Trained Biased Number Representation for ReRAM-Based Neural Network Accelerators

WEIJIA WANG and BILL LIN, University of California

Recent works have demonstrated the promise of using resistive random access memory (ReRAM) to perform neural network computations in memory. In particular, ReRAM-based crossbar structures can perform matrix-vector multiplication directly in the analog domain, but the resolutions of ReRAM cells and digital/analog converters limit the precisions of inputs and weights that can be directly supported. Although convolutional neural networks (CNNs) can be trained with low-precision weights and activations, previous quantization approaches are either not amenable to ReRAM-based crossbar implementations or have poor accuracies when applied to deep CNNs on complex datasets. In this article, we propose a new CNN training and implementation approach that implements weights using a *trained biased number representation*, which can achieve near full-precision model accuracy with as little as 2-bit weights and 2-bit activations on the CIFAR datasets. The proposed approach is compatible with a ReRAM-based crossbar implementation. We also propose an *activation-side coalescing* technique that combines the steps of batch normalization, non-linear activation, and quantization into a single stage that simply performs a clipped-rounding operation. Experiments demonstrate that our approach outperforms previous low-precision number representations for VGG-11, VGG-13, and VGG-19 models on both the CIFAR-10 and CIFAR-100 datasets.

CCS Concepts: • **Computer systems organization → Architectures**; **Other architectures**; **Neural networks**;

Additional Key Words and Phrases: Resistive Memory, convolutional neural networks, quantization, machine learning, processing-in-memory

ACM Reference format:
Weijia Wang and Bill Lin. 2019. Trained Biased Number Representation for ReRAM-Based Neural Network Accelerators. *J. Emerg. Technol. Comput. Syst.* 15, 2, Article 15 (March 2019), 17 pages.
https://doi.org/10.1145/3304107

1 INTRODUCTION

Convolutional neural networks (CNNs) have achieved breakthrough performance on a variety of artificial intelligence applications, including image classification, video object tracking, natural language processing, two-player games, and autonomous-driving vehicles. However, to continue breakthrough performance on increasingly complex artificial intelligence problems, CNNs have steadily increased in complexity, with recent CNNs requiring more than 16 billion floating point operations for a single inference across a deep network with nearly 140 million parameters [13, 14].

Although conventional processor architectures provide plenty of processing power for training deep CNNs, they are often not well suited for deployment in mobile and wearable applications

Authors' addresses: W. Wang and B. Lin, Department of Electrical and Computer Engineering, University of California, San Diego, La Jolla, CA 92093-0407; emails: {wweijia, billlin}@eng.ucsd.edu.

1550-4832/2019/03-ART15 $15.00
https://doi.org/10.1145/3304107

where energy efficiency is paramount. In particular, conventional processor architectures typically require frequent data movements between the processor and off-chip memory, which consume enormous amounts of energy. Moreover, although not as significant as the energy cost for data movements, the tens of billions of full-precision floating point operations per inference are also often cost prohibitive in terms of energy consumption.

Recently, there has been considerable excitement surrounding the use of emerging non-volatile memory technologies for the implementation of neural network accelerators. In particular, recent efforts have demonstrated that metal-oxide resistive random access memory (ReRAM) [15] can be used to efficiently implement crossbar structures that provide both storage and computation capabilities. For neural network computations, ReRAM crossbars can be used to both store synaptic weights and perform matrix-vector multiplications directly in the analog domain [16–25]. A number of promising dataflow-like ReRAM-based neural network accelerator architectures (e.g., ISAAC [1], PRIME [2], and PipeLayer [3]) have been proposed that show a substantial advantage in energy efficiency over conventional processor architectures.

Although a ReRAM crossbar can directly perform matrix-vector multiplication, several critical challenges are presented to ReRAM-based neural network acceleration:

- The precision of weights that can be stored in the crossbar is limited by the resolution of the ReRAM cells, and the precision of inputs to the crossbar is limited by the resolutions of the digital-to-analog converters (DACs) and analog-to-digital converters (ADCs) that are used at the crossbar interface. In particular, practical implementations of ReRAM crossbars are limited to some m-bit weight precision and some p-bit input precision. For example, in ISAAC [1], the weight precision is $m = 2$ bits, and the input precision is just $p = 1$ bit. In PRIME [2], the weight precision is $m = 4$ bits, and the input precision is $p = 3$ bits. In PipeLayer [3], a spike-based scheme is used in which the inputs are provided as spikes, which eliminates the need for DACs. This effectively corresponds to an input precision of just $p = 1$ bit, whereas ADCs are replaced with integrate and fire units. PipeLayer supports a weight precision of $m = 4$ bits. Higher-precision inputs can be achieved by evaluating the ReRAM crossbar multiple times with successive p-bit inputs. For example, both ISAAC [1] and PipeLayer [3] support 16-bit inputs by evaluating the ReRAM crossbar 1 bit at a time successively 16 times. However, this way of achieving higher-precision inputs increases the processing time. To increase the precision of weights, a group of multiple crossbars can be used, where the j^{th} column of each crossbar in the group logically implements a portion of the weights for the same kernel. However, both means of increasing input and weight precisions cost proportional increases in energy consumption. Moreover, the use of multiple crossbars to increase weight precision limits the size of CNNs that can be implemented as weights are persistent in ReRAM-based neural network implementations.
- Ideally, we would like to use native ReRAM cell precisions of $m = 2$ to 4 bits. However, as observed in Song et al. [3], the accuracies of ReRAM-based neural network accelerators are sensitive to weight precisions. For deep CNNs on complex datasets, accuracies drop sharply when weight and activation precisions are decreased to low bit widths. In particular, the full-precision VGG-19 network achieves about 6.7% test error on the CIFAR-10 dataset, but this error dramatically increases to 90% if we simply "truncate" it into 4-bit weights and activations. Alternatively, CNNs can be trained with low-precision weights and activations [4–10, 27, 28]. In particular, in Chi et al. [2], a low-precision number representation called *dynamic fixed point* (DFP) [7, 8] is used for ReRAM-based accelerators, in which an m-bit number is viewed as a 2's complement number that is scaled by a power-of-2 fractional scaling factor M: $\{-M \cdot 2^{m-1}, \ldots, 0, \ldots, M \cdot (2^{m-1} - 1)\}$. This means that the representation

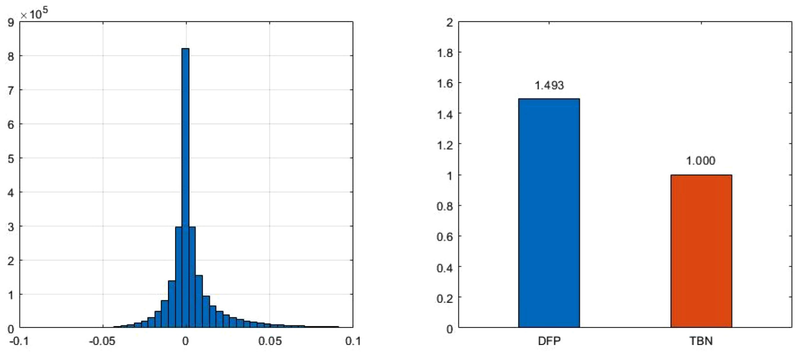

Fig. 1. The distribution of full-precision weights on the Conv5-2 layer (left) and the comparison of the normalized mean square error between our trained biased number (TBN) representation and DFP representation for $m = 2$ bits (right).

is symmetric in the range of positive and negative numbers. However, when examining actual weights that these number representations are suppose to approximate, we find that the set of weights on a given CNN layer is often not symmetric. Consider the VGG-11 CNN architecture [14] trained on the CIFAR-10 dataset. The distribution of weights on the Conv5-2 layer, which contains about 2.36 million weights, spans the range $[-0.075, 0.128]$, with the positive range 71% larger than the negative range. This leads to poor approximations when using low-precision numbers in DFP to represent the weights.

- ReRAM crossbar cells can only represent "positive" conductance values. However, neural networks generally require both positive and negative weights. To implement both positive and negative weights in ReRAM-based neural network accelerators, previous approaches [2, 3] have implemented kernels with positive and negative weights as two separate crossbar arrays. This "sign-splitting" approach significantly increases the hardware cost.

In this article, we propose to use a *trained biased number* (TBN) *representation* to approximate low-precision weights. In particular, we view an m-bit number as an unsigned integer that is scaled by a fractional scaling factor M and offset by a *biasing* term K. Each m-bit integer therefore represents a number from the set $\{0 - K, M - K, \ldots, M \cdot (2^m - 1) - K\}$, where the range of positive and negative numbers can be arbitrarily shifted by the biasing term K, and the step size M can be any fractional scaling factor. The parameters M and K can be independently trained on a *per-layer* basis to best approximate the distribution of weights on a given layer.

To illustrate the benefits of our proposed TBN representation, let us consider again the weights on the Conv5-2 layer of VGG-11 that have been trained on the CIFAR-10 dataset. For a precision of $m = 2$ bits, we computed the optimal parameters for these weights for DFP and our proposed TBN representation, and we computed the mean square error of each representation relative to full-precision weights. Figure 1 shows the normalized mean square errors, which shows that DFP has 49.3% higher mean square errors vs. our proposed TBN representation.

The main contributions of this article are as follows:

- We propose a new low-precision quantization approach for CNNs based on a novel TBN representation of weights. Our representation can represent both positive and negative numbers for ReRAM-based implementations without the need for separate crossbar arrays. Moreover, the trained biasing term in our approach enables our representation to approximate well sets of weights that have asymmetric ranges of positive and negative numbers.

- Our number representation is well suited to the inherent matrix-vector multiplication capabilities of ReRAM-based crossbar structures. In particular, our low-precision quantization approach can match the resolution limitations of digital/analog converters and memory cells in ReRAM-based crossbars.
- To take full advantage of ReRAM-based analog computational capabilities, the amount of computations that must be performed in the digital domain should be minimized. To this end, we propose a novel activation-side coalescing approach that coalesces the steps of batch normalization (BN), non-linear activation, and quantization into a single stage that simply performs a clipped-rounding operation.
- We explore the configuration space of different combinations of weight precisions and activation precisions by training different versions of the popular VGG deep CNN architecture [14] on the CIFAR datasets. Experimental results show that our approach substantially outperforms previous low-precision number representations and can achieve near full-precision model accuracy with as little as 2-bit weights and 2-bit activations.

The remainder of the article is organized as follows. Section 2 introduces some background on CNNs and ReRAM-based acceleration. Section 3 describes our training algorithm for training CNNs with TBN representations. Section 4 describes our activation-side coalescing approach that combines BN, non-linear activation, and quantization into a single efficient stage. Sections 5 and 6 present our evaluation results. Section 7 concludes the article.

2 BACKGROUND

2.1 Convolutional Neural Networks

A CNN is a class of deep, feed-forward artificial neural networks that has successfully been applied to multi-channel image classification. A typical CNN comprises a pipeline of connected layers, each performing transformations from a set of input feature maps to a new set of output feature maps. The inputs to the first layer correspond to the channels of an input image, and the outputs of the last layer correspond to the probabilities of classes that best describe that image. Each layer of the CNN is associated with a set of parameters, usually called *weights*, that are typically trained offline with a labeled dataset. The goal of supervised learning of CNNs is to train these parameters so that the CNN can accurately classify new data points.

In a standard CNN structure, the layers are typically convolutional layers, pooling layers, or fully connected (FC) layers. Each convolutional (Conv) layer consists of a number of $h \times w \times C_{in}$ kernels, each of which is convolved with an $H_{in} \times W_{in} \times C_{in}$ multi-channel input feature map to produce the corresponding $H_{out} \times W_{out}$ output channel. Together, a three-dimensional $H_{out} \times W_{out} \times C_{out}$ output feature map is produced from C_{out} kernels. The convolution operation for the z^{th} kernel can be expressed as follows:

$$\mathbf{z}_{out}(x, y, z) = \sum_{i=0}^{h-1} \sum_{j=0}^{w-1} \sum_{k=0}^{C_{in}-1} \mathbf{w}_z(i, j, k) \cdot \mathbf{x}_{in}(x + i, y + j, k). \tag{1}$$

The elements of a $h \times w \times C_{in}$ kernel are weights to be trained, and a bias term is usually added to $\mathbf{z}_{out}(x, y, z)$, which is also trained.

A pooling layer maps each input feature map to an output feature map, where each output feature is the maximum or average of an $h \times w$ window of input features. A pooling layer reduces the height and width dimensions of the output feature map by a factor of h and w, respectively. Pooling layers are inserted throughout a CNN to gradually reduce the size of the intermediate feature maps.

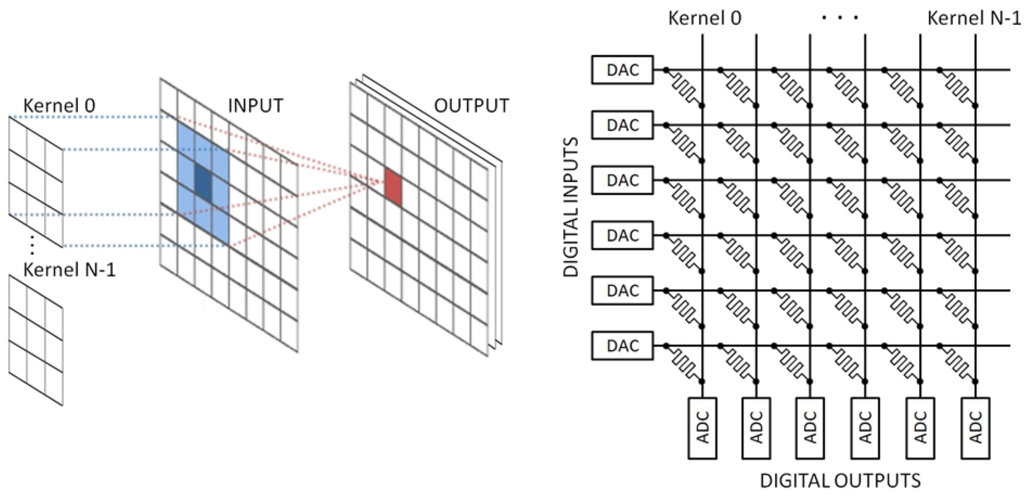

Fig. 2. ReRAM-based crossbar structure.

An FC layer takes an input vector and performs a dot product with a weight vector, which can be expressed as follows:

$$\mathbf{z}_{out} = \sum_{i=0}^{C_{in}-1} \mathbf{w}(i) \cdot \mathbf{x}_{in}(i). \tag{2}$$

A bias term is also usually added to this output, and this bias term together with the weights is also trained.

For Conv and FC layers, the result of Equation (1) or (2) is usually passed through a BN layer [11], which solves the problem of internal covariate shift. The BN operation can be expressed as

$$\mathbf{y} = \gamma \left(\frac{\mathbf{z}_{out} - \mu}{\sqrt{\sigma^2 + \epsilon}} \right) + \beta, \tag{3}$$

where μ and σ are statistics collected over the training set, γ and β are trained parameters, ϵ is used to avoid round-off errors.

Finally, the output of BN is usually passed through a non-linear activation function like ReLU or Sigmoid. In this work, we will assume ReLU activation, which performs $\max(0, \mathbf{y})$.

2.2 ReRAM-Based Crossbar Structure

Figure 2 depicts an $N \times N$ ReRAM crossbar structure where each ReRAM cell can be programmed with one of multiple possible resistance states. The corresponding conductance of an ReRAM cell at the i^{th} row and j^{th} column of the crossbar is represented by $g_{i,j}$. These ReRAM cells can be used to encode the synaptic weights of a neural net. In the case of a convolutional (Conv) layer in a CNN, each column of the ReRAM crossbar, $j = 0, 1, \ldots, N-1$, can be used to implement a different Conv kernel. The input voltage of each row is represented by v_i, which can be used to encode an input feature of a neural net. Each column j of the ReRAM crossbar can then perform an *analog dot product* of the input voltage vector $\{v_0, v_1, \ldots, v_{N-1}\}$ with the corresponding conductivity vector $\{g_{0,j}, g_{1,j}, \ldots, g_{1,N-1}\}$ as follows:

$$I_{out}^{j} = \sum_{i=0}^{N-1} g_{i,j} \cdot v_i. \tag{4}$$

These dot product operations across the columns are performed simultaneously as a single matrix-vector multiplication operation. A DAC is used to convert a digital input into an analog voltage v_i, and an ADC is used to convert an output voltage derived from I^j_{out} into a digital output. In the case of an FC layer, the entire crossbar can be used to implement the corresponding weight matrix.

3 LEARNING THE BIASED NUMBER REPRESENTATIONS AND WEIGHT APPROXIMATIONS

3.1 Gradient Calculations

In our training algorithm, we begin with a pre-trained model with full-precision weights, with each latent full-precision weight denoted as $\tilde{w}_i$. The goal of our training procedure is to assign an m-bit integer $g_i = 0, 1, \ldots, 2^m - 1$ to each $\tilde{w}_i$ so that the latent full-precision weight can be approximated with the following biased number representation:

$$\hat{w}_i = Mg_i - K. \tag{5}$$

Here, M is the scaling factor, and K is the *biasing* term that gets subtracted from the scaled term Mg_i. Note that in this biased number representation, both positive and *negative* numbers are represented using the same m-bit integers $g_i = 0, 1, \ldots, 2^m - 1$. However, unlike signed fixed point representations that represent a symmetric range of positive and negative numbers, our utilization of a biasing term K allows us to *asymmetrically* partition the range of positive and negative numbers. Further, the scaling factor M allows us to provide the *appropriate resolution* for approximating full-precision weights. For example, for $m = 2$ bits, $M = 0.541$, and $K = 1.182$, $g_i = 0, 1, 2, 3$ correspond to the approximate weights $\hat{w}_i = -1.182, -0.641, -0.1, 0.441$, respectively.

The key idea in our training procedure is that M and K are independent parameters that are trained together with other parameters, including the latent full-precision weights. These independent parameters M and K are defined on a *per-layer* basis, meaning that a different pair of parameters is used for each layer to approximate the latent full-precision weights.

During each feed-forward pass, we assign g_i to $\tilde{w}_i$ as follows:

$$g_i = \text{clip}\left(\text{round}\left(\frac{\tilde{w}_i + K}{M}\right), 0, 2^m - 1\right), \tag{6}$$

where

$$\text{clip}(x, x_{\min}, x_{\max}) = \max(x_{\min}, \min(x, x_{\max})). \tag{7}$$

During backpropagation, we calculate the gradient for the scaling factor M as follows:

$$\frac{\partial L}{\partial M} = \sum_i g_i \frac{\partial L}{\partial \hat{w}_i}, \tag{8}$$

where L is the loss to be optimized.

For the biasing term K, we calculate its gradient as follows:

$$\frac{\partial L}{\partial K} = -\left(\sum_i \frac{\partial L}{\partial \hat{w}_i}\right). \tag{9}$$

Finally, the gradient that we use to update each latent full-precision weight is simply the gradient of the corresponding approximate weight:

$$\frac{\partial L}{\partial \tilde{w}_i} = \frac{\partial L}{\partial \hat{w}_i}. \tag{10}$$

Since the latent full-precision weights $\tilde{w}_i$ are updated together with the independent parameters M and K during backpropagation, a different g_i may get assigned to approximate $\tilde{w}_i$ in the next feed-forward pass. In turn, the new weight approximations $\hat{w}_i$ would be used to derive gradients to update the latent full-precision weights $\tilde{w}_i$ and the independent parameters M and K in the next backpropagation phase. This way, the *biased number representations* are *trained together* with the weights and other parameters of the neural network to minimize classification loss.

The benefits of using a TBN representation is that the trained scaling factor provides the appropriate resolution to represent the weights and the trained biasing term provides an asymmetric partitioning of the number range between positive and negative weights. Together, these trained parameters enable our biased number representation to provide more model capacity to the neural network.

3.2 The Initialization of M and K

As discussed earlier, M and K are independent parameters that are defined and trained on a per-layer basis. Before we start on the training procedure described in Section 3.1, we must first initialize M and K for each layer based on the latent full-precision weights from the pre-trained model. The main idea is that each $g_i = 0, 1, \ldots, 2^m - 1$ defines a separate centroid $\hat{w}_i = Mg_i - K$, and we want to initialize M and K so that the corresponding centroids are linearly spaced across the range of pre-trained full-precision weights in a layer. Let $[r_{\min}, r_{\max}]$ denote this range. Then, we initialize M and K as follows:

$$M = \frac{r_{\max} - r_{\min}}{2^m - 1}, \tag{11}$$

$$K = -r_{\min}. \tag{12}$$

Experimentally, we have found that limiting the range of weights to those within two standard deviations from the mean weight of a layer, which covers 95.4% of the weights, leads to a better initialization of M and K. In particular, let $\mu_{\tilde{w}}$ and $\sigma_{\tilde{w}}$ be the mean and standard deviation of the latent full-precision weights in a layer. Then, we define $r_{\min}$ and $r_{\max}$ as follows:

$$r_{\min} = \mu_{\tilde{w}} - 2\sigma_{\tilde{w}}, \tag{13}$$

$$r_{\max} = \mu_{\tilde{w}} + 2\sigma_{\tilde{w}}. \tag{14}$$

Then, M and K are defined accordingly. We have found that if the number range needs to be increased to minimize loss, then gradient descent can quickly update M and K accordingly to increase the range.

4 ACTIVATION-SIDE COALESCING

In the previous section, we described how latent full-precision weights can be accurately approximated using an m-bit integer that matches the weight precision of an ReRAM cell. For example, in ISAAC [1], PRIME [2], and PipeLayer [3], the ReRAM cell precisions are 2 bits, 4 bits, and 4 bits, respectively. Besides overcoming the precision challenge of ReRAM cells, we also have to overcome the precision challenge of input precision to the ReRAM crossbar. In ISAAC [1] and PipeLayer [3], a 1-bit input precision is used, whereas a 3-bit input precision is used in PRIME [2]. In general, a p-bit input precision can be used, which means that we are limited to a p-bit activation and intermediate features are store using p-bits. Unfortunately, when p is small, for example $p = 2$ bits, the results for deep CNNs on complex datasets can be prohibitively inaccurate. Higher effective input precision p can be achieved by evaluating the ReRAM crossbar multiple times. For example, an effective input precision of $p = 4$ can be achieved by evaluating the input 2 bits at a time using an ReRAM crossbar with a 2-bit input precision. However, this incurs proportionally more energy and more processing time, both of which are undesirable. Thus, in general, it is important to minimize the number of bits p for representing the activations as long as high accuracy can be maintained. This problem is discussed in this section.

In particular, we first describe in Section 4.1 how activations are quantized based on a Gaussian distribution. Then, in Section 4.2, we describe an *activation-side coalescing* technique that combines the steps of BN, activation, and quantization into a single stage that simply performs a clipped-rounding operation. We describe in Section 4.2 how our TBN representation of weights described in Section 3 can be combined with activation-side coalescing.

4.1 Gaussian-Based Quantization

To achieve accuracy when using low-bitwidth quantized activations, we use the half-wave Gaussian quantization (HWGQ) approach proposed in Cai et al. [10]. The HWGQ idea is based on the observation that state-of-the-art neural network architectures generally employ BN [11], which forces the responses of each network layer to be a Gaussian distribution with zero mean and unit variance. Moreover, ReLU is widely used as the activation function in state-of-the-art neural network architectures, which acts as an half-wave rectifier that produces linear outputs for non-negative responses. Therefore, the p-bits used to encode the activations only needs to quantized the non-negative range of responses. For example, an activation x_i can be quantized and encoded with an p-bit integer $q_i = 0, 1, \ldots, 2^p - 1$ so that the activation x_i can be approximated with a uniform quantizer as follows:

$$Q(x_i) = \hat{x}_i = Sq_i, \tag{15}$$

where $\hat{x}_i = 0, S, 2S, \ldots (2^p - 1)S$ are the corresponding quantization levels, which in general can be floating point values since the quantization step S can in general be a floating point number. For a uniform quantizer, we can derive q_i from x_i as follows:

$$q_i = \text{clip}\left(\text{round}\left(\frac{x_i}{S}\right), 0, 2^p - 1\right). \tag{16}$$

In Cai et al. [10], an optimal uniform quantizer is derived from a Gaussian distribution with zero mean and unit variance. This is based on the observation that BN network layers generally produce response distributions that are approximately Gaussian with zero mean and unit variance across all units and layers. Therefore, the same uniform quantizer can be used for all activations. There is no need to train a number representation for activations. The activations are indirectly trained by training the BN parameters. In particular, the optimal uniform quantization step S can be derived from a Gaussian distribution with zero mean and unit variance by minimizing the following mean square error:

$$\arg\min_{Q} \int \varphi(x)(Q(x) - x)^2 dx, \tag{17}$$

where $\varphi(x)$ is the corresponding probability density function. The optimal step S can be derived by solving Equation (17) by adding the constraint that $Q(x)$ is a uniform quantizer with step S [10, 12].

4.2 Combining Trained Biased Weights With Activation-Side Coalescing

In this section, we discuss how our TBN representation of weights described in Section 3 can be combined with the Gaussian-based quantized activation approach described earlier in Section 4.1. In particular, we wish to store each weight as an m-bit integer g_i and each activation as a p-bit integer q_i, which can be interpreted by the corresponding parameter S for activation and the corresponding per-layer parameters M and K for the weights. For inference, a naive implementation would operate as follows:

A1. $\hat{x}_i = Sq_i$;
A2. $\hat{w}_i = Mg_i - K$;

A3. $z = \sum_i \hat{w}_i \hat{x}_i + b$;
A4. $y = \text{BatchNorm}(z) = \gamma \left(\frac{z-\mu}{\sqrt{\sigma^2+\epsilon}} \right) + \beta$;
A5. $r = \text{ReLU}(y)$;
A6. $q_{out} = \text{clip}(r/S, 0, 2^p - 1)$;

In Step A6, $2^p - 1$ is the maximum integer for a p-bit integer encoding of the activations.

As can be readily observed in the preceding inference algorithm, the steps do not match the hardware capabilities of a ReRAM crossbar since the steps involve floating point operations. In particular, a ReRAM crossbar is capable of efficiently computing in the analog domain an integer dot product operation of the form

$$\sum_i g_i q_i, \tag{18}$$

where each g_i is an m-bit integer that corresponds to the ReRAM cell precision, and each q_i is a p-bit integer that corresponds to the input precision of the ReRAM crossbar. Ideally, we would like to avoid expanding the integers g_i and q_i into floating point numbers in Steps A1 and A2 and performing the dot product in the floating point domain in Step A3. Instead, we would like to perform the dot product in the integer domain to match the hardware capabilities of a ReRAM crossbar. This can be achieved by rewriting Steps A1 through A3 as follows:

$$\begin{aligned} \sum_i \hat{w}_i \hat{x}_i + b &= \sum_i (Mg_i - K)(Sq_i) + b, & (19)\\ &= S \sum_i (Mg_i q_i - Kq_i) + b, & (20)\\ &= S\left(M \left(\sum_i g_i q_i \right) - K \left(\sum_i q_i \right) \right) + b, & (21)\\ &= S(Mp_1 - Kp_2) + b, & (22) \end{aligned}$$

where

$$\begin{aligned} p_1 &= \sum_i g_i q_i, & (23)\\ p_2 &= \sum_i q_i. & (24) \end{aligned}$$

As can be readily observed, p_1 can be directly implemented as an integer dot product using a column in a ReRAM crossbar. In particular, each column in a ReRAM crossbar can be used to implement this p_1 computation for a different kernel, as depicted in Figure 2.

The computation for p_2 can also readily be implemented directly as an integer dot product using a column in a ReRAM crossbar by programming the corresponding ReRAM cells with unit weights. However, since the computation of p_2 is *kernel independent*, the p_2 computation can be *shared by all the kernels* that are implemented on the same ReRAM crossbar. For example, if we have a baseline 128×128 ReRAM crossbar, we can add one more column to create a 128×129 array and use the last column to implement p_2, which can be shared by all 128 kernels implemented in the array. Thus, the amortized cost of p_2 is negligible.

Besides performing integer dot products directly using g_i and q_i, we propose to further optimize BN, ReLU activation, and quantization steps (Steps A4 through A6) by combining them into a single step. In particular, we wish to eliminate the floating point operations in BN and absorb the BN parameters as well as the quantization step parameter S in Equation (22) directly into a simple clipped-rounding operation.

The optimized algorithm is as follows:

B1. $p_1 = \sum_i g_i q_i$;
B2. $p_2 = \sum_i q_i$;
B3. q_{out} = activation_side_coalescing(p_1, p_2);

In particular, given Equation (22), we have $z = S(Mp_1 - Kp_2) + b$. Then the BN operation can be stated as follows:

$$y = \gamma\left(\frac{z-\mu}{\sqrt{\sigma^2+\epsilon}}\right) + \beta, \tag{25}$$

$$= \gamma\left(\frac{SMp_1 - SKp_2 + b - \mu}{\sqrt{\sigma^2+\epsilon}}\right) + \beta, \tag{26}$$

$$= \frac{\gamma S(Mp_1 - Kp_2)}{\sqrt{\sigma^2+\epsilon}} + \frac{\gamma(b-\mu)}{\sqrt{\sigma^2+\epsilon}} + \beta. \tag{27}$$

Then we can compute

$$\hat{q} = \text{round}(y/S), \tag{28}$$

$$= \text{round}\left[\frac{\gamma(Mp_1 - Kp_2)}{\sqrt{\sigma^2+\epsilon}} + \frac{\gamma(b-\mu)}{S\sqrt{\sigma^2+\epsilon}} + \frac{\beta}{S}\right]. \tag{29}$$

Given that $\gamma, \mu, \sigma, \epsilon, \beta$, and S are all constants after training, we can pre-compute these constants for all activations:

$$A = \frac{\gamma M}{\sqrt{\sigma^2+\epsilon}}, \tag{30}$$

$$B = \frac{\gamma K}{\sqrt{\sigma^2+\epsilon}}, \tag{31}$$

$$C = \frac{\gamma(b-\mu)}{S\sqrt{\sigma^2+\epsilon}} + \frac{\beta}{S}. \tag{32}$$

Then we have

$$\hat{q} = \text{round}(Ap_1 - Bp_2 + C), \tag{33}$$

$$q_{out} = \text{clip}(\hat{q}, 0, 2^p - 1). \tag{34}$$

We can directly implement Equations (33) and (34) in a single clipped-rounding operation in activation_side_coalescing(p_1, p_2). Note that the clip operation effectively performs a clipped ReLU activation. The operations in Equations (33) and (34) can be easily implemented in the digital domain.[1]

5 EVALUATION

5.1 Evaluation Setup

We have implemented our proposed training algorithm based on a TBN representation described in Section 3 and our proposed activation-side coalescing technique described in Section 4 in the PyTorch framework [26]. We use the CIFAR-10 and CIFAR-100 datasets to evaluate our solutions on three versions of the popular VGG deep CNN architecture [14]: VGG-11, VGG-13, and VGG-19. Since VGG networks were originally proposed for the ImageNet dataset whose input size is

[1]Given the clipped-rounding operations to just a few bits, the operands in Equation (33) do not need to be full precision. Experimentally, we found that an 8-bit fixed point representation of the operands is more than enough for small values of p.

Table 1. CNN Configurations

Layer	VGG-11	VGG-13	VGG-19
Conv1	3×3×3, 64	3×3×3, 64	3×3×3, 64
		3×3×64, 64	3×3×64, 64
Max-pool	2×2		
Conv2	3×3×64, 128	3×3×64, 128	3×3×64, 128
		3×3×128, 128	3×3×128, 128
Max-pool	2×2		
Conv3	3×3×128, 256	3×3×128, 256	3×3×128, 256
			3×3×256, 256
	3×3×256, 256	3×3×256, 256	3×3×256, 256
			3×3×256, 256
Max-pool	2×2		
Conv4	3×3×256, 512	3×3×256, 512	3×3×256, 512
			3×3×512, 512
	3×3×512, 512	3×3×512, 512	3×3×512, 512
			3×3×512, 512
Max-pool	2×2		
Conv5	3×3×512, 512	3×3×512, 512	3×3×512, 512
			3×3×512, 512
	3×3×512, 512	3×3×512, 512	3×3×512, 512
			3×3×512, 512
Max-pool	2×2		
FC1	512×512		
FC2	512×512		
FC3	512×10 (100)		

FC layers have been adjusted to fit the size of the CIFAR dataset.

$224 \times 224 \times 3$, whereas the CIFAR images are $32 \times 32 \times 3$, we reduce the size of the FC layers to match the input features. The configurations of networks are summarized in Table 1. All convolutional and FC layers except the last layer are followed by a BN layer and ReLU non-linearity in sequence. The Adam optimizer is used to update the scaling and shifting factors, with the learning rate initialized to be 1e-6. The other parameters of the network are optimized by stochastic gradient descent (SGD) with a learning rate starting at 0.01. The momentum and weight decay factor that we use for SGD are 0.9 and 1e-4, respectively. We use a mini-batch size of 128 and divide the learning rates for both optimizers by 10 every 75 epochs. The minimal learning rates for SGD and Adam are 1e-7 and 1e-8, respectively. The results are computed by taking the average of the last seven test accuracy numbers with the maximal and minimal values removed.

5.2 Evaluation Results

As discussed in Section 1, weights can be quantized into low-precision representations using DFP [7, 8] implementations. To evaluate the performance of our TBN representation, we apply these two methods to compress weights into the resolutions from 2 to 8 bits with features remaining in full precision. The scaling factors of DFP are derived by minimizing the quadratic error from a pre-trained model, whereas the scaling and shifting factors of TBN are trained according to (8) and (9) from the same pre-trained model. The results are shown in Table 2 together with the full-precision accuracies added as a baseline.

Table 2. Classification Accuracy on CIFAR-10

Weight Bits			2	3	4	5	6	7	8	Full Precision
CIFAR-10	VGG-11	DFP	10.0%	71.6%	86.9%	88.1%	90.0%	90.0%	90.0%	91.6%
		TBN	91.1%	91.2%	91.3%	91.3%	91.4%	91.4%	91.5%	
	VGG-13	DFP	10.0%	35.2%	87.6%	91.6%	93.1%	93.4%	93.3%	93.7%
		TBN	93.4%	93.3%	93.5%	93.4%	93.3%	93.4%	93.5%	
	VGG-19	DFP	10.0%	10.0%	71.5%	90.0%	92.7%	93.1%	93.1%	93.3%
		TBN	93.0%	93.2%	93.2%	93.0%	93.1%	93.2%	93.1%	
CIFAR-100	VGG-11	DFP	1.0%	5.0%	61.6%	67.7%	69.0%	70.2%	70.1%	70.2%
		TBN	68.2%	69.4%	69.7%	70.1%	70.1%	70.2%	70.2%	
	VGG-13	DFP	1.0%	4.7%	61.3%	69.0%	72.4%	72.7%	72.8%	73.3%
		TBN	72.9%	73.2%	73.2%	73.0%	73.1%	73.2%	73.5%	
	VGG-19	DFP	1.0%	1.0%	36.5%	61.7%	69.0%	70.4%	71.2%	72.2%
		TBN	71.6%	72.3%	72.2%	72.2%	72.3%	72.3%	72.2%	

It can be seen that, compared to DFP, TBN achieves a significantly higher accuracy. On the CIFAR-10 dataset, TBN keeps the loss within 1% with 2-bit weights while the performance of DFP drops below 30%, in comparison with full-precision weights. In particular, since there are only 10 classes in total in CIFAR-10, an accuracy of 10% implies that the model is seriously damaged and fails to produce any reasonable classification. When evaluated using the CIFAR-100 dataset, 2-bit TBN results in a more significant accuracy drop on VGG-11, which is 2% worse than the full-precision model, whereas this loss shrinks sharply with the increase of bitwidth and reduces to 0.5% with 4-bit weights. However, DFP turns to have an extremely poor performance with 2-bit weights and the 4-bit degradation is still larger than 5%, compared to the full-precision models. Moreover, on the VGG-11 network and CIFAR-10 dataset, even when using 8-bit representations, DFP still results in an obvious loss in accuracy, due to the fact that the quantization errors from the pre-trained model remain substantial, which determines an upper bound on the accuracy. Here, we note that M and K for TBN are updated according to gradients that are derived to directly minimize classification loss, and that the latent full-precision weights are also trained to compensate for errors caused by low precision.

Next, we compare our TBN representation approach with activation-side coalescing on different combinations of precisions for weights and activations on the same VGG network configurations (VGG-11, VGG-13, and VGG-19). In particular, for both weights and activations, we vary the precision from 2 to 8 bits, and we compare each configuration to full-precision accuracy. The results of CIFAR-10 and CIFAR-100 are illustrated in Figure 3 and Figure 4, respectively, as a function of weight precision with multiple lines to show different activation precisions.

In all cases, the bottom blue line corresponding to 2-bit activations shows the lowest accuracy. However, even with 2-bit activations, our approach achieves accuracies within about 1% of full-precision activations on average under different weight precisions. With 3-bit activations, our approach achieves accuracies within just 0.5% of full-precision activations on average, which is negligible consider the substantial energy savings in ReRAM-based implementations. The slopes of the lines reflect the accuracy loss due to decreasing weight precisions. There exists some ripples in the curves due to the inherent randomness of SGD-based training in the experiments, but it can be identified that, as expected, the accuracy tends to decrease as the precision of weights is reduced. It can be seen that the drops corresponding to CIFAR-100 are sharper than those of CIFAR-10, potentially because of the less redundancy in networks for more complicated datasets.

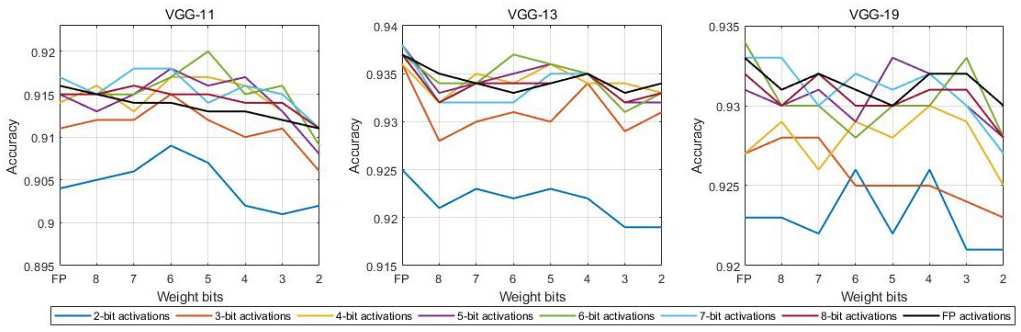

Fig. 3. Classification errors of CIFAR-10 dataset on VGG-11 (left), VGG-13 (middle), and VGG-19 (right) with different weight and activation precisions.

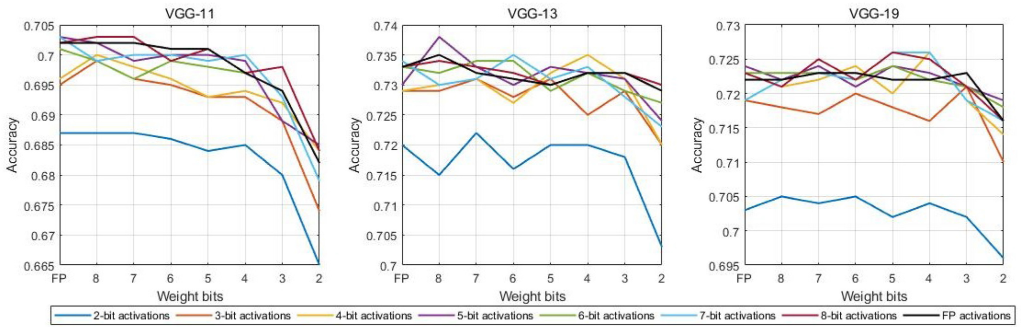

Fig. 4. Classification errors of CIFAR-100 dataset on VGG-11 (left), VGG-13 (middle), and VGG-19 (right) with different weight and activation precisions.

On CIFAR-10, the largest degradation from 2-bit to full precision is only about 0.5%, and when using 4-bit weights and activations, we achieve virtually no loss relative to full-precision models on both the CIFAR-10 and CIFAR-100 datasets.

In particular, with 2-bit weights and 2-bit activations, our proposed approach on the CIFAR-10 dataset achieves accuracies of 90.2%, 91.9%, and 92.1% on VGG-11, VGG-13, and VGG-19, respectively. Comparing with full-precision model accuracy, which are 91.6%, 93.7%, and 93.3%, the accuracy loss is about 1.5%. Moreover, the difference in accuracies vs. full-precision accuracies on both the CIFAR-10 and CIFAR-100 datasets can be further reduced to <1% with 3-bit weights and activations and to virtually no loss with 4-bit weights and activations.

6 ENERGY AND AREA ESTIMATION

In this section, we evaluate the energy and area consumption of our approach based on a recent ReRAM-based hardware accelerator architecture [1]. The analyses are performed on the VGG-11 network trained on the CIFAR datasets.[2] As shown in Shafiee et al. [1], at 32nm, the optimal design point is achieved by using 128×128 ReRAM crossbar arrays with a cell precision of 2 bits, which are considered as the basic unit in our evaluation. To increase the precision of weights, a

[2]In particular, the energy and area estimations are based on the CIFAR-10 dataset, noting that the difference in overhead between CIFAR-10 and CIFAR-100 is negligibly close to 0 due to the fact that their architectures only differ in the last FC layer.

Table 3. Unit Cost of the ReRAM Array and eDRAM Buffer

ReRAM Properties				
Component	Parameter	Spec	Power (mW)	Area (mm^2)
ReRAM array	Resolution	2 bits	0.3	0.000025
	Size	128×128		
ADC	Resolution	8 bits	2	0.0012
	Number	1		
DAC	Resolution	1 bit	0.5	0.000021
	Number	128		
Interface[a]			0.319375	0.000787
Total			3.119375	0.002033
eDRAM Properties				
eDRAM[b]	Size	1KB	0.432813	0.002703

[a]The interface component comprises the amortized cost of input/output registers and routers to interface with eDRAM buffers and other ReRAM arrays, as well as the sample-and-hold and shift-and-add units for data accumulation.
[b]The unit cost of the eDRAM component is provided on a per-page (1KB) basis, and this unit cost includes the amortized cost of the memory bus for interfacing with ReRAM arrays.

group of multiple crossbars can be used, which spatially increases both energy and area consumption. Also as proposed in Shafiee et al. [1], we use an input precision of 1 bit, which effectively replaces the DACs with trivial inverters, and 128 such 1-bit DACs are used for the 128 rows of every crossbar array. Higher-precision inputs can be achieved by evaluating the ReRAM crossbar multiple times with successive 1-bit inputs, which temporally increases energy consumption. At the output side of a crossbar array, an 8-bit ADC is shared by all 128 columns. As shown in Shafiee et al. [1], the cycle time is bounded by the large latency of crossbar arrays, which is on the order of 100ns, whereas a frequency on the level of giga samples per second (GSps) can be achieved for an 8-bit ADC. Thus, an 8-bit ADC can be time multiplexed by the columns of the crossbar without performance degradation. Moreover, an additional column per crossbar is also added in Shafiee et al. [1], and the extra cost has already been accounted for, so there is no need to further re-scale the overhead. For the storage of intermediate features, the architecture proposed in Shafiee et al. [1] uses on-chip eDRAM buffers. In Table 3, we summarize the power and area costs of crossbar arrays and eDRAM buffers, as derived from Shafiee et al. [1].

We evaluate our TBN representation approach in comparison with a DFP approach using 2-bit weights and activations (namely 2-bit precision for both weights and activations). For the DFP approach, "sign splitting" is required since an ReRAM crossbar cannot directly implement both positive and negative weights when the DFP representation is used, as explained in Chi et al. [2] and Section 1. Therefore, two separate crossbars are required to represent positive and negative weights, respectively, and the final results can be obtained by subtracting the outputs of two arrays. The costs of two separate crossbars are reflected in the DFP results. In addition, we include the estimations of a 16-bit fixed point implementation (labeled as "16-bit") to provide a baseline for comparison.

In Table 4, we summarize the energy and area results for the three models. In particular, the table reports the number of ReRAM crossbar arrays and the size of the eDRAM buffers for each of the three implementations. As explained in Shafiee et al. [1] and Song et al. [3], with the dataflows fully pipelined, the energy consumption is proportional to the processing time of dot-product operations, whose bottleneck is the layer with the largest latency, which can be up to thousands

Table 4. Area and Energy Estimations

Parameter	16 Bit	2-Bit DFP	2-Bit TBN
Number of ReRAM arrays	4,948	1,352	742
eDRAM size (KB)	298	40	40
Area (mm^2)	10.87	2.86	1.62
Normalized area	6.72	1.77	1
Energy (μJ/img)	1,593.72	54.20	29.85
Normalized energy	53.39	1.82	1

of cycles. To optimize the critical layers, we adopt the parallelism granularity scheme described in Song et al. [3]. In particular, for the 16-bit fixed point implementation, the number of ReRAM crossbar arrays reported in Table 4 has the Conv1 and Conv2 layers duplicated 16 and 4 times, respectively, to match the speed of other layers. As the input images of the Conv1 layer have a precision of 16 bits, the Conv1 layer for the 2-bit TBN and DFP implementations has to be further duplicated by another factor of $16/2 = 8$ times to match the processing times of the other layers with 2-bit input features. In addition to reporting the absolute energy and area results for the three implementations, we also provide in Table 4 the normalized results with respect to the 2-bit TBN cost to illustrate our overhead reduction.

As shown in Table 4, our TBN approach has lower energy and area costs than both the DFP and 16-bit models. In particular, in comparison to the 16-bit results, 2-bit TBN achieves 6.7 times area reduction since it uses fewer crossbar arrays to encode synaptic weights and smaller buffers to store intermediate features. Moreover, the 16-bit model consumes 53.4 times as much energy as TBN does to process an image, due to the quadratic decrease in energy caused by lower power and faster speed of the low-precision models. When compared to 2-bit DFP results, TBN achieves 1.8 times reduction in both area and energy costs because of the double crossbar arrays introduced in DFP to represent positive and negative weights.

7 CONCLUSION

In this article, we consider the problem of training CNNs with low-precision weights and activations in a manner that is compatible with an implementation on ReRAM-based neural network accelerators. Low-precision weights and activations are needed to match the low resolutions of memory cells and input voltages in ReRAM-based structures. In particular, non-uniform quantization approaches are not amenable to ReRAM-based crossbar implementations, and previous uniform quantization approaches have poor accuracies when applied to deep CNNs on complex datasets. We propose a TBN representation with trainable scaling and shifting factors that can approximate well asymmetric number ranges, which can achieve near full-precision model accuracy with as little as 2-bit weights and 2-bit activations on difficult datasets. Moreover, we propose an activation-side coalescing technique that combines the steps of BN, non-linear activation, and quantization into a single stage that simply performs a clipped-rounding operation. Evaluation results show that our TBN representation significantly outperforms previous quantization approach in terms of both classification errors and the costs of energy and area. In particular, our models achieve accuracies within about of 1.5% of full-precision model accuracy with 2-bit weights and activations on the CIFAR-10 dataset and about 0.1% of accuracy degradation with 4-bit weights and activations on both the CIFAR-10 and CIFAR-100 datasets. Moreover, when using 2-bit weights and activations, our proposed approach yields about 6.7 and 53.4 times reduction in terms of area and energy consumption, respectively.

REFERENCES

[1] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, et al. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. 14–26.

[2] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, et al. 2016. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. 27–39.

[3] Linghao Song, Xuehai Qiany, Hai Li, and Yiran Chen. 2017. PipeLayer: A pipelined ReRAM-based accelerator for deep learning. In *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*. 541–552.

[4] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. 2016. Convolutional neural networks using logarithmic data representation. arXiv:1603.01025.

[5] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Quantized neural networks: Training neural networks with low precision weights and activations. arXiv:1609.07061.

[6] 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. arXiv:1606.06160.

[7] Matthieu Courbariaux, Jean-Pierre David, and Yoshua Bengio. 2014. Low precision storage for deep learning. arXiv:1412.7024.

[8] Yu Ji, Youhui Zhang, Wenguang Chen, and Yuan Xie. 2018. Bridge the gap between neural networks and neuromorphic hardware with a neural network compiler. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, 448–460.

[9] Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. arXiv:1510.00149.

[10] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. 2017. Deep learning with low precision by half-wave Gaussian quantization. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'17)*.

[11] S. Ioffe and C. Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the IEEE International Conference on Machine Learning*. 448–456.

[12] S. P. Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (1982), 129–136.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*. 770–778.

[14] K. Simonyan and A. Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations (ICLR'15)*.

[15] H.-S. Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, et al. 2012. Metal-oxide RRAM. *Proceedings of the IEEE* 100, 6 (2012), 1951–1970.

[16] Miao Hu, John Paul Strachan, Zhiyong Li, Emmanuelle M. Grafals, Noraica Davila, Catherine Graves, Sity Lam, et al. 2016. Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication. In *Proceedings of the 53rd ACM/IEEE Design Automation Conference*. 19.

[17] Miao Hu, Hai Li, Qing Wu, and Garrett S. Rose. 2012. Hardware realization of BSB recall function using memristor crossbar arrays. In *Proceedings of the 49th ACM/IEEE Annual Design Automation Conference*. 498–503.

[18] Boxun Li, Yi Shan, Miao Hu, Yu Wang, Yiran Chen, and Huazhong Yang. 2013. Memristor-based approximated computation. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*. IEEE, Los Alamitos, CA, 242–247.

[19] M. Prezioso, F. Merrikh-Bayat, B. Hoskins, G. Adam, K. K. Likharev, and D. B. Strukov. 2015. Training and operation of an integrated neuromorphic network based on metal-oxide memristors. *Nature* 521, 7550 (2015), 61–64.

[20] Y. Kim, Y. Zhang, and P. Li. 2015. A reconfigurable digital neuromorphic processor with memristive synaptic crossbar for cognitive computing. *Journal of Emerging Technologies for Computing Systems* 11, 4 (2015), Article 38, 25 pages.

[21] Zhe Chen, Bin Gao, Zheng Zhou, Peng Huang, Haitong Li, Wenjia Ma, Dongbin Zhu, et al. 2015. Optimized learning scheme for grayscale image recognition in a RRAM based analog neuromorphic system. In *Proceedings of the 2015 IEEE International Electron Devices Meeting (IEDM'15)*. IEEE, Los Alamitos, CA, 17–7.

[22] G. W. Burr, P. Narayanan, R. M. Shelby, Severin Sidler, Irem Boybat, Carmelo di Nolfo, and Yusuf Leblebici. 2015. Large-scale neural networks implemented with non-volatile memory as the synaptic weight element: Comparative performance analysis (accuracy, speed, and power). In *Proceedings of the 2015 IEEE International Electron Devices Meeting (IEDM'15)*. IEEE, Los Alamitos, CA, 4.

[23] X. Liu, M. Mao, B. Liu, H. Li, Y. Chen, B. Li, Y. Wang, et al. 2015. RENO: A high-efficient reconfigurable neuromorphic computing accelerator design. In *Proceedings of the Design Automation Conference (DAC'15)*.

[24] T. Taha, R. Hasan, C. Yakopcic, and M. McLean. 2013. Exploring the design space of specialized multicore neural processors. In *Proceedings of the 2013 International Joint Conference on Neural Networks (IJCNN'13)*.

[25] C. Yakopcic and T. M. Taha. 2013. Energy efficient perceptron pattern recognition using segmented memristor crossbar arrays. In *Proceedings of the 2013 International Joint Conference on Neural Networks (IJCNN'13)*.

[26] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, et al. 2017. Automatic differentiation in PyTorch. In *Proceedings of the 2017 NIPS Workshop*.

[27] Peisong Wang, Qinghao Hu, Yifan Zhang, Chunjie Zhang, Yang Liu, and Jian Cheng. 2018. Two-step quantization for low-bit neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[28] Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian D. Reid. 2017. Towards effective low-bitwidth convolutional neural networks. arXiv:1711.00205.

Received July 2018; revised November 2018; accepted January 2019