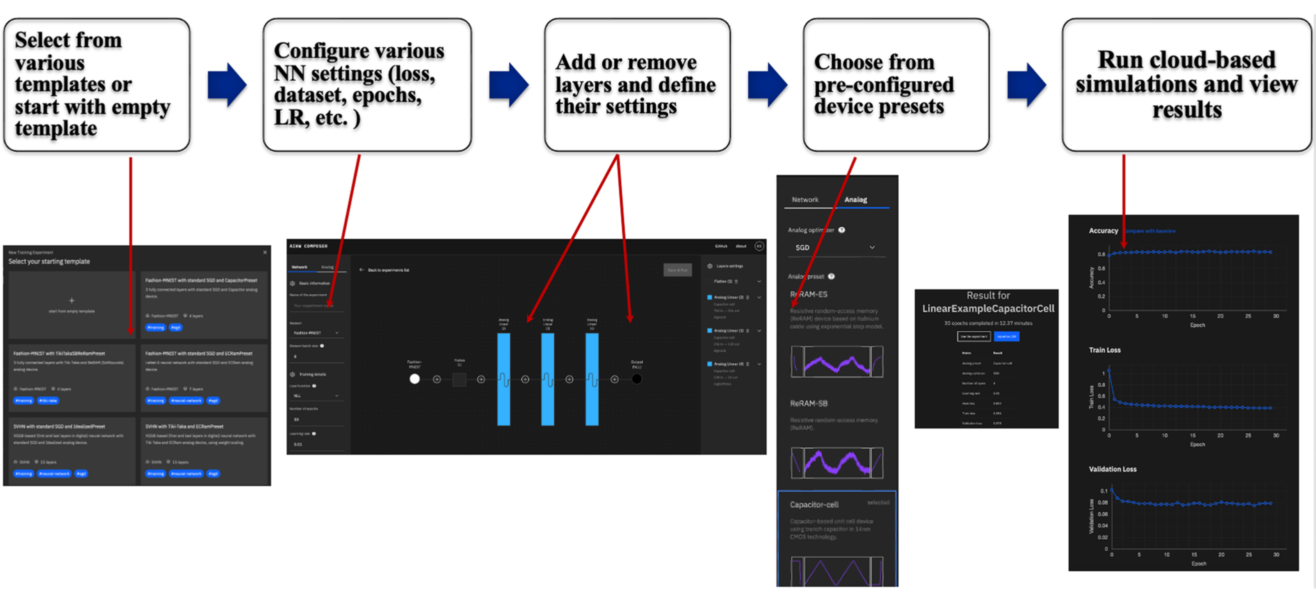

FIG. 14. Analog AI Cloud Composer (AAICC) in-memory training workflow.

02 July 2024 08:09:40

© Author(s) 2023

neural network training. Many of these presets are calibrated based on the measured characteristics of real hardware devices. Device non-ideal characteristics, noise, and variability are accurately simulated in all presets (see Table X for a selection of device presets). Many of these presets are readily available in AAICC, and the user can choose one of several in-memory optimizers and, thus, conveniently investigate the accuracy impacts of various nonidealities and material choices on the DNN at hand.

Figure 13 shows the composer training interface. The steps used to launch a training experiment and visualize its results are detailed below and summarized in Fig. 14:

1. The user can start a new experiment or select one of the existing templates.
2. After picking a template or choosing to compose a network from scratch, the user is then shown the composer playground interface, where one can choose or configure various parameters. In the middle, the neural network layers are visualized. The left and right sides of the screen provide tabs that allow the user to set training hyperparameters, analog-related configurations, or layer specific parameters. The user needs to first choose a proper name for the experiment to be created.
3. The next step is to input the training hyperparameters, such as the batch size, the loss function, the number of epochs, and the learning rate.
4. The user can also add or select a layer to configure its type, size, and activation function.
5. One of the key features of this interface is the ability to explore and apply an analog device preset, as shown in Fig. 15. The interface also provides useful documentation about each preset. The user can learn about the technology and device materials used in each preset and view the conductance response curve.
6. Once the user defines all training and analog related parameters, a training experiment can be launched on the cloud by clicking on the save and run button to launch. The jobs will be accelerated by GPUs in the cloud. The experiment is validated first by the back-end to ensure the correctness of the user-provided input before invoking the AIHWKit to run the AIMC training simulation.
7. Upon completion of the training experiment, the results page, as shown in Fig. 16 summarizes the key training job parameters that were used, such as the analog preset and the analog optimizer algorithm, and plots the trained model's accuracy, validation loss, and training loss per epoch.

C. Analog AI inference service

Similar to the training service, the AAICC inference service provides a template-based interactive no-code user experiences that allow creating analog inference experiments and launching them in the cloud. Figure 17 illustrates the high-level workflow used, which is detailed below:

1. First, the user can pick one of the pre-trained model templates. We provide models that are either hardware-aware trained or trained in digital hardware, such as GPUs. There are a number

02 July 2024 08:09:40

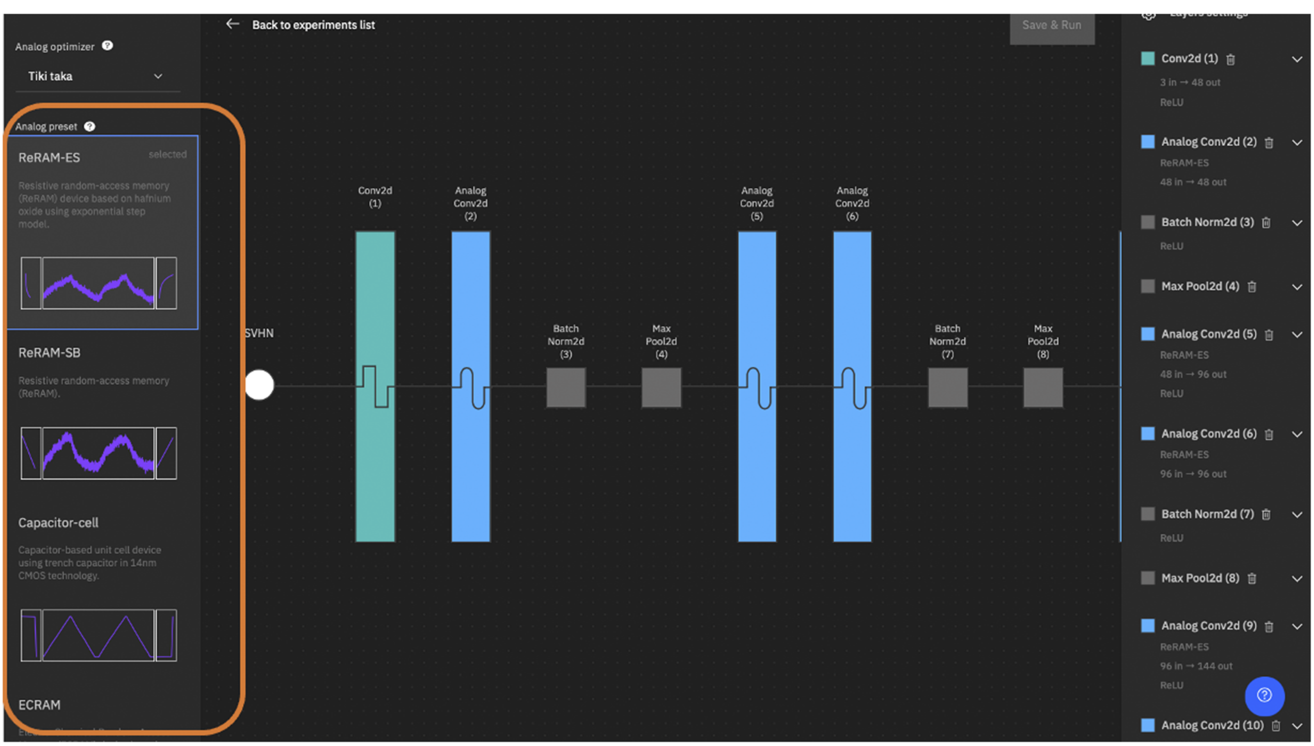

FIG. 15. Apply an analog training preset.

© Author(s) 2023

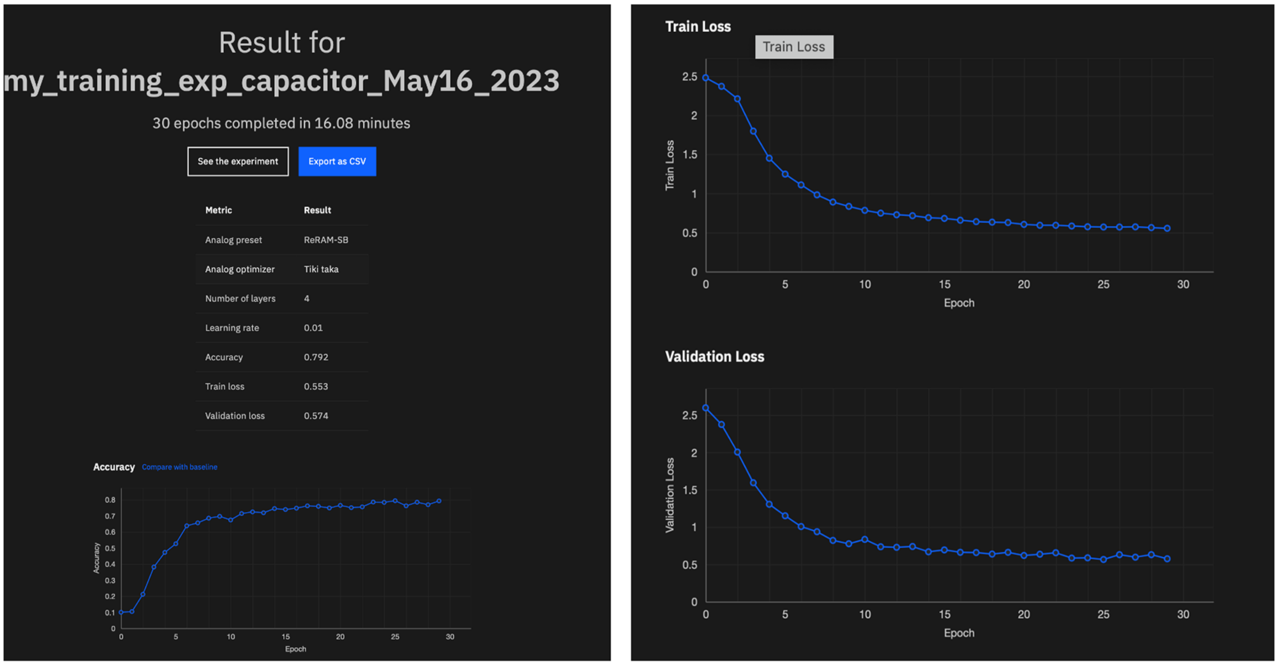

FIG. 16. Analog AI Cloud Composer (AACC) training results page.

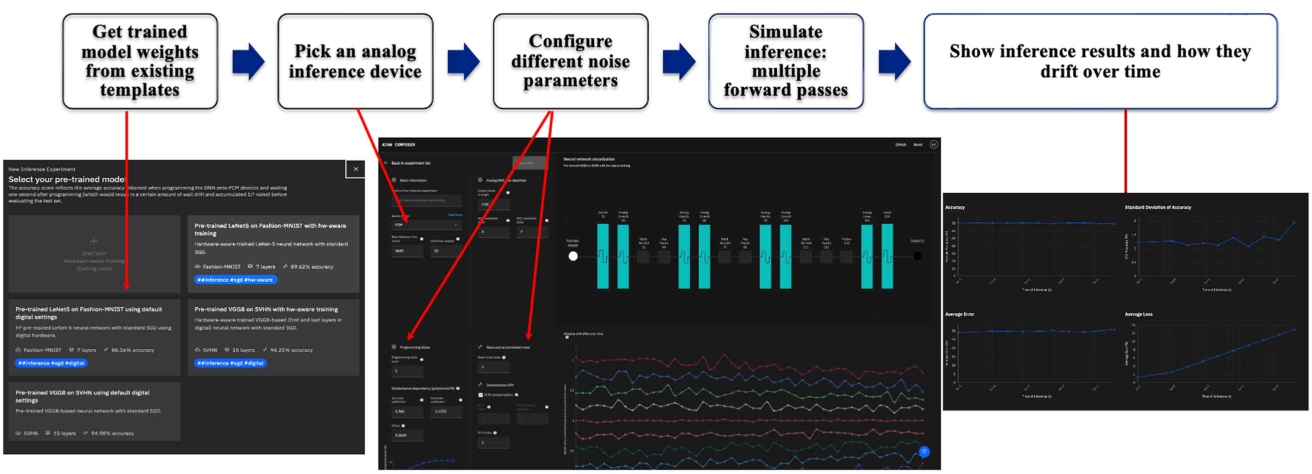

FIG. 17. Analog AI Cloud Composer (AACC) inference workflow.

of available pre-trained models and their characteristics (using VGG8 and LeNet DNNs for image classification with a combination of digital and analog layers). Future work will enable hardware-aware training directly from the composer interface that can feed into this interface.

2. An AIMC inference device needs to be chosen. We provide two choices: a PCM abstract device or a state-independent generic device. The PCM model (`PCMLikeNoiseModel`) is described in detail in Sec. IV A.
3. The next step is to configure different noise parameters and drift strengths. Different MVM nonideality sources can be tuned to study their effect on the accuracy, as shown in Fig. 18. These nonideality settings correspond to the *RPUConfig* choices for inference (see Sec. IV and Table V for details).
4. Depending on the configured parameters, the inference service provides an interactive graph that visualizes the drift effect over time of the hardware device that is simulated (PCM or generic device). As shown in Fig. 18, the graph shows how the weights are drifting over time for different weight values after they have been programmed on the device.

02 July 2024 08:09:40

© Author(s) 2023

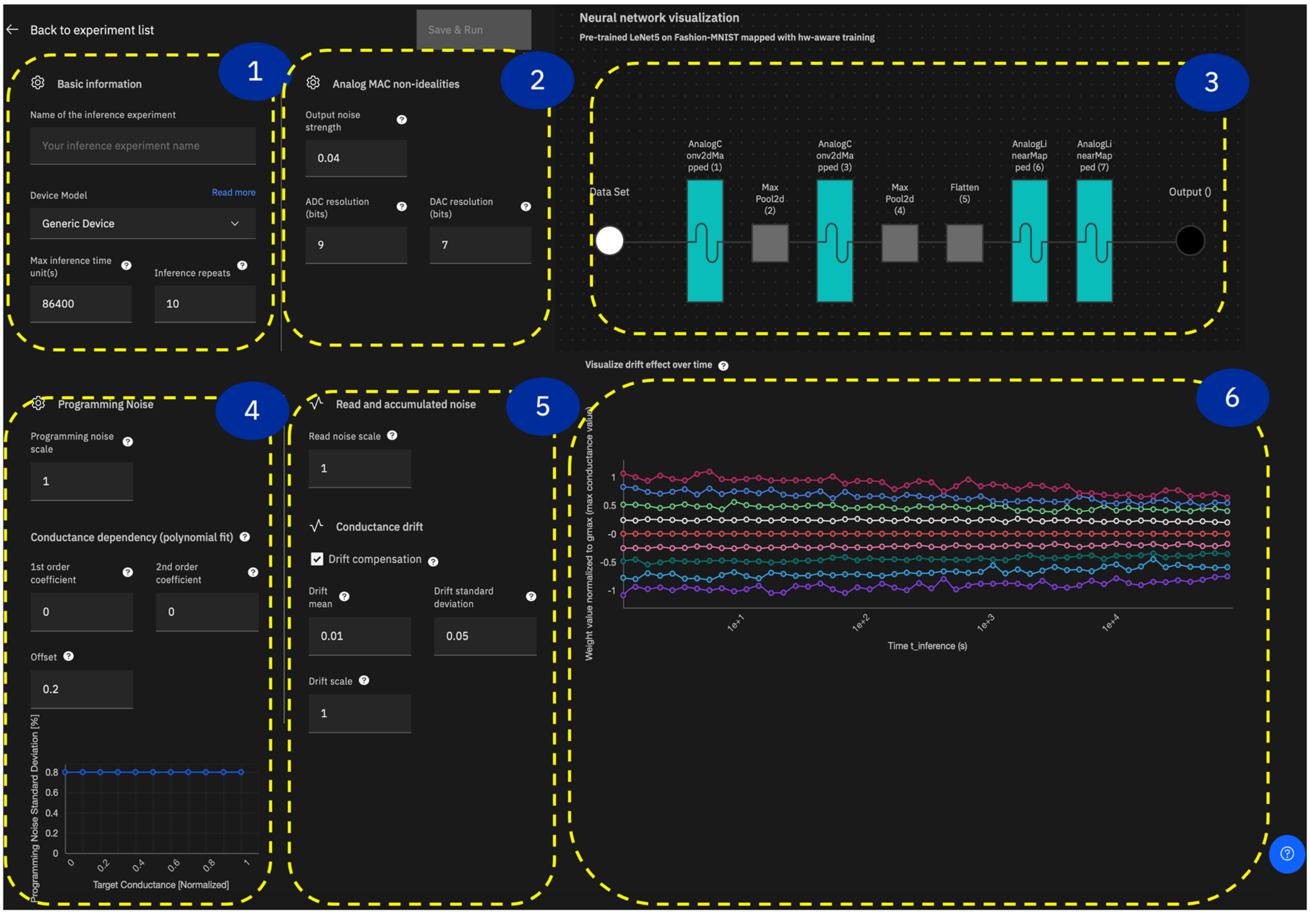

FIG. 18. Design of the Analog AI Cloud Composer (AAICC) inference user interface.

5. The inference simulation using AIHWKit can then be launched as a job in the cloud. The user can visualize the results of the inference, including model accuracy and drift effects over time.

D. Access to analog IMC hardware

In addition to the inference and training simulations using the AIHWKit, the composer application provides a framework for accessing real IBM IMC chips as they become available. The first IMC chip that we will expose is the Fusion PCM chip.[21] As shown in Fig. 19, the Fusion chip has 512 word lines (WL) and 2048 bit lines (BL). Each WL/BL address has a PCM device and an access transistor, which can be individually accessed. Hence, there are 512 × 2048 PCM devices in total. Because the chip only stores the weights and does not perform an explicit MVM on-chip, they can be placed at any arbitrary location on the chip, independently of which layer they encode. Each PCM device stores the absolute value of a weight in its conductance state. The sign information is stored in the Python client software. Figure 19 shows a high level description of how the Fusion chip interacts with the composer. Trained weights from the user are converted to conductance values in G_{train} and then sent to the Python client running on a local host to program the weights on the PCM chip. After programming, conductance values are read from the PCM chip through the local Python client and, thus, provides an accurate measurement of the programmed weight and its deviation from the target conductance. The hardware conductance measurements are sent to the AIHWKit running on the IBM cloud, which will then perform inference on them and simulate the additional MVM nonidealities shown in Table V. Inference results are displayed on the UI or can be retrieved via a command line interface. Figure 20 shows a preview of the AAICC user experience that allows access to our first analog inference Fusion chip. This capability is still in beta version and under active development.

E. Road-map and future directions

A cloud no-code interactive experience has been developed to provide a platform for cloud-based experiments, access to IBM Research hardware technology, and the creation of a vibrant ecosystem around Analog AI. AIHWKit can be used online through

02 July 2024 08:09:40

© Author(s) 2023

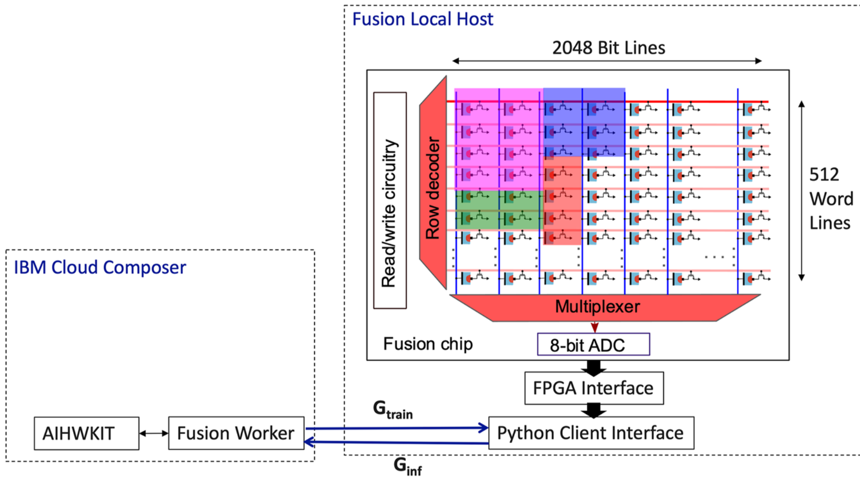

FIG. 19. Access to the fusion chip through the Analog AI Cloud Composer (AAICC).

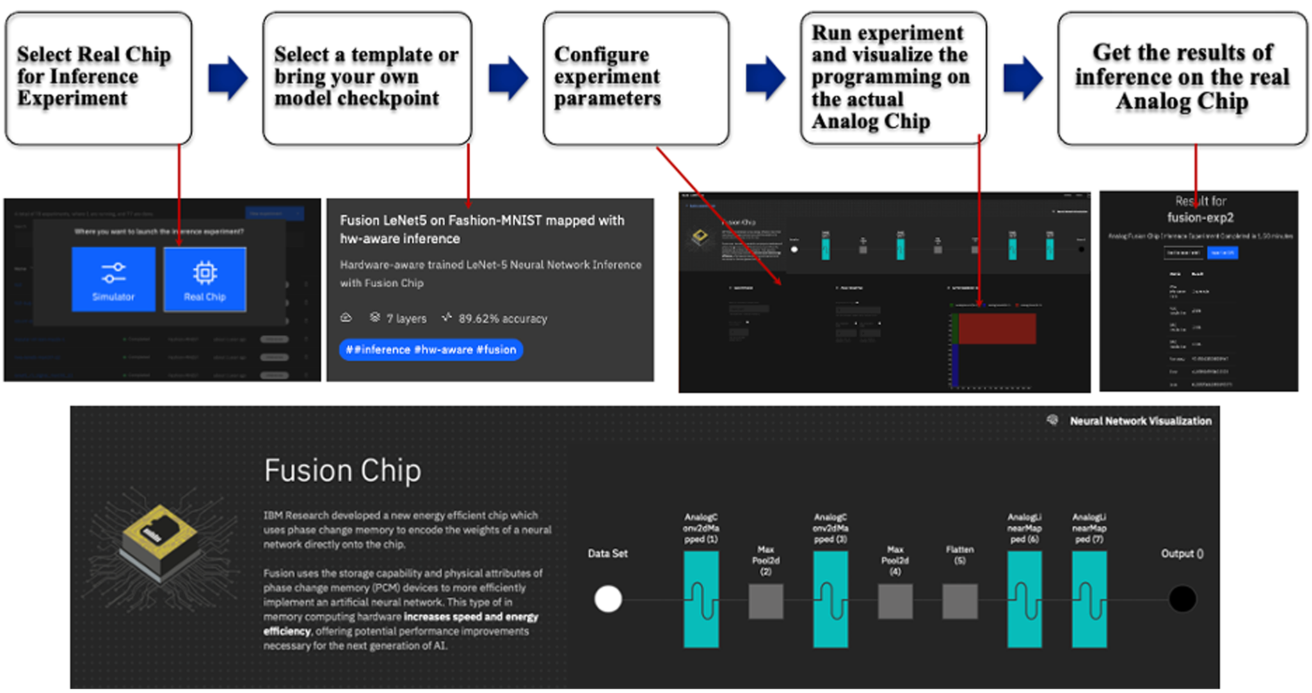

FIG. 20. Workflow of accessing the analog inference PCM-based fusion chip.

a web-based, front-end AAICC. The composer provides a set of templates and a no-code experience to introduce the concepts of Analog AI, configure experiments, and launch training and inference experiments on the IBM public cloud. The future road-map includes adding hardware-aware training, energy, and latency performance models' estimators, access to more IBM Research premium Analog AI chips as a service, adding additional advanced capabilities such as a material builder for training and inference, and continuing to expose the latest algorithmic innovations from IBM Research to the open-source community as consumable services. Figure 21 summarizes our short-term and long-term plans.

VII. HOW TO EXTEND AND CUSTOMIZE THE AIHWKIT

The AIHWKit has been designed to be easily customizable and modular to ease any feature extensions. Moreover, AIHWKit is

02 July 2024 08:09:40

© Author(s) 2023

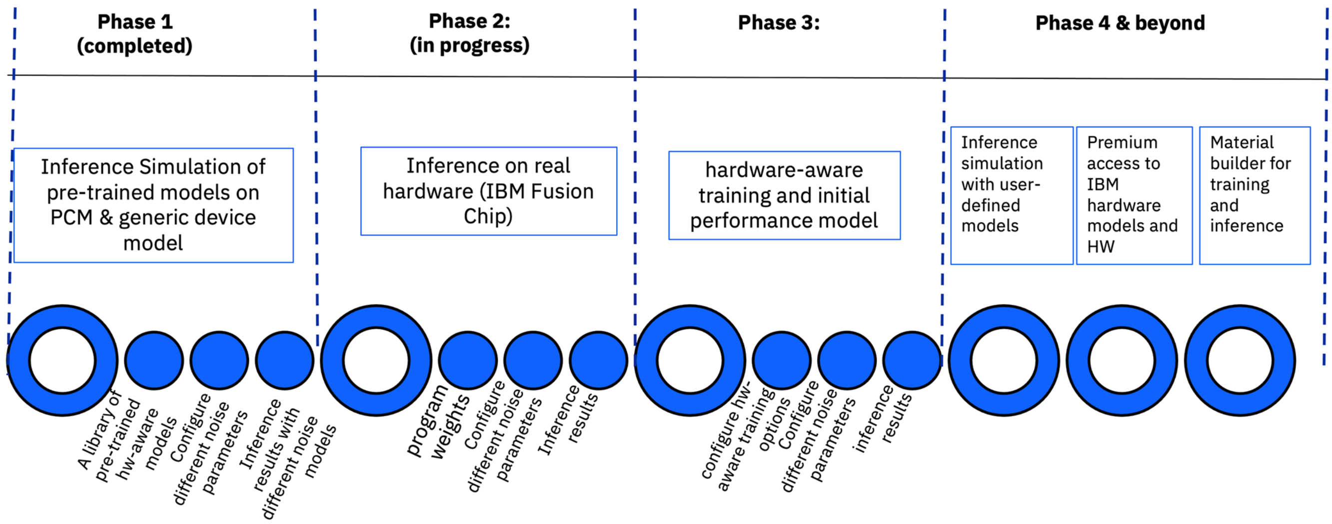

FIG. 21. AAICC application roadmap.

implemented using modern coding and open source practices, such as Python code formatting guidelines, versioning, GitHub integration for collaborative coding, and unit testing to ensure quality and back-functionality when adding new code (see Ref. 29 and the online documentation for more details).

In the following, we give a number of examples of how to extend functionality. In particular, we show how a new phenomenological inference noise model can be added, how a custom drift compensation is implemented, and how the AIMC crossbar simulation could be enhanced.

A. Custom phenomenological inference noise model

Phenomenological inference noise models are applied to model the long-term noise effects of the NVM device (see Sec. IV A). To capture the initial programming error as well as the long-term temporal component of the conductance changes, AIHWKit allows for defining inference noise models (such as `PCMLikeNoiseModel`).

Let us assume one has a new material, and matching the measurements with the provided noise models is not possible even when changing the parameters. In this case, one needs to implement a customized noise model. For that, one needs to derive a new class from the `BaseNoiseModel`, and override a number of methods that define what noise is added. First, `apply_programming_noise_to_conductance`, that applies programming noise to given conductances (in μS) and returns the programmed conductances. Second, `apply_drift_noise_to_conductance`, which applies long-term noise (e.g., drift and $1/f$ noise) to the programmed conductances. Finally, `generate_drift_coefficients`, which generates the drift coefficients during the programming that will be given as input when applying the drift, if needed.

In the following example, we implement a very simple model that just assumes a Gaussian additive programming model and constant conductance drift. The new noise model class could look like this (omitting import statements; see notebook `extending_functionality.ipynb`[74]):

```
class SimpleNVMNoiseModel(BaseNoiseModel):
    """Very simple noise model of a new material """

    def __init__(self,  nu=0.1, prog_std=0.1, **kwargs):
        super().__init__(**kwargs)
        self.nu = nu
        self.prog_std = prog_std  # in muS

    def apply_programming_noise_to_conductance(self, g_target: Tensor) -> Tensor:
        """Apply programming noise to a target conductance Tensor. """
        g_prog = g_target + self.prog_std * randn_like(g_target)
        g_prog.clamp_(min=0.0)  # no negative conductances allowed
        return g_prog

    def generate_drift_coefficients(self, g_target: Tensor) ->Tensor:
        """Just constant nu"""

        return tensor(self.nu)

    def apply_drift_noise_to_conductance(self, g_prog, nu, t_inference) -> Tensor:
        """Apply drift up to the assumed inference time"""
        t_0 = 1  # assume 1 sec as drift reference
        t = t_inference + t_0
        if t <= t_0:
            return g_prog
        return g_prog * ((t / t_0) ** (-nu))
```

Note that before the noise model is applied for inference accuracy evaluation (see Sec. IV), the learned target weight values are passed through a conductance converter to get a list of conductances, for which the noise model is applied [i.e., when `analog_model.drift_analog_weights(t_inference)` is called].

To describe this process in more detail, let us first get the target analog weight values of an analog tile. The target analog weight values are the tile weight values without applying any digital output scales. We thus get these target analog weight values with (here simply for the first analog tile of a model):

```
analog_tile = next(analog_model.analog_tiles())
target_analog_weights, _ = analog_tile.get_weights(apply_weight_scaling=False)
```

02 July 2024 08:09:40

© Author(s) 2023

These target analog weight values are, however, still in normalized units (typically in the range $-1, \ldots, 1$), thus having both negative and positive values. To get the conductances from these normalized target analog weight values, a conductance converter is used (see `aihwkit.inference.converter.conductance`). For instance, the `SinglePairConductanceConverter` would return a list of conductance matrices, one for positive values of the analog weights (setting negative values to zero and scaling it by $g_{\max}$ to get values in μS) and one for negative values (setting positive values to zero and scale it similarly):

```
g_converter = SinglePairConductanceConverter(g_max=25.0)
g_tuple = g_converter.convert_to_conductances(analog_weights)
```

This list of conductance matrices can be converted back to

```
new_analog_weights = g_converter.convert_back_to_weights(*g_tuple)
```

where, in this case, the new analog weights are simply the old ones because the noise model was not yet applied. Note that the conversion from normalized target analog weight values to conductances can also be customized by adding a new conductance converter.

However, this conductance conversion happens internally when the programming is applied. In other words, the noise model above is applied after the conversion to conductances, and then conductances are internally converted back to normalized analog weights and applied back to each analog tile. Therefore, to use the above new noise, model one can simply do, for instance, evaluate a ResNet:

```
rpu_config = InferenceRPUConfig(
                 noise_model=SimpleNVMNoiseModel(prog_std=0.3,
                     g_converter=SinglePairConductanceConverter(g_max=25.)))
analog_model = convert_to_analog(resnet32(), rpu_config)
analog_model.eval()
analog_model.program_analog_weights()
analog_model.drift_analog_weights(3600) # 1 hour
```

Now the analog weights are programmed and drifted, and one could evaluate the accuracy with such long-term noise sources applied to the analog weights.

B. Custom drift compensation

In the above example, no drift compensation was used. Drift compensations are needed for inference with materials that exhibit conductance drift, and they are modular classes in the AIHWKit that can be easily customized.

For instance, assume that the baseline of the drift compensation should be read multiple times (instead of a single time) to improve the signal-to-noise ratio when applying the drift compensation during inference. For that, one could implement a new custom drift class that derives from the base drift compensation class. The custom drift compensation class could look like the following (all import statements are omitted for brevity; see notebook[74] for more details):

```
class NTimesDriftCompensation(BaseDriftCompensation):
    """Global drift compensation with multiple read-outs."""
    def __init__(self, n_times: int = 1) -> None:
        self.n_times = n_times
    def readout(self, out_tensor: Tensor) -> Tensor:
        return clamp(torch_abs(out_tensor).mean(), min=0.0001)
    def get_readout_tensor(self, in_size: int) -> Tensor:
        """Return the read-out tensor with n-times one-hot vectors (eye)."""
        return tile(eye(in_size), [self.n_times, 1])
```

Now this new drift compensation can be simply set when specifying the *RPUConfig*, such as

```
rpu_config = InferenceRPUConfig(
                 drift_compensation=NTimesDriftCompensation(n_times=10))
```

This *RPUConfig* can then be used to define an analog model and will be used for inference evaluation as described in the previous example.

C. Modifying the AIMC MVM for each analog tile

The basic AIMC MVM is typically part of the C++ RPUCUDA engine for speed and, thus, less easily extended using Python. However, AIHWKit provides a separate Python implementation of (some of) the AIMC MVM nonidealities. This analog MVM is encapsulated in the base class `SimulatorTile`. Here we show how one could add changes to the way the analog MVM is performed. In this example, we only show it for inference (deriving from the inference-only tile `TorchSimulatorTile` and modifying the forward pass), but a custom in-memory training tile can similarly be implemented by deriving from the `CustomSimulatorTile` in `aihwkit.simulator.tiles.custom` by overriding the forward, backward, or update methods.

In a simple example, we create a new simulator tile class that modifies the forward pass of the inference evaluation. Currently, the implementation in `TorchSimulatorTile` does negative and positive inputs in one MVM pass. Let us assume one wants to simulate two analog MVMs instead, one for positive and one for negative inputs, and add their results together in digital.

This could be simply done by defining a new `SimulatorTile` that derives from the `TorchSimulatorTile` but overrides the forward pass accordingly. Parameters from the *RPUConfig* can be passed during the initialization and are considered constant. The new class could be defined as (omitting the import statements; see notebook[74] for details)

```
class TwoPassTorchSimulatorTile(TorchSimulatorTile):
    """New class where two forwards are done optionally"""

    def __init__(self, x_size: int, d_size: int,
            rpu_config: "TwoPassTorchInferenceRPUConfig", bias: bool = False):
        super().__init__(x_size, d_size, rpu_config, bias)
        self._one_pass = rpu_config.one_pass

    def forward(self, x_input: Tensor, **kwargs) -> Tensor:
        if self._one_pass:
            return super().forward(x_input, **kwargs)
        x_pos, x_neg = clamp(x_input, min=0.0), -clamp(x_input, max=0.0)
        return super().forward(x_pos, **kwargs) - super().forward(x_neg, **kwargs)
```

Note that this new class defines a new simulator tile that modifies the way the MVM is computed and defines new parameters (`one_pass`). To use this tile in DNN, we need to provide a compatible *RPUConfig* that uses this simulator tile:

```
@dataclass
class TwoPassTorchInferenceRPUConfig(TorchInferenceRPUConfig):
    """Optionally using two forward passes for negative and positive inputs"""
    simulator_tile_class: ClassVar[Type] = TwoPassTorchSimulatorTile
    one_pass: bool = True
    """Optionally turn on the two passes"""
```

Now we can simply use this new *RPUConfig* for model conversion, e.g.:

02 July 2024 08:09:40

© Author(s) 2023

```
rpu_config = TwoPassTorchInferenceRPUConfig(one_pass=False)
analog_model = convert_to_analog(resnet32, rpu_config)
```

The analog model will now use the new simulator tile.

Similarly, other aspects of the AIMC compute can be extended by an analogous approach. For instance, one could add a new peripheral (digital) computation, which would then require overriding methods of the `AnalogTile` or `InferenceTile`, that encapsulate the full tile operations on a higher level (that is, analog MVM simulations in the lower-level `SimulatorTile` and also digital periphery, such as output scaling).

If users decide to implement custom functionality, we highly encourage them to share the new code with the community. Integrating the new addition to the open source community is as easy as raising a new pull request on the AIHWKit GitHub.

VIII. OUTLOOK

Having described in detail the functionality of AIHWKit and how to customize it, we would like to briefly highlight a few possible research directions that could be pursued with the toolkit in this last section. The primary use case for AIHWKit is, of course, the exploration of device-level parameter specifications for inference and training, which has already been the subject of several publications.[22,61,75,76] In addition, novel analog optimizers for on-chip training could be implemented and tested to demonstrate improvement over the existing ones on a wide range of device parameters.[43] For inference, a noteworthy addition to AIHWKit could be to implement the auxiliary digital operations for affine scaling, batch normalization, and activation functions with low-precision arithmetic to study the digital precision requirements on a wide range of networks. Another interesting direction would be to implement input and weight bit slicing[19] and evaluate the impact of those schemes for inference and training. While (almost) arbitrary pre-trained models can already be converted by AIHWKit and custom trained, it would still be worthwhile in the future to make (HWA) training compatible with other training pipeline libraries, such as DeepSpeed,[77] HuggingFace,[78] or Fairseq,[79] in order to conveniently re-use preexisting code using these pipelines. Finally, extending AIHWKit to generate approximate power and latency estimates, using some fairly generic assumptions on the hardware architecture being modeled, would be desirable to explore optimal AIMC design approaches using neural architecture search.[80]

We hope that this Tutorial will make the barrier of entry more accessible for new users to adopt AIHWKit to simulate the inference and training of DNNs with AIMC. AIHWKit not only provides accurate hardware-calibrated models of AIMC devices and the main peripheral circuit nonidealities present in a AIMC chip, but it is also continuously being maintained by a team of developers who are actively fixing issues and adding new features. Therefore, user-made contributions to AIHWKit will be integrated into a well-maintained toolkit and will benefit from being further improved as the toolkit develops, instead of getting "lost" into a private repository that would involve too much overhead from the user to be maintained properly. For this reason, we strongly encourage users working in the AIMC field to adopt actively maintained toolkits, such as AIHWKit, and make the effort to integrate their contributions to them. Otherwise, contributions put in individual repositories will likely get abandoned and just add up to the excessive tool fragmentation that already prevails. It is only with active contributions from the community and by bringing all those contributions together into a single tool that AIMC can eventually become commercially successful and lead to a new era of efficient and sustainable non-von Neumann accelerators.

SUPPLEMENTARY MATERIAL

The supplementary material contains four Jupyter Notebooks that accompany this Tutorial.

ACKNOWLEDGMENTS

We thank Geoffrey Burr, An Chen, Andrea Fasoli, Pritish Narayanan, Tayfun Gokmen, Takashi Ando, John Rozen, Irem Boybat, Athanasios Vasilopoulos, Hadjer Benmeziane, and Ghazi Sarwat Syed for technical input on AIHWKit; Kim Tran, Kurtis Ruby, Borja Godoy, Jordan Murray, Todd Deshane, Diego Moreda, and Kevin Johnson for developing the Analog AI Cloud Composer; and Jeff Burns for managerial support. This work was supported by the IBM Research AI Hardware Center. This work also received funding from the European Union's Horizon Europe research and innovation program under-Grant Agreement No. 101046878 and was supported by the Swiss State Secretariat for Education, Research and Innovation (SERI) under Contract No. 22.00029. We thank the computational support from AiMOS, an AI supercomputer made available by the IBM Research AI Hardware Center and Rensselaer Polytechnic Institute's Center for Computational Innovations (CCI).

AUTHOR DECLARATIONS

Conflict of Interest

The authors have no conflicts to disclose.

Author Contributions

Manuel Le Gallo: Methodology (lead); Project administration (equal); Supervision (equal); Writing – original draft (equal); Writing – review & editing (lead). **Corey Lammie**: Investigation (equal); Methodology (equal); Writing – original draft (equal); Writing – review & editing (equal). **Julian Büchel**: Investigation (equal); Software (equal); Writing – original draft (equal). **Fabio Carta**: Investigation (equal); Software (equal); Writing – original draft (equal). **Omobayode Fagbohungbe**: Investigation (equal); Writing - original draft (equal). **Charles Mackin**: Software (supporting). **Hsinyu Tsai**: Supervision (equal). **Vijay Narayanan**: Supervision (equal). **Abu Sebastian**: Supervision (equal); Writing – review & editing (supporting). **Kaoutar El Maghraoui**: Project administration (lead); Software (equal); Supervision (equal); Writing – original draft (equal); Writing – review & editing (equal). **Malte Rasch**: Methodology (equal); Project administration (equal); Software (lead); Writing - original draft (lead); Writing – review & editing (equal).

02 July 2024 08:09:40

© Author(s) 2023

DATA AVAILABILITY

The data that support the findings of this study are openly available in the AIHWKit repository at https://github.com/IBM/aihwkit.[81]

REFERENCES

[1] J. Burns and L. Chang, "IBM's new AIU artificial intelligence chip," https://research.ibm.com/blog/ibm-artificial-intelligence-unit-aiu (2023).
[2] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," Nat. Nanotechnol. **15**, 529–544 (2020).
[3] M. Lanza, A. Sebastian, W. D. Lu, M. Le Gallo, M.-F. Chang, D. Akinwande, F. M. Puglisi, H. N. Alshareef, M. Liu, and J. B. Roldan, "Memristive technologies for data storage, computation, encryption, and radio-frequency communication," Science **376**, eabj9979 (2022).
[4] S. Yu, H. Jiang, S. Huang, X. Peng, and A. Lu, "Compute-in-memory chips for deep learning: Recent trends and prospects," IEEE Circuits Syst. Mag. **21**, 31–56 (2021).
[5] R. Khaddam-Aljameh, M. Stanisavljevic, J. Fornt Mas, G. Karunaratne, M. Brändli, F. Liu, A. Singh, S. M. Müller, U. Egger, A. Petropoulos, T. Antonakopoulos, K. Brew, S. Choi, I. Ok, F. L. Lie, N. Saulnier, V. Chan, I. Ahsan, V. Narayanan, S. R. Nandakumar, M. Le Gallo, P. A. Francese, A. Sebastian, and E. Eleftheriou, "HERMES-core—A 1.59-TOPS/mm^2 PCM on 14-nm CMOS in-memory compute core using 300-ps/LSB linearized CCO-based ADCs," IEEE J. Solid-State Circuits **57**, 1027–1038 (2022).
[6] P. Narayanan, S. Ambrogio, A. Okazaki, K. Hosokawa, H. Tsai, A. Nomura, T. Yasuda, C. Mackin, S. C. Lewis, A. Friz, M. Ishii, Y. Kohda, H. Mori, K. Spoon, R. Khaddam-Aljameh, N. Saulnier, M. Bergendahl, J. Demarest, K. W. Brew, V. Chan, S. Choi, I. Ok, I. Ahsan, F. L. Lie, W. Haensch, V. Narayanan, and G. W. Burr, "Fully on-chip MAC at 14 nm enabled by accurate row-wise programming of PCM-based weights and parallel vector-transport in duration-format," IEEE Trans. Electron Devices **68**, 6629–6636 (2021).
[7] M. Le Gallo, R. Khaddam-Aljameh, M. Stanisavljevic, A. Vasilopoulos, B. Kersting, M. Dazzi, G. Karunaratne, M. Brändli, A. Singh, S. M. Müller *et al.*, "A 64-core mixed-signal in-memory compute chip based on phase-change memory for deep neural network inference," Nat. Electron. **6**, 680–693 (2023).
[8] S. Ambrogio, P. Narayanan, A. Okazaki, A. Fasoli, C. Mackin, K. Hosokawa, A. Nomura, T. Yasuda, A. Chen, A. Friz *et al.*, "An analog-AI chip for energy-efficient speech recognition and transcription," Nature **620**, 768–775 (2023).
[9] W. Wan, R. Kubendran, C. Schaefer, S. B. Eryilmaz, W. Zhang, D. Wu, S. Deiss, P. Raina, H. Qian, B. Gao *et al.*, "A compute-in-memory chip based on resistive random-access memory," Nature **608**, 504–512 (2022).
[10] J.-M. Hung, C.-X. Xue, H.-Y. Kao, Y.-H. Huang, F.-C. Chang, S.-P. Huang, T.-W. Liu, C.-J. Jhang, C.-I. Su, W.-S. Khwa *et al.*, "A four-megabit compute-in-memory macro with eight-bit precision based on CMOS and resistive random-access memory for AI edge devices," Nat. Electron. **4**, 921–930 (2021).
[11] W. Zhang, P. Yao, B. Gao, Q. Liu, D. Wu, Q. Zhang, Y. Li, Q. Qin, J. Li, Z. Zhu, Y. Cai, D. Wu, J. Tang, H. Qian, Y. Wang, and H. Wu, "Edge learning using a fully integrated neuro-inspired memristor chip," Science **381**, 1205–1211 (2023).
[12] J.-M. Hung, T.-H. Wen, Y.-H. Huang, S.-P. Huang, F.-C. Chang, C.-I. Su, W.-S. Khwa, C.-C. Lo, R.-S. Liu, C.-C. Hsieh, K.-T. Tang, Y.-D. Chih, T.-Y. J. Chang, and M.-F. Chang, "8-b precision 8-Mb ReRAM compute-in-memory macro using direct-current-free time-domain readout scheme for AI edge devices," IEEE J. Solid-State Circuits **58**, 303–315 (2023).
[13] P. Deaville, B. Zhang, and N. Verma, "A 22 nm 128-kb MRAM row/column-parallel in-memory computing macro with memory-resistance boosting and multi-column ADC readout," in *2022 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)* (IEEE, 2022) pp. 268–269.
[14] S. Jain, H. Tsai, C.-T. Chen, R. Muralidhar, I. Boybat, M. M. Frank, S. Woźniak, M. Stanisavljevic, P. Adusumilli, P. Narayanan, K. Hosokawa, M. Ishii, A. Kumar, V. Narayanan, and G. W. Burr, "A heterogeneous and programmable compute-in-memory accelerator architecture for analog-AI using dense 2-D mesh," IEEE Trans. Very Large Scale Integr. Syst. **31**, 114–127 (2023).
[15] G. W. Burr, R. M. Shelby, A. Sebastian, S. Kim, S. Kim, S. Sidler, K. Virwani, M. Ishii, P. Narayanan, A. Fumarola *et al.*, "Neuromorphic computing using non-volatile memory," Adv. Phys.: X **2**, 89–124 (2017).
[16] T. Gokmen and Y. Vlasov, "Acceleration of deep neural network training with resistive cross-point devices: Design considerations," Front. Neurosci. **10**, 333 (2016).
[17] S. Agarwal, R. B. J. Gedrim, A. H. Hsia, D. R. Hughart, E. J. Fuller, A. A. Talin, C. D. James, S. J. Plimpton, and M. J. Marinella, "Achieving ideal accuracies in analog neuromorphic computing using periodic carry," in *2017 Symposium on VLSI Technology* (IEEE, 2017), pp. T174–T175.
[18] S. R. Nandakumar, I. Boybat, J.-P. Han, S. Ambrogio, P. Adusumilli, R. L. Bruce, M. BrightSky, M. J. Rasch, M. L. Gallo, and A. Sebastian, "Precision of synaptic weights programmed in phase-change memory devices for deep learning inference," in *IEEE International Electron Devices Meeting (IEDM)* (IEEE, 2020).
[19] M. Le Gallo, S. Nandakumar, L. Ciric, I. Boybat, R. Khaddam-Aljameh, C. Mackin, and A. Sebastian, "Precision of bit slicing with in-memory computing based on analog phase-change memory crossbars," Neuromorphic Comput. Eng. **2**, 014009 (2022).
[20] C. Mackin, M. J. Rasch, A. Chen, J. Timcheck, R. L. Bruce, N. Li, P. Narayanan, S. Ambrogio, M. Le Gallo, S. Nandakumar *et al.*, "Optimised weight programming for analogue memory-based deep neural networks," Nat. Commun. **13**, 3765 (2022).
[21] V. Joshi, M. Le Gallo, S. Haefeli, I. Boybat, S. R. Nandakumar, C. Piveteau, M. Dazzi, B. Rajendran, A. Sebastian, and E. Eleftheriou, "Accurate deep neural network inference using computational phase-change memory," Nat. Commun. **11**, 2473 (2020).
[22] M. J. Rasch, C. Mackin, M. Le Gallo, A. Chen, A. Fasoli, F. Odermatt, N. Li, S. Nandakumar, P. Narayanan, H. Tsai *et al.*, "Hardware-aware training for large-scale and diverse deep learning inference workloads using in-memory computing-based accelerators," Nat. Commun. **14**, 5282 (2023).
[23] X. Liu and Z. Zeng, "Memristor crossbar architectures for implementing deep neural networks," Complex Intell. Syst. **8**, 787–802 (2022).
[24] T. P. Xiao, C. H. Bennett, B. Feinberg, S. Agarwal, and M. J. Marinella, "Analog architectures for neural network acceleration based on non-volatile memory," Appl. Phys. Rev. **7**, 031301 (2020).
[25] A. Lu, X. Peng, W. Li, H. Jiang, and S. Yu, "NeuroSim simulator for compute-in-memory hardware accelerator: Validation and benchmark," Front. Artif. Intell. **4**, 659060 (2021).
[26] C. Lammie, W. Xiang, and M. Rahimi Azghadi, "Modeling and simulating in-memory memristive deep learning systems: An overview of current efforts," Array **13**, 100116 (2022).
[27] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **31**, 994–1007 (2012).
[28] L. Xia, B. Li, T. Tang, P. Gu, P.-Y. Chen, S. Yu, Y. Cao, Y. Wang, Y. Xie, and H. Yang, "MNSIM: Simulation platform for memristor-based neuromorphic computing system," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **37**, 1009–1022 (2018).
[29] M. J. Rasch, D. Moreda, T. Gokmen, M. Le Gallo, F. Carta, C. Goldberg, K. El Maghraoui, A. Sebastian, and V. Narayanan, "A flexible and fast PyTorch toolkit for simulating training and inference on analog crossbar arrays," in *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)* (IEEE, 2021), pp. 1–4.
[30] CPU and GPU versions can be installed from https://anaconda.org/conda-forge/aihwkit and https://anaconda.org/conda-forge/aihwkit-gpu, respectively.
[31] L. Chen, J. Li, Y. Chen, Q. Deng, J. Shen, X. Liang, and L. Jiang, "Accelerator-friendly neural-network training: Learning variations and defects in RRAM crossbar," in *Proceedings of the Conference on Design, Automation and Test in Europe* (European Design and Automation Association, 2017), pp. 19–24.
[32] P.-Y. Chen, X. Peng, and S. Yu, "NeuroSim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **37**, 3067–3080 (2018).
[33] Y. Luo, X. Peng, and S. Yu, "MLP+NeuroSimV3.0," in *Proceedings of the International Conference on Neuromorphic Systems* (ACM, 2019).
[34] X. Peng, S. Huang, Y. Luo, X. Sun, and S. Yu, "DNN+NeuroSim: An end-to-end benchmarking framework for compute-in-memory accelerators with versatile

02 July 2024 08:09:40

© Author(s) 2023

device technologies," in *2019 IEEE International Electron Devices Meeting (IEDM)* (IEEE, 2019).

[35] X. Peng, S. Huang, H. Jiang, A. Lu, and S. Yu, "DNN+NeuroSim v2.0: An end-to-end benchmarking framework for compute-in-memory accelerators for on-chip training," IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **40**, 2306–2319 (2021).

[36] H. Liu, J. Han, and Y. Zhang, "A unified framework for training, mapping and simulation of ReRAM-based convolutional neural network acceleration," IEEE Comput. Archit. Lett. **18**, 63–66 (2019).

[37] C. Lammie and M. R. Azghadi, "MemTorch: A simulation framework for deep memristive cross-bar architectures," in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)* (IEEE, 2020).

[38] C. Lammie, W. Xiang, B. Linares-Barranco, and M. Rahimi Azghadi, "MemTorch: An open-source simulation framework for memristive deep learning systems," Neurocomputing **485**, 124–133 (2022).

[39] T. P. Xiao, C. H. Bennett, B. Feinberg, M. J. Marinella, and S. Agarwal, "CrossSim: Accuracy simulation of analog in-memory computing," GitHub, v2.0, https://github.com/sandialabs/cross-sim (2022).

[40] G. W. Burr, R. M. Shelby, S. Sidler, C. Di Nolfo, J. Jang, I. Boybat, R. S. Shenoy, P. Narayanan, K. Virwani, E. U. Giacometti *et al.*, "Experimental demonstration and tolerancing of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element," IEEE Trans. Electron Devices **62**, 3498–3507 (2015).

[41] T. Gokmen and W. Haensch, "Algorithm for training neural networks on resistive device arrays," Front. Neurosci. **14**, 103 (2020).

[42] T. Gokmen, "Enabling training of neural networks on noisy hardware," Front. Artif. Intell. **4**, 1–14 (2021).

[43] M. J. Rasch, F. Carta, O. Fagbohungbe, and T. Gokmen, "Fast offset corrected in-memory training," arXiv:2303.04721 (2023).

[44] S. R. Nandakumar, M. Le Gallo, C. Piveteau, V. Joshi, G. Mariani, I. Boybat, G. Karunaratne, R. Khaddam-Aljameh, U. Egger, A. Petropoulos, T. Antonakopoulos, B. Rajendran, A. Sebastian, and E. Eleftheriou, "Mixed-precision deep learning based on computational memory," Front. Neurosci. **14**, 406 (2020).

[45] T. Gokmen, M. Onen, and W. Haensch, "Training deep convolutional neural networks with resistive cross-point devices," Front. Neurosci. **11**, 1–22 (2017).

[46] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems* (IEEE, 2019), Vol. 32.

[47] M. J. Rasch, T. Gokmen, and W. Haensch, "Training large-scale artificial neural networks on simulated resistive crossbar arrays," IEEE Des. Test **37**, 19–29 (2020).

[48] M. Croci, M. Fasi, N. J. Higham, T. Mary, and M. Mikaitis, "Stochastic rounding: Implementation, error analysis and applications," R. Soc. Open Sci. **9**, 211631 (2022).

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

[50] S. R. Nandakumar, I. Boybat, V. Joshi, C. Piveteau, M. Le Gallo, B. Rajendran, A. Sebastian, and E. Eleftheriou, "Phase-change memory models for deep learning training and inference," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)* (IEEE, 2019), pp. 727–730.

[51] S. G. Sarwat, M. Le Gallo, R. L. Bruce, K. Brew, B. Kersting, V. P. Jonnalagadda, I. Ok, N. Saulnier, M. BrightSky, and A. Sebastian, "Mechanism and impact of bipolar current voltage asymmetry in computational phase-change memory," Adv. Mater. **35**, 2201238 (2022).

[52] I. Boybat, M. Le Gallo, S. Nandakumar, T. Moraitis, T. Parnell, T. Tuma, B. Rajendran, Y. Leblebici, A. Sebastian, and E. Eleftheriou, "Neuromorphic computing with multi-memristive synapses," Nat. Commun. **9**, 2514 (2018).

[53] S. Ambrogio, M. Gallot, K. Spoon, H. Tsai, C. Mackin, M. Wesson, S. Kariyappa, P. Narayanan, C.-C. Liu, A. Kumar, A. Chen, and G. W. Burr, "Reducing the impact of phase-change memory conductance drift on the inference of large-scale hardware neural networks," in *2019 IEEE International Electron Devices Meeting (IEDM)* (IEEE, 2019), pp. 6.1.1–6.1.4.

[54] M. Le Gallo, A. Sebastian, G. Cherubini, H. Giefers, and E. Eleftheriou, "Compressed sensing with approximate message passing using in-memory computing," IEEE Trans. Electron Devices **65**, 4304–4312 (2018).

[55] H. Tsai, S. Ambrogio, C. Mackin, P. Narayanan, R. M. Shelby, K. Rocki, A. Chen, and G. W. Burr, "Inference of long-short term memory networks at software-equivalent accuracy using 2.5 M analog phase change memory devices," in *2019 Symposium on VLSI Technology* (IEEE, 2019), pp. T82–T83.

[56] X. Yang, C. Wu, M. Li, and Y. Chen, "Tolerating noise effects in processing-in-memory systems for neural networks: A hardware–software codesign perspective," Adv. Intell. Syst. **4**, 2200029 (2022).

[57] T. Gokmen, M. J. Rasch, and W. Haensch, "The marriage of training and inference for scaled deep learning analog hardware," in *2019 IEEE International Electron Devices Meeting (IEDM)* (IEEE, 2019), pp. 22–23.

[58] S. Kariyappa, H. Tsai, K. Spoon, S. Ambrogio, P. Narayanan, C. Mackin, A. Chen, M. Qureshi, and G. W. Burr, "Noise-resilient DNN: Tolerating noise in PCM-based AI accelerators via noise-aware training," IEEE Trans. Electron Devices **68**, 4356–4362 (2021).

[59] K. Spoon, H. Tsai, A. Chen, M. J. Rasch, S. Ambrogio, C. Mackin, A. Fasoli, A. M. Friz, P. Narayanan, M. Stanisavljevic, and G. W. Burr, "Toward software-equivalent accuracy on transformer-based deep neural networks with analog memory devices," Front. Comput. Neurosci. **15**, 1–9 (2021).

[60] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using dropconnect," in *Proceedings of the 30th International Conference on Machine Learning, Proceedings of the Machine Learning Research*, edited by S. Dasgupta and D. McAllester (PMLR, Atlanta, Georgia, 2013), Vol. 28, pp. 1058–1066.

[61] N. Li, H. Tsai, V. Narayanan, and M. Rasch, "Impact of analog memory device failure on in-memory computing inference accuracy," APL Mach. Learn. **1**, 016104 (2023).

[62] J. Büchel, M. Le Gallo, and K. El Maghraoui, https://github.com/IBM/aihwkit/tree/master/notebooks/tutorial/hw_aware_training.ipynb, 2023.

[63] J. Büchel, M. Le Gallo, and K. El Maghraoui, https://github.com/IBM/aihwkit/tree/master/notebooks/tutorial/post_training_input_range_calibration.ipynb, 2023.

[64] See https://github.com/ysh329/deep-learning-model-convertor for more information about external libraries that can be used to convert trained models to PYTORCH-based models.

[65] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," technical report, https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf (2009).

[66] N. Gong, T. Idé, S. Kim, I. Boybat, A. Sebastian, V. Narayanan, and T. Ando, "Signal and noise extraction from analog memory elements for neuromorphic computing," Nat. Commun. **9**, 2102 (2018).

[67] N. Gong, M. J. Rasch, S.-C. Seo, A. Gasasira, P. Solomon, V. Bragaglia, S. Consiglio, H. Higuchi, C. Park, K. Brew *et al.*, "Deep learning acceleration in 14 nm CMOS compatible ReRAM array: Device, material and algorithm co-optimization," in *IEEE International Electron Devices Meeting* (IEEE, 2022).

[68] Y. Li, S. Kim, X. Sun, P. Solomon, T. Gokmen, H. Tsai, S. Koswatta, Z. Ren, R. Mo, C. C. Yeh *et al.*, "Capacitor-based cross-point array for analog neural network with record symmetry and linearity," in *2018 IEEE Symposium on VLSI Technology* (IEEE, 2018), pp. 25–26.

[69] J. Tang, D. Bishop, S. Kim, M. Copel, T. Gokmen, T. Todorov, S. Shin, K.-T. Lee, P. Solomon, K. Chan, W. Haensch, and J. Rozen, "ECRAM as scalable synaptic cell for high-speed, low-power neuromorphic computing," in *2018 IEEE International Electron Devices Meeting (IEDM)* (IEEE, 2018), pp. 13.1.1–13.1.4.

[70] S. Kim, T. Todorov, M. Onen, T. Gokmen, D. Bishop, P. Solomon, K.-T. Lee, M. Copel, D. B. Farmer, J. A. Ott *et al.*, "Metal-oxide based, CMOS-compatible ECRAM for deep learning accelerator," in *2019 IEEE International Electron Devices Meeting (IEDM)* (IEEE, 2019), pp. 35–37.

[71] F. Carta, M. J. Rasch, and K. El Maghraoui, https://github.com/IBM/aihwkit/tree/master/notebooks/tutorial/analog_training.ipynb, 2023.

[72] H. Kim, M. J. Rasch, T. Gokmen, T. Ando, H. Miyazoe, J.-J. Kim, J. Rozen, and S. Kim, "Zero-shifting technique for deep neural network training on resistive cross-point arrays," arXiv:1907.10228 [cs.ET] (2019).

[73] Note that the weight in this context refers to the *analog weight* $\breve{W}$, that is the normalized conductance value that is stored in physical units in the crossbar devices (see Sec. III D 1). The full mathematical weight that is responsible for the MVM as seen by the next layer is given by the product of the analog weight and the digital output scales after the ADC.

02 July 2024 08:09:40

© Author(s) 2023

[74]M. J. Rasch, https://github.com/IBM/aihwkit/blob/master/notebooks/tutorial/extending_functionality.ipynb, 2023.
[75]Z. Yu, S. Menzel, J. P. Strachan, and E. Neftci, "Integration of physics-derived memristor models with machine learning frameworks," in *2022 56th Asilomar Conference on Signals, Systems, and Computers* (IEEE, 2022), pp. 1142–1146.
[76]C. Lee, K. Noh, W. Ji, T. Gokmen, and S. Kim, "Impact of asymmetric weight update on neural network training with Tiki-Taka algorithm," Front. Neurosci. **15**, 767953 (2022).
[77]J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'20* (Association for Computing Machinery, New York, 2020), pp. 3505–3506.
[78]T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, and A. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations* (Association for Computational Linguistics, Online, 2020), pp. 38–45.
[79]M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A fast, extensible toolkit for sequence modeling," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)* (Association for Computational Linguistics, Minneapolis, Minnesota, 2019), pp. 48–53.
[80]H. Benmeziane, C. Lammie, I. Boybat, M. Rasch, M. Le Gallo, H. Tsai, R. Muralidhar, S. Niar, O. Hamza, V. Narayanan, A. Sebastian, and K. El Maghraoui, "AnalogNAS: A neural network design framework for accurate inference with analog in-memory computing," in *2023 IEEE International Conference on Edge Computing and Communications (EDGE)* (IEEE, 2023), pp. 233–244.
[81]M. Rasch, T. Gokmen, D. Moreda, M. Le Gallo, and K. El Maghraoui (2023). "IBM analog hardware acceleration kit," *GitHub*, v. 0.8.0. https://github.com/IBM/aihwkit

02 July 2024 08:09:40

© Author(s) 2023