CxDNN: Hardware-software Compensation Methods for Deep Neural Networks on Resistive Crossbar Systems

SHUBHAM JAIN and ANAND RAGHUNATHAN, School of Electrical and Computer Engineering, Purdue University

113

Resistive crossbars have shown strong potential as the building blocks of future neural fabrics, due to their ability to natively execute vector-matrix multiplication (the dominant computational kernel in DNNs). However, a key challenge that arises in resistive crossbars is that non-idealities in the synaptic devices, interconnects, and peripheral circuits of resistive crossbars lead to errors in the computations performed. When large-scale DNNs are executed on resistive crossbar systems, these errors compound and result in unacceptable degradation in application-level accuracy. We propose CxDNN, a hardware-software methodology that enables the realization of large-scale DNNs on crossbar systems by compensating for errors due to non-idealities, greatly mitigating the degradation in accuracy. CxDNN is composed of (i) an optimized mapping technique to convert floating-point weights and activations to crossbar conductances and input voltages, (ii) a fast one-time re-training method to recover accuracy loss due to this conversion, and (iii) low-overhead compensation hardware to mitigate dynamic and hardware-instance-specific errors. Unlike previous efforts that are limited to small networks and require the training and deployment of hardware-instance-specific models, CxDNN presents a scalable compensation methodology that can address large DNNs (e.g., ResNet-50 on ImageNet) and maintains the train-once-deploy-anywhere tenet of current DNN application. We evaluated CxDNN on six top DNNs on the ImageNet dataset with 0.5–13.8 million neurons and 0.5–15.5 billion connections. CxDNN achieves 16.9%–49% improvement in the top-1 classification accuracy, effectively mitigating a key challenge to the use of resistive crossbar-based neural fabrics.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Neural networks**; • **Hardware** → **Memory and dense storage**;

Additional Key Words and Phrases: Resistive crossbars, in-memory computing, crossbar non-idealities, Deep Neural Networks (DNNs), emerging memories

ACM Reference format:

Shubham Jain and Anand Raghunathan. 2019. CxDNN: Hardware-software Compensation Methods for Deep Neural Networks on Resistive Crossbar Systems. *ACM Trans. Embed. Comput. Syst.* 18, 6, Article 113 (November 2019), 23 pages.

<https://doi.org/10.1145/3362035>

This work was supported in part by C-BRIC, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Authors' addresses: S. Jain and A. Raghunathan, 465 Northwestern Ave, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907; emails: {jain130, raghunathan}@purdue.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

1539-9087/2019/11-ART113 \$15.00

<https://doi.org/10.1145/3362035>

1 INTRODUCTION

Deep neural networks (DNNs) have greatly advanced the state-of-the-art in machine learning tasks involving video, image, speech, and natural language processing, and are currently used in several real-world products and services [26]. The high computation and memory requirements of DNNs often pose challenges to the computing platforms on which they are executed. Currently, GPUs and specialized accelerators like Google's TPU and Microsoft's Brainwave are the mainstays for DNN execution. However, the continuing increase in DNN complexity and amount of data processed has fueled an active interest in emerging hardware fabrics that can deliver further improvements in efficiency, and thereby enable the virtuous cycle of advances in deep learning algorithms and hardware to continue.

Resistive crossbars have attracted significant interest in recent years due to their ability to natively execute vector-matrix multiplication, which is the dominant compute kernel in DNNs. Resistive crossbars designed using various emerging Non-Volatile Memory (NVM) technologies, *viz.*, Phase change memory (PCM), Resistive RAM (ReRAM), and Spintronics, have shown promise in enabling highly dense and energy-efficient DNN implementations [6, 7, 9, 10, 18, 29, 32, 33]. Consequently, there have been several research efforts on resistive crossbars at various levels of abstraction [6, 7, 9, 10, 18, 29, 32, 33].

A key challenge with resistive crossbars is that the computed function (input voltages multiplied with the weights stored as conductances to obtain output currents) is only an approximation of the desired vector-matrix multiplications. In reality, the computed outputs can deviate significantly from the desired value due to limited precision, as well as numerous device and circuit-level non-idealities, *viz.*, interconnect resistance, sensing and driver resistances, noise in synaptic devices, ADC and DAC non-linearity, sneak paths, imperfect write operations, and process variations [2–5, 14, 15, 19, 21–23, 34, 35]. The cumulative effect of all these non-idealities manifests as errors in the vector-matrix multiplication that depend on data (programmed weights and applied inputs), how the computation is mapped to the crossbar (since errors vary across crossbar columns), and the hardware instance performing the computation (due to variations). These errors in vector-matrix multiplications can accumulate over numerous crossbar operations within a DNN layer and propagate through the different network layers, eventually impacting the accuracy of the DNN. The overall impact of non-idealities on DNN accuracy strongly depends on the scale of the network. Figure 1 illustrates the accuracy degradation due to limited precision and device and circuit-level non-idealities, for simple and complex DNNs (details in Section 5). In the figure, FP32 represents a floating-point implementation, Cross-Ideal6 is an “ideal” crossbar implementation that only considers limited precision (6 bits), and Cross-NI6-64 $\times$ 64 is an implementation with 64 $\times$ 64 crossbars that considers all crossbar non-idealities. We observe the accuracy degradation to be minimal for simple networks (LeNet and ConvNet) with smaller and fewer layers. However, the accuracy degradation can be substantial (19.8%–57.8%) for large-scale DNNs such as VGG-16, GoogleNet, and ResNet-50. *Therefore, it is essential to address the challenges posed by crossbar non-idealities to enable adoption of resistive crossbar-based systems.*

Prior efforts that have addressed non-idealities in resistive crossbars are limited by the scale of networks that they consider (*there is no reported work on ImageNet DNNs such as AlexNet, VGG-16, GoogLeNet, and ResNet*) and only mitigate a subset of crossbar non-idealities [3, 5, 14, 19, 21–23, 35]. Re-training DNNs using software models of the crossbar is a popular approach to compensating for non-idealities [2, 3, 5, 9, 10, 19, 23]. However, re-training can only mitigate accuracy degradation to a limited extent. Moreover, a key limitation of re-training-based compensation is that each hardware instance requires separate model re-training and the trained network is tied to the specific hardware instance. Breaking the train-once-deploy-anywhere tenet of current DNN applications raises significant challenges. For example, consider a potential deployment of

Fig. 1. Accuracy degradation due to non-idealities.

resistive-crossbar-based inference accelerators in a million edge devices. Deployment of a new DNN on these devices would require characterizing the non-idealities of each instance, communicating these non-idealities to a cloud-based infrastructure, invoking a million runs of re-training in the cloud, and transferring a unique model back to each device, thereby greatly raising deployment complexity and computational requirements. Compensation mechanisms without re-training have only been explored in the limited context of small-scale auto-associative neural networks to mitigate specific non-idealities such as imperfect programming and drift in synaptic conductances [21, 35]. Therefore, there is a need for a design methodology that: (i) can effectively compensate for errors due to a wide range of crossbar non-idealities, (ii) does not use hardware-instance-specific training, (iii) incurs low hardware overhead, and (iv) is scalable to large DNNs.

We propose CxDNN, a hardware-software methodology to realize high-accuracy DNNs on resistive crossbar systems. CxDNN is composed of a conversion algorithm, a fast re-training method, and a low-overhead hardware compensation scheme. The conversion algorithm takes any given floating-point DNN and converts its weights to conductances and activation values to voltages by determining appropriate scaling factors. Subsequently, CxDNN uses a fast re-training method to recover accuracy loss due to the conversion with very few iterations.¹ We note that our re-training method is hardware-instance-independent and therefore needs to be performed only once overall (and not once per instance). Finally, CxDNN mitigates data-dependent and hardware-instance-specific errors using low-overhead error compensation circuits that post-process the crossbar outputs. We evaluate CxDNN using six state-of-the-art DNNs on the ImageNet dataset. Our results show that CxDNN improves top-1 accuracy by 16%–49% and obtains classification accuracies comparable to a software fixed-point implementation, thereby alleviating non-idealities as an impediment to the use of resistive crossbar-based neural fabrics.

We organize the rest of the article as follows: Section 2 provides an overview of prior efforts that address non-idealities in resistive crossbars. Section 3 presents the necessary background on resistive crossbar systems and crossbar non-idealities. Section 4 details the proposed CxDNN compensation methodology. We outline the experimental methodology in Section 5. The experimental results are presented in Section 6, and Section 7 concludes the article.

¹Training ResNet-50 on the ImageNet dataset requires ~365 K iterations with a batch size of 50. In comparison, CxDNN requires <500 iterations.

2 RELATED WORK

In recent years, many research efforts have focused on designing resistive crossbar-based systems for DNNs at the device [18, 33], architectural [6, 7, 29], and algorithmic [9, 10, 32] levels. We limit our discussion to prior efforts that attempt to mitigate the effects of crossbar non-idealities. These efforts can be grouped into (i) methods that use (re-)training to compensate for errors [2, 3, 5, 19, 23], (ii) mapping to reduce computational errors [14], (iii) closed-loop re-programming methods to overcome imperfect programming and drift in synaptic conductances [21, 22, 35], and (iv) ECC schemes to reduce errors due to a specific device non-ideality [8].

As mentioned in Section 1, training-based compensation [2, 3, 5, 19, 23] using software models of the crossbar is the most popular approach to compensating for non-idealities. However, re-training can only mitigate accuracy degradation to a limited extent. Further, it ties the network to the specific hardware instance and therefore is not scalable. For example, to create re-trained models for crossbar-based inference accelerators deployed in millions of IoT devices will correspondingly require millions of training runs. Another limitation of these efforts [2, 3, 5, 19, 23] is that they correct errors due to only a subset of crossbar non-idealities. For example, Reference [2] can only mitigate errors due to driver resistance, source resistance, and chip-to-chip variations. It ignores other prominent non-idealities such as wire resistance, sneak paths, non-linear ADCs and DACs, and intra-chip process variations (each crossbar or synaptic device within a chip is uniquely impacted by process-variations). Finally, all (re-)training-based prior efforts [2, 3, 5, 14, 19, 21–23, 35] are limited to small-scale networks and datasets (we are unaware of any reported training-based compensation work on ImageNet DNNs).

ECC scheme using AN codes have also been proposed to correct errors in resistive crossbars [8]. However, AN codes [1] can correct errors only in linear functions realized using digital systems, wherein errors are discrete (at the bit-level) and can be quantified using the digital number representation (multiple bits). Although vector-matrix multiplications realized using digital systems are linear, the functions executed in resistive crossbars are not. For example, vector-matrix multiplications in resistive crossbars consist of several non-linear operations (*viz.*, non-linear DACs and ADCs functions). The errors in resistive crossbars due to the cumulative effect of all non-idealities are analog and non-linear in nature. Therefore, ECC schemes cannot effectively mitigate errors in analog systems (experimental results shown in Section 6.5). Further, apart from assuming crossbar operations to be linear, another limitation of the prior effort [8] is that it only models errors due to a specific device non-ideality (random telegraph noise) and ignores circuit-level non-idealities, *viz.*, wire resistances, sensing resistances, sneak paths, driver resistances, and non-linear ADCs and DACs.

Other prior efforts [21, 22, 35] using closed-loop programming may be combined with CxDNN to specifically address programming errors.

CxDNN complements and improves upon previous efforts by proposing a hardware-software methodology that scales to large DNNs, avoids hardware-instance-specific (re-)training, can correct errors due to a wider range of crossbar non-idealities (including both device and circuit-level non-idealities), and uses low-overhead compensation circuits to better alleviate dynamic and hardware-instance-specific errors.

3 PRELIMINARIES

In this section, we provide a brief background on resistive crossbar-based accelerator. We also discuss vector-matrix multiplication using resistive crossbars and errors introduced therein due to various crossbar non-idealities.

Fig. 2. Architecture for resistive crossbar system.

3.1 Resistive Crossbar System

Figure 2 (left side) depicts a representative resistive crossbar-based accelerator, viz., Resistive crossbar processing unit (XPU), connected to a host processor and main memory via a system bus. The host processor off-loads execution of compute kernels having vector-matrix multiplications (e.g., convolution and fully-connected layers in DNNs) to the XPU, which in turn realizes them using multiple Crossbar processing tiles (XPTs) and a scheduler. XPTs are composed of Resistive Crossbar Arrays (RCAs) that store weights and compute vector-matrix multiplications, local memories in the form of input and output buffers to store activations, a local bus, a controller to orchestrate various operations, and a special function unit (SFU) to execute other DNN operations such as ReLU, max pooling, and so on. Next, we describe RCAs, the fundamental compute units of resistive crossbar systems, in detail.

As shown in Figure 2 (right side), a resistive crossbar array is composed of a 2D array of synaptic elements, Analog-to-Digital Converters (ADCs), Digital-to-Analog Converters (DACs), registers, a decoder, a column MUX, a reduce unit, and write circuits. The synaptic elements are programmable resistors that store DNN weights as conductances. The number of available conductance levels determines the precision of the synaptic element. They can be realized using emerging NVMs technologies, viz., PCM, ReRAM, and spintronics [18, 28, 33]. RCA supports two main operations: (i) vector-matrix multiplication and (ii) programming. A vector-matrix multiplication is performed by driving all wordlines (WL) to analog voltages using DACs and sensing the resultant currents flowing through the columns (BL) using ADCs. The vector-matrix multiplication ideally realized in an RCA can be expressed as $I_{out_ideal} = V_{in_ideal} * G_{ideal}$, where V_{in_ideal} represents input voltages, G_{ideal} denotes programmed conductances, and I_{out_ideal} represents output currents. In contrast, the programming operations, i.e., write operations on synaptic elements, are performed row-wise, wherein the write circuitry applies the necessary currents and sets them to the desired conductance.

Fig. 3. Crossbar non-idealities and error characteristics.

An XPT realizes vector-matrix multiplications involving a large weight matrix by partitioning it into fragments and mapping them to multiple RCAs. The partitions can have dependencies, i.e., they can either share inputs or outputs (produce partial sums for the same output). We map partitions sharing the inputs (outputs) to same rows (columns). RCAs also have additional logic units (e.g., reduce unit) to handle partial sums and to support input sharing across rows.

3.2 Errors Due to Crossbar Non-idealities

The computed vector-matrix multiplications in resistive crossbars suffer from various device and circuit-level non-idealities, viz., wire resistances, driver and sensing resistances, non-ideal DACs and ADCs, sneak paths, imperfect write operations, and process variations (shown in Figure 3(a)) [2–5, 14, 15, 19, 21–23, 34, 35]. Cumulatively, these non-idealities introduce errors at the crossbar outputs. The scatter plot in Figure 3(b) depicts these errors across 500 vector-matrix multiplication operations. In the figure, the x-axis is the expected (ideal crossbar) output and the y-axis is the actual crossbar output (with non-idealities). We observe that the non-ideal outputs deviate significantly from the expected outputs as the data-points in the scatter plot (Figure 3(b)). We also find the non-ideal outputs to be almost always smaller than the ideal outputs due to the additional parasitics such as wire resistances, sensing resistances, and driver resistances that result in smaller output currents due to IR-drops. Further, the errors in resistive crossbars are data-dependent as well as hardware-instance specific. In other words, the errors are sensitive to

Fig. 4. CxDNN usage model.

programmed weights, applied inputs, the mapping of weight matrices to crossbars, and the variations impacting the crossbar elements.

To study the error characteristics, we perform SPICE simulations of 1 K resistive crossbar arrays (RCAs) executing vector-matrix multiplications. We vary the programmed conductances and the inputs across these RCA instances and consider random variations in synaptic conductances ($\sigma/\mu = 10\%$). Figure 3(c) presents the sensitivity of errors to synaptic conductances, applied inputs, crossbar columns, and process variations for a 64×64 resistive crossbar. The plots show the maximum (MAX-Error) and minimum (MIN-Error) error for each crossbar column across 1 K SPICE simulations. We also show error dependence on the crossbar instance by plotting the errors for individual crossbar instances (Error-RCA). We observe that errors are dynamic and hardware-instance-specific, motivating a joint hardware-software compensation approach to mitigate them. These errors at the crossbar-level accumulate over numerous vector-matrix multiplications and propagate across DNN layers, leading to unacceptable degradation (19.8%–57.8%) in application-level accuracy (as shown in Figure 1).

4 CxDNN: COMPENSATION METHODS FOR DNNs ON RESISTIVE CROSSBARS

In this section, we present CxDNN, a hardware-software methodology to realize DNNs with high accuracy on *resistive crossbar-based accelerators for DNN inference*. We first motivate the approach adopted by CxDNN and subsequently detail the software and hardware compensation methods.

4.1 CxDNN: Approach and Overview

Figure 4 contrasts the design approach adopted by CxDNN with prior efforts by highlighting the difference in their usage models. We consider a scenario in which XPUs are deployed on millions of mobile and IoT devices to accelerate DNN inference. As shown in Figure 4(a), prior schemes that rely on (re-)training to mitigate hardware-instance-specific errors will have to characterize non-idealities for each instance and communicate these characteristics to a cloud-based infrastructure. Subsequently, a run of re-training is invoked for each device, and a unique model is transferred back to each device. Therefore, these schemes are not scalable. In contrast, CxDNN preserves the train-once-deploy-anywhere tenet of current DNN implementations (shown in Figure 4(b)). It designs DNNs that are re-trained once and deployed on many XPU instances. Further, the XPU instances (XPU⁺) are equipped with low-overhead compensation hardware to mitigate instance-specific errors.

Fig. 5. CxDNN Overview.

Figure 5 outlines the CxDNN compensation flow, which consists of a quantization and conversion algorithm, hardware-independent re-training, and hardware compensation. CxDNN takes a readily available floating-point (FP32) DNN as an input. It first quantizes the weights and activations of the FP32 network to match the precisions of the synaptic devices and ADCs/DACs, resulting in a fixed-point (FxP) DNN. Subsequently, it transforms the weights into conductances and activations to voltages to obtain a DNN for an “ideal” crossbar (with no device and circuit non-idealities other than limited precision), denoted DNN_{XI} . CxDNN then uses a fast hardware-instance-independent re-training method to recover accuracy lost during the conversion process. Next, CxDNN maps the DNN_{XI} to a resistive crossbar system that suffers from crossbar non-idealities (denoted DNN_{XNI}), which results in further degradation in accuracy. CxDNN overcomes this accuracy drop using a hardware compensation scheme. Note that DNN_{XI} can also be obtained using end-to-end DNN training, i.e., used for designing FP32 DNNs. However, training is computationally very expensive, requiring exa-ops of compute and hence days to weeks to complete. In contrast, CxDNN designs DNN_{XI} by converting FP32 models and utilizing a fast re-training method to recover accuracy loss during conversion with very few iterations.² We next describe CxDNN’s software and hardware methods in detail.

4.2 Conversion Algorithm

CxDNN uses a two-step conversion algorithm to convert a floating-point DNN (DNN_{FP}) to a DNN for ideal crossbar (DNN_{XI}). Algorithm 1 details the pseudo code for the conversion algorithm. It takes as inputs DNN_{FP} , precision constraints for weights and activations (pW, pA), conductance (g_{max}, g_{min}) and voltage (v_{max}, v_{min}) ranges, and a validation dataset ($ValData$), and produces DNN_{XI} as its output. First, CxDNN quantizes DNN_{FP} to get an optimized fixed-point DNN (DNN_{FXP}) (line 2). Subsequently, it converts DNN_{FXP} to DNN_{XI} by transforming weights to conductances (lines 3–10). To avoid hardware overhead incurred by variable precision, we assume the same precision for all layers. However, Algorithm 1 can be easily extended to support variable precision across DNN layers and data-structures (weights and activations).

To convert DNN_{FP} to DNN_{FXP} (line 2), several algorithms have been proposed in recent years [16, 17, 25, 30, 37]. We adopt the quantization method proposed in Reference [16], wherein we explore the conversion design space (trade-off between range and resolution) to identify the

²Training ResNet-50 on the ImageNet dataset requires ~365 K iterations with a batch size of 50. In comparison, our re-training requires <500 iterations.

Fig. 6. FxP to ideal crossbar conversion function.

ALGORITHM 1: DNN Conversion for an ideal crossbar

Input: DNN_{FP} : floating-point DNN, $[pW, pA]$: weights and activation precision, ValData: Validation Dataset, $[g_{max}, g_{min}]$: Conductance range, $[v_{max}, v_{min}]$: voltage range

Output: DNN_{XI} : DNN converted for ideal crossbar

```

1: Begin
2:  $DNN_{FXP} = \text{convertToFxp}(DNN_{FP}, \text{ValData}, pW, pA)$  //  $DNN_{FXP}$ : Fixed-Point DNN
3:  $[\alpha_g, \beta_g] = \text{get-Conversion-Params}(g_{max}, g_{min}, pW)$ 
4:  $DNN_{XI} = DNN_{FXP}$ 
5: for Each layer  $k$ : 1 to  $N$  do
6:    $DNN_{XI} = \text{convertToConductances}(DNN_{XI}, \alpha_g, \beta_g, k)$ 
7:    $[WD_k, AD_k] = \text{get\_magnitude\_distribution}(DNN_{XI}, \text{ValData}, k)$ 
8:    $[SF_{in}, SF_{out}] = \text{getOptScalingFactor}(DNN_{XI}, \text{ValData}, k, v_{max}, v_{min}, pA, WD_k, AD_k)$ 
9:    $DNN_{XI} = \text{applyScalingFactor}(DNN_{XI}, SF_{in}, SF_{out}, k)$ 
10: end for
11: Return  $DNN_{XI}$ 

```

best FxP format for each DNN layer and data-structure. Next, we convert DNN_{FXP} to DNN_{XI} using the weight-to-conductance conversion parameters (α_g, β_g) and scaling factors (SF_{in}, SF_{out}) that are used for converting digital inputs to voltages and currents to digital outputs, respectively. Figure 6 depicts our DNN_{FXP} to DNN_{XI} conversion process. We develop a statistics-driven approach to identify the best scaling factors (SF_{in}, SF_{out}) for each DNN layer. For each layer in the network, we derive weight and activation distributions (line 7) by simulating DNN_{XI} with the validation dataset and determine a list of possible scaling factors. We systematically explore these choices and identify the scaling factors that yield the highest accuracy (line 8). We convert each layer separately starting with the first DNN layer, and once all layers are converted DNN_{XI} is returned (line 11).

4.3 Re-training Method

The conversion of DNN_{FXP} to DNN_{XI} is not perfect and leads to accuracy loss due to two major factors: (i) non-linear ADC and DAC functions introduce distortions in the computed outputs and (ii) the representation of numerical “0” with a non-zero (g_{min}) conductance for weights and a non-zero voltage (v_{min}) for inputs. For example, multiplications with zero values produce “0” in FxP but yield a non-zero value in ideal crossbar-based implementation. Further, the magnitude of the non-zero value is not constant, rather data-dependent. Therefore, the output (D_{out}) computed using ideal crossbar differs from the output (F_{out}) calculated using FxP. Figure 7 shows the impact

Fig. 7. Activation distortion during conversion.

of the incurred data distortion during conversion by highlighting the differences in the distribution of the layer outputs for 6-bit FxP (DNN_{FxP6}) and ideal crossbar (DNN_{XI-6}) implementations of the VGG-16 network. We observe that for the same primary input (shown on the left), the histograms of the output activations get distorted for DNN_{XI-6} design in comparison to DNN_{FxP6} . Moreover, the distortions propagate through DNN layers and increase as we go deeper into the network, eventually impacting DNN accuracy. For example, the output distribution of CONV1-2 layer shows higher distortion than the CONV1-1 layer outputs.

To recover the accuracy lost during conversion, CxDNN uses a hardware-instance-independent re-training method. DNN training consists of three major steps—forward propagation, backward propagation, and gradient computation followed by weight update. CxDNN uses a software model of the resistive crossbar during the forward propagation step, reflecting the fact that we are modeling inference on a resistive crossbar-based hardware fabric, and a floating point (FP32) model for the backward propagation, gradient computation, and weight update steps.

Figure 8 presents the CxDNN re-training process that uses a conductance (G) weight matrix during the forward pass and FP32 weight matrices during the backward pass. The DAC and ADC models apply the scaling factors and convert digital inputs (D_{in}) to input voltages (V_{in}) and crossbar output currents (I_{out}) to digital output activations (D_{out}). We synchronize both weight matrices after each mini-batch iteration by converting the updated FP32 weights to conductances. Figure 8 illustrates the main steps in the forward and backward passes (steps 1–4 and steps 5–8, respectively). First, we convert inputs (D_{in}) and weights to voltages (V_{in}) and conductances (G) (Step 1–2). Next, we perform the crossbar-based vector-matrix multiplication to obtain output currents (I_{out}) (Step 3). The ADC model converts the output currents to digital outputs (D_{out}) (Step 4). We perform Steps 1–4 for each convolution and fully connected DNN layer. On completion of the forward pass, the loss and error functions are computed using the final layer outputs and a label (Step 5). Next, we execute backward pass, gradient computation, and weight update functions for each DNN layer. Figure 8 illustrates these operations for the K th DNN layer, wherein the gradients (ΔW) (Step 6) and output errors ($Error_K$) (Step 8) are computed using the errors from layer $K+1$ ($Error_{K+1}$). Next, the computed error ($Error_K$) propagates in the backward direction to the previous layer ($K-1$), and the FP32 DNN model is updated using the computed gradient (ΔW) (Step 7).

Fig. 8. Re-training method for DNNs on crossbar system.

We re-train DNN_{XI} by repeating Steps 1–8 for a small set of images to compensate for the accuracy drop during conversion.

4.4 Hardware Compensation

As mentioned in Section 4.1, DNNs executed on crossbar-based systems will suffer from accuracy degradation due to data-dependent and hardware-instance-specific non-idealities. CxDNN employs a hardware compensation method to overcome this degradation. Our hardware compensation is motivated by a key observation that errors in vector-matrix multiplications realized using resistive crossbars result from the cumulative effect of all non-idealities. It is not possible to isolate the effect of individual non-idealities on the executed vector-matrix multiplication, as all non-idealities kick in simultaneously (as described in Section 3.1). Therefore, we propose error compensation at the crossbar-level, where the outputs of the realized vector-matrix multiplications are compensated using Compensation Factors (CFs). CxDNN uses a separate compensation factor for each crossbar column of each crossbar instance, which allows many degrees of freedom in tuning the compensation process. The use of per-column compensation factors is motivated by the error characteristics in Figure 3(c) that show a significant variance across both crossbar instances and columns within an instance.

Next, we determine the type of compensation factor that should be used to mitigate errors due to crossbar non-idealities based upon insights from the error characteristics at the crossbar-level. To that end, we compute errors at each crossbar column by considering all crossbar non-idealities and plot the error histogram. We examine two error metrics—relative error (RE) and absolute error (AE). Figure 9 shows the histogram of RE and AE normalized with their respective mean (μ) across these input vectors. We observe that the absolute errors vary substantially across different input vectors, as they have large variance ($\sigma/\mu = 8\%$) with a spread of 0.78μ to 1.3μ . In contrast, the relative errors show significantly lower spread and have very small variance ($\sigma/\mu = 0.5\%$). Based on this observation, we can conclude that additive compensation factors where a constant offset is added to correct the errors at crossbar columns will not work, as the absolute errors show significant input dependence. Further, it also suggests that we can effectively compensate errors by selecting multiplicative compensation factors (CFs) computed using the mean relative error (RE_{mean}). For example, we can compute the compensation factor (CF_i) for the i th crossbar column using RE_{mean} as shown in Equation (1), and correct errors using CF_i as a multiplicative factor

Fig. 9. Histogram of absolute and relative errors at crossbar columns.

(shown in Equation (2)).

$$CF_i = 1/(1 - RE_{mean}), \quad (1)$$

$$RE_{mean} = \frac{\sum_{j=1}^k \frac{|Expected_{val-j} - Actual_{val-j}|}{|Expected_{val-j}|}}{k}$$

$$Compensated_{val-i} = Actual_{val-i} * CF_i. \quad (2)$$

CxDNN's hardware compensation method consists of two phases—a calibration phase to determine CFs and a runtime phase to mitigate errors using these CFs. The host processor initiates the calibration phase after the DNN_{XI} is mapped and programmed to the resistive crossbar system. The calibration phase consists of three main steps: In step 1, the host sends predetermined inputs (Inp) to each RCA in the resistive crossbar system. In step 2, RCAs perform vector-matrix multiplications using these calibration inputs (Inp) and stored weights to obtain the actual (erroneous) output vectors (Out-act) and subsequently send them to the host. In step 3, the host processor computes compensation factors using the actual outputs (Out-act) and the ideal outputs (Out-idl). The host processor could either store or generate the calibration inputs (Inp) and expected outputs (Out-idl). We generate the calibration inputs using a characterization dataset. We note that our method does not restrict the nature or size of the characterization dataset—it can be as large as required, subject to the characterization time being acceptable. In our experiments, we found that a small subset of the validation set was sufficient to compute compensation factors that led to the accuracy improvements reported in the article. Figure 10 depicts the CF computation process for an RCA using actual and ideal outputs (Out-idl). As shown in the figure, we compute a vector of CFs, one for each column. The computed CFs are then sent back to the respective crossbars to be stored locally in the compensation logic. While all RCAs execute vector-matrix multiplications in parallel, the host processor computes the compensation factor for each RCA sequentially. Therefore, the overall calibration time is dominated by the host processor. Further, the calibration time depends on the total number of RCAs utilized by the DNN. In our evaluation, the overall calibration time was 0.2 s to 1.3 s across our benchmark DNNs for a single core Intel-Atom C2350 as our host processor. We invoke the calibration phase whenever a new DNN is mapped to the XPU system and subsequently execute millions of inference operations during the runtime phase. We can also periodically invoke the calibration phase to cope with aging issues and drift in synaptic conductances. Aging and synaptic drifts are very slow phenomena, hence the calibration phase will only need to be performed rarely (e.g., once every 10 K–100 K inference operations). The execution

Fig. 10. Hardware compensation: Calibration phase.

Fig. 11. Compensation logic.

time required for calibration is negligible when amortized over 10 K–100 K inference operations. We also note that the computed CFs are not exact but approximate factors that compensate the errors effectively with limited hardware overhead, as discussed next.

Figure 11 shows an RCA with the compensation logic, which consists of registers for storing CFs, a column mux, and a multiplier. Using the compensation logic, we mitigate errors in vector-matrix multiplications at runtime to obtain compensated outputs given by $\text{Out-comp} = \text{Out-act} * \text{CF}$. Further, we hide the latency of the compensation logic using a two-stage pipeline, where we pipeline crossbar evaluation and error compensation operations. We note that the slowest stage, which determines the pipeline throughput, is still the crossbar evaluation stage [29]. In our evaluation,

Table 1. Modeling details for crossbar non-idealities

Type	Non-idealities modeled
Device-level non-idealities	Process variations (Gaussian), device-level noises (Gaussian), imperfect writes (Gaussian)
Circuit-level non-idealities	Wire resistances ($r_{row} = 1\Omega$, $r_{col} = 4.6\Omega$), driver resistances ($1.5\text{ K}\Omega$), sensing resistances ($500\ \Omega$), sneak path (data-dependent), non-linear ADCs and DACs (lookup tables)

the latency of the crossbar and the compensation logic are $\sim 60\text{ ns}$ and $\sim 0.78\text{ ns}$, respectively. We require $\sim 49\text{ ns}$ to compensate 64 (all) columns of an RCA using the compensation logic. Thus, the compensation logic adds no throughput penalty. Further, we observe the compensation logic area and power overheads to be 7.3% and 4%, respectively, at the RCA-level; and 5.5% and 3.5%, respectively, at the system-level. We consider these overheads to be reasonable, considering the accuracy improvements they enable. The major contributor to the overhead are the additional registers required for storing the CFs.

We note that CxDNN offers multiple design choices that differ based on which non-idealities are compensated in software and which are compensated in hardware. In our implementation, we compensated for limited precision and conversion of weights to conductances and activation values to voltages/currents in software (through re-training), leaving the remaining non-idealities to be compensated in hardware. Alternative design choices are possible where additional non-idealities (e.g., sneak paths) are included during (re-)training to obtain a compensated DNN model. However, as motivated in Section 4.1, we cannot model instance-specific errors (e.g., chip-to-chip variations) during re-training, as it ties the trained network to the specific hardware instance. This breaks the train-once-deploy-anywhere tenet of current DNN applications. Therefore, the overhead of hardware compensation to mitigate instance-specific errors must be incurred irrespective of the degree of non-idealities considered during re-training.

5 EXPERIMENTAL METHODOLOGY

In this section, we present the experimental framework used for evaluating CxDNN.

Accuracy evaluation. To evaluate the application-level accuracy of DNNs on resistive crossbar systems, we use a modified version of the Caffe deep learning framework [15]. The Caffe framework is enhanced with a crossbar model, ADC/DAC models, and the CxDNN design methodology. It takes a resistive crossbar system description as an input and maps convolution (Conv) and fully connected (FC) DNN layers to this system. Like Caffe, it models Conv and FC layers as matrix-matrix and vector-matrix multiplications. It functionally evaluates DNNs by mimicking the vector-matrix multiplications realized in resistive crossbars to obtain application-level accuracy. We consider several non-idealities, *viz.*, wire resistances, driver and sensing resistances, non-ideal DACs and ADCs, sneak paths, imperfect write operations, device-level noise (e.g., thermal noise) and process variations in our crossbar model. We provide the modeling details in Table 1. We use a Gaussian distribution in synaptic conductance to model non-idealities in synaptic devices. Table 2 details the benchmark DNNs that we use to evaluate CxDNN. To demonstrate the scalability of CxDNN, we focus on large-scale DNNs, *viz.*, AlexNet, Network-in-Network (NiN), OverFeat, VGG-16, GoogleNet, and ResNet-50.

Device, circuit, and system modeling. Table 3 details the synaptic device [28], circuit parameters, interconnect technology [24], and variation technology parameters [31] used in our experiments. It also provides the RCA, the host processor, and the compensation logic configurations.

Table 2. Benchmark Applications

Data-Set	Network	# (Conv, FC) Layers	#Synapses (in billions)	#Neurons (in millions)	Relative Model Size
MNIST	LeNet	(2,2)	0.0005	0.02	1
CIFAR-10	ConvNet	(3,2)	0.01	0.05	20
ImageNet	AlexNet	(5,3)	0.5	0.5	1000
	NiN	(12,0)	1.1	1.7	2200
	OverFeat	(5,3)	2.6	1.9	5200
	VGG-16	(13,3)	15.5	13.6	31000
	GoogleNet	(59,5)	1.6	3.2	3200
	ResNet-50	(53,1)	5.1	13.8	10200

Table 3. Device, Technology, Circuit, and System Parameters

Components	Values
Synaptic Device [28]	Precision = 6 bits, $R_{min} = 200 \text{ K}\Omega$, $R_{max} = 1.4 \text{ M}\Omega$ Length = 1,280 nm, Width = 40 nm, Tox = 2.12 nm
Wire Technology [24]	Resistance = $3.3 \Omega/\mu\text{m}$, Capacitance = $0.2 \text{ fF}/\mu\text{m}$
Variation	Synaptic conductance (σ/μ) = 5% to 15%
CMOS Technology	45 nm Bulk CMOS
Resistive Crossbar Array (RCA)	64 DACs (6 bits each), 16 shared ADCs (10 bits each), 64 $\times$ 64 synaptic elements, 64 input registers (6 bits), 64 output registers (10 bits), 1 Adder (16 bits)
Compensation logic	64 registers (6 bits), 1 multiplier (10 bit)
SFU	32 ReLUs, 4 vPE with 4 lanes, 8 SPEs, 16 QUs
SFU throughput	1,066 GOPs/s for ReLU, 35 GOPs/s for normalization, 57 GOPs/s for pooling, 5.3 GOPs for special functions, 45 GOPs/s for Quantization
Extracted Parasitic	$r_{row} = 1\Omega$, $c_{row} = 0.05 \text{ fF}$, $r_{col} = 4.6\Omega$, $c_{col} = 0.3 \text{ fF}$
Host processor	Intel-Atom C2350 @1.7 GHz (1 core)

We limit the synaptic device precision to 6 bits, considering the maximum precision that can be achieved by practical synaptic devices [13]. We utilize 64×64 crossbar arrays in our evaluations, as they show a good balance between accuracy and efficiency. Prior efforts [34] that study the trade-off between area, power, and accuracy based on crossbar size also report similar observations. We use an in-house device model of the synaptic element [28] that is based on the solution of Landau-Lifshitz-Gilbert (LLG) magnetization dynamics and Non-Equilibrium-Green's Function (NEGF) electron transport. We perform the device simulation to abstract the circuit-level parameters such as conductance range (R_{min} to R_{max}) and the number of conductive states (Precision) for a nominal device (without any non-idealities). Next, we characterize the impact of process variations in physical parameters (e.g., synaptic device dimensions, Tox) and intrinsic device-level noise (e.g., thermal noise) that results in variations in the synaptic conductance. Subsequently, we model all device-level non-idealities using a Gaussian distribution in the synaptic conductance (the abstracted circuit-level parameter). We consider the variation in the synaptic conductance to be $\sigma/\mu = 5\%$ to 15% based on variation data reported in Reference [31]. We show the efficacy of CxDNN on this range of variation values [$(\sigma/\mu) = 5\%$ to 15%]. We perform layouts of the synaptic device and

the crossbar array (using design rules from Reference [27]) to extract interconnect parasitics (detailed in Table 3). Our simulations utilize the ADC and DAC circuits proposed in References [20, 36]. We evaluate RCA energy consumption and latency using SPICE simulations. Finally, we synthesize the digital logic using Synopsys Design Compiler and a commercial 45 nm technology to estimate their energy, delay, and area at the gate-level. We assume a spatial architecture with sufficient RCAs to map the largest DNN (i.e., VGG-16). Operations other than vector-matrix multiplications (max-pooling, normalization, etc.) are performed on the Special Function Unit (SFU). The SFU consists of 32 ReLU units, 4 vector processing elements (vPEs) each with 4 lanes, 8 special function processing elements (SPEs), and 16 Quantization units (QUs). SPEs compute special functions such as Sigmoid, and vPEs compute normalization and pooling operations. SFUs occupy 1.9% of the total XPU area and consume 1.6% of the total XPU power. The SFU throughput for supported operations is detailed in Table 3.

6 RESULTS

We now present results to show the effectiveness of CxDNN in enabling high-accuracy large-scale DNNs on resistive crossbar-based systems.

6.1 Application-level Compensation

We first show CxDNN's compensation ability at the application-level. To this end, we compare DNNs designed using three different compensation techniques. We use (i) a baseline DNN obtained using only the CxDNN conversion algorithm, (ii) a software compensated DNN obtained using the CxDNN conversion algorithm and re-training mechanism, and (iii) a software and hardware compensated DNN obtained using both hardware and software compensation techniques of CxDNN. Figure 12 shows the classification accuracy of these DNNs realized on two different resistive crossbar systems—an "ideal" system with 6-bit precision but no device and circuit non-idealities (Cross-ideal6) and a non-ideal crossbar system (Cross-NI6) designed using 64×64 resistive crossbar arrays. In Figure 12, FxP6 is a software 6-bit fixed-point implementation. Across all benchmark applications, we observe that DNNs compensated using both software and hardware techniques are considerably superior to the other two versions (baseline and Software Compensated). The baseline DNN suffers distortion due to conversion, and both the baseline and the software compensated DNNs suffer from device and circuit-level non-idealities, leading to poor application-level accuracy on real resistive crossbar systems (Cross-NI6). Overall, CxDNN achieves 16.9%–49% improvement in accuracy relative to the baseline. We observe the effect of crossbar non-idealities to be more prominent on deeper networks with >50 layers (e.g., ResNet and GoogleNet), as the accuracy degradation for the software compensated design (with no compensation for crossbar non-idealities) on Cross-NI6 is substantial. In contrast, we find wider DNNs (e.g., AlexNet and VGG-16) with fewer (9 to 16) layers to show more resilience to crossbar noise, as the software compensation is effective in restoring application-level accuracy. We observed an interesting correlation between the resilience of DNNs to crossbar non-idealities and their resilience to other traditional optimizations, such as pruning [11, 12]. For example, efforts on DNN pruning [11, 12] show that AlexNet and VGG-16 networks can be pruned by 90%–95%, but parameters in ResNets can only be pruned by about 50%–60%. Correspondingly, we observe that AlexNet and VGG-16 networks are more resilient (robust) than ResNet-50 to crossbar non-idealities.

Next, we provide insights on how errors in vector-matrix multiplication at the crossbar-level interact over various crossbar operations and propagate through the network. To that end, we visualize the heat map of output activations obtained from various layers of the GoogleNet DNN as it processes an oxcart image from the ImageNet dataset. Figure 13 illustrates the heat map of the first layer, some middle layers, and the last layer of the GoogleNet DNN implemented on an

Fig. 12. Application-level accuracy benefits.

Fig. 13. Visual demonstration of HW error compensation.

ideal crossbar system (Cross-ideal) and a non-ideal crossbar system (Cross-NI) with and without hardware compensation. We observe that the initial layers of the uncompensated CrossNI and the Cross-ideal cases are very similar. However, the later layers are noticeably different. This is because the errors accumulate as we go deeper into the network, leading to significant and visible distortions in output features of the later layers. This observation also concurs well with the result

Fig. 14. HW compensation for instance specific errors.

Fig. 15. HW compensation with increasing process variations.

shown in Figure 12, where the impact of crossbar non-idealities on deeper networks is found to be more prominent. Further, we observe the output feature-maps of the Cross-ideal and Compensated Cross-NI cases to be very similar, thereby demonstrating the effectiveness of CxDNN in mitigating errors due to crossbar non-idealities.

6.2 Compensating Hardware-instance-specific Errors

We now focus on hardware-instance-specific errors due to process variations and evaluate CxDNN's efficacy in mitigating them. We assume random variations ($\sigma/\mu = 5\%$) in the synaptic elements of a resistive crossbar system as the source of non-ideality. Figure 14 shows the accuracy of an ideal crossbar system (Cross-Ideal) and a non-ideal crossbar system (Cross-NI) with and without hardware compensation. We observe that hardware compensation recovers the drop in DNN accuracy due to process variation very effectively. On average, we observe an accuracy improvement of 25.83% over the Cross-NI Uncompensated case. Moreover, Hardware compensation achieves accuracy within 1.8%–2.8% of the Cross-ideal implementation.

We also study CxDNN's effectiveness with increasing process variations on the GoogleNet and NiN networks. Figure 15 shows the accuracy of the compensated and the uncompensated Cross-NI designs for various degrees of random variations (σ/μ) in the synaptic devices. We observe that the HW compensation effectively mitigates errors in resistive crossbar systems with high process variations ($\sigma/\mu = 15\%$) and achieves considerable accuracy improvement over the uncompensated design irrespective of the degree of variations. As mentioned in Section 4.4, CxDNN's HW compensation is a low-overhead approximate compensation scheme that does not correct errors exactly. Figure 15 suggests that the CxDNN can achieve accuracy close to Cross-ideal case for σ/μ up to 10%, with some degradation for larger values.

Fig. 16. Efficiency of the re-training method.

Fig. 17. Hardware compensation in action.

6.3 Re-training Efficiency

Next, we demonstrate the effectiveness of the proposed hardware-instance-independent re-training mechanism. Figure 16 shows the accuracy improvement with re-training iterations for various DNNs. As shown, we achieve an improvement of 7%–27.4% for a minimal amount of re-training (150 iterations). In summary, our re-training method is able to bridge the accuracy gap between the fixed-point and the ideal crossbar (Cross-Ideal) DNN implementations effectively.

6.4 Hardware Compensation at the Array-level

We now show the effectiveness of the hardware compensation method in mitigating errors at the crossbar-array level. Figure 17 shows the observed errors for the Uncompensated and Compensated RCA instances with respect to ideal crossbar across 1 K input vectors. We plot the average error at each column of a 64×64 crossbar. We also show the error spread (maximum and minimum error) for each crossbar column across the 1 K vectors. We use four different crossbar instances with 0%, 5%, 10%, and 15% random variations in synaptic conductances. As evident, the observed errors for each crossbar column in the compensated design are drastically smaller than the uncompensated design, underscoring the effectiveness of CxDNN's hardware compensation scheme in mitigating errors at the crossbar-array level.

Fig. 18. Comparison of CxDNN and ECC scheme-based compensation [8].

Fig. 19. Area and power breakdown of a crossbar array with compensation logic.

6.5 Comparison with ECC Scheme

Next, we compare the proposed hardware compensation with the AN code-based ECC compensation scheme [8] at the crossbar-level. Figure 18 shows the computed relative errors across crossbar columns of an RCA instance for three designs—(i) an uncompensated design, (ii) the CxDNN compensated design, and (iii) the ECC compensated design. In the plots, the markers (orange square, green diamond, and blue circle) represent the mean relative error for each crossbar column across 1 K input vectors, and the error bars show the spread. We implemented the AN code-based ECC scheme with $N = 19$, which is equivalent to a 5-bit Hamming code and can correct up to 19 syndromes. We note that for 6-bit weights and inputs, the overhead due to the ECC scheme requiring 5 additional bits will be substantial. However, we found the $N = 19$ code to provide the best result, and therefore use it for the comparison. For instance, $N > 19$ and $N < 19$ both had higher mean error due to overcompensation and inferior compensation, respectively. In Figure 18, we observe that the ECC scheme reduces the mean error for each crossbar column. However, it cannot detect and correct a significant number of errors and wrongly compensates a few. In contrast, CxDNN is quite effective in compensating errors at the crossbar-level. Further, as mentioned in Section 2, AN codes [1] are for correcting errors in linear functions realized using digital systems, where the errors are discrete (at bit-level) and can be quantified using the digital number representation (multiple bits). However, the functions realized in resistive crossbars are analog and non-linear due to several non-linear functions (e.g., ADCs and DACs). Therefore, ECC schemes cannot effectively mitigate errors in analog systems.

6.6 Area and Power Breakdown

Finally, we discuss the area and the power breakdown of various components in a Resistive Crossbar Array (RCA) including the compensation logic. Figure 19 shows the major components, viz., ADC, DAC, input and output registers, and compensation logic (Comp-Logic). The Comp-Logic includes registers for storing CFs, a mux, and a multiplier. ADCs and DACs dominate the area and power consumption in the RCA. The fraction of total area and power consumed by them are 79% and 90%, respectively. The area and power overhead due to the Comp-Logic with respect to

RCA are about 7.3% and 4%, respectively. However, the system-level area and power overhead are lower, i.e., 3.5% and 5.5%, respectively, as the resistive crossbar processing unit (XPU) consists of other non-RCA logic units such as input and output buffers, controllers, and special function units (SFUs). These non-RCA logic units occupy 24% of the total XPU area and consume 13% of the total XPU power. The major contributors to the overhead are the additional registers required for storing compensation factors (CFs). Therefore, more efficient methods for storing CF can effectively reduce the overall overhead. For example, the registers for storing CFs can be replaced by denser memories like SRAM, and CFs can also be provided as inputs along with the activations and the partial sums. The overhead can also be improved by sharing CFs across crossbar columns. In this article, we did not optimize the overheads further, as we believe them to be reasonable, considering the benefits of CxDNN and its ability in mitigating a key challenge to the use of resistive crossbar-based neural fabrics.

7 CONCLUSION

Resistive crossbar systems are a promising solution to the energy-efficient realization of DNNs. However, they suffer from numerous device and circuit-level non-idealities leading to degradation in application-level accuracy of large-scale DNNs. In this work, we present CxDNN, a hardware-software methodology composed of a conversion algorithm, a hardware-instance-independent re-training mechanism, and a low-overhead hardware compensation scheme to design high-accuracy DNNs on resistive crossbars. CxDNN preserves the train-once-deploy-anywhere tenet of current DNN implementations, wherein DNNs are re-trained once to obtain a common model that can be deployed on many resistive crossbar systems (XPUs). Further, CxDNN utilizes low-overhead hardware to mitigate instance-specific errors. Our evaluations reveal that CxDNN can largely restore the accuracy loss due to non-idealities by achieving 16.9%–49% improvements over a range of large-scale DNN benchmarks. We also find CxDNN to be very effective in compensating hardware-instance-specific errors, as it can achieve accuracy within 1.8%–2.8% of the crossbar implementation with no non-idealities. The system-level area and power overheads of the compensation hardware are minimal, i.e., 5.5% and 3.5%, respectively. Finally, we conclude that CxDNN presents a promising approach to compensate for errors due to crossbar non-idealities and addresses a key bottleneck to the adoption of resistive crossbar-based neural fabrics.

REFERENCES

- [1] A. Avizienis. 1971. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. *IEEE Trans. Comput.* C-20, 11 (Nov. 1971), 1322–1331. DOI: <https://doi.org/10.1109/T-C.1971.223134>
- [2] I. Chakraborty, D. Roy, and K. Roy. 2018. Technology aware training in memristive neuromorphic systems for non-ideal synaptic crossbars. *IEEE Trans. Emerg. Topics. Comput. Intell.* 2, 5 (Oct. 2018), 335–344. DOI: <https://doi.org/10.1109/tetci.2018.2829919>
- [3] L. Chen, J. Li, Y. Chen, Q. Deng, J. Shen, X. Liang, and L. Jiang. 2017. Accelerator-friendly neural-network training: Learning variations and defects in RRAM crossbar. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'17)*. 19–24. DOI: <https://doi.org/10.23919/DATE.2017.7926952>
- [4] P. Chen, X. Peng, and S. Yu. 2018. NeuroSim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning. *IEEE Trans. Comput.-Aided Des. Integ. Circ. Syst.* (2018), 1–1. DOI: <https://doi.org/10.1109/TCAD.2018.2789723>
- [5] Pai-Yu Chen, Binbin Lin, I-Ting Wang, Tuo-Hung Hou, Jieping Ye, Sarma Vrudhula, Jae-sun Seo, Yu Cao, and Shimeng Yu. 2015. Mitigating effects of non-ideal synaptic device characteristics for on-chip learning. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'15)*. IEEE Press, Piscataway, NJ, 194–199. Retrieved from: <http://dl.acm.org/citation.cfm?id=2840819.2840848>.
- [6] Ming Cheng, Lixue Xia, Zhenhua Zhu, Yi Cai, Yuan Xie, Yu Wang, and Huazhong Yang. 2017. TIME: A training-in-memory architecture for memristor-based deep neural networks. In *Proceedings of the 54th Design Automation Conference (DAC'17)*. ACM, New York, NY, Article 26, 6 pages. DOI: <https://doi.org/10.1145/3061639.3062326>

- [7] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. 2016. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *Proceedings of the ACM/IEEE 43rd International Symposium on Computer Architecture (ISCA'16)*. 27–39. DOI : <https://doi.org/10.1109/ISCA.2016.13>
- [8] B. Feinberg, S. Wang, and E. Ipek. 2018. Making memristive neural network accelerators reliable. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. 52–65. DOI : <https://doi.org/10.1109/HPCA.2018.00015>
- [9] T. Gokmen, O. Murat Onen, and W. Haensch. 2017. Training deep convolutional neural networks with resistive cross-point devices. *Front. Neurosci.* 11 (2017), 538. DOI : <https://doi.org/10.3389/fnins.2017.00538>
- [10] T. Gokmen and Y. Vlasov. 2016. Acceleration of deep neural network training with resistive cross-point devices: Design considerations. *Front. Neurosci.* 10 (2016), 333. DOI : <https://doi.org/10.3389/fnins.2016.00333>
- [11] Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization, and Huffman coding. Retrieved from: *Arxiv Preprint Arxiv:1510.00149* (2015).
- [12] Yihui He and Song Han. 2018. ADC: Automated deep compression and acceleration with reinforcement learning. *CoRR abs/1802.03494* (2018). Retrieved from: <http://arxiv.org/abs/1802.03494>.
- [13] Miao Hu, Catherine Graves, Can Li, Yunning Li, Ning Ge, Eric Montgomery, Noraica Davila, Hao Jiang, R. Stanley Williams, J. Joshua Yang, Qiangfei Xia, and John Paul Strachan. 2018. Memristor-based analog computation and neural network classification with a dot product engine. *Advanced Materials* 30, 9 (2018), 1705914.
- [14] M. Hu, J. P. Strachan, Z. Li, E. M. Grafals, N. Davila, C. Graves, S. Lam, N. Ge, J. J. Yang, and R. S. Williams. 2016. Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication. In *Proceedings of the 53rd ACM/EDAC/IEEE Design Automation Conference (DAC'16)*. 1–6. DOI : <https://doi.org/10.1145/2897937.2898010>
- [15] Shubham Jain, Abhronil Sengupta, Kaushik Roy, and Anand Raghunathan. 2018. Rx-Caffe: Framework for evaluating and training deep neural networks on resistive crossbars. *CoRR abs/1809.00072* (2018). Retrieved from: <http://arxiv.org/abs/1809.00072>.
- [16] Shubham Jain, Swagath Venkataramani, Vijayalakshmi Srinivasan, Jungwook Choi, Pierce Chuang, and Leland Chang. 2018. Compensated-DNN: Energy efficient low-precision deep neural networks by compensating quantization errors. In *Proceedings of the 55th Design Automation Conference (DAC'18)*. ACM, New York, NY, Article 38, 6 pages. DOI : <https://doi.org/10.1145/3195970.3196012>
- [17] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, K. Gopalakrishnan, and L. Chang. 2019. BiScaled-DNN: Quantizing long-tailed datastructures with two scale factors for deep neural networks. In *Proceedings of the 56th ACM/IEEE Design Automation Conference (DAC'19)*. 1–6.
- [18] Sung Hyun Jo, Ting Chang, Idongesit Ebong, Bhavitavya B. Bhadviya, Pinaki Mazumder, and Wei Lu. 2010. Nanoscale memristor device as synapse in neuromorphic systems. *Nano Lett.* 10, 4 (2010), 1297–1301. DOI : <https://doi.org/10.1021/nl904092h> PMID: 20192230.
- [19] Irina Kataeva, Farnood Merrikh-Bayat, Elham Zamanidoost, and Dmitri Strukov. 2015. Efficient training algorithms for neural networks based on memristive crossbar circuits. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN'15)*. IEEE, 1–8.
- [20] L. Kull, T. Toifl, M. Schmatz, P. A. Francese, C. Menolfi, M. Braendli, M. Kossel, T. Morf, T. M. Andersen, and Y. Leblebici. 2013. A 3.1mW 8b 1.2GS/s single-channel asynchronous SAR ADC with alternate comparators for enhanced speed in 32nm digital SOI CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference Digest of Technical Papers*. 468–469. DOI : <https://doi.org/10.1109/ISSCC.2013.6487818>
- [21] Beiyue Liu, M. Hu, Hai Li, Zhi-Hong Mao, Yiran Chen, Tingwen Huang, and Wei Zhang. 2013. Digital-assisted noise-eliminating training for memristor crossbar-based analog neuromorphic computing engine. In *Proceedings of the 50th ACM/EDAC/IEEE Design Automation Conference (DAC'13)*. 1–6.
- [22] B. Liu, H. Li, Y. Chen, X. Li, T. Huang, Q. Wu, and M. Barnell. 2014. Reduction and IR-drop compensations techniques for reliable neuromorphic computing systems. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'14)*. 63–70. DOI : <https://doi.org/10.1109/ICCAD.2014.7001330>
- [23] C. Liu, M. Hu, J. P. Strachan, and H. Li. 2017. Rescuing memristor-based neuromorphic design with high defects. In *Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference (DAC'17)*. 1–6. DOI : <https://doi.org/10.1145/3061639.3062310>
- [24] Peter Moon, Vinay Chikarmane, Kevin Fischer, Rohit Grover, Tarek A. Ibrahim, Doug Ingerly, Kevin J. Lee, Chris Litteken, Tony Mule, and Sarah Williams. 2008. Process and electrical results for the on-die interconnect stack for Intel's 45nm process generation. *Intel Technol. J.* 12, 2 (2008).
- [25] E. Park, D. Kim, and S. Yoo. 2018. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *Proceedings of the ACM/IEEE 45th International Symposium on Computer Architecture (ISCA'18)*. 688–698. DOI : <https://doi.org/10.1109/ISCA.2018.00063>

- [26] R. Parloff. 2016. Why deep learning is suddenly changing your life. Fortune.com. 9/28/16. Retrieved from: <http://fortune.com/ai-artificial-intelligence-deep-machine-learning/>
- [27] Design Rules. [n.d.]. MOSIS scalable CMOS (SCMOS). Retrieved from: <https://www.mosis.com/files/scmos/scmos.pdf>.
- [28] Abhronil Sengupta, Yong Shim, and Kaushik Roy. 2016. Proposal for an all-spin artificial neural network: Emulating neural and synaptic functionalities through domain wall motion in ferromagnets. *IEEE Trans. Biomed. Circ. Syst.* 10, 6 (2016), 1152–1160.
- [29] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the ACM/IEEE 43rd International Symposium on Computer Architecture (ISCA'16)*. 14–26. DOI : <https://doi.org/10.1109/ISCA.2016.12>
- [30] Hokchhay Tann, Soheil Hashemi, R. Iris Bahar, and Sherief Reda. 2017. Hardware-software codesign of accurate, multiplier-free deep neural networks. In *Proceedings of the 54th Design Automation Conference (DAC'17)*. ACM, New York, NY, Article 28, 6 pages. DOI : <https://doi.org/10.1145/3061639.3062259>
- [31] A. F. Vincent, J. Larroque, N. Locatelli, N. Ben Romdhane, O. Bichler, C. Gamrat, W. S. Zhao, J. O. Klein, S. Galdin-Retailleau, and D. Querlioz. 2015. Spin-transfer torque magnetic memory as a stochastic memristive synapse for neuromorphic systems. *IEEE Trans. Biomed. Circ. Syst.* 9, 2 (Apr. 2015), 166–174. DOI : <https://doi.org/10.1109/TBCAS.2015.2414423>
- [32] Yandan Wang, Wei Wen, Beiye Liu, Donald M. Chiarulli, and Hai Helen Li. 2017. Group scissor: Scaling neuromorphic computing design to big neural networks. *CoRR* abs/1702.03443 (2017). Retrieved from: <http://arxiv.org/abs/1702.03443>.
- [33] H. S Philip Wong, Heng Yuan Lee, Shimeng Yu, Yu Sheng Chen, Yi Wu, Pang Shiu Chen, Byoungil Lee, Frederick T. Chen, and Ming Jinn Tsai. 2012. Metal-oxide RRAM. *Proc. IEEE* 100, 6 (6 2012), 1951–1970. DOI : <https://doi.org/10.1109/JPROC.2012.2190369>
- [34] L. Xia, B. Li, T. Tang, P. Gu, X. Yin, W. Huangfu, P. Y. Chen, S. Yu, Y. Cao, Y. Wang, Y. Xie, and H. Yang. 2016. MNSIM: Simulation platform for memristor-based neuromorphic computing system. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE'16)*. 469–474.
- [35] B. Yan, J. Yang, Q. Wu, Y. Chen, and H. Li. 2017. A closed-loop design to enhance weight stability of memristor based neural network chips. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'17)*. 541–548. DOI : <https://doi.org/10.1109/ICCAD.2017.8203824>
- [36] Jintao Zhang, Zhuo Wang, and Naveen Verma. June 2016. A machine-learning classifier implemented in a standard 6T SRAM array. In *Proceedings of the IEEE Symposium on VLSI Circuits (VLSI-Circuits'16)*. IEEE, 1–2.
- [37] Shuchang Zhou, Zekun Ni, Xinyu Zhou, He Wen, Yuxin Wu, and Yuheng Zou. 2016. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR* abs/1606.06160 (2016). Retrieved from: <http://arxiv.org/abs/1606.06160>.

Received August 2019; revised September 2019; accepted September 2019