Supporting the Momentum Training Algorithm Using a Memristor-Based Synapse

Tzofnat Greenberg-Toledo, Roee Mazor, Ameer Haj-Ali, *Student Member, IEEE*, and Shahar Kvatinsky, *Senior Member, IEEE*

***Abstract*— Despite the increasing popularity of deep neural networks (DNNs), they cannot be trained efficiently on existing platforms, and efforts have thus been devoted to designing dedicated hardware for DNNs. In our recent work, we have provided direct support for the stochastic gradient descent (SGD) training algorithm by constructing the basic element of neural networks, the synapse, using emerging technologies, namely memristors. Due to the limited performance of SGD, optimization algorithms are commonly employed in DNN training. Therefore, DNN accelerators that only support SGD might not meet DNN training requirements. In this paper, we present a memristor-based synapse that supports the commonly used momentum algorithm. Momentum significantly improves the convergence of SGD and facilitates the DNN training stage. We propose two design approaches to support momentum: 1) a hardware friendly modification of the momentum algorithm using memory external to the synapse structure, and 2) updating each synapse with a built-in memory. Our simulations show that the proposed DNN training solutions are as accurate as training on a GPU platform while speeding up the performance by 886× and decreasing energy consumption by 7×, on average.**

***Index Terms*— Memristor, deep neural networks, training, momentum, synapse, stochastic gradient descent, hardware, VTEAM.**

I. Introduction

ARTIFICIAL Neural Networks, and especially Deep Neural Networks (DNNs) and Convolutional Neural Networks (CNN), achieve state-of-the-art results for different tasks. Therefore, these algorithms have gained increasing interest from the scientific community and are widely used in a broad range of applications, such as image and speech recognition, natural language processing, automotive and finance [1]–[3].

Manuscript received June 9, 2018; revised August 30, 2018, October 12, 2018, and November 13, 2018; accepted December 10, 2018. Date of publication January 3, 2019; date of current version March 15, 2019. This work was supported by the European Research Council under the European Union's Horizon 2020 Research and Innovation Programme under Grant 757259. This paper was recommended by Associate Editor A. James. *(Corresponding author: Tzofnat Greenberg-Toledo.)*

T. Greenberg-Toledo, R. Mazor, and S. Kvatinsky are with the Andrew and Erna Viterbi Department of Electrical Engineering, Technion–Israel Institute of Technology, Haifa 3200003, Israel (e-mail: stzgrin@tx.technion.ac.il; roeemz@tx.technion.ac.il; shahar@ee.technion.ac.il).

A. Haj-Ali is with the Faculty of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA 94720 USA (e-mail: ameerh@berkeley.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2018.2888538

DNN architecture is structured by layers of neurons connected by synapses. Each synapse is weighted, and the functionality of the network is set by supplying different values to those weights. To find the values suitable for a specific task, machine learning algorithms are used to train the network. After the training is complete, the network is provided with new data and it infers the result based on its training; this stage is called the inference stage. One of the most common DNN training algorithms is *stochastic gradient descent* (SGD) [4]–[7]. The SGD algorithm minimizes a defined cost function, which represents the error value of the network output. However, the simplest form of SGD (sometimes called *vanilla SGD*) does not guarantee a good convergence rate or convergence to the global minimum [8]. Different optimization algorithms have thus been proposed to improve SGD performance; examples include Momentum, Adagrad, and Adam [9]–[11].

Both the training and inference stages in DNNs are usually executed by commodity hardware (mostly FPGA and GPU platforms), as they are compute and memory intensive. Yet the intensive data movement required as a result of the separation of memory and processing units in such hardware poses significant limitations on the performance and energy efficiency of those systems. Many studies have thus been devoted to developing dedicated hardware, optimized for executing DNN tasks. One approach is to move the computation closer to the memory [12]. Another approach is to accelerate the performance of the NN by improving the performance of the matrix-vector multiplication, which is known to be computation-intensive for CMOS logic circuits. Therefore, in recent years several works have proposed analog implementations of the matrix-vector multiplication, based on the new emerging memristor technologies [13], [14]. The memristor is used to implement the synapse, and it stores the weight as well as computes the matrix-vector multiplication in the synapse array. Such solutions accelerate only the inference stage, and the network must be trained beforehand.

Providing hardware support for the DNN training stage is more complex than supporting only the inference stage. The training includes differential derivative calculations, updating the weights, and propagating the result through the network. Previous designs that attempted to support the DNN training stage support only SGD or simpler modifications of it [12], [15], [16]. We previously proposed a synapse circuit based on two CMOS transistors and a single memristor to support

vanilla SGD [17], [18]. In this paper, we propose to modify the synapse circuit to support the momentum algorithm, a common optimization algorithm which accelerates the convergence of SGD. We propose two circuit-level designs. The first design is a hardware friendly modification of the momentum algorithm that employs memory external to the synapse array. The second design supports the original momentum algorithm and adds memory to the synapse circuit. The proposed designs and the principles presented in this paper can be integrated into full architecture solutions such as Pipelayer [19] and TIME [20].

Both solutions have been designed using the VTEAM model [21] for the memristor and 180nm CMOS process. The area overhead incurred by the synapse modification is compared between the two solutions, and the accuracy of the training algorithms is validated using the standard software implementation. The two solutions are evaluated and compared to a GPU platform in terms of power and run time. Our simulations show that training a DNN with the momentum algorithm using our design can improve over GPU runtime by 886× on average.

The rest of the paper is organized as follows. In Section II, background on DNNs and memristor-based hardware for DNNs is given. In Section III, the proposed synapse circuits are described and qualitatively compared. In Section IV, the proposed circuits are evaluated for their compatibility to the momentum algorithm, their run time, and their energy consumption. Conclusions and future research directions are given in Section V.

II. Hardware-Based DNN

This section provides the required background on deep neural networks. First, we describe the topology of DNN algorithms (Section II-A) and the SGD algorithm (Section II-B). Then, previous hardware designs of DNNs using memristors are described (Section II-C), including our previously proposed synapse that supports vanilla SGD, which is the basis for this paper (Section II-D).

A. Deep Neural Networks

The basic computation element in a DNN is the neuron. DNNs are constructed from layers of neurons, each of which determines its own value from a set of inputs connected to the neuron through a weighted connection called a synapse. Therefore, the value of the output is given by the weighted sum of the input,

$$r_n = \sum_{m=1}^{M} w_{nm} x_m, \tag{1}$$

where M, x_m, w_{mn}, and r_n are, respectively, the total number of input neurons, the value of input neuron m, the connection weights (synapse weights) between neuron n and neuron m, and output n. The structure of a single neuron layer is shown in Figure 1. In the general case, each connection has its own weight, and thus the output vector $\vec{r}$ is determined by a matrix-vector multiplication,

$$\vec{r} = W\vec{x}, \tag{2}$$

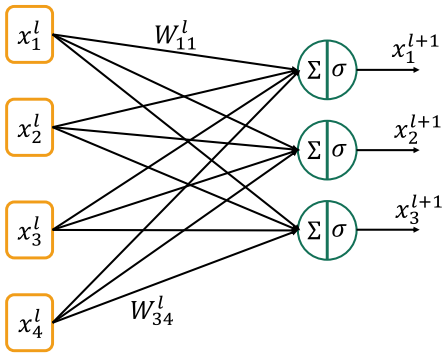


Fig. 1. Single layer in an Artificial Neural Network. Each neuron in the input neuron vector $\vec{x}^l \in \mathbb{R}^4$ is connected to an output neuron in the output neuron vector $\vec{x}^{l+1} \in \mathbb{R}^3$ by a synapse. Each synapse has its own weight W_{nm}^l, $W^l \in \mathbb{R}^{4x3}$. By feeding the weighted sum into an activation function, the output neuron value is determined.

where the elements of matrix W are the synapse weights and $\vec{x}$ is the input neuron vector. The neuron vector of the next layer $\vec{x}^{l+1}$is calculated by passing the output vector of the previous layer $\vec{r}^l$ through an activation function, which is some nonlinear function $\sigma(\cdot)$. Hence,

$$\vec{x}^{(l+1)} = \sigma(W^{(l)}\vec{x}^{(l)}) = \sigma(\vec{r}^{(l)}). \tag{3}$$

The network is therefore constructed by cascading matrices of synapses and activation function layers. Each time a neuron input vector is fed to the layer, the output neuron vector is determined. For simplicity, the discussion in this paper will focus on fully connected layers, where all the input neurons are connected to each output neuron and the weights are not shared among synapses.

B. Stochastic Gradient Descent

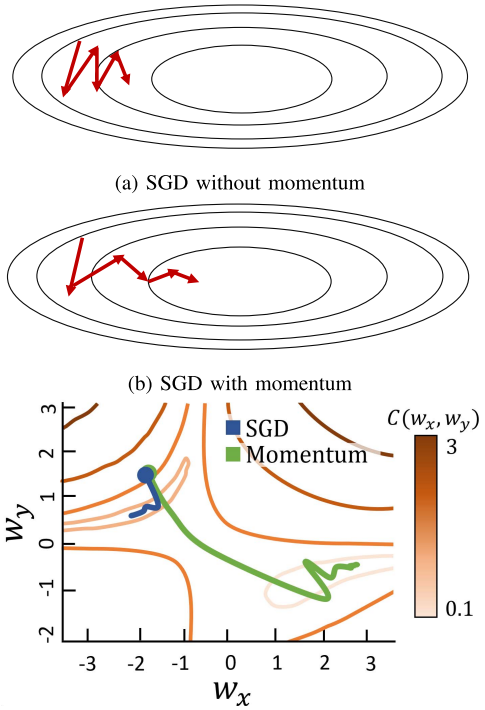
In supervised learning, the network is trained to find the set of synapse weights that approximates the desired functionality. In the training stage, sets of pairs $\{\vec{x}, \vec{d}\}$ are provided to the network, where $\vec{x}$ is the input vector of the network and $\vec{d}$ is the desired output. To define the performance of the DNN, a measure of quality is defined: the cost function $C(\vec{d}, \vec{z})$, where $\vec{z}$ is the output of the network. The goal of the learning algorithm is to minimize the value of $C(\vec{d}, \vec{z})$ with respect to the weights of the network. The training process is iterative: for iteration k, the error is determined for the given $\{\vec{x}_k, \vec{d}_k\}$ pair. Using SGD, the weights are updated in the opposite direction to the cost function derivative with respect to each weight:

$$w_{nm}^{(l)} = w_{nm}^{(l)} - \eta \frac{\partial C(d,z)}{\partial w_{nm}^{(l)}}, \tag{4}$$

where l is the synapse layer, nm is the weight location at that layer (connecting input neuron m to output neuron n), $w_{nm}^{(l)}$ is the value of the weight and η is the learning rate. For DNNs with a cost function such as mean square error (MSE) or cross-entropy, the update rule is

$$w_{nm}^{(l)} = w_{nm}^{(l)} + \eta x_m^{(l)} y_n^{(l)}, \tag{5}$$

$$\vec{y}^{(l)} \triangleq ((W^{(l+1)})^T \vec{y}^{(l+1)}) \cdot \sigma'(\vec{r}^{(l)}), \tag{6}$$



(a) SGD without momentum

(b) SGD with momentum

(c) SGD with momentum vs. SGD without momentum

Fig. 2. Using momentum helps SGD to converge faster around ravines. (a) Convergence is slow because of oscillations, while (b) adding momentum lead to a drift of the current update in the common direction of the previous updates, thus helping to speed up the convergence. (c) SGD vs. momentum convergence. $C(W_x, W_y)$ is the cost function, the blue line is the SGD cost and green line is the momentum cost during training. SGD updates the weights according to the current gradient. Therefore, when reaching a "valley" it might be trapped at the local minimum. By considering the previous update directions, SGD with momentum helps to avoid the local minimum.

where $\vec{y}^{(l)}$ represents the error of layer l. As can be seen from (5) and (6), the error is determined at the output layer of the network and then back-propagated into the network.

SGD performs poorly in areas of shallow ravines, where the curve of the cost function surface is steeper in one dimension than in the other [22], [23]. In these cases, oscillations will occur, slowing down the convergence, as illustrated in Figure 2. To improve the convergence of SGD, optimization algorithms such as the momentum algorithm [9] are commonly employed. The momentum algorithm helps navigate the update values and directions, at shallow ravine areas, by adding a fraction γ of the previous updates (the history) to the current update. Thus, the current update takes into account the direction trend from the previous updates and reduces the oscillations, as illustrated in Figure 2. The update rule using momentum is

$$v_{t,nm}^{(l)} = \gamma\, v_{t-1,nm}^{(l)} + \eta \frac{\partial C(d,z)}{\partial w_{nm}^{(l)}}, \tag{7}$$

$$w_{nm}^{(l)} = w_{nm}^{(l)} - v_{t,nm}^{(l)}, \tag{8}$$

where γ is the momentum value. The weights of the synapses are updated according to v_t, which is comprised of all previous v_t values and the current gradient value. Therefore, using momentum and (5), the weight update rule at iteration k is

$$\Delta w_{nm}^{k} = \eta \sum_{j=1}^{k-1} \gamma^{\,k-j} x_m^{(j)} y_n^{(j)} + \eta x_m^{(k)} y_n^{(k)}. \tag{9}$$

The motivation for using the momentum algorithm is well established by dedicated DNN research and this algorithm is widely used to train DNNs [1], [24]–[26]. However, training with momentum causes a hardware overhead, which is addressed in section IV-C.

C. Hardware DNN Using Memristors

The emergence of memristors has enabled combining data storage and computation using the same physical entities. Hence, it is possible to compute the multiplication between the input neuron vector and the synaptic weight matrix in-place and with analog computation. In recent years, different design approaches for memristor-based synapses have been investigated [13]–[17]. In all of these approaches, the memristors are organized in a crossbar array and are used to store the weights of the synapses. Figure 3 illustrates the structure of a neural network layer, where each cross-section represents a memristive synapse. This structure leverages the analog nature of these devices, thus improving the power consumption and latency of synapse related computation. Using an analog regime, the input neuron vector is represented by the voltage drops over the columns u_m, and the weight of the synapse is represented by the conductance of the memristor G_{nm}. The current of each synapse is thus given by Ohm's law,

$$I_{nm} = G_{nm} u_m = w_{nm} x_m, \tag{10}$$

and, using KCL, the current through each row is

$$I_n = \sum_m w_{nm} x_m. \tag{11}$$

Hence, by simply applying a voltage over the columns and summing the currents per row, a highly parallel multiply-accumulate (MAC) operation is computed, which is a fundamental operation of DNN. This analog computation eliminates the need to read the weight values outside of the arrays and the use of special MAC circuits; thus, reduces the power consumption and latency of the MAC operation.

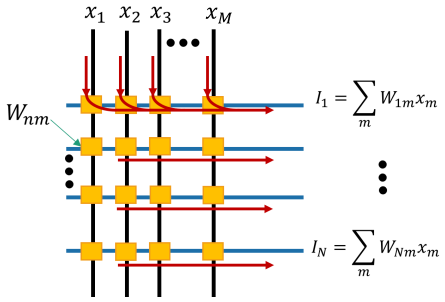


Fig. 3. Memristor-based synapse array. The input vector $x \in \mathbb{R}^{1xM}$ is converted to the corresponding voltage value per column $U \in \mathbb{R}^{1xM}$. The current per row is the sum of currents flowing through the matching synapse cells of the array.

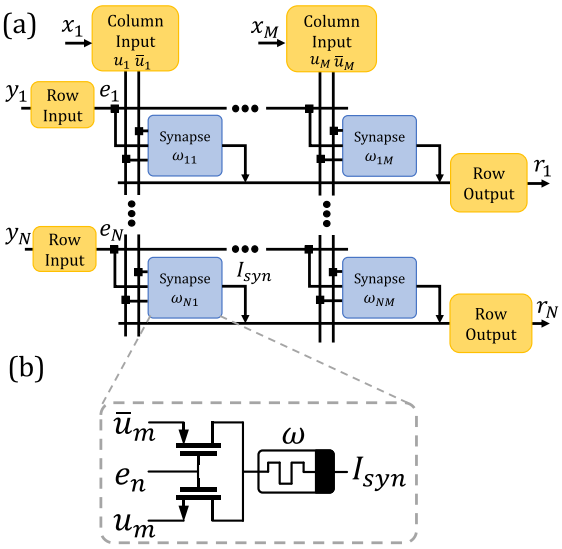


Fig. 4. Baseline synapse. (a) Synaptic grid structure. Each ω_{nm} node is implemented by the memristor-based synapse circuit and represents its weight at location (n, m). Each column receives voltage level u_m and $\bar{u}_m$, which represent the neuron input value scaled to the corresponding voltage level. Each row receives control signal e_n. (b) The baseline memristor-based synapse. Each synapse circuit contains two transistors and a single memristor. The control signal e_n selects the input voltage and controls the effective write duration for training.

In this manner, the feed-forward phase — propagating the input data toward the output — occurs naturally due to the structure of the memristor-based synapse array. Nonetheless, the back propagation phase — propagating the error backwards into the network and updating the weights accordingly — is not so simple. Thus, different approaches for updating the weights have been proposed [15]–[17]. However, due to the complexity involved in supporting full SGD, those approaches implemented only vanilla SGD or even a simple modification of it.

D. Baseline Synapse to Support SGD

We have recently proposed a memristive synapse that can support SGD training [17], [18]. In [17] the analysis was done for the mean squared error (MSE). However, the proposed circuits can be used with any cost function when training with the backpropagation algorithm [27]. In this topology, synapses consist of two transistors and a single memristor, shown in Figure 4b, and structured in a synaptic grid as illustrated in Figure 4a. The weight of the synapse is represented by a single memristor, so the weight is a function, $w(s)$, of the memristor internal state s $(0 \leq s \leq 1)$. To achieve negative weight values, the synapse zero weight is defined for $s = 0.5$, so $w(s = 0.5) = 0$. This is done by adding a reference resistor, $R_{ref} = R_{mem}(s = 0.5)$ [18]. Therefore, the value of the weight is

$$w_{nm} = \frac{1}{R_{mem}} - \frac{1}{R_{ref}}. \tag{12}$$

To support negative weights, the reference resistors are added to each row, as suggested in [18] and shown in Figure 5. Each resistor is connected to a voltage source that matches the negative input voltage source of the corresponding synapse.

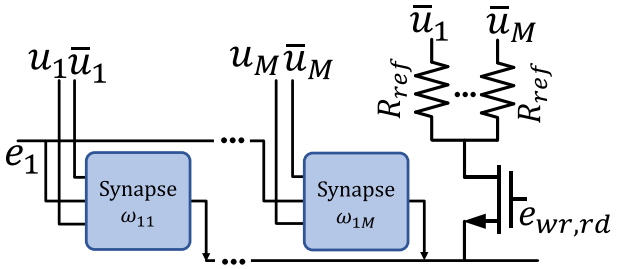


Fig. 5. Single row of a synapse array with M synapse. Each synapse has a reference resistors connected to the negative input source $\bar{u}_i$ of the matching synapse.

Thus, resistor i is connected to $v_i = \bar{u}_i$. Therefore, the current summed through the row is

$$I_{out,n} = \sum_{i=1}^{M} \left(\frac{1}{R_{mem,i}} - \frac{1}{R_{ref}}\right) u_i = \sum_{i=1}^{N} w_{ni} u_i. \tag{13}$$

Assume the state of the memristor $s(t)$ is restricted to relatively small changes around an operation point s^*. By linearizing the conductivity of the memristor around s^*, the conductivity of the memristor is

$$G(s(t)) = \bar{g} + \hat{g} \cdot s(t), \tag{14}$$

where $\bar{g} = G(s^*) - \hat{g} \cdot s^*$ and $\hat{g} = [dG(s)/ds]_{s=s^*}$. In this circuit, $\vec{u}, \vec{\bar{u}} \in \mathbb{R}^{1xM}$ are the input voltages, shared among all the synapses in the same column. A control signal $\vec{e} \in \mathbb{R}^{1xN}$ is shared among all the synapses in the same row. If $e_n = 0$, the current through the memristor is $I = 0$. If $e_n = \pm V_{DD}$, the voltage drop across the memristor is approximately $\pm u$, assuming that the transistor conductivity is significantly higher than the conductivity of the memristor. To support DNN functionality, the synapse circuits support two operation modes: read and write.

1) Read Mode: To perform the matrix-vector multiplication during inference, the input vector x is encoded to voltage as follows: $u_m = -\bar{u}_m = ax_m, v_{on} < ax_m < v_{off}$, where v_{on} and v_{off} are the threshold voltages of the memristor and a is a normalization factor controlled by the digital-to-analog conversion process. Thus, the read procedure is non-destructive. The current through all synapses in the same row is summed according to (11). By defining

$$w_{nm} = ac\hat{g}s_{nm}, \tag{15}$$

the current output at each row is

$$r_n = ac\hat{g}\sum_m s_{nm}x_m, \tag{16}$$

where c is a constant converting the results to a unit-less number, r_n, controlled by the analog-to-digital conversion process. Thus,

$$\vec{r} = W\vec{x}. \tag{17}$$

2) Write Mode (Update Weights/Training): In this mode, the inputs hold the same value as in the read mode, the control signal e_n changes so the value of u_m is selected to match $sign(y_n)x_m$, and the duration of the write interval matches $b|y_n|$, where b is a constant, in the order of magnitude of

$T_{rd/wr}$, converting $|y_n|$ to time units. Therefore, the update rule of the memristor state is

$$\Delta s_{nm} = \int_{T_{rd}}^{T_{rd}+b|y_n|} (asign(y_n)x_m)dt = abx_m y_n. \quad (18)$$

Using (15), the update value per synapse is

$$\Delta W = \eta \vec{y}\vec{x}^T, \quad (19)$$

which matches the vanilla SGD as determined in (5) for $\eta = a^2bc\hat{g}$.

III. Momentum Hardware Support Schemes

As described in Section II-B, the momentum algorithm adds a fraction of the past updates to the current update vector. Thus, in order to support this algorithm, the previous updates (*i.e.*, the history) must be preserved in some form of memory. In this section, two circuits for supporting the momentum algorithm are proposed. The designs are built upon the baseline synapse circuit [17]. For simplicity, the analysis shown in this section assumes a linear memristor model. In Section IV-B, we evaluate the proposed circuits using VTEAM, a realistic memristor model [21].

A. Supporting Momentum With Baseline Synapse Circuit

Keeping the history value in any form of memory will require at least the same number of memory cells as the number of synapses. Serial read-write operations will also be required to update the synapse value according to the momentum value. Thus, for a synapse layer of size $n \times m$, $n \cdot m$ memory cells and m read and write operations will be needed. Rather than keeping the value of the momentum for each synapse, we suggest keeping the value of the vector pair $\{\vec{x}, \vec{y}\}$: the input and error vectors, for each layer and iteration as determined by (9). Although keeping all of the $\{\vec{x}, \vec{y}\}$ pairs for each layer from all of the previous updates is impractical, storing only the pairs from the recent iterations may be sufficient. Since the momentum algorithm adds a fraction, $\gamma < 1$, of the past updates to the current update vector, the contribution of the gradients from previous computations decays with each iteration. Assume that the contribution of the previous updates becomes negligible after the following h iterations. Under this assumption, we can ignore the contribution of the sum elements that are multiplied by factor $\gamma^n < \gamma^h$ to the total update sum. With that in mind, we propose to store only the pairs from the last h iterations that contribute to the momentum value. For the rest of the paper, we refer to this solution as the "Limited History" solution. Thus, (9) can be written as

$$\Delta w_{nm,history}(k-1) \approx \eta \sum_{j=k-h}^{k} \gamma^{k-j} x_m^{(j)} y_n^{(j)}. \quad (20)$$

For this solution, we use a system architecture composed of tiles, each containing several synapse grids, an SRAM-based memory (referred to as external memory), and a digital computation unit, as shown in Figure 6. A single DNN layer can be stored in one or more synapse grids. The activation of each layer is computed in the computation unit and then sent to the synapse grid, which stores the following layer.

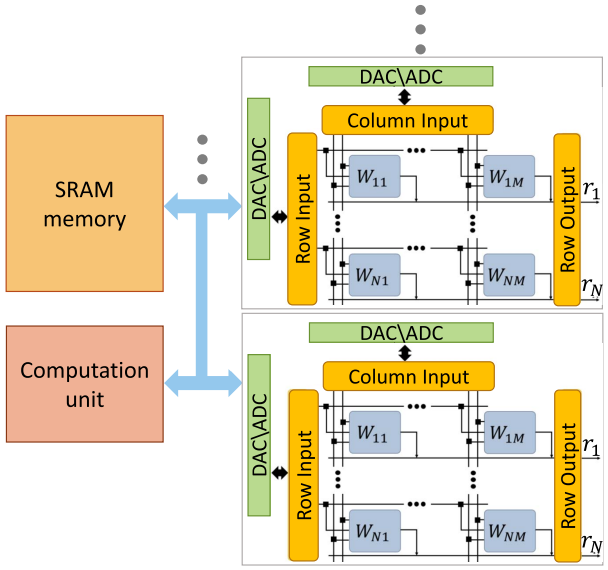


Fig. 6. Limited history solution. External memory is used to store the history of the updates in order to support the momentum algorithm. Several synaptic grids are used to store the different DNN layers. The neuron computation unit is used to compute the values of the neurons from the synaptic grid data. The neuron values are passed between two synaptic grids through the computation unit.

While the read mode remains unchanged from vanilla SGD, the write mode requires a new algorithm, described in Algorithm 1. The h pairs of $\{\vec{x}, \vec{y}\}$ are read sequentially from the external memory, one pair at a time, from the newest to the oldest. For iteration i, the $\vec{y}$ values are multiplied by γ^i. The $\{\vec{x}, \gamma^i\vec{y}\}$ data is mapped into the matching synapse grids, so $\vec{x}$ is mapped to the grid's column interface and $\vec{y}$ to the row interface. Then the weights are updated based on the vanilla SGD algorithm as in Section II-D.2. Thus,

$$\Delta W = \eta\gamma^i \vec{y}\vec{x}^T, \quad (21)$$

and after h iterations,

$$\Delta W = \eta \sum_{j=k-h}^{k} \gamma^{k-j} \vec{y}^{(j)}(\vec{x}^{(j)})^T. \quad (22)$$

After h iterations, the gradient value is updated. The memory arrays that hold the $\{\vec{x}, \vec{y}\}$ pair are shifted left so that the oldest pair is removed and the current $\{\vec{x}, \vec{y}\}$ pair is stored in the memory.

This solution uses a simple synapse circuit and can be supported by any memristor-based solution for DNNs that contains a memory structure sufficiently large to store all the momentum algorithm history data. The drawback of this solution is the need to modify the momentum algorithm, add external memory, and move data between the synaptic array and the memory, which might lead to greater energy consumption and higher latency. The solution is also sensitive to the γ value: for γ closer to 1, the number of pairs that have to be stored increases. The effect of the modified algorithm on the network performance is investigated in Section IV.

Algorithm 1 Limited History. h Pairs of $\{\vec{x}, \vec{y}\}$ Are Stored in Memory Arrays $\{X, Y\}$ External to the Synapse Array

1: Gradient update, according to Section II-D.2
2: **for** i = 1,2...,h **do**
3: Read $\vec{X}(i), \gamma^i \vec{Y}(i)$ pair from external memory
4: Map $\{\vec{X}(i), \gamma^i \vec{Y}(i)\}$ to column and row interface
5: Update synapse weights, as described in Section II-D.2.
6: Multiply $\gamma^{i-1} \vec{Y}(i-1)$ with γ at the computation unit
7: **end for**
8: Remove "oldest" $\{\vec{X}, \gamma^h \vec{Y}\}$ pair and store new $\{\vec{X}, \gamma \vec{Y}\}$ pair in external memory arrays

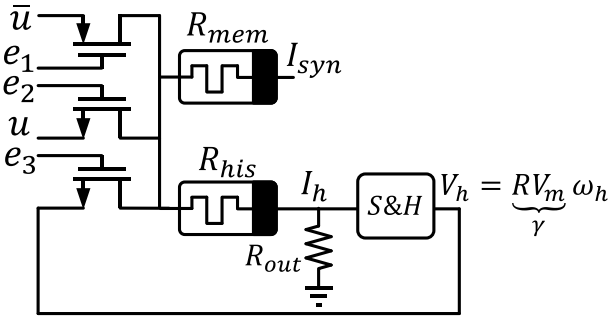


Fig. 7. Internal memory solution. The synapse holds an additional memristor as an internal memory to store the accumulative history of the previous updates.

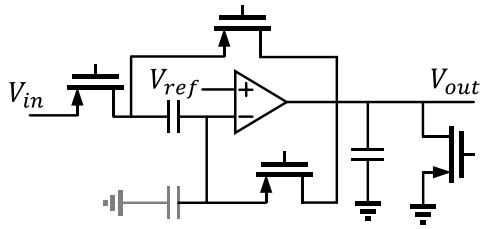


Fig. 8. Schematic of the Sample and Hold circuit [28].

B. Supporting Momentum With a Dedicated Synapse Circuit

The momentum information is the weighted sum of all the previous updates, as described in (9). Rather than using an "external" memory, we investigate adding more memory capabilities to the synapse circuit, such that the accumulative value is stored by an additional memristor. Thus, each synapse is constructed from two memristors, marked as R_{mem} and R_{his}, as shown in Figure 7. While R_{mem} holds w, the current value of the weight, R_{his} holds w_h, the sum value from (9) and acts as a private memory of the synapse. Thus, a simple sample and hold (S&H) circuit is added in order to read and keep the stored data from R_{his} [28]. The S&H circuit is shown in Figure 8. Additionally, two transistors and a reference resistor R_{ref} are added to control the new functionality of the synapse. As mentioned in Section II-D, the conductivity is assumed to be linearized around s^* and the weight of the synapse is equal to zero for s^*. Therefore, another reference resistor R_{ref} is added to the new memristor path to subtract the operation point when the momentum value is read from R_{his} [18]. Since the weight is stored in R_{mem}, read and write operations are performed with respect to R_{mem}. The working modes of the synapse deviate from the description in Section II-D as follows:

Algorithm 2 Write Operation for the Internal Memory Solution

1: Read w_h, feed into the Sample and Hold unit
$e_1 =$'1', $e_2 =$ '0', $e_3 =$ '0'
$u = V_{mom}$
$I_{nm} = a(\bar{g} + \hat{g}s_{nm})V_{mom}$
$I_{h,nm} = a\hat{g}s_{nm}V_{mom}$
$V_{h,nm} = R_{out}I_{h,nm} = RV_{mom}w_{h,nm} = \bar{\gamma}\, w_{h,nm}$
2: Reset $w_h = 0$:
$u = \begin{cases} V_{reset} & t_0 <= t <= t_1 \\ V_{set} & t_1 < t <= t_2 \end{cases}$
$e_1 =$ '1', $e_2 =$ '0', $e_3 =$ '0'
$\Delta s_{nm} = \int_{t_0}^{t_1} V_{reset}dt \Rightarrow s = 0$
$\Delta s_{nm} = \int_{t_1}^{t_2} V_{set}dt \Rightarrow s = 0.5$
3: Accumulate w and w_h by $\gamma\, w_h$
$e_1 =$ '0'
$e_2 = e_3 = \begin{cases} \text{'1'} & t_2 < t \le t_2 + \hat{\gamma} \\ \text{'0'} & t_2 + \hat{\gamma} < t \le t_3 \end{cases}$
$\Delta s_{nm} = \int_{t_2}^{t_2+\hat{\gamma}} \bar{\gamma}\, w_{h,nm}dt = \hat{\gamma}\,\bar{\gamma}\, w_{h,nm}$
$\Delta w_{nm} = ac\hat{\gamma}\,\bar{\gamma}\, w_{h,nm} = \gamma\, w_{h,nm}$
4: Write gradient to w and w_h $e_1 = e_2 =$ '1', $e_3 =$ '0'
$\Delta s_{nm} = \int_{T_{rd}}^{T_{rd}+b|y_n|} (asign(y_n)x_m)dt = abx_m y_n$
$\Delta W = a^2bc\hat{g}yx^T = \eta\vec{y}\vec{x}^T$

1) Read Mode: Setting $e_2 =$ '1' and $e_3 =$ '0' to avoid affecting R_{his} during the read operation.

2) Write Mode (Update Weights): The write mode is described in Algorithm 2. Before updating the weights, the value of w_h is read and stored in the S&H circuit. The input voltage per column is set to $u = V_{mom}$, where V_{mom} is a constant voltage level that is used while reading w_h. Thus, the current that passes through $w_{h,nm}$ is

$$I_{nm} = a(\bar{g} + \hat{g}s_{nm})V_{mom}. \tag{23}$$

After subtracting the operation point, the current value is

$$I_{h,nm} = a\hat{g}s_{nm}V_{mom}. \tag{24}$$

Using the output resistor R_{out}, the current I_h value is converted to voltage. Hence, by defining $\bar{\gamma} = R_{out}V_{mom}$, the voltage value stored in the S&H is

$$V_{h,nm} = \bar{\gamma}\, w_{h,nm}. \tag{25}$$

From (25), V_h holds the value of the history multiplied by factor $\bar{\gamma}$. Before the update stage, w_h is set back to zero, as can be seen in step 2 of Algorithm 2. The next step is to update both w and w_h with $\gamma\, w_h$. This is done in a manner similar to the write mode described in Section II-D.2, although now the voltage source is connected to V_h, meaning that $e_1 =$ '0', $e_2 = e3 =$ '1'. The duration of the update phase is set by $\hat{\gamma}$. Therefore,

$$\Delta s_{nm} = \int_{t_2}^{t_2+\hat{\gamma}} \bar{\gamma}\, w_{h,nm}dt = \hat{\gamma}\,\bar{\gamma}\, w_{h,nm} \tag{26}$$

and

$$\Delta w_{nm} = ac\hat{g}\hat{\gamma}\,\bar{\gamma}\,w_{h,nm} = \gamma\,w_{h,nm}, \tag{27}$$

where $\gamma = ac\hat{g}\hat{\gamma}\,\bar{\gamma}$. Thus, we obtain that

$$\begin{aligned} w_{h,nm} &= \gamma\,w_{h,nm}, \\ w_{nm} &= w_{nm} + \gamma\,w_{h,nm}. \end{aligned} \tag{28}$$

The last step is performed on the gradient value, as described in Section II-D.2, except that both w and w_h need to be updated. Using the updated value given by (19), we obtain that

$$\begin{aligned} w_{h,nm} &= \eta x_m y_n + \gamma\,w_{h,nm}, \\ w_{nm} &= w + \eta x_m y_n + \gamma\,w_{h,nm}. \end{aligned} \tag{29}$$

So,

$$\begin{aligned} W_h &= \eta\vec{y}\vec{x}^T + \gamma\,W_h, \\ W &= W + \eta\vec{y}\vec{x}^T + \gamma\,W_h, \end{aligned} \tag{30}$$

where W_h is the synapse history matrix.

The use of internal memory allows the maximum information used by the momentum algorithm to be kept; it allows each synapse to track the full update history by saving the value of the sum of updates. The data is kept per synapse, thus reducing data movement out of the synapse, which should reduce latency and energy. The cost of this solution is the addition of read and write cycles to the update phase and the S&H energy consumption. Furthermore, the internal memory requires extra area and a more complex controller.

C. Qualitative Comparison of the Proposed Solutions

We performed a qualitative comparison of the accuracy, latency, power, and area of the two proposed solutions.

1) Accuracy: The "Limited History" solution modifies the momentum algorithm by changing the update rule. Therefore, the convergence behavior and the classification accuracy rates are expected to be less than the rates of the "internal memory" counterpart solution.

2) Latency and Power: The use of external memory results in a longer weight-update stage as compared to the internal memory solution, due to sequential access to this memory. However, in the second solution, more energy is consumed per synapse access due to the circuit complexity. Therefore, the optimal solution depends on the history depth h, and on the neuron precision.

3) Area: The synapse cell area limits the number of synapse cells per synaptic grid. Thus, it defines the number of synapse grids needed to match the DNN layer.

a) Limited history: This solution uses the baseline synapse. Thus, the synapse contains only two transistors. However, as described in Section III-A, an external memory is used to hold h pairs of input-error vectors per layer. Therefore, the memory area overhead must be considered as well. Consider the following simplified example, assuming that the area per memory cell is similar for both solutions. The internal memory solution holds extra memory circuits per synapse. Thus, for $n \times m$ synapse matrix, $n \cdot m$ memory circuits are required. For the same matrix dimensions, the external memory solution requires $h(n+m)$ memory circuits. Therefore, for large layers and a given h whose value is $h < \frac{nm}{n+m}$, the area overhead of the external memory solution is lower.

b) Internal memory: Our design uses a simple S&H circuit for the internal memory. S&H circuits, even the simplest one, will use sense amplifiers and complex circuit elements. Thus, this approach will significantly increase the synapse circuit area and reduce the capacity of the synapse array.

IV. Evaluation

In this section, we verify the compatibility of the proposed circuits to the momentum algorithm, and evaluate their computation runtime and energy consumption. To validate that our designs are compatible with the original momentum algorithm, we compare the performance (*i.e.*, accuracy) of the modified momentum algorithm to the original momentum algorithm implemented in software. Additionally, we compare the training performance of the memristor-based designs to the performance of the software implementation. We extract the energy and latency of the proposed circuits and compare their total runtime and energy consumption to the runtime and energy required when training on a GPU.

A. Circuit Evaluation

1) Methodology: The system described in Section III-A is used to simulate and evaluate the proposed solutions. The analog values of the weights for each NN layer are stored in a single synaptic grid. To move the synapse and neuron values from and to the synaptic grid, 16-bit ADCs and DACs are used. Hence, the neurons are represented by 16-bit resolution while the weights keep their full resolution. The neuron activation is determined by the computation unit. For the limited history solution, SRAM is used to store the h history pairs of all layers. The simulated system is non-pipelined and at each iteration a single input is computed.

To extract the power and operation latency of the two proposed circuits, both solutions were designed and evaluated using Cadence Virtuoso in 180nm CMOS process. To fit the GPU platform technology and allow a fair comparison, we scaled down the results to 16nm using the model proposed in [29], which fits polynomial equations to accurately evaluate the scaling factor. To simulate the memristor we used the VTEAM model [21]. The parameters of the VTEAM device were selected to simulate a symmetric, non-linear memristor with high R_{off}/R_{on} ratio. Due to the short write intervals, the non-linearity is important for significant transitions of the memristor state. Memristors are back-end-of-line (BEOL) to the CMOS process, and as such they can be modeled independently of the CMOS node. Thus, the memristor model was not affected by the process scaling. The circuit parameters are listed in Table I. For the peripheral ADC and DAC, 1.55 GSps ADC converters were used [30] with power consumption of 3.83mW. The DAC operates at the same frequency as the ADC with power consumption of 2.92mW [31]. For the internal memory solution, the S&H circuit was implemented in VerilogA, with power consumption of 4nW and area consumption

TABLE I
CIRCUIT PARAMETERS FOR SPICE SIMULATIONS FOR 180nm CMOS PROCESS

Parameter	Value	Units	Parameter	Value	Units	Parameter	Value	Units
Memristor			Power			Timing		
R_{off}	$200K$	Ω	V_{DD}	1.8	V	T_{read}	1	μsec
R_{on}	100	Ω	V_{EE}	-1.8	V	T_{write}	5	μsec
k_{off}	0.01	m/sec	V_{in}	$[-1.24, 1.24]$	V	T_{reset,w_h}	2	μsec
k_{on}	-0.01	m/sec	PMOS			Circuit		
v_{off}	0.1	V	W/L	20	–	$r_{line}*$	0.15	Ω/μ
v_{on}	-0.1	V	V_T	-0.57	V	$c_{line}*$	0.2	$fF/\mu m$
α_{off}	3	–	NMOS			R_{out}	$1K$	Ω
α_{on}	3	–	W/L	10	–	R_{ref}	$100.05K$	Ω
f	$s \cdot (1-s)$	–	V_T	0.56	V			

TABLE II
SRAM CONFIGURATION USED BY THE LIMITED HISTORY SOLUTION. EACH NN ARCHITECTURE HAS ITS OWN SRAM, FITTED TO THE DIMENSIONS OF THE LAYERS

SRAM		
Block Size		64 Bytes
Area Per NN		
MNIST1	16KB	0.026mm^2
MNIST2	64KB	0.091mm^2
MNIST3	128KB	0.143mm^2

TABLE III
DNN ARCHITECTURES. EACH NUMBER REPRESENTS THE NUMBER OF NEURONS PER LAYER

Layer	MNIST1	MNIST2	MNIST3
1	400	400	400
2	100	1000	1000
3	10	100	500
4	SoftMax	10	100
5	–	SoftMax	10
6	–	–	SoftMax

of $0.016\mu m^2$ as in [28]. The additional S&H circuit has little effect on the circuit transient response and delay due to the relatively long time intervals of the synapse circuit and the low capacitance. The time intervals used in this work are long enough to guarantee that the circuit parasitics will have little effect. For future work, further optimization and performance can be gained by increasing the frequency, when the circuit parasitic becomes dominant. For the limited history solution, each neural network had its own SRAM-based memory, best suited to its layers and history size. The configuration of the SRAM-based memory per NN is listed in Table II and was modeled using NVSim [32].

2) Area Overhead: The area overhead was compared for three DNN architectures listed in Table III. To compare the area overhead of both solutions, we focus on the area overhead of the synaptic grid and SRAM. We do not consider the area of the ADC and DAC, which are identical for both solutions. We assume $h = 10$, which provides promising results, as shown later in Section IV-B. The S&H circuit added $0.0163\mu m^2$ to each synapse of the internal memory solution. For the limited history solution, the SRAM area is listed in Table II. The minimum required area overhead of both solutions as compared to the baseline synapse is shown in Figure 9. The extra S&H circuit added to each synapse in the internal memory solution increases the area of the synaptic array by $9.45\times$. In contrast, the relative area overhead of the limited history solution can reach two orders of magnitude over the baseline synapse due to the SRAM area. This overhead substantially decreases for large networks, as discussed in Section III-C.3.a and shown in Figure 10.

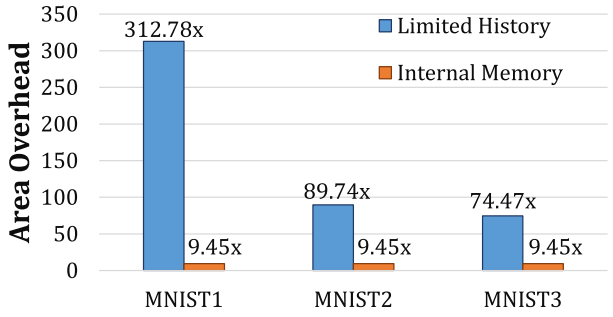


Fig. 9. Area overhead of the momentum algorithm circuits as compared to the baseline synapse area.

B. Verifying the "Limited History" Momentum Algorithm

To verify that the accuracy of the modified momentum algorithm is sufficient, we trained two DNN architectures implemented in PyTorch on the MNIST data-sets: MNIST2 (Table III) and PyMNIST [33]. In addition, we trained the Resnet 18 [26] and ResNext [34] architectures on the CIFER10 and CIFER100 data-sets [35]. All the networks were trained with momentum and $\gamma = 0.9$. Table IV shows the performance of the limited history algorithm as compared to the original algorithm and training without momentum. As expected, the accuracy of the nets is higher when trained with momentum. The limited history algorithm with $h = 10$ reached similar results as compared to the original momentum algorithm.

The convergence is defined as the iteration after which the cost function does not exceed 0.3, which corresponds to a probability greater than 0.5 for a correct output by the SoftMax function. The error rate — the percentage of inference errors in the test phase — and convergence of the limited history approach depend on the length of the history kept for each layer, *i.e.*, the value of h. Figure 11 shows the number of

TABLE IV

ACCURACY OF LIMITED HISTORY MOMENTUM AS COMPARED TO THE ORIGINAL MOMENTUM ALGORITHM. FOR THE MNIST2 NET, THE NUMBER OF ITERATIONS UNTIL CONVERGENCE (WHEN THE COST FUNCTION IS LOWER THAN 0.5) IS LISTED IN BRACKETS, AND IT IS HIGHER FOR THE LIMITED HISTORY. FOR DEEPER NETWORKS, THE LIMITED HISTORY OBTAINS RESULTS SIMILAR TO THOSE OF THE ORIGINAL ALGORITHM

Net	Dataset	No Momentum	Original Momentum	Limited History Momentum	h	Epochs
MNIST2	MNIST	89.59%	91.57% (14900)	90.29% (37700)	10	1
PyMNIST	MNIST	98%	99%	99%	10	10
Resnet 18	CIFER10	89.59%	91.53%	91.18%	10	200
ResNext	CIFER10	94.48%	96.36%	96.36%	10	300
ResNext	CIFER100	77.28%	82.13%	81.27%	10	300
ResNext	CIFER100	77.28%	82.13%	81.67%	20	300

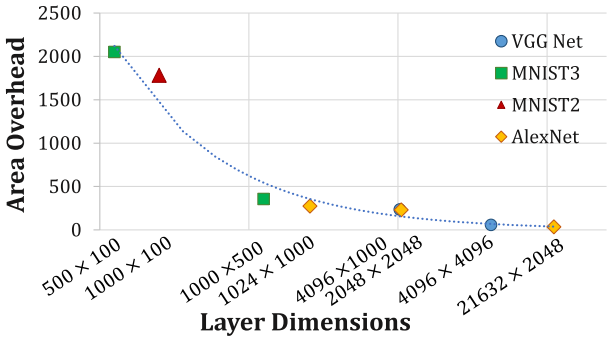


Fig. 10. Area overhead of the limited history circuit for different layer dimensions. The overhead is lower for larger layers. The layer dimensions are taken from the AlexNet, VGG Net, MNIST2, and MNIST3 (Table III) architectures. For each layer, the best suited SRAM size is used. The area was modeled using NVSim [32]. In practice, the SRAM can be shared between several layers, thus further reducing the area overhead.

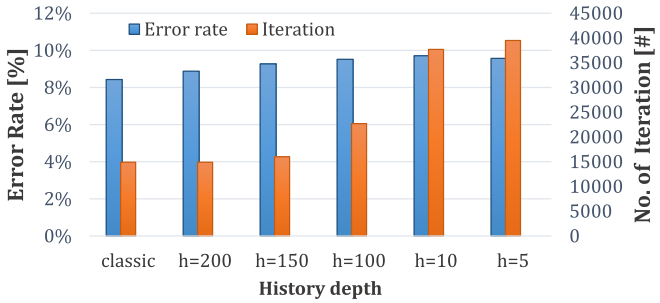


Fig. 11. Limited history momentum algorithm performance (error rate) and the number of samples until convergence for MNIST2 with different history values h.

iterations until convergence and the testing error rate for the MNIST2 network for different history values. For longer history, the performance of the network approaches the classic momentum performance with regard to convergence and error. This can also be seen in Table IV for ResNext trained on the CIFER100 dataset. Nevertheless, the history length determines the number of write operations per weight update and the size of the memory needed to support the algorithm. Thus, the weight update period increases for a longer history.

Note that like the RRAM-based accelerators, CMOS-based accelerators have limited memory capacity; thus, CMOS-based accelerators and commodity hardware platforms can also use the limited history principle.

C. DNN Hardware Simulations

To evaluate the proposed hardware solutions, we designed the DNN and simulated the training in MATLAB, using circuit parameters extracted from SPICE simulations. We evaluated the accuracy of the circuits and their compatibility to the original momentum algorithm. Although numerous DNN hardware solutions have been proposed in recent years (*e.g.*, DaDianNao [12], PipeLayer [19] and TIME [20]), these ASIC-based solutions lack the computational capabilities to execute training with momentum. Therefore, to evaluate the potential of the proposed circuits, we compare their performance to that of GPU, which can support momentum training.

Methodology: The simulations were conducted using the Intel Xeon E5-2683 2GHz processor [36] and the MATLAB 2017a environment. Three DNNs were designed using the MNIST data-set [37], as listed in Table III. The MNIST data-set includes 60000 examples for training and 10000 examples for testing. The ReLU activation function was used by the hidden layers, SoftMax at the output layer, and cross-entropy as the cost function. The DNNs were trained over several epochs — one iteration over the training data-set — as shown in Table V, after which the performance does not change significantly. For each DNN architecture simulation, the error rate was compared to the error rate of the software implementation of the original momentum algorithm for the same networks. The power consumption and runtime of the training stage were evaluated using the results extracted from the circuit evaluation. Both energy and computation runtime were evaluated with a focus on the synapse-related computation and compared to the same computation on the Nvidia GeForce GTX 1080 Ti GPU, 1.6GHz. To evaluate the GPU performance, the three DNNs were implemented using PyTorch [33], and their energy consumption and runtime were computed using Python's time counters and the nvidia-smi tool provided by NVIDIA CUDA [38].

1) Compatibility With the Momentum Algorithm: The metrics used for comparison are convergence and error rate. Figure 12 shows the convergence of MNIST1 training for all implementations. The convergence trend is similar for all the implementations, and the inset shows that they all converged to similar values. Table V lists the testing phase error, where both hardware designs show surprisingly better results than

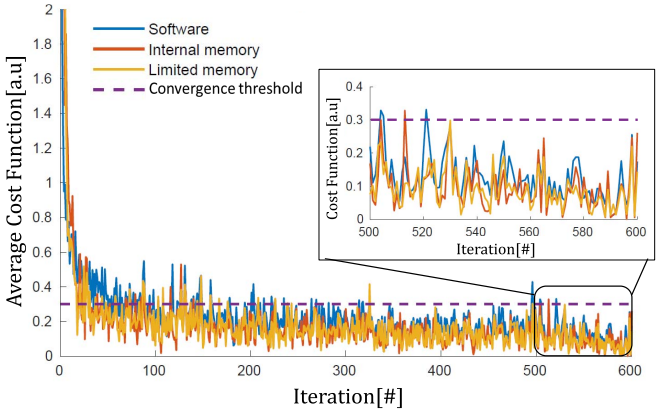


Fig. 12. Training stage cost function value for the three implementations of MNIST2 – internal memory, limited history, and software. The cost function value was averaged over a set of 100 iterations for training. For smaller values of the cost function, the probability given to the correct output increases. When the cost function does not exceed the dashed line, we consider the network as converged.

TABLE V
TEST PHASE AVERAGE ERROR FOR THE SOFTWARE IMPLEMENTATION AND THE TWO PROPOSED HARDWARE SOLUTIONS

Error Rate [%]				
NN Arch.	Software	Int. Mem	Lim. His	#Epochs
MNIST1	5.46	4.02	4.96	6
MNIST2	5.2	4.01	3.97	6
MNIST3	7.8	3.42	2.78	4

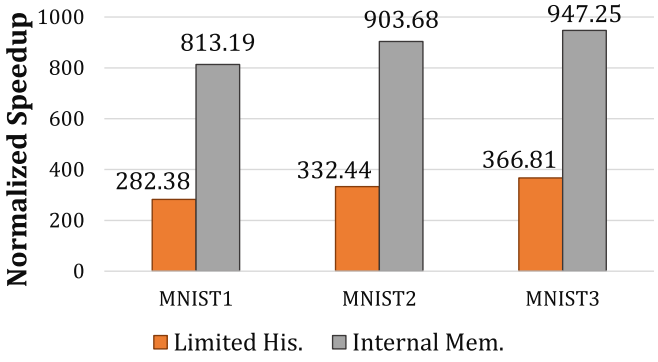


Fig. 13. Computation runtime speedup of the synaptic grid normalized to GPU runtime.

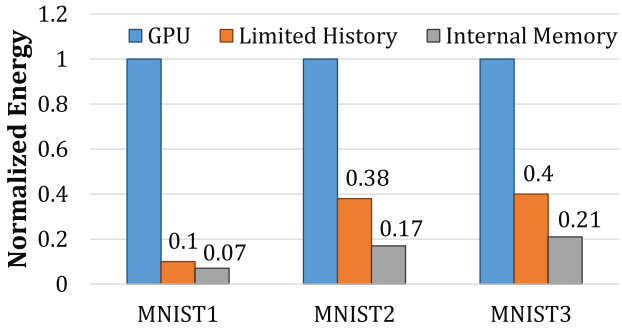


Fig. 14. Dynamic energy consumption of each DNN architecture using the proposed synapse circuits vs. GPU energy consumption.

the software implementation. The differences in the results between the software and hardware implementations are due to the dependency of the values of η and γ on the memristor state, which changes during the training phase [18]. This dependency causes the η and γ values of each individual memristor to decay when the $|\omega|$ value increases [18]. The behavior described above resembles that of training methods such as ADAM and NADAM, which use the adaptive learning rate and adaptive momentum, and are known to improve performance [8], [11], [39]. The limited history is a simplified version of the momentum algorithm; nevertheless, it shows good training performance, which surpasses, for the MNIST2 and MNIST3 architectures, the performance of the internal memory solution.

2) Computation Runtime: Figure 13 shows the computation runtime speedup versus the GPU runtime for both hardware solutions. The limited history runtime is approximately 325× faster than GPU execution, where the internal memory solution exhibits greater speedup (886×). The performance speedup of the proposed solutions is attributed to their high parallelism and the proximity of the new momentum data to the synaptic grid. The proposed solutions are less sensitive to the size of the neural network layers.

3) Energy Consumption: The energy consumption for each hardware solution normalized to the GPU energy consumption is shown in Figure 14. For both solutions, the ADC and DAC accesses are major energy consumers, as illustrated in Figure 15, for the MNIST3 architecture. For the limited history solution, the energy consumption of the synaptic grid and SRAM accesses is negligible as compared to that of the ADC and DAC. However, the low activity of the SRAM makes it a significant consumer of static energy. As shown in Figure 15, the SRAM static energy is approximately 64% of the total energy consumption. A careful design of the memory hierarchy is required to reduce the static energy. For example, we might consider replacing the SRAM with RRAM or other emerging memory technologies such as MRAM [40]. Thus, for larger layers and shallow history depth, the energy efficiency of the internal memory solution is lower than the external memory solution.

4) Process Variation: The limited memory solution uses the baseline synapse, which demonstrated high robustness to process variation [17], [18]. The internal memory synapse is more complex than the baseline solution; therefore, we evaluate its robustness to process variations. As mentioned in section IV-A, the S&H circuit and the transistors have little effect on the circuit response; therefore, we focused the evaluation on the memristor parameters.

a) Memristor process variation: To evaluate the robustness of the internal memory approach in the presence of process variations, we simulated training with variations of the memristor device. The average error rate of the MNIST1 network was evaluated over one epoch, for several Gaussian distributions with different Coefficients of Variation (CV= (standard deviation)/mean). We considered variations in

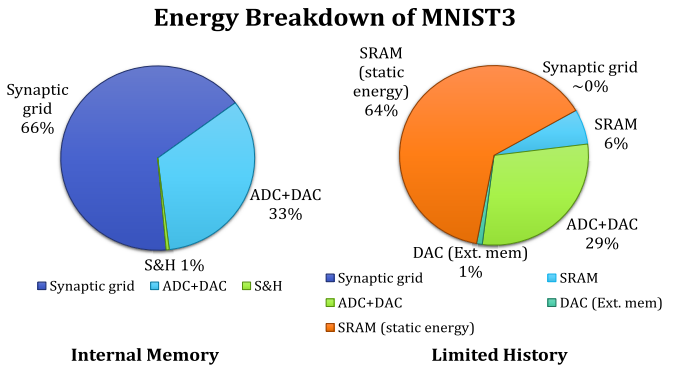


Fig. 15. Energy breakdown for MNIST3. The DAC (Lim. his.) represents the energy consumed by the data conversions of the h X, Y pairs stored in SRAM.

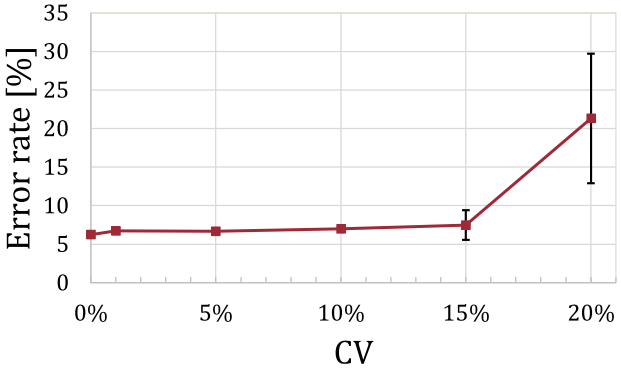


Fig. 16. MNIST1 error rate for variation of the memristor parameters.

the device resistance ($R_{off/on}$), threshold voltages ($V_{off/on}$), and the device dynamic response ($\alpha_{off/on}$ and $K_{off/on}$). Each iteration the memristor parameters were sampled from Gaussian distribution with the same CV. Figure 16 shows the average error rate for different values of CV. The training performance is relatively robust to process variations by up to CV= 15%, when most of the sensitivity is due to variations in the threshold voltages of the memristor. For large variations of the threshold voltage (over 20%), there are many memristors which are active even for a zero voltage drop, causing their state to change. Additionally, there are many memristors with a high threshold voltage, so their state will not change for low voltage drop.

b) Environmental parameters variation: A non-ideal power grid might lead to variability in the voltage source. A serially connected resistor models this variability. Figure 17 shows the training performance for range of source resistances, which is robust against changes in the source variations. Also, Figure 17 shows that the accuracy improves for higher source resistances. When the resistance is higher, the voltage drop on the memristor decreases and leads to a smaller change in the memristor state (smaller update steps). This behavior can be looked at as a learning rate, where in this case smaller learning rate improves the results. However, in other DNN architectures, lower learning rate might lead to suboptimal results. Variation in the temperature changes the R_{off} and R_{on} resistance. At high temperature, R_{off} decreases and R_{on} increases, while the change of R_{off} is more significant than of the change of R_{on} [41]. Furthermore, the memristor

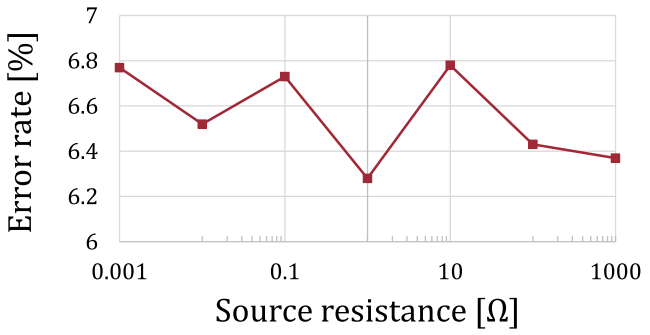


Fig. 17. The effect of variations in the voltage source on MNIST1 error rate.

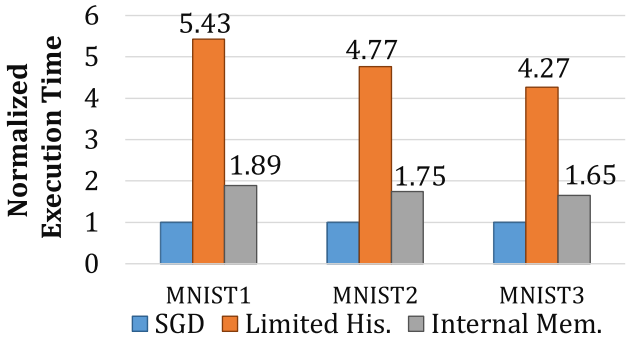
TABLE VI
THE TEMPERATURE EFFECT ON MNIST1 ERROR RATE. THE VALUES IN THE TABLE SIMULATES THE THEORETICAL BEHAVIOR. THE ROOM TEMPERATURE COLUMN IS THE BASELINE AND GIVEN IN THE BOLD

Parameter	Low Temp.		Room Temp.	High Temp.	
$\mathbf{R_{on}}$	95Ω	99Ω	$\mathbf{100\Omega}$	101Ω	105Ω
$\mathbf{R_{off}}$	$210K\Omega$	$202K\Omega$	$\mathbf{200K\Omega}$	$198K\Omega$	$195K\Omega$
$\mathbf{K_{on}}$	-0.95	-0.99	$\mathbf{-1}$	-1.01	-1.05
$\mathbf{K_{off}}$	0.95	0.99	$\mathbf{1}$	1.01	1.05
Error rate	17.32	7.11%	**6.23%**	6.66%	9.62%

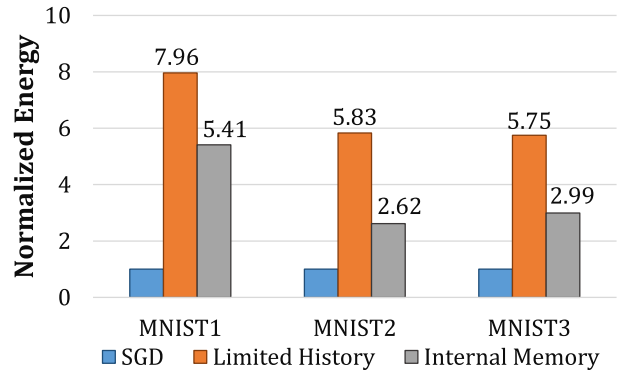
dynamics depends on the temperature. In the VTEAM model, this change is expressed by $K_{off/on}$, which increases for high temperature. We illustrated the temperature effect on the training performance. Table VI lists the training performance for values which simulate the projected behavior. The training is sensitive to temperature variation, and the performance might degrade due to large variations. Therefore, the RRAM temperature response is a critical consideration for selecting a RRAM device for training. This response varies among different RRAM devices and need to be characterized and optimized for training.

5) Comparison to Baseline Synapse: The baseline synapse supports momentum using the limited history method. As shown in Figure 15, the limited history algorithm contributes approximately 71% of the total energy consumption. When considering only the dynamic energy, the limited history algorithm contributes only 7%. Executing training using the memory solution consumes on average 0.46× lower energy than the limited history solution. The internal memory solution is approximately 3× faster than the limited history solution. The latency of the limited solution can be reduced by designing a pipelined system which hides the memory access time. Although our design is based on [17], the proposed circuits can be adopted with minor modifications for use in other memristor-based circuits, such as the one in [15]. Therefore, we expect similar results from other memristive-based solutions.

6) Momentum Hardware Overhead: The added momentum hardware leads to higher power consumption and longer runtime. In Figure 18, the overhead of executing the momentum algorithms is compared to the training with vanilla SGD using the baseline synapse. As shown Figure 18a, for the same number of epochs, executing the momentum algorithm



(a) Momentum execution time normalized to the SGD execution time.



(b) The momentum dynamic energy consumption normalized to the SGD.

Fig. 18. Momentum hardware performance compared to SGD hardware performance. (a) Computation runtime speedup of the momentum algorithm normalized to SGD runtime. (b) Dynamic energy consumption of the momentum algorithms vs. SGD energy consumption.

increased the runtime over SGD. Similarly, Figure18b shows the momentum energy consumption overhead over SGD. The momentum algorithm executed with the internal memory solution increases the run-time by approximately 2×, and the energy consumption by approximately 4×. The momentum algorithm executed with the limited history solution increases the run-time by approximately 5×, and the energy consumption by approximately 6.5×. The area overhead of the momentum hardware compared to the SGD hardware (baseline) is given in Figure 9. Although executing momentum increases the area, power consumption and execution time, this is the cost for improving training accuracy and accelerating the convergence. Supporting the momentum algorithm with accelerators such as DaDianNao [12], PipeLayer [19], and TIME [20] requires keeping the momentum value for each weight of the DNN (we refer to this solution as "Full History"). Therefore, the memory capacity and the number of memory accesses dramatically increase. To compare the overhead of momentum with the full history approach to that of the other momentum-based solutions presented in this work, we considered an example network with a single 256 × 256 layer, a 64 byte SRAM cache line, 16-bit data size, and a history depth of 16. (For the full history approach, the momentum value is computed and stored in the SRAM). Table VII lists the overhead for the solutions suggested in this work compared to the full history. In this example, the full history approach consumes 58× and 3.5× more energy than the internal memory and limited history solutions, respectively. Also, the full history update duration is 848× and 6× longer than the internal memory and limited history solutions, respectively.

TABLE VII
QUALITATIVE COMPARISON TO OTHER NN HARDWARE SOLUTIONS

	Int. mem	Lim. His	Full His.
Memory overhead	64K[1]	16KB	256KB
No. Update Cycles	1	128	4096
Update duration	0.5ns	71ns	424ns
Area overhead	1.04μm^2	0.026mm^2	0.488mm^2
Energy overhead	4.53nJ	72.9nJ	261nJ

[1] The memory overhead refers to the synapse circuit.

V. CONCLUSIONS

Neural networks with memristor-based synapses pose challenges when supporting various training algorithms. In this paper, support of the momentum algorithm was investigated. Since memory abilities are necessary to support the momentum algorithm, the basic memristor-based hardware synapse, generally implemented within a crossbar, needs to be modified. Two circuit solutions were suggested. The first is a hardware-friendly modification of the momentum algorithm, which has a negligible effect on the training accuracy. The second proposed solution relies on an additional memory circuit within the basic synapse circuit to support the original momentum algorithm. Both solutions were shown to speed up the training runtime (325× and 886× respectively) and improve the energy consumption (4× and 7× respectively) as compared to GPU execution.

Further research must be done to design synapses that are more flexible and to define the working mode of memristor-based DNN accelerators. In future work, we intend to support widely used state-of-the-art training algorithms. The synapse circuit will be used as the foundation of future full system architectures, dedicated to working with DNN workloads.

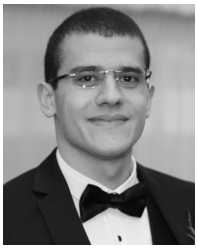
REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, Dec. 2012, pp. 1097–1105.
[2] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proc. 25th Int. Conf. Mach. Learn.*, Jul. 2008, pp. 160–167.
[3] I. Kaastra and M. Boyd, "Designing a neural network for forecasting financial and economic time series," *Neurocomputing*, vol. 10, no. 3, pp. 215–236, 1996.
[4] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Proc. Adv. Neural Inf. Process. Syst.* Red Hook, NY, USA: Curran Associates, Inc., 2010, pp. 2595–2603.
[5] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Proc. Adv. Neural Inf. Process. Syst.* Red Hook, NY, USA: Curran Associates, Inc., 2013, pp. 315–323.
[6] T. Zhang, "Solving large scale linear prediction problems using stochastic gradient descent algorithms," in *Proc. 21st Int. Conf. Mach. Learn.*, 2004, Art. no. 116.
[7] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Proc. Adv. Neural Inf. Process. Syst.* Red Hook, NY, USA: Curran Associates, Inc., 2011, pp. 693–701.
[8] S. Ruder. (Sep. 2016). "An overview of gradient descent optimization algorithms." [Online]. Available: https://arxiv.org/abs/1609.04747
[9] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural Netw.*, vol. 12, no. 1, pp. 145–151, 1999.

[10] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, Feb. 2011.
[11] D. P. Kingma and J. Ba. (Dec. 2014). "Adam: A method for stochastic optimization." [Online]. Available: https://arxiv.org/abs/1412.6980
[12] Y. Chen *et al.*, "DaDianNao: A machine-learning supercomputer," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2014, pp. 609–622.
[13] A. Shafiee *et al.*, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 14–26.
[14] P. Chi *et al.*, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 27–39.
[15] M. Prezioso, F. Merrikh-Bayat, B. D. Hoskins, G. C. Adam, K. K. Likharev, and D. B. Strukov, "Training and operation of an integrated neuromorphic network based on metal-oxide memristors," *Nature*, vol. 521, pp. 61–64, May 2015.
[16] T. Gokmen and Y. Vlasov, "Acceleration of deep neural network training with resistive cross-point devices: Design considerations," *Frontiers Neurosci.*, vol. 10, p. 333, Jul. 2016.
[17] D. Soudry, D. Di Castro, A. Gal, A. Kolodny, and S. Kvatinsky, "Memristor-based multilayer neural networks with online gradient descent training," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 26, no. 10, pp. 2408–2421, Oct. 2015.
[18] E. Rosenthal, S. Greshnikov, D. Soudry, and S. Kvatinsky, "A fully analog memristor-based neural network with online gradient training," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2016, pp. 1394–1397.
[19] L. Song, X. Qian, H. Li, and Y. Chen, "PipeLayer: A pipelined ReRAM-based accelerator for deep learning," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2017, pp. 541–552.
[20] M. Cheng *et al.*, "TIME: A training-in-memory architecture for memristor-based deep neural networks," in *Proc. 54th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2017, pp. 1–6.
[21] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "VTEAM: A general model for voltage-controlled memristors," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 62, no. 8, pp. 786–790, Aug. 2015.
[22] B. T. Polyak, "Some methods of speeding up the convergence of iteration methods," *USSR Comput. Math. Math. Phys.*, vol. 4, no. 5, pp. 1–17, 1964.
[23] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proc. 30th Int. Conf. Mach. Learn.*, Feb. 2013, pp. 1139–1147.
[24] N. Zhang, "An online gradient method with momentum for two-layer feedforward neural networks," *Appl. Math. Comput.*, vol. 212, pp. 488–498, Jun. 2009.
[25] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.
[26] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.
[27] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst. Cogn. Sci., La Jolla, CA, USA, ICS Rep. 8506, 1985.
[28] M. O'Halloran and R. Sarpeshkar, "A 10-nW 12-bit accurate analog storage cell with 10-aA leakage," *IEEE J. Solid-State Circuits*, vol. 39, no. 11, pp. 1985–1996, Nov. 2004.
[29] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm," *Integration*, vol. 58, pp. 74–81, Jun. 2017.
[30] S. Devarajan, L. Singer, D. Kelly, S. Decker, A. Kamath, and P. Wilkins, "A 16-bit, 125 MS/s, 385 mW, 78.7 dB SNR CMOS pipeline ADC," *IEEE J. Solid-State Circuits*, vol. 44, no. 12, pp. 3305–3313, Dec. 2009.
[31] M. Saberi, R. Lotfi, K. Mafinezhad, and W. A. Serdijn, "Analysis of power consumption and linearity in capacitive digital-to-analog converters used in successive approximation ADCs," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 58, no. 8, pp. 1736–1748, Aug. 2011.
[32] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Trans. Comput.-Aided Design Integr.*, vol. 31, no. 7, pp. 994–1007, Jul. 2012.
[33] A. Paszke *et al.*, "Automatic differentiation in PyTorch," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst. (NIPS), Autodiff Workshop*, Dec. 2017.
[34] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jul. 2017, pp. 5987–5995.
[35] A. Krizhevsky, "Learning multiple layers of features from tiny images," M.S. thesis, Dept. Comput. Sci., Univ. Toronto, Toronto, ON, USA, Apr. 2009.
[36] *Intel Xeon Processor E5-2683 V4 Specification*, Intel Inc., Santa Clara, CA, USA, 2016.
[37] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[38] *NVIDIA System Management Interface*, Nvidia Corp., Santa Clara, CA, USA, 2012.
[39] T. Dozat, "Incorporating nesterov momentum into adam," in *Proc. Int. Conf. Learn. Represent. (ICLR), Workshop Track*, May 2016.
[40] K. L. Wang, J. G. Alzate, and P. K. Amiri, "Low-power non-volatile spintronic memory: STT-RAM and beyond," *J. Phys. D, Appl. Phys.*, vol. 46, no. 7, p. 074003, 2013.
[41] C. Walczyk *et al.*, "Impact of temperature on the resistive switching behavior of embedded HfO_2-based RRAM devices," *IEEE Trans. Electron Devices*, vol. 58, no. 9, pp. 3124–3131, Sep. 2011.

Tzofnat Greenberg-Toledo received the B.Sc. degree from the Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion–Israel Institute of Technology, in 2015, where she is currently a Graduate Student in electrical engineering. From 2014 to 2016, she was a Logic Design Engineer with Intel Corporation. Her research interests are computer architecture and accelerators for deep neural networks with the use of memristors.

Roee Mazor received the B.Sc. degree *(cum laude)* in computer engineering from the Technion–Israel Institute of Technology in 2017, where he is currently a Graduate Student with the Andrew and Erna Viterbi Faculty of Electrical Engineering. From 2015 to 2016, he was with Intel Technologies as an Intern and an Engineer. His current research is focused on efficient neural networks.



Ameer Haj-Ali received the B.Sc. degree *(summa cum laude)* in computer engineering and the M.Sc. degree with the Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion–Israel Institute of Technology, in 2017 and 2018, respectively. He is currently pursuing the Ph.D. degree with the Department of Electrical Engineering and Computer Science, University of California at Berkeley. From 2015 to 2016, he was with Mellanox Technologies as a Chip Designer. His current research is focused on auto-tuning, reinforcement learning, ASIC design, and high performance computing.

Shahar Kvatinsky received the B.Sc. degree in computer engineering and applied physics and the M.B.A. degree from The Hebrew University of Jerusalem in 2009 and 2010, respectively, and the Ph.D. degree in electrical engineering from the Technion–Israel Institute of Technology in 2014. From 2006 to 2009, he was with Intel as a Circuit Designer and was a Post-Doctoral Research Fellow with Stanford University from 2014 to 2015. He is currently an Assistant Professor with the Andrew and Erna Viterbi Faculty of Electrical Engineering, Technion–Israel Institute of Technology. His current research is focused on circuits and architectures with emerging memory technologies and design of energy efficient architectures. He has been a recipient of the 2010 Benin Prize, the 2013 Sanford Kaplan Prize for Creative Management in High Tech, the 2014 and 2017 Hershel Rich Technion Innovation Awards, the 2015 IEEE Guillemin-Cauer Best Paper Award, the 2015 Best Paper of COMPUTER ARCHITECTURE LETTERS, Viterbi Fellowship, Jacobs Fellowship, ERC Starting Grant, the 2017 Pazy Memorial Award, and seven Technion excellence teaching awards. He is an Editor of *Microelectronics Journal*.