Local Learning in RRAM Neural Networks with Sparse Direct Feedback Alignment

Brian Crafton[1], Matt West[2], Padip Basnet[2], Eric Vogel[2], and Arijit Raychowdhury[1]

Georgia Institute of Technology, Atlanta, GA
[1]School of Electrical and Computer Engineering
[2]School of Materials Science and Engineering
brian.crafton@gatech.edu, arijit.raychowdhury@ece.gatech.edu

***Abstract*—Neural networks utilizing non-volatile random access memory (NVM) exhibit excellent power reduction over traditional CMOS implementations. RRAM (resistive random access memory) is one such emerging memory technology offering low energy, good endurance, and a large analog conductance window. When implemented in a crossbar architecture, these networks are able to bypass the von-Neumann bottleneck by performing compute in-memory. This architecture works well for inference; however, training the network is far more challenging. Networks built using RRAM can be trained on-chip with gradient descent or off-chip where weights are transferred. Backpropagation, while effective in training von-Neumann architectures, is inefficient when memory and compute are partitioned together. Commonly referred to as the weight transport problem, each neuron's dependence on the weights and errors located deeper in the network requires reading the weights in each layer before computing and applying the error. This presents a key challenge in performing efficient on chip training for non von-Neumann architectures. In this work we demonstrate an alternative to backpropagation called sparse direct feedback alignment which bypasses the weight transport problem. We simulate crossbars of HfOx RRAM based on experimental data to explore the performance, area, and energy trade-offs of using bio-plausible algorithms on the MNIST and EMNIST datasets.**

I. Introduction

Hardware implementations of deep neural networks (DNNs) and their many variants rely on arrays of cores to carry out multiply-and-accumulate (MAC) operations. To feed each of the cores, memory must be funneled through a high speed memory controller which represents the von-Neumann bottleneck. Because we have separated compute and memory in this way, the von-Neumann bottleneck becomes the limiting factor in all data intensive applications. To solve this problem we can better map our applications to hardware where processing and memory are local to one another and global movement of data is minimized. Computing locally finds its inspiration from the human brain, where neurons communicate with adjacent neurons and synapses learn based on locally available information. Much like the brain we expect that moving forward, the separation of compute and memory will be heavily penalized in data intensive applications.

Near memory and in-memory computing are techniques inspired by the brain to combine compute and memory together and reduce global data movement. Near memory computing attempts to physically place compute and memory together. This can be done with traditional CMOS logic and SRAM [1]. While this is a useful technique, the focus of this work is on in-memory computing. In-memory computing uses physical properties of the devices to do computation without a dedicated compute unit. Many emerging non-volatile devices are examples of this, where using Ohm's law a voltage applied across a device's conductance results in a current. Using Kirchhoff's current law (KCL) we can sum along the columns of our memory crossbar to perform matrix multiplication in O(1).

RRAM is one such device that have been experimentally demonstrated in data-intensive machine learning applications [2], [3]. RRAM offers a small foot print [4] and sufficient endurance for such applications. Their low power and large conductance windows are highly desirable traits for an artificial synapse, enabling us to store multiple bits in a small area and perform matrix multiplication in place. Their voltage and process compatibility with CMOS allows them to be fabricated on the same platform and enables CMOS transistor based selectors as access devices for RRAM storage.

Using RRAM for in-memory processing and multi-level memories we can reduce the power of read and MAC operations which represent the majority of energy consumed by neural networks. This architecture performs especially well for the inference phase of neural networks, since there is no global movement of data. However, such an architecture fails to deliver high efficiency when functions that require global information are implemented. This is especially true when training DNNs with backpropagation (BP) and gradient descent (GD). In BP with GD, the transpose of the weights of the deeper layers in a network is needed to compute error gradients and weight updates of the shallower layers, thus requiring global movement of data. In the course of training networks based on in-memory computing, we must read the weights out to compute the backwards pass using backpropagation. Reading the weights out of the crossbar and performing the operation in CMOS logic requires significant time and power overhead that reduce the advantages of using in-memory computing. Another technique that can be used is transpose memory. By transposing the memory and performing in-memory compute, we no longer need to read the weights out of the crossbar to perform backpropagation. Although this is a significant advantage in terms of power and latency, transpose memory requires significant area overhead for the additional selectors, digital-to-analog converters (DACs), and analog-to-digital converters (ADCs) required.

Instead of using training algorithms that map well to von-Neumann architectures, like BP and GD, we pro-

978-1-7281-2954-9/19/$31.00 ©2019 IEEE

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on June 28,2021 at 13:43:12 UTC from IEEE Xplore. Restrictions apply.

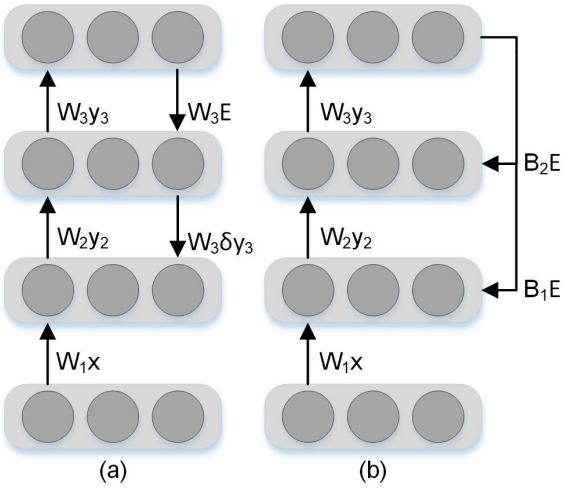

Fig. 1. **A. Backpropagation:** The backwards pass is computed by applying the transpose of the weight matrix to the error at each layer. **B. Direct Feedback Alignment:** The backwards pass is computed by applying a random matrix B to the error for each layer

pose using bio plausible algorithms which map well to distributed in-memory computing architectures. We show how a new algorithm, single connection sparse direct feedback alignment (SSDFA) [5], can be used to train an RRAM network without reading the weights or performing transposed matrix multiplication. As a result, we show a circuit design and algorithm which yields improvements in area, power, and performance.

II. Direct Feedback Alignment

A simple solution to the weight transport problem, namely feedback alignment, was proposed in [6]. Feedback Alignment (FA) uses fixed random feedback weights to propagate the errors back through the layers of a DNN rather than using the actual network weights to compute the partial error. Consequently, the weights in the shallow layers of the network no longer need information of the weights of all the deeper layers. Building on top of this, [7] proposes Direct Feedback Alignment (DFA), where it was shown that the feedback to shallow layers need not be propagated through all the layers. DFA showed that instead, a random matrix can be used to compute the error at each layer. Such a matrix can be randomly initialized with size $N \times C$, where N is the number of hidden neurons at a hidden layer and C is the number of classes (or outputs) in the network. This matrix is used to create a linear projection of the error for each hidden neuron.

For both BP and DFA, the feed forward computation can be written as

$$y_1 = W_1 \cdot x, a_1 = f(y_1) \tag{1}$$
$$y_2 = W_2 \cdot a_1, a_2 = f(y_2) \tag{2}$$
$$y_n = W_n \cdot a_{n-1}, a_n = f(y_n) \tag{3}$$

where x is the feature vector and W_i is the weight matrix connecting layer $i-1$ to layer i ($y_0 = x$). The dot product of x and W_i yields y_i, and applying the non-linear activation function f results in the activation at neuron i, a_i. Each of these algorithms computes the error at a specific layer. The error at the last layer of the network, n, is the classification error e. BP computes the error at each hidden layer l, δa_l, by transposing the weight matrices W and multiplying by the gradient of the activation function. These layerwise computations for BP can be written as

$$\delta a_n = e \tag{4}$$
$$\delta a_2 = W_2^T \cdot \delta a_3 \odot f^{'}(a_2) \tag{5}$$
$$\delta a_1 = W_1^T \cdot \delta a_2 \odot f^{'}(a_1) \tag{6}$$

where $\odot$ is the element-wise multiplication operator. BP requires all the deeper weights in the network in order to compute the error at a layer earlier in the network. DFA bypasses this dependence, only requiring a random matrix B to be multiplied with the error vector. The layerwise error computations for DFA are

$$\delta a_n = e \tag{7}$$
$$\delta a_2 = B_2 \cdot e \odot f^{'}(a_2) \tag{8}$$
$$\delta a_1 = B_1 \cdot e \odot f^{'}(a_1) \tag{9}$$

where we observe that the error at each layer does not depend on the error at any other layer. In Figure 1, we compare the error computation at each layer. In BP (Figure 1a), the error must be sent back through the same exact matrix W as in the forward direction. Consequently, we must compute the error at each layer one at a time before sending the data backwards to be used at the previous layer. DFA bypasses this issue by simply using a random matrix to create an error for all layers independently of one another. This is observed in Figure 1b where layers 2, 3, and 4 can be updated in parallel. Instead of moving data from previous layers, the error can be computed at the layers locally. These advantages allow us to use DFA to do all updates in parallel, while using only local information without the need to propagate backwards.

Decoupling the forwards and backwards weights makes DFA a suitable algorithm for training networks that rely on in-memory computing. The advantage of in-memory computing is that in the forwards pass, the weights do not need to be read out and brought to compute. Similarly, with DFA we no longer need to read weights out during the backwards pass. Instead a random feedback matrix can be used to compute the error and update the network independent of the forward weights.

While DFA does not require information about the forward pass or the feedback weights of other layers, it still requires feedback through dense matrices. Recent work [5] has shown that an extremely sparse feedback matrix can produce similar results to BP and DFA. In the special case, single connection sparse direct feedback alignment (SSDFA), the matrix contains only a single non-zero valued weight per row. As a result, each neuron is connected to only a single error through a single feedback weight. Using this technique local learning can be achieved. As the size of the network grows, the information required to compute the error at each neuron remains fixed.

III. Circuit Architecture

In the implementation of a hardware DNN, there are three phases of computation to consider: forward propagation, backward propagation (backpropagation), and the network update. Each of these phases has its own challenges that we consider in the following sections.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on June 28,2021 at 13:43:12 UTC from IEEE Xplore. Restrictions apply.

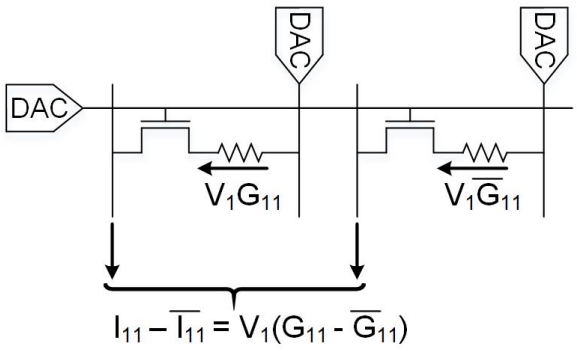

Fig. 2. 1T1R circuit with a differential pair for signed weights.

A. Forward Propagation

Using in memory processing with the RRAM we can apply voltages proportional to the input data along the RRAM cells. Using Ohm's law and KCL, we will perform matrix multiplication, the primary operation in the forward pass in our neural network. Traditional artificial neural networks are trained without a positive or negative weight constraint. As a result nearly half of the weights are negative values. However, because RRAM cannot be signed, we use two RRAM cells to represent a single synapse. The value of the weight is represented as the difference of the two RRAM's conductances as shown in Figure 2. While this is considerable overhead, it is necessary to achieve results consistent with CMOS neural network implementations.

B. Backpropagation

To compute the error at each of the hidden layers in our network, we use the weights in the previous layer to backpropagate the error to the current layer. In RRAM crossbars, this is a significant problem that can be solved with two methods:

1) Read conductance values from the crossbar, and perform the multiplication in CMOS peripherals.
2) Transposed matrix multiplication in the crossbar.

Reading the weights out requires that we spend significant time reading each weight one at a time, or a row at a time described by [2]. Once complete, we must then perform the matrix multiplication in CMOS. This method limits the advantages of using in memory computing because we revert to a von-Neumann computing architecture in the backwards direction.

Using additional circuitry it is possible to transpose the weight matrix so that the backwards pass can be computed in memory [8], [9]. Transposing the weight matrix requires significant circuit overhead which increases area, power, and design complexity. The 2T1R synapse circuit is shown in a crossbar array in Figure 3. The circuit elements highlighted in red represent the portion of the circuit for the transposed read and matrix multiplication. This additional circuity comes at a significant overhead in area and design complexity. An additional select transistor, DAC, and ADC are required to read the transposed data.

One of these two methods are required because backpropagation requires information about the forward weights. Rather than attempting to optimize circuits for backpropagation, we can instead use one of the feedback alignment algorithms. The family of feedback alignment

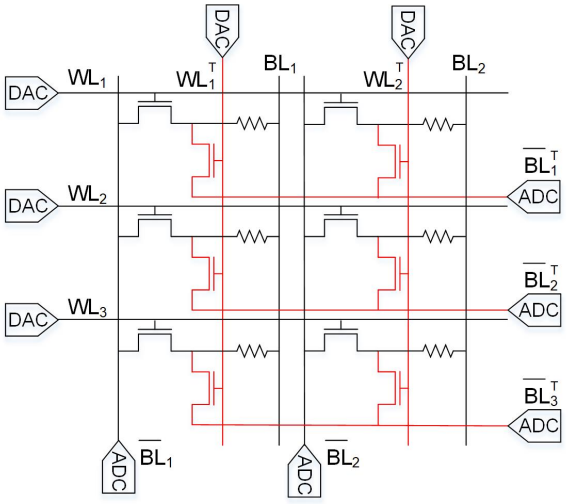

Fig. 3. 2T1R circuit for transposed matrix multiplication. The transpose logic is shown in red. The 2T1R configuration requires additional transistor, word line, and bit line.

algorithms discussed in section II do not require any information from the forwards pass. DFA allows us to compute the error independent of the layers to reduce latency and the energy cost to move data from layer to layer. However, DFA requires that we compute the error at each layer using another RRAM crossbar or CMOS and SRAM. Later in this section we show how a sparse matrix can be used so that neither of these need to be used.

C. Network Update

Once we have computed the error using backpropagation, we must apply the error to the network. The error with respect to each weight is computed as the outer product of input values and error: $\vec{a} \cdot \vec{e}$. This results in a matrix, δW, that we must apply to our crossbar. To update an RRAM, we must apply a positive voltage to "set" the device (lower resistance) or negative voltage to "reset" (increase resistance). There are different ways to apply the error update to the crossbar. The most common approach is to compute δW in CMOS and then write each row one at a time. Like computing the backwards pass, this limits the advantages of using compute in memory because we need to move the two vectors $\vec{a}$ and $\vec{e}$ to the CMOS and then perform $N \cdot M$ MAC operations. [10] proposed a more efficient way to compute and apply the error to the network. To update the RRAM network we perform matrix multiplication with time and the results are integrated by the RRAM. If we apply pulses proportional to the input values on the word lines, and pulses proportional to the errors on the columns then the time overlap of the two is proportional to the product.

To perform time based multiplication, we use a digital time converter shown in Figure 4. In this circuit the error value, $\vec{e}$, is input to the circuit to control the delay of the inverter chain. The duty cycle of pulses can controlled by increasing or decreasing the delay of the inverter chain. In Figure 5 we demonstrate an example where WL is asserted for 40% of the period and $\overline{BL}$ is asserted for 80% of its period. It is necessary that the period of $\overline{BL}$ is several times shorter than WL, otherwise a min function is performed

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on June 28,2021 at 13:43:12 UTC from IEEE Xplore. Restrictions apply.

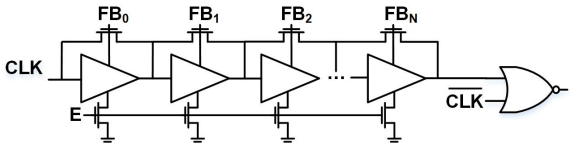

Fig. 4. A digital time converter for time based multiplication. The NMOS transistors connecting the buffers short the delay to implement a random feedback value for SSDFA. When one of the feedback weight bits are high, the tri-state buffers can be switched to high-impedance mode, while the NMOS transistors provide a shorter delay.

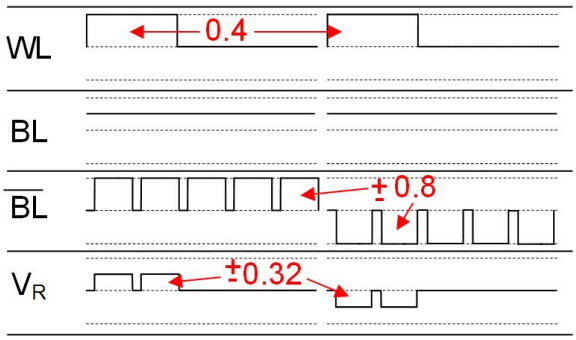

Fig. 5. Waveform showing pulsed-based multiplication of positive and negative voltages applied to RRAM array. $Wordline$ (WL) and $\overline{Bitline}$ ($\overline{BL}$) are the two inputs. The result of multiplication is V_R.

rather than multiplication. The resulting voltage across the RRAM is roughly 32% of the period.

Performing gradient descent with DFA to compute the error at each layer, we no longer need to read out the forward weights from RRAM. However, using DFA still requires that we compute the error at each layer using either another RRAM array or CMOS and SRAM. Recent work [5] showed that an extremely sparse feedback matrix can produce similar results to BP and DFA. The matrix will contain only one non-zero valued weight per row. This means that each neuron is connected to only a single error through a single feedback weight. This algorithm has a special application in the context of training RRAM arrays. Because only a single error and *random fixed* feedback weight are required we can randomly alter our DTC with a *random fixed* scalar proportional to the feedback weight. Using this technique, no additional memory for the feedback weights is required. An example is shown in Figure 4, where an additional path through NMOS transistors is given and the random feedback weight is used to randomly decrease the delay.

In this work we consider three different circuit implementations for training RRAM DNNs:

1) **1T1R RRAM with CMOS BP:** Using 1T1R we must read the values from RRAM and then compute the error at each hidden layer with a CMOS MAC unit.
2) **2T1R RRAM:** Using 2T1R we can compute transposed matrix multiplication with increased area and circuit complexity.
3) **1T1R RRAM with SSDFA:** Using SSDFA we no longer need access to the weights in the forward pass. Instead we can use a random feedback value for each neuron. When updating the network we can simply apply a random error for a randomly modulated amount of time to each neuron. To implement this we make the DTC programmable such that we can randomly bypass the buffer delays as shown in Figure 4 thus realizing a random constant. Using SSDFA, in this way, eliminates the error computation from the backwards pass and requires no additional time or power.

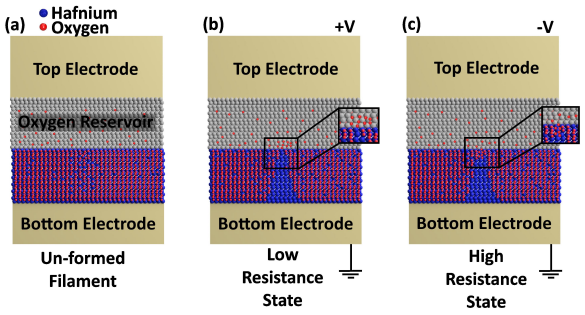

Fig. 6. (a) A fabricated device. (b) RRAM after forming and device in the LRS. The conducting filament is formed and will be used to toggle between the LRS and the HRS. (c) RRAM after being reset into the HRS. The tip of the filament is re-oxidized which increases the resistance of the cell.

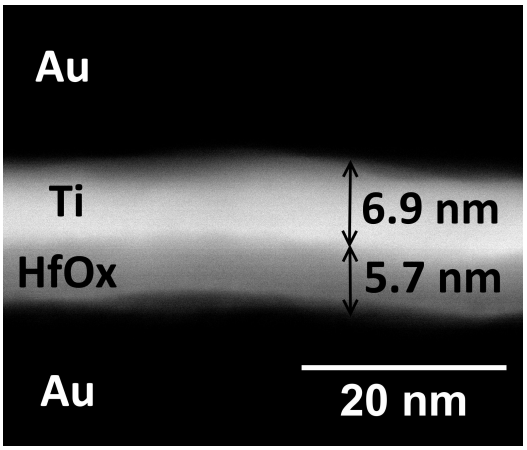

Fig. 7. Scanning transmission electron microscope cross-section of a RRAM device.

IV. Device Fabrication and Model Development

Prototypical RRAM devices and small arrays have been fabricated in clean-room facilities. The crossbar RRAM structures are fabricated using standard lift-off lithography practices. The bottom electrode is deposited under vacuum using electron beam evaporation at pressures lower than $5 \cdot 10^{-6}$. This electrode consists of a 20nm adhesion layer of titanium capped by 70nm of gold deposited at a rate of 1Å/s and 3Å/s, respectively. A blanket 5nm thick hafnium oxide layer is deposited over the whole wafer by using thermal atomic layer deposition at 250 °C. This is followed by another lithography step to pattern the top electrode. A titanium (5nm) and gold (150nm) top electrode is deposited under vacuum using electron beam evaporation under the same conditions as the bottom electrode. The titanium layer in contact with the hafnium oxide acts as an adhesion layer for the gold, as well as, an oxygen ion reservoir which helps with stable, non-volatile operation of the RRAM [11]. The device structure can be seen below in Figure 7 and in the insets of Figure 8.

Electrical testing of the devices is performed on a Keithley 4200-SCS Parameter Analyzer. Testing begins by

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on June 28,2021 at 13:43:12 UTC from IEEE Xplore. Restrictions apply.

sweeping voltages from 0 V to 5 V on the top electrode. This creates the initial conducting filament which acts as the memory element. A MOSFET is used to control the upper limit of the current through the filament to prevent damage, this is called the current compliance (Icc); the Icc is set to 1 mA. The creation of the filament is schematically shown in Figure 6a and 6b. After filament formation, the device is in the low resistance state (LRS) and will remain in the LRS without any external bias. To return the device to the high resistance state (HRS) a negative bias is applied to the top electrode causing the filament tip to re-oxidize, shown in Figure 6c. Device operation then follows the curve shown in Figure 6a, continuously toggling between the LRS and the HRS. The current-voltage characteristics of the device is shown in Figure 8.

To model this device in our simulation we modified a metal oxide RRAM model presented in [12]. The model shares a similar binary set and analog reset with the actual device. In our model we adjust the base resistance to match the target resistance in a scaled node so as to maximize the readability of the array. The current-voltage characteristics, as derived from the model is shown in Figure 9. We can observe a good qualitative match and a significant quantitative match between the experimental device and the model.

V. Experimental Results

To compare the different training methods, we benchmark the algorithms on standard vision data sets. We empirically evaluate power, performance, and area for the implementations of the different algorithms. The device characterization and the calibrated models have been discussed in the previous section. We use these models in a circuit simulation environment to explore the design space of RRAM based in-memory compute, in particular for DFA. We simulate our results using a framework developed in Python based on the calibrated models of the metal oxide RRAM from [12] and 45nm CMOS models from [13]. We feed in each 28x28 pixel image one by one as voltages. The simulation infrastructure includes

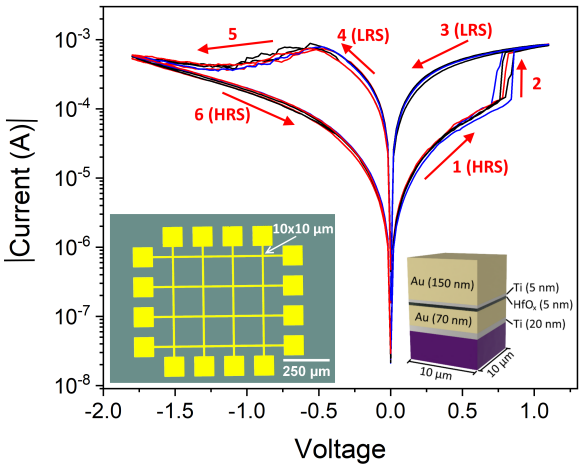

Fig. 8. Graph showing the operation of a Au/TiOx/HfOx/Au RRAM device. After initial filament formation, the device is operated by sweeping a bias from 0 V to 1.1 V on the top electrode. The device is reset to the HRS by sweeping from 0 V to -1.8 V. Inset (left) shows an optical microscopy image of 16 crossbar devices and (right) a schematic representation of the device structure.

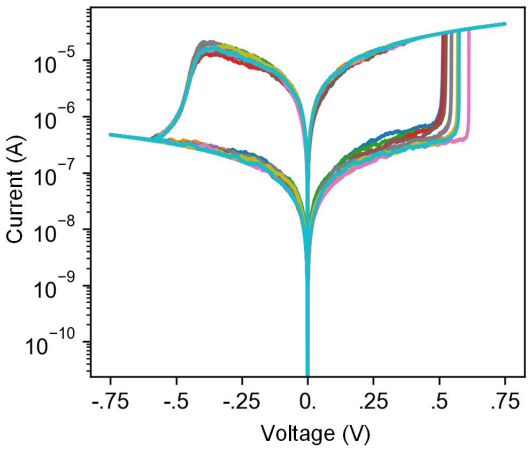

Fig. 9. Simulated RRAM device for 10 cycles with variability. This model is modified from the metal oxide RRAM model presented in [12].

the bit-cells, and all the peripheral circuits including the analog-to-digital converters, digital-to-analog converters, multiply-and-accumulate units, digital comparators for the ReLU operation, and CMOS memory to store the activation values of the neurons. For energy and performance we synthesized SRAM, MAC, and comparators in 45nm CMOS and assumed 8-bit precision and $100\mu W$ for each ADC and DAC pair operating at $10MHz$.

A. Benchmarks and Network Architectures

In our evaluation we used two different data sets: MNIST [14] and EMNIST [15]. EMNIST is an extended version of MNIST, which contains 47 classes instead of 10. This makes for a more challenging benchmark to address the concern of scaling to more complex problems. We use a simple 2 layer network for MNIST, and a 3 layer network for EMNIST. For both benchmarks we test a different number of hidden neurons to evaluate the performance and cost trade off.

B. Simulation Results

Using our simulation framework we collected results for performance and energy consumption. Performance is measured as the time to complete a single training example. Energy consumption is measured as the total energy per training example. We consider the three training methods described in section III. In Table I we show our results collected from running the three training implementations on MNIST with varying hidden layer sizes. Although BP does outperform SSDFA, it is only by a small margin. SSDFA trains the network 1.7x faster with 23% lower energy compared to a baseline 1T1R design running BP (Figure 10). Time and power savings from using SSDFA are a function of the number of layers. As the number of layers increases, the cost of the backwards pass relative to the forwards pass also increases. This is evident when training EMNIST on the larger network. We use an additional hidden layer and more hidden neurons per layer. As a result, the backwards pass accounts for a larger percentage of energy consumption and training time. In Table II and Figure 11 we observe a 20% reduction in energy and 2.3x reduction in training time for SSDFA versus BP in the

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on June 28,2021 at 13:43:12 UTC from IEEE Xplore. Restrictions apply.

TABLE I
MNIST CLASSIFICATION SIMULATION RESULTS

Network size	Algorithm	Accuracy (%)	Energy (nJ)	Time (us)
784-200-10	BP	97.0	90.17	5.80
784-400-10	BP	97.5	165.36	5.84
784-800-10	BP	97.5	307.41	5.92
784-200-10	SSDFA	96.7	82.49	3.20
784-400-10	SSDFA	97.1	150.93	3.23
784-800-10	SSDFA	97.2	281.77	3.32

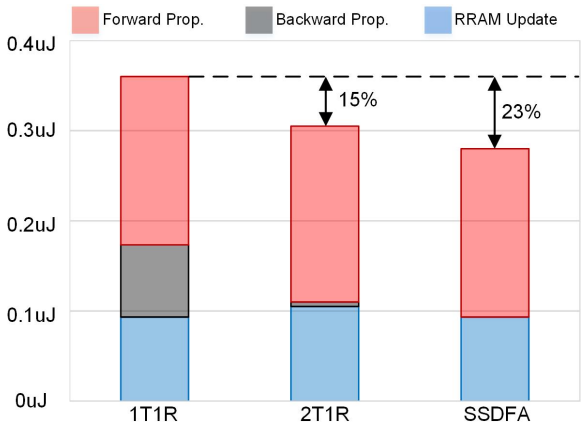

Fig. 10. Energy breakdown for MNIST 3 layer network (784-800-10)

2T1R configuration. The reduction in training time occurs because the backwards pass for SSDFA can be done in parallel, where as the backwards pass for BP must be done one at a time.

TABLE II
EMNIST CLASSIFICATION SIMULATION RESULTS

Network Size	Algorithm	Accuracy (%)	Energy (nJ)	Time (us)
784-500-500-47	BP	84.5	362.86	9.07
784-1000-1000-47	BP	85.1	823.34	9.37
784-500-500-47	SSDFA	82.1	320.70	3.76
784-1000-1000-47	SSDFA	83.1	719.12	3.96

In Figure 10 and 11 we show the energy breakdown for the forward pass, the backward pass, and the RRAM update. For MNIST, the energy costs are very similar because the error is computed for only the last hidden layer which is much smaller than the first hidden layer. For EMNIST this percentage increases more than two fold because we must read the second hidden layer which is the largest in the network. We note a 23% decrease in the total energy for SSDFA compared to a 2T1R implementation of BP for a smaller network running MNIST. For a larger network running EMNIST, we estimate a 20% decrease in energy compared to a 2T1R implementation of BP.

VI. CONCLUSIONS

In this paper we demonstrate the efficacy of single error sparse direct feedback alignment (SSDFA) as a bio-plausible training algorithm for deep neural networks. In particular, the use of local updates improves the energy-efficiency in training. We fabricate prototypical RRAM devices, model their characteristics and demonstrate 20%-23% improvement in energy-efficiency and 2.3x improvement in training time, compared to the baseline backpropagation and gradient descent algorithm.

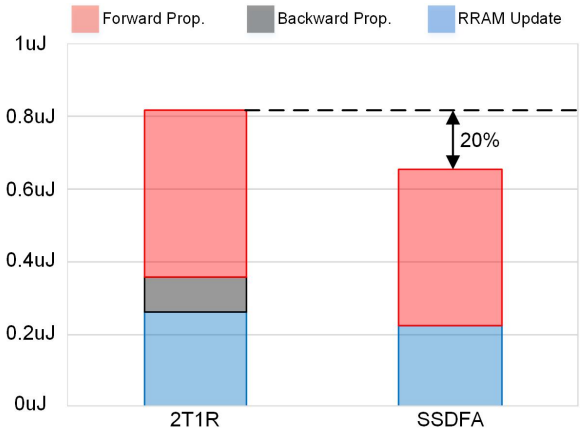

Fig. 11. Energy breakdown for EMNIST 4 layer network (784-1000-1000-47). We do not include 1T1R because it consumes significantly more power than 2T1R and SSDFA.

VII. ACKNOWLEDGEMENT

This work was funded by the U.S. Department of Defenses Multidisciplinary University Research Initiatives (MURI) Program under grant number FOA: N00014-16-R-FO05.

REFERENCES

[1] P. A. Merolla *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014.
[2] C. Li *et al.*, "Efficient and self-adaptive in-situ learning in multilayer memristor neural networks," *Nature Communications*, vol. 9, no. 1, p. 2385, 2018.
[3] C. Du *et al.*, "Reservoir computing using dynamic memristors for temporal information processing," *Nature communications*, vol. 8, no. 1, p. 2204, 2017.
[4] S. Pi *et al.*, "Memristor crossbars with 4.5 terabits-per-inch-square density and two nanometer dimension," *arXiv preprint arXiv:1804.09848*, 2018.
[5] B. Crafton *et al.*, "Direct feedback alignment with sparse connections for local learning," *arXiv preprint arXiv:1903.02083*, 2019.
[6] T. P. Lillicrap *et al.*, "Random synaptic feedback weights support error backpropagation for deep learning," *Nature communications*, vol. 7, p. 13276, 2016.
[7] A. Nøkland, "Direct feedback alignment provides learning in deep neural networks," in *Advances in neural information processing systems*, pp. 1037–1045, 2016.
[8] R. Hasan *et al.*, "Enabling back propagation training of memristor crossbar neuromorphic processors," in *2014 International Joint Conference on Neural Networks (IJCNN)*, pp. 21–28, IEEE, 2014.
[9] D. Soudry *et al.*, "Memristor-based multilayer neural networks with online gradient descent training," *IEEE transactions on neural networks and learning systems*, vol. 26, no. 10, pp. 2408–2421, 2015.
[10] T. Gokmen *et al.*, "Training deep convolutional neural networks with resistive cross-point devices," *Frontiers in neuroscience*, vol. 11, p. 538, 2017.
[11] S. Z. Rahaman *et al.*, "The role of ti buffer layer thickness on the resistive switching properties of hafnium oxide-based resistive switching memories," *Langmuir*, vol. 33, no. 19, pp. 4654–4665, 2017.
[12] Z. Jiang *et al.*, "A compact model for metal–oxide resistive random access memory with experiment verification," *IEEE Transactions on Electron Devices*, vol. 63, no. 5, pp. 1884–1892, 2016.
[13] W. Zhao and Y. Cao, "New generation of predictive technology model for sub-45 nm early design exploration," *IEEE Transactions on Electron Devices*, vol. 53, no. 11, pp. 2816–2823, 2006.
[14] Y. LeCun *et al.*, "Handwritten digit recognition with a back-propagation network," in *Advances in neural information processing systems*, pp. 396–404, 1990.
[15] G. Cohen *et al.*, "Emnist: an extension of mnist to handwritten letters," *arXiv preprint arXiv:1702.05373*, 2017.

Authorized licensed use limited to: Georgia Institute of Technology. Downloaded on June 28,2021 at 13:43:12 UTC from IEEE Xplore. Restrictions apply.