Master's Thesis

Efficient Sneak Path-aware Training of Binarized Neural Networks for RRAM Crossbar Arrays

Sugil Lee

Department of Computer Science and Engineering

Graduate School of UNIST

2019

Efficient Sneak Path-aware Training of Binarized Neural Networks for RRAM Crossbar Arrays

Sugil Lee

Department of Computer Science and Engineering

Graduate School of UNIST

Efficient Sneak Path-aware Training of Binarized Neural Networks for RRAM Crossbar Arrays

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Sugil Lee

6. 21. 2019

Approved by

Advisor
Jongeun Lee

Efficient Sneak Path-aware Training of Binarized Neural Networks for RRAM Crossbar Arrays

Sugil Lee

This certifies that the thesis of Sugil Lee is approved.

6. 21. 2019

signature

Advisor: Jongeun Lee

signature

Myeongjae Jeon

signature

Kyung Rok Kim

Abstract

Although RRAM crossbar arrays have been suggested as an efficient way to implement MVM for DNNS, the sneak path problem of RRAM crossbar arrays due to wire resistance can distort the result of MVM quite significantly, resulting harsh performance degradation of the network. Therefore, a software solution that can predict the effect of sneak paths to mitigate the impact without permanent hardware cost or expensive SPICE simulations is very desirable. In this paper, a novel method to incorporate the sneak path problem during training with a negligible overhead is proposed. The test validation results, done through accurate SPICE simulations, show very high improvement in the performance close to the baseline BNNs on GPU, which demonstrates the efficiency of the proposed method to capture the sneak path problem.

Contents

List of Figures

List of Tables

Chapter I

Introduction

RRAM (Resistive RAM) crossbar arrays have been suggested as an efficient way to implement matrix-vector multiplication (MVM) [1], which is the main computation for deep neural networks (DNNs) and many others. To perform the MVM, the input matrix is programmed on the RRAM array cells as conductance values first, and the input vector is applied as voltage at rows. Then the current generated at each cell is proportional to the voltage and conductance, and summed along each column. Therefore, we can get MVM result vectors as the output currents at columns.

In the real devices, however, the nonidealities such as write disturbance, read disturbance, device variabilities, inherent noise and others make it difficult to get right results. [2]. Especially, the sneak path problem caused by wire resistance and surrounding cells' resistance states can harshly distort the result of MVM computation. The voltage drop from unwanted resistance will change current contribution of each cell and the output currents. Though the sneak path problem for memory applications, activating one row at a time, is a solved issue, the simultaneous activation of parallel rows as for MVM computation still suffers from it.

We show that the sneak path problem can degrade quality of simple BNN (binary neural network) inference even at moderate wire resistance values if the array size is large, as in Fig. 1.1 for 128×128 RRAM crossbar arrays (RCAs) (see Section 6.1 for experimental setup), where the accuracy starts to decline only with $R_w = 0.1\ \Omega$. Moreover, according to the ITRS roadmap [3], wire resistance is expected to keep increasing because while wires getting shorter linearly, the cross-sectional areas of wires decrease quadratically.

In this paper we focus on the sneak path problem among the device nonidealities. We first check the impact of wire resistance on BNN inference using generic RRAM crossbar SPICE model [4]. Then, to recover the significant accuracy drop due to sneak paths on BNNs, we propose a novel training method based on the prediction of RRAM nonidealities and the compensation of them through weight adjustment during training (see Fig. 1.2), without any hardware overhead. Our experimental results based on accurate SPICE-level simulations demonstrate that our sneak path-aware training method can reclaim many crossbars with considerable sneak paths, enabling them to run complex BNNs with near software-level accuracy.

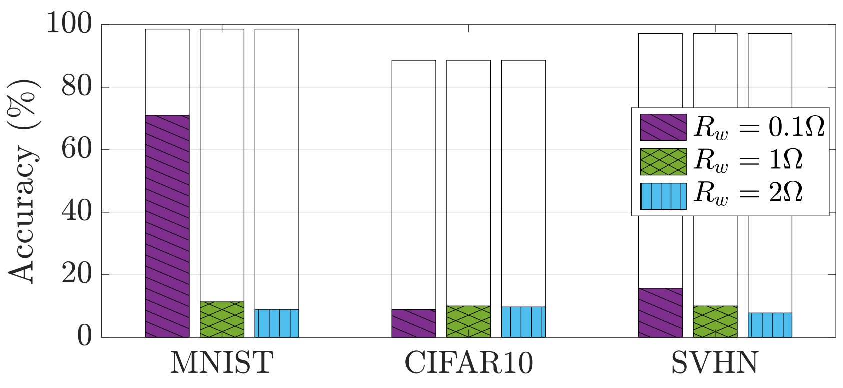


Figure 1.1: Test validation accuracy vs. wire resistance per cell (R_w) before retraining; white bars represent the baseline accuracy on GPU.

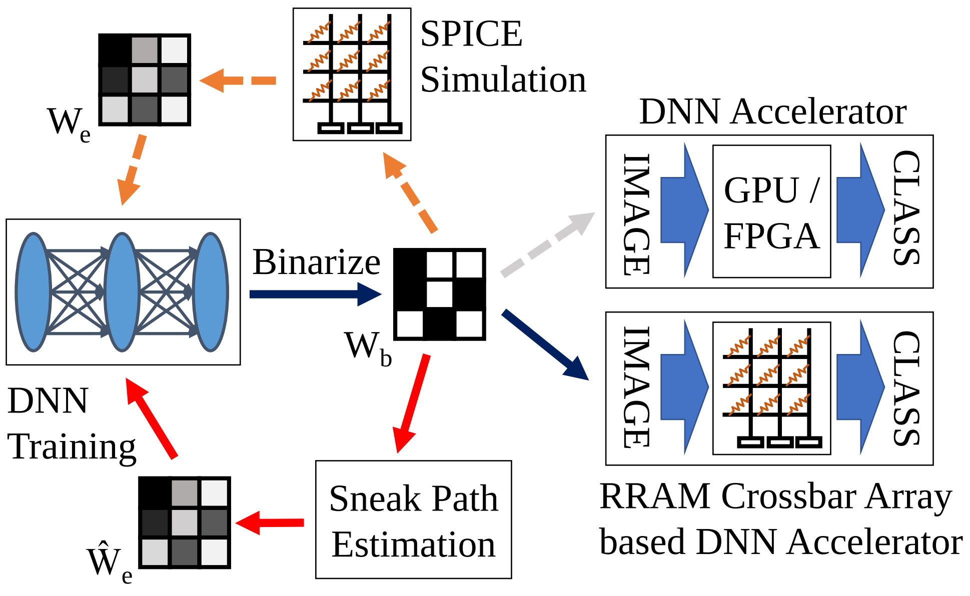


Figure 1.2: Our SPICE-based validation flow (in orange) and sneak path-aware training flow (in red).

Chapter II

Background and Related Work

2.1 RRAM Crossbar Array

Crossbar structures offer a fast MVM platform, especially for neural network acceleration, which reduces MVM's time complexity to $O(1)$ instead of $O(N^2)$ using conventional approaches. The multiplication operation is performed through sensing the current at the output ports, which can be written as

$$I_j = \sum_{i=1}^{m} G_{ij} V_i \tag{2.1}$$

where I_j is the neural current of the j^{th} neuron, G_{ij} the synaptic weight in conductance, and V_i the i^{th} input voltage. Afterward the neural current passes through activation circuit. Usually two memory devices are needed to realize each weight to have bipolar values, but different methods have been proposed to realize weights such as RRAM (or memristor), floating-gate transistor [5], and capacitor [6]. Memristors have a good potential to be used for MVM operations due to better retention, endurance, and stackability. While continuous memristors (RRAMs) show good potential for full-precision neural networks, current device fabrication techniques are not mature enough to support fine-tuning of a device's resistance, endurance, and variability [7]. On the other hand, binary devices that switch between two levels offer high retention and endurance performance [8].

2.2 Binary Weight Realization

In BNNs, weights are quantized into two levels, $\{-1, +1\}$, while activation function output can be continuous or binarized depending on the network. In order to store and represent negative weights in RRAMs, typically two devices are needed. Alternatively, one can use a shared reference RRAM with the conductance of $G_r = (G_{\max} + G_{\min})/2 \approx G_{\max}/2$ for high switching ratio as shown in Fig. 2.1 [9], which is referred to as unbalanced realization. The figure show the example for n wordlines and m bitlines. A variable weight is set to either $G_{\max}$ or $G_{\min}$. Hence, the realized weight values are $G_{\max}/2$ and $-G_{\max}/2$ for 1 and -1, respectively. This realization requires smaller crossbar arrays (i.e., less area and less power dissipation) compared with two-device realization.

2.3 DNN on RRAM Array

Recently, different experiments on RCA-based accelerator have been introduced utilizing the analogue behavior of RRAMs to solve classification and regression problems [10, 11, 12]. These experimental examples are performed for networks that require small arrays to perform MVM. The training of these experiments is performed through *in-situ* learning where RCAs are used for the inference, and the gradients and the weight updates are calculated on software. This approach would be useful for small

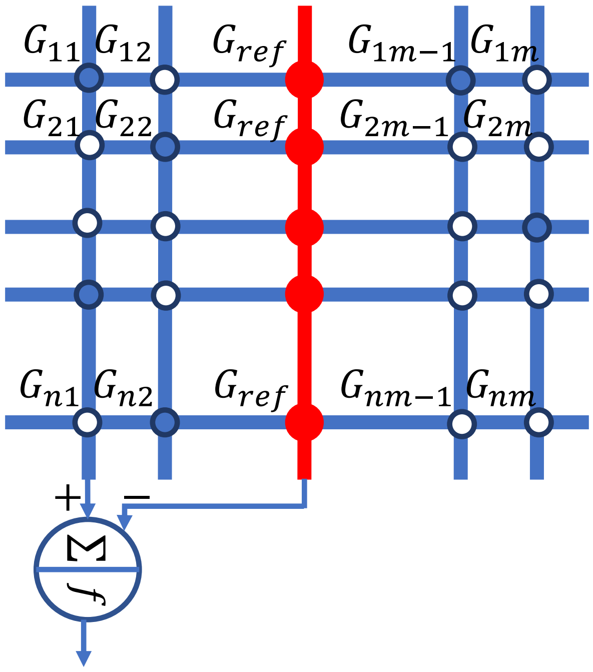
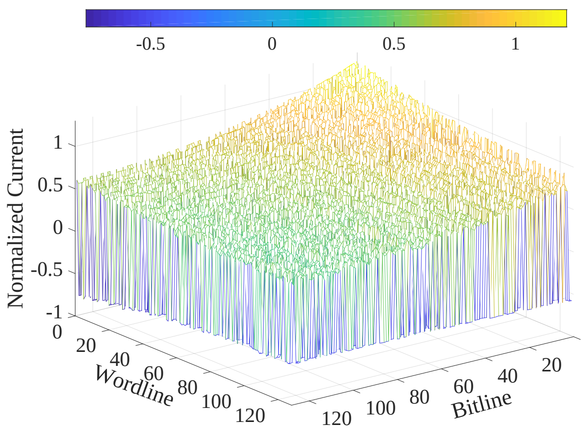


Figure 2.1: Unbalanced synaptic weight realization using RCAs.

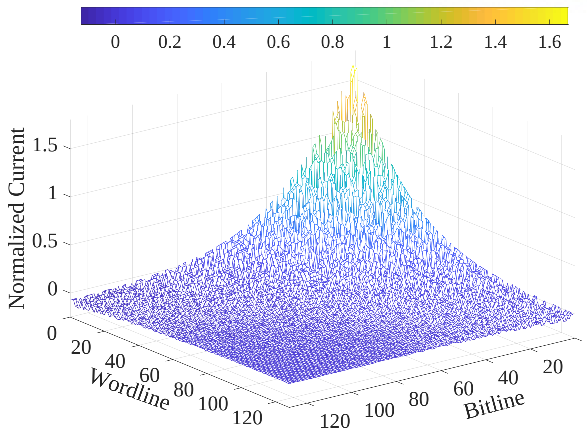
networks where the number of trainable parameters is small. However, in case of large and dense networks, the *in-situ* learning would be impossible. Thus, several works have proposed to train the network while taking the nonidealities of the RCA into consideration to ensure correct functionality after weight transfer [13, 14].

BNNs are considered to be a promising way that can avoid nonidealities that exist in analogue memristors such as asymmetric non-linearity, variability, and low retention. Hence several works have shown through simulations and some hardware implementations that RRAM crossbar-based neural networks can be trained with high accuracy—typically with less than 1%p drop in accuracy for networks designed for MNIST and CIFAR10. In [15, 16] the authors used a 1T1R crossbar structure and proposed *sequential* BNNs where the current sensing circuit (neuron) is shared between all the neurons so only one column is activated at a time. In [17], a binary RRAM-accelerated CNN was designed and optimized to feature a massive parallelism with high energy efficiency. In [18], multi-bit binary conventional neural network was introduced using selector-less crossbar arrays with pipeline implementation, and included the device variations as well. In [19], the authors presented a full hardware implementation for robust RRAM-based convolutional blocks using single-ended XNOR sensing capable of performing dot product operations in a single cycle, for computer vision and image processing applications. Although those work provide good implementation techniques of BNNs using RRAMs, none of them consider wire resistance which is inevitable in the crossbar arrays and would highly degrade the results especially for both 0T1R arrays and 1T1R with *parallel* computation.

Using 1T1R-based RCA architectures such as in [16] helps avoid the sneak path problem, thanks to the sequential operation. However, as shown in Section 6.5, it requires far more processing time because the time complexity of MVM becomes $O(N)$ instead of $O(1)$.



(a) $R_w = 0.1\ \Omega$



(b) $R_w = 2\ \Omega$

Figure 3.1: Normalized sensed current for 128×128 crossbar array with one reference column.

Chapter III

Problem Description

3.1 Sneak Path Problem

Ideally, the absolute value of the sensed current per a RRAM cell should be constant regardless of the position of the cell in the array. But due to the existence of wire resistance, which is inevitable in any interconnect, the sensed current can vary depending on the location. The variation in the sensed current is due to the voltage drops over the wire resistances that create leakage paths throughout the array which is referred as the sneak path problem.

To illustrate the effect of sneak paths, Fig. 3.1 plots the sensed current from each cell while simulating random binary weights. The sensed current is the *perceived* weight value as seen at the output port. The graphs clearly show that the sensed current decays exponentially across the diagonal direction of an array instead of having constant values in $\{-1, 1\}$. Also, with increasing wire resistance, the weights decay much faster.

One way to mitigate this problem is to use a selector device/material in series with a switching device. However, selector has voltage-dependent nonlinearity (e.g., exponential or quadratic), which disturbs MVM computation (i.e., $I = G \cdot \sinh(V)$) in addition to increasing the area of a cell. This nonlinearity must be taken into consideration during training, which would require re-designing the neural network frameworks. Our proposed technique can be used in conjunction with the selector approach, but does not require it.

3.2 DNN Application

The problem can be stated as follows: Given a BNN, a train dataset, and the physical parameters of RCAs such as LRS, HRS, wire resistance, array size, etc., to find the best synaptic weights for the BNN to be programmed into RCAs such that the accuracy of inference on RCAs can be maximized despite the existence of sneak path currents.

The BNN part of the problem can be solved easily using the back-propagation algorithm, with a proper consideration of sneak path effects. The new challenge is how to quickly and accurately estimate the effect of sneak path currents during training so that we can guide the back-propagation algorithm to minimize or even compensate for the sneak paths. Note that sneak path patterns can vary widely and nonlinearly depending on the programmed binary weights, requiring re-evaluation of sneak paths for every weight update. Also, since the number of RCAs in a BNN can be quite large, a naïve integration of SPICE simulation during training would require a prohibitive amount of resources.

Now, if we limit ourselves to the steady-state behavior, a passive RCA can be seen as a resistive network with constant, albeit programmable, resistance values. Then we can use the result of [4] on a generic resistive network model, which is shown to be as accurate as SPICE simulation. Importantly, this model implies that despite the presence of sneak path currents, a resistive network's output should be linear to input. Therefore, we can express the observed output current $\mathbf{y}$ as the product of input voltage $\mathbf{x}$ and some matrix $\mathbf{W}_e$:

$$\mathbf{y} = f(\mathbf{W}_b, \mathbf{x}) = g(\mathbf{W}_b)\mathbf{x} = \mathbf{W}_e\mathbf{x} \tag{3.1}$$

Here $\mathbf{W}_b$ is the *programmed weight matrix* and $\mathbf{W}_e$ the *effective weight matrix*. They have the same size as RCAs, which is $n \times m$. For BNN applications, both input $\mathbf{x}$ and programmed weight matrix $\mathbf{W}_b$ are binary, whereas effective weight matrix $\mathbf{W}_e$ is real-valued.

Now the goal is to find $\mathbf{W}_e$ from $\mathbf{W}_b$. One may use SPICE simulation or equivalent numerical models such as [4], both of which are extremely slow. Inference, fortunately, does not require many SPICE simulations; one simulation per RCA is enough due to (3.1). This allows us to build an accurate and practical evaluation setup for sneak path-aware inference on RCAs. For training, however, we need a very fast model, and preferably in a closed-form expression so that it can be directly integrated into an existing DNN training framework. Also, the model must be differentiable for back-propagation to work. This suggests that we can use regression methods to identify the function g, treating $\mathbf{W}_b$ and $\mathbf{W}_e$ as the input and output of the system to be identified.

In the next section we present the solution to the regression problem. Note that though in this paper we obtain $\mathbf{W}_e$ from SPICE or a SPICE-equivalent model, it could also be derived from real device measurement data, to which our technique should be applicable as well.

Chapter IV

Prediction of Effective Weights

In this section we present different models to estimate effective weight matrix, including one that was recently proposed.

4.1 Mask Method

Recently the Mask method was proposed [20], which is to use a single matrix $\mathbf{M}$, called *mask*, of the same size as the RCA to compensate for the difference between $\mathbf{W}_b$ and $\mathbf{W}_e$ as

$$\mathbf{W}_e = \mathbf{W}_b \circ \mathbf{M} \tag{4.1}$$

where $\circ$ is the Hadamard product (i.e., element-wise multiplication). The intuition behind this method is that the physical location within an RCA has a dominant effect on the deviation of $\mathbf{W}_e$ from $\mathbf{W}_b$ as seen in Fig. 3.1, which is captured by a mask.

It works on a small dataset (MNIST) with some easy settings. It is also very fast and can be easily integrated into any training framework. For more challenging datasets and settings, however, we find that it does not give satisfactory accuracy. This is because Mask captures only the first-order effect, but ignores the effect of the other cells on the same crossbar, which may be crucial in estimating the sneak path effect.

4.2 Row-wise Linear Regression

While linear and polynomial regressions are popular, there is one problem to apply them to our problem: our input and output data are very high-dimensional. For a 128×128 RCA, input $\mathbf{W}_b$ (or output $\mathbf{W}_e$) has 16K dimensions, requiring 256 million parameters for linear regression.

Instead, we perform linear regression on each row, meaning that each row is treated as an independent data sample. Specifically,

$$\mathbf{W}_e^{[i]} = \mathbf{W}_b^{[i]} \cdot \mathbf{R} + \mathbf{b} \tag{4.2}$$

where $\mathbf{W}_b^{[i]}$ and $\mathbf{W}_e^{[i]}$ are the i^{th} row vector in the programmed and effective weight matrices, respectively. In addition, $\mathbf{R}$ and $\mathbf{b}$ are an $m \times m$ matrix and an m-dim bias vector, respectively, which are determined through regression.

Note that here all rows share the same regression parameters, which ignores the effect of physical location and may lead to low regression accuracy. Moreover, the sneak path effect as seen in Fig. 3.1 is clearly nonlinear, which may motivate the use of a nonlinear model.

4.3 Parallel Linear Network

We now present Parallel Linear Network (PLN). Despite its name, it is a nonlinear model, built by stacking multiple neural network layers. There are many factors suggesting the use of neural networks. Not only can they model nonlinear systems, they allow stacking, which is very useful for learning high-level features. Sneak paths exist in both row and column directions, thus combining row-wise and column-wise models makes sense. We first define two layers, and combine them to create PLN. Like in row-wise linear regression, we divide the input and output matrices into rows, but apply fully-connected layers (also known as linear layers) between each pair of input/output rows. In what follows, we use X and Y to refer to input and output matrices:

$$\mathbf{PL_{row}} : \mathbf{X} \rightarrow \mathbf{Y} \text{ where } \mathbf{Y}^{[i]} = f(\mathbf{X}^{[i]} \cdot \mathbf{R}_i + \mathbf{b}_i) \tag{4.3}$$

where $\mathbf{X}^{[i]}$ is the i-th row of $\mathbf{X}$ and f a nonlinear activation function (we use *tanh*), $\mathbf{R}_i$ a weight matrix of size $m \times m$, and $\mathbf{b}_i$ an m-dim bias vector. Unlike row-wise linear regression, each row has its own set of parameters, which can ensure better regression quality. In neural network frameworks, this layer can be easily implemented as a hierarchy of primitive layers: a parallel layer and linear layers below it (hence called *parallel linear layer*).

Column-wise parallel linear layer is defined similarly:

$$\mathbf{PL_{col}}(\mathbf{X}) = \mathbf{PL_{row}}(\mathbf{X}^T)^T \tag{4.4}$$

Then we define a *parallel linear network* to capture both row- and column-wise dependency:

$$\text{PLN}: \quad \mathbf{W}_e = \mathbf{PL_{col}}(\mathbf{PL_{row}}(\mathbf{W}_b)) \circ \mathbf{U} \tag{4.5}$$

where $\mathbf{U}$ is an $n \times m$ parameter matrix for an element-wise multiplication layer, which is needed because the range of $\mathbf{W}_e$ can go beyond $[-1, 1]$, as seen in Fig. 3.1.

One concern on this method is that each linear layer has its independent parameters and PLN has many of it in parallel, resulting more memory requirement during training. Though it does not matter on inference since the prediction models will not be included in the BNN validations once training is over.

4.4 Convolutional Layers

Another possible approach for considering the effect of other cells on the same crossbar is that using convolutional layers, specialized in capturing spatial information on 2D data. Here we feed each 128x128 array as a single-channel input through convolutional layers. We use fixed filter size as 3 and the number of hidden channels is same through layers in a network for simpler design space exploration. Not to change the feature map size, padding and stride is set to 1 and no pooling layer is included.

Now we define an n stack of convolutional layers given c hidden channels with activation function f, $\mathbf{CL_{n,c}}$, as:

$$\begin{aligned} \mathbf{CL_{n,c}}: \quad & \mathbf{X} \rightarrow \mathbf{Y} \text{ where } \mathbf{X_1} = \mathbf{X}, \\ & \mathbf{X_{k+1}} = f(Conv(\mathbf{X_k})) \text{ for } 1 \leq k \leq n-1, \mathbf{Y} = Conv(\mathbf{X_n}) \end{aligned} \tag{4.6}$$

Here only the last layer's output channel size is 1 and all others are c. We apply *ReLU* activation function on each layer's output but the last one's, not to lost negative-valued information.

We expect that a stack of convolutional layers can capture the sneak path effect from the adjacent cells, but still cannot deal with the effect of physical locations as linear regression and may show low performance.

4.5 Scaled Convolutional Network

To manage the locational information, simple yet powerful method is to scale the data according to the location, using element-wise multiplication layers. It has similar insight as Mask method, but the difference is that it is trainable through backpropagation along with other layers. We add 2 element-wise multiplication layers, one before the convolutional layers to scale input data, and the other one after the convolutional layers to adjust output range, as follows:

$$\text{SCN} : \mathbf{CL_{n,c}}(\mathbf{X} \circ \mathbf{U_1}) \circ \mathbf{U_2} \tag{4.7}$$

The first element-wise multiplication layer $\mathbf{U_1}$ scales the input according to the physical location to hide the locational impact, so that convolutional layers in the middle can solely recognize the effect of adjacent cells. After that, the activation is scaled through the last element-wise multiplication layer $\mathbf{U_2}$, applying locational information again to the output. We denote it *scaled convolutional Network* (SCN).

To train the networks we use *mean squared error* (MSE) loss, defined as

$$L = \frac{1}{N} \|\hat{\mathbf{W}}_e - \mathbf{W}_e\|_2^2 \tag{4.8}$$

where $N = nm$ is the number of elements in $\mathbf{W}_e$, and $\hat{\mathbf{W}}_e$ is the estimated effective weight matrix. After exploring some combination of n and c, we apply $\mathbf{CL_{7,32}}$ for our SCN which gives the least loss considering the model size.

Chapter V

Sneak Path-aware Training Rule

5.1 HW-Accurate BNN Inference for Validation

We have built a validation flow (see Fig. 1.2) to accurately account for the effect of sneak paths during BNN inference. After training is finished, binarized weights are first exported, which consist of several binary weight matrices as each layer requires many RCAs to represent a large weight matrix. Then the set of binary weight matrices $\{\mathbf{W}_b\}$ is fed to the SPICE or an equivalent numerical model simulation. Simulation generates a set of real-valued weight matrices $\{\mathbf{W}_e\}$ as per (3.1). Finally, in the BNN framework we substitute $\mathbf{W}_e$ for $\mathbf{W}_b$ in the forward path. That is, $x^{(l+1)} = f(\mathbf{W}_b x^{(l)})$ is replaced with $x^{(l+1)} = f(\mathbf{W}_e x^{(l)})$, where $x^{(l)}$ is activation at layer l, and f an activation function.

5.2 Modification for Training

Unlike numerical simulation, regression models can be evaluated within a DNN framework itself, which allows fast training based on estimated sneak path effect (see Fig. 1.2). For training, we perform back-propagation using $\mathbf{W}_e$ instead of $\mathbf{W}_b$:

$$\delta_x^{(l)} = {\mathbf{W}_e}^T \cdot (\delta_x^{(l+1)} \circ f') \tag{5.1}$$

$$\delta_{\mathbf{W}_e}^{(l)} = (\delta_x^{(l+1)} \circ f') \cdot {x^{(l)}}^T \tag{5.2}$$

where $\delta_X = \frac{\partial L}{\partial X}$.

While (5.2) gives the weight gradient for $\mathbf{W}_e$, what we need is the weight gradient for $\mathbf{W}$, the real-valued weights before binarization. It is possible to derive $\delta_{\mathbf{W}}$ from $\delta_{\mathbf{W}_e}$ using $\mathbf{W}_e = g(\mathbf{W}_b) \approx g(\mathbf{W})$ through chain rule, but we found that just approximating $\delta_{\mathbf{W}} \approx \delta_{\mathbf{W}_e}$ is fine to train the network to simplify the implementation.

Chapter VI

Experiments

6.1 Experimental Setup

To evaluate our proposed technique we use published BNNs [21] for MNIST, CIFAR10, and SVHN datasets. Table 6.1 lists key parameters of BNNs and training. Our primary metric is test accuracy, or the accuracy on unseen data, using the validation setup as described in Section 5.1. The baseline accuracy is that on GPU, which is the highest accuracy we can expect. While we mostly follow [21] for training setting, we reduce the learning rate by 1/8 at the beginning of retraining. The crossbar array is 128×128, with the number of reference columns varied. The wire resistance per cell is varied from 0.1 Ω to 2 Ω. For device parameters, we consider Ta/HfO$_2$/Pd device that has linear switching behavior [22] with these parameters: $G_{\min} = 1\ \mu\mathrm{S}, G_{\max} = 1$ mS$, V_{set} = 1.1$ V and $V_{reset} = -1.3$ V. For the simulation of RCAs, we use the model published in [4], which has the same steady-state behavior as SPICE simulation.

To train the regression models, we use 60,000 randomly generated 128$\times$128 binary data as inputs and corresponding SPICE-equivalent simulation results as targets, except for the Mask method, for which we follow the procedure in [20] using 100 data samples only. We use Adam optimizer for DNN-based regression models.

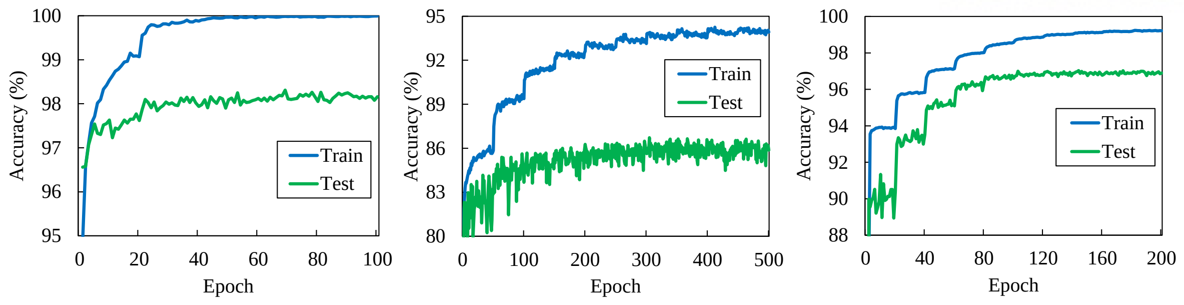
6.2 Comparison Among Regression Models

Table 6.2 compares the performance of regression models. For this experiment we use 1 Ω wire resistance and 1 reference column. MSE is as per (4.8), measuring the accuracy of predicting $\mathbf{W}_e$. Unsurprisingly, there is a clear difference in performance among the models. The (row-wise) linear regression model performs the worst, while our PLN and SCN perform the best, with an order of magnitude difference in MSE.

Using the trained models, we then train the MNIST BNN. We only retrain the BNN as opposed to training from scratch, as detailed in Section 6.1. At the end of the training we measure MNIST test accuracy, which is pre-validation accuracy, and RCA validation accuracy. In terms of the MNIST test accuracy, they all achieve near-baseline accuracy; the accuracy drop from the baseline model is

Table 6.1: BNNs and training parameters.

	MNIST	CIFAR10	SVHN
Network type	MLP	CNN	CNN
#Layers	4	9	9
Initial training #epochs	100	500	200
Retraining #epochs	50	200	80
Baseline test accuracy on GPU	98.41%	88.62%	97.18%



(a) MNIST (b) CIFAR10 (c) SVHN

Figure 6.1: Accuracy curves for PLN during retraining ($R_w = 1\ \Omega$, 9 Reference Columns).

Table 6.2: MSE and MNIST accuracy for different predictors.

	Mask	LR	PLN	SCN
MSE on random weights	9.28e-3	1.79e-2	1.32e-4	1.16e-4
Pre-validation accuracy	98.39%	96.88%	98.03%	98.25%
Validation accuracy	37.10%	44.86%	97.80%	97.71%

within 1%p$\sim$1.5%p. This accuracy is one that is reported by the BNN framework, calculated using the prediction models, and may not be representative of the real accuracy on real RCAs. Nonetheless, the high pre-validation accuracy indicates that the training itself was successful, somewhat validating our training rule modifications.

The ultimate measure of success, however, is validation accuracy, which is the accuracy obtained using the method explained in Section 5.1. Considering that validation accuracy before retraining (i.e., after initial training) was around only 10%, all methods did improve, but only PLN and SCN achieve near-baseline accuracy on validation test. That our models perform much better than the others is no coincidence, given they have much lower MSE than any other.

6.3 Results on Various RRAM Conditions

To see the effect of various RRAM conditions, we vary wire resistance (R_w) and the number of reference columns (#RefCols). Fig. 6.2 shows the validation accuracy for all four networks, where the white bars represent the baseline accuracy on GPU. Again, these results are not simply DNN training results, but obtained by exporting binary weights from the BNN framework, doing SPICE simulation on many RCAs, and feeding them back to the BNN framework to get very realistic inference accuracy.

First, we observe that the validation accuracy without retraining is universally low, at typically 10%. This is due to the discrepancy between the programmed RCA weights ($\mathbf{W}_b$) vs. the effective RCA weights ($\mathbf{W}_e$), which is ultimately caused by the existence of sneak paths. The only exception is the MNIST case at $R_w = 0.1$, which is because the MNIST task is much easier and the sneak path effect at $R_w = 0.1$ is relatively mild as evidenced by Fig. 3.1a.

Second, our proposed PLN and SCN methods perform consistently better than the Mask method, sometimes with a large margin. For MNIST, SCN achieves the baseline accuracy in all cases, and PLN does except for one condition ($R_w = 2$, $\#\text{RefCols} = 1$), which still is much better than Mask. CIFAR10 is the most challenging dataset, and though sometimes not as good as it could be, SCN achieves near-baseline performance in most cases, narrowing the gap with the baseline accuracy to as low as 4%p in one case. PLN also performs well in about half the cases and does better than SCN in one case ($R_w = 0.1$, $\#\text{RefCols} = 15$), achieving lower accuracy gap (3.5%p). For SVHN, PLN and SCN achieves near-baseline accuracy again in most cases, which is very impressive compared with the alternatives as Mask achieves in just 2 cases.

Table 6.3: CIFAR10 Pre-validation accuracy (%).

R_w	#RefCols	Mask	PLN	SCN
0.1	1	81.77	84.10	84.09
	9	84.04	83.75	84.85
	15	82.96	85.85	83.39
1	1	84.29	79.96	82.69
	9	83.79	85.42	85.27
	15	83.68	84.47	85.89
2	1	82.20	67.81	78.27
	9	82.18	79.06	84.19
	15	82.94	81.21	84.67

To find out the reason for lower performance in some cases, we made a closer examination of the CIFAR10 training result. Table 6.3 compares the pre-validation accuracy of Mask, PLN, and SCN. Interestingly, while Mask always achieves over 80% accuracy *before validation*, PLN's accuracy varies with the RRAM condition, reaching below 70% in one case. SCN also shows below 80% accuracy in one case. This result suggests that in the case of Mask, the prediction model is clearly the culprit. In the case of PLN and SCN, the training method could be improved, to increase either the result quality, convergence speed, or both. But another possibility is that the BNN model itself, so much damaged by the sneak path-ridden device imperfections, might be unable to learn the complex task. In the extreme case, a network wouldn't be able to learn well if half of the neurons/synapses were gone bad. A more analysis to pinpoint the exact cause of this quantitatively is left for future work.

Fig. 6.1 reports accuracy curves, showing the progress of train/test accuracy during retraining ($R_w = 1\,\Omega$, $\#\text{RefCols} = 9$). As can be seen in another figure (the *w/o retraining* accuracy in Fig. 6.2), the initial accuracy at the beginning of retraining is very low, at around 10% for all three curves, even though we start from the fully trained baseline models. However, our BNN training can quickly recover most of the accuracy within a few epochs. These graphs also confirm that the numbers of epochs we use for retraining, as listed in Table 6.1, are sufficient to reach convergence.

6.4 Time Overhead Comparison

The low training time overhead is a definite advantage of our proposed scheme, as shown in Fig. 6.3. The BNN framework is implemented in Torch7, and evaluation as well as training with Mask/PLN/SCN is done using GPU on a system with Intel Xeon CPU E5-2630 v4 and Nvidia GPU GeForce GTX 1080Ti. The total training time depends also on the number of iterations and the number of epochs.

To put this in perspective, replacing the regression models with a SPICE simulation would take exceedingly longer. Even a SPICE-equivalent numerical simulation [4] we use for validation, which is at least an order of magnitude faster than SPICE simulation, takes several minutes per iteration on 30 CPU cores. At that rate, retraining the MNIST BNN would take over 5,000 hours, or about 7 months while SCN requires only around 6 hours.

6.5 Comparison with 1T1R Scheme

Table 6.4 shows a comparison of the area and latency while processing one input image. The reported area is calculated for RCAs, drivers, sensing circuits, and registers (wire resistance is 1 Ω, RRAM feature size is $5nm$). 0T1R-based implementations have slightly less area compared to 1T1R-based implementations. It is worth mentioning that in the implementation based on 0T1R, the CMOS circuit area (e.g., sensing circuits, and registers) is approximately equal to the RCAs area, which means that a

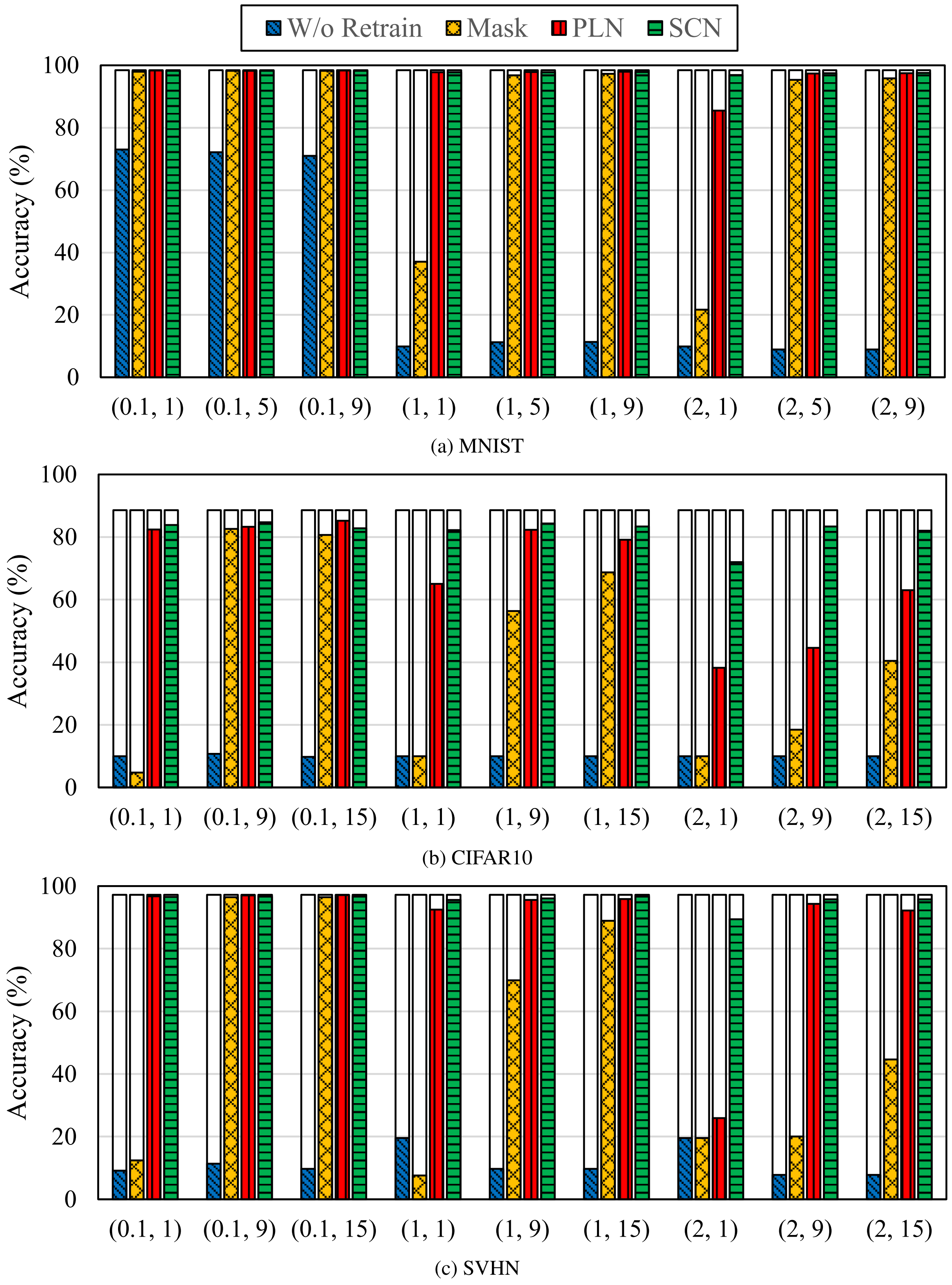


Figure 6.2: Test validation accuracy under different RRAM conditions.

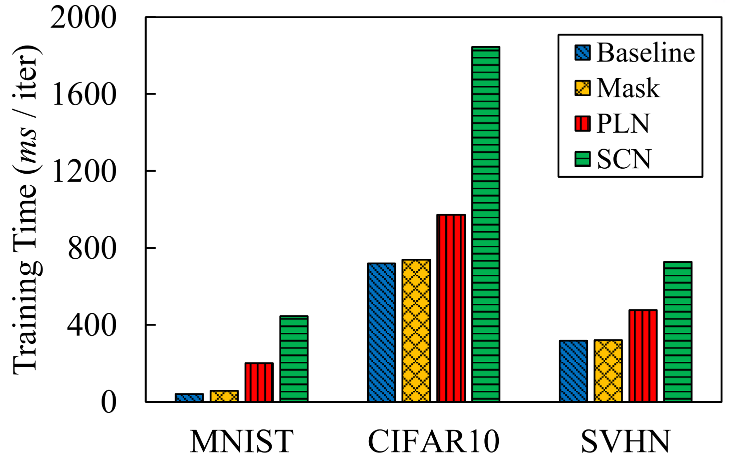


Figure 6.3: Training time comparison (per iteration).

Table 6.4: Area and latency comparison between 1T1R and 0T1R architectures for image classification; 1T1R is serial implementation, 0T1R full parallel implementation.

		MNIST	CIFAR10	SVHN
Area (μm^2)	1T1R	2.54e+3	1.47e+4	1.06e+4
	0T1R	2.41e+3	1.36e+4	1.03e+4
Latency (ns)	1T1R	9.23e+3	6.91e+5	5.44e+5
	0T1R	3.00e+0	4.04e+3	4.04e+3

further area saving of up to 50% is possible if CMOS circuit is placed under the RCAs, which is not possible in 1T1R. At the same time, 0T1R is 3000$\times$, 170$\times$ and 134.7$\times$ faster compared to 1T1R-based implementation for MNIST, CIFAR10 and SVHN networks, respectively, which is due to the serial processing of 1T1R scheme.

Chapter VII

Conclusion

In this paper we presented novel methods to incorporate the sneak path problem during BNN training with a negligible overhead. Compared to hardware methods (e.g., new device/selector material, error compensating circuitry), our training method is essentially free, and applicable on top of any hardware methods. Our experimental results demonstrate that while the sneak path problem renders many RRAM crossbar configurations unsuitable for DNN inference, our proposed methods can extend the range of usable configurations significantly, achieving near-baseline level test validation accuracy with MNIST and SVHN BNNs, and significant boost with CIFAR10 BNN.

We see many paths for future work. While our experiments in this paper are conducted assuming SPICE simulation as the ground truth, it could be extended to using real measurement data. Using two crossbar arrays instead of reference columns could be another way to improve accuracy, in exchange for more area and power dissipation. We can explore various neural network models for the prediction model, surely including fully-connected and convolutional neural networks, which would take a lot of exploration to find the best set of hyperparameters.

References

[1] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, “Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 14–26, 2016.

[2] S. Ambrogio, S. Balatti, A. Cubeta, A. Calderoni, N. Ramaswamy, and D. Ielmini, “Understanding switching variability and random telegraph noise in resistive ram,” in *2013 IEEE International Electron Devices Meeting*. IEEE, 2013, pp. 31–5.

[3] L. Wilson, “International technology roadmap for semiconductors (itrs),” *Semiconductor Industry Association*, 2013.

[4] M. Fouda, A. Eltawil, and F. Kurdahi, “Modeling and analysis of passive switching crossbar arrays,” *IEEE Trans. on Circuits and Systems I: Regular Papers*, vol. 65, no. 1, pp. 270–282, 2018.

[5] F. Merrikh-Bayat, X. Guo, M. Klachko, M. Prezioso, K. K. Likharev, and D. B. Strukov, “High-performance mixed-signal neurocomputing with nanoscale floating-gate memory cell arrays,” *IEEE transactions on neural networks and learning systems*, no. 99, pp. 1–9, 2017.

[6] Z. Wang, M. Rao, J.-W. Han, J. Zhang, P. Lin, Y. Li, C. Li, W. Song, S. Asapu, R. Midya *et al.*, “Capacitive neural network with neuro-transistors,” *Nature communications*, vol. 9, no. 1, p. 3208, 2018.

[7] M. Azzaz, E. Vianello, B. Sklenard, P. Blaise, A. Roule, C. Sabbione, S. Bernasconi, C. Charpin, C. Cagli, E. Jalaguier *et al.*, “Endurance/retention trade off in hfox and taox based rram,” in *2016 IEEE 8th International Memory Workshop (IMW)*. IEEE, 2016, pp. 1–4.

[8] Y. Yang, P. Sheridan, and W. Lu, “Complementary resistive switching in tantalum oxide-based resistive memory devices,” *Applied Physics Letters*, vol. 100, no. 20, p. 203112, 2012.

[9] C.-C. C. et al., “Mitigating asymmetric nonlinear weight update effects in hardware neural network based on analog resistive synapse,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2017.

[10] Z. Wang, C. Li, W. Song, M. Rao, D. Belkin, Y. Li, P. Yan, H. Jiang, P. Lin, M. Hu *et al.*, “Reinforcement learning with analogue memristor arrays,” *Nature Electronics*, p. 1, 2019.

[11] C. Du, F. Cai, M. A. Zidan, W. Ma, S. H. Lee, and W. D. Lu, “Reservoir computing using dynamic memristors for temporal information processing,” *Nature communications*, vol. 8, no. 1, p. 2204, 2017.

[12] M. Prezioso, M. Mahmoodi, F. M. Bayat, H. Nili, H. Kim, A. Vincent, and D. Strukov, “Spike-timing-dependent plasticity learning of coincidence detection with passively integrated memristive circuits,” *Nature communications*, vol. 9, no. 1, p. 5311, 2018.

[13] S. Yu, "Neuro-inspired computing with emerging nonvolatile memorys," *Proceedings of the IEEE*, vol. 106, no. 2, pp. 260–285, 2018.

[14] M. Fouda, E. Neftci, A. Eltawil, and F. Kurdahi, "Independent component analysis using rrams," *Nanotechnology, IEEE Transactions on*, 2018.

[15] S. Yu, Z. Li, P. Chen, H. Wu, B. Gao, D. Wang, W. Wu, and H. Qian, "Binary neural network with 16 mb rram macro chip for classification and online training," in *2016 IEEE International Electron Devices Meeting (IEDM)*, Dec 2016, pp. 16.2.1–16.2.4.

[16] X. Sun, X. Peng, P.-Y. Chen, R. Liu, J.-s. Seo, and S. Yu, "Fully parallel rram synaptic array for implementing binary neural network with (+ 1,- 1) weights and (+ 1, 0) neurons," in *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*. IEEE Press, 2018, pp. 574–579.

[17] L. Ni, Z. Liu, H. Yu, and R. V. Joshi, "An energy-efficient digital reram-crossbar-based cnn with bitwise parallelism," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 3, pp. 37–46, Dec 2017.

[18] T. Tang, L. Xia, B. Li, Y. Wang, and H. Yang, "Binary convolutional neural network on rram," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 2017, pp. 782–787.

[19] E. Giacomin, T. Greenberg-Toledo, S. Kvatinsky, and P.-E. Gaillardon, "A robust digital rram-based convolutional block for low-power image processing and learning applications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 2, pp. 643–654, 2019.

[20] M. E. Fouda, J. Lee, A. M. Eltawil, and F. Kurdahi, "Overcoming crossbar nonidealities in binary neural networks through learning," in *2018 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*. IEEE, 2018, pp. 1–3.

[21] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. Curran Associates Inc., 2016, pp. 4114–4122.

[22] M. Hu, C. E. Graves, C. Li, Y. Li, N. Ge, E. Montgomery, N. Davila, H. Jiang, R. S. Williams, J. J. Yang *et al.*, "Memristor-based analog computation and neural network classification with a dot product engine," *Advanced Materials*, vol. 30, no. 9, p. 1705914, 2018.