Framework for Implementing DNN using Memristor Arrays

Andrew McLean, Alvin Wong, Kody Ferguson, Arash Fayyazi
Department of Electrical Engineering, University of Southern California
Los Angeles, CA 90007
{amclean, alvindwo, kgfergus, fayyazi}@usc.edu

***Abstract* — In this paper, we propose an improved training framework for an inverter-based memristive neuromorphic hardware that is research community friendly. The original training framework, referred to as PHAX, is a physical characteristics aware artificial neural network (ANN) that incorporates an ex-situ training approach. The considered circuit is a highly energy-efficient hybrid-CMOS memristive implementation. We aim to improve training time and efficiency by creating a framework that utilizes declarative programming to parallelize network dependencies. As such, a network can be trained on graphic processing units (GPUs) or tensor processing units (TPUs), enabling the use of larger datasets and improving inference accuracy. In addition to faster training, using industry-standard tools like TensorFlow reduces the effort needed to train specialized hardware as well as incentivizes collaboration through open-sourced projects. Our framework opens the door for other research in cutting-edge training optimizations to easily explore their applications on neuromorphic hardware.**

I. INTRODUCTION

The ongoing interest in big data processing and cognitive computing has fostered a large community of researchers focused on creating high speed and energy-efficient platforms to accomplish recognition, approximation, and classification tasks. However, stark increases in the amount of data to be processed has caused a significant power overhead that cannot be executed in traditional hardware [1]. To solve the issue of high power consumption, alternative computing paradigms have been explored. A popular approach is to mimic the most power-efficient and highly parallelized computing system known to this day, the human brain.

The complexity of the brain has been modeled in a relatively new computing paradigm called Neuromorphic Computing. There are two main sub-disciplines of this computing architecture. One is based on conventional artificial neural networks (ANNs), while the other attempts to capture the biological spiking activity of real neurons and is referred to as spiking neural networks (SNNs) [2]. The latter has its advantages with respect to energy efficiency over ANNs because signals are spike-encoded instead of analytically modeled. However, when considering hardware area cost constraints, the ANN implementation has more area and energy efficiency compared to that of an SNN [3].

ANN neuromorphic hardware can be implemented digitally or by exploiting the analog characteristics of physical circuit elements. Although digital implementations are more predictable as they are less prone to device variation, they consume a greater amount of silicon area and power. Therefore, the analog implementation will be the main focus of our paper.

A neural network is constructed as layers of neurons connected by synapses that lead from the output of one neuron to the input of another in the following layer. Each neuron represents an activation function, with each synaptic connection having an associated weight. Prior research has shown that the memristor is a highly desirable circuit element that can model the plasticity of synaptic arrays [4]. In general, the memristor is called the fourth fundamental passive two-terminal electrical component, which relates the electric charge (q) and magnetic flux (ψ) and serves as a resistive non-volatile memory [5].

Prior analog implementations of DNNs focusing on memristors widely accept two architectures for synaptic weight storage implementation: a memristive crossbar [6], [7], [8], [9] and a memristive bridge [10], [11]. The activation functions of these neuromorphic, memristive crossbar circuits can be implemented using different circuit elements such as an analog comparator [7], operational amplifier (op-amp) [6], or a CMOS inverter [1], [8]. The inspiration for this work implements its activation function using a CMOS inverter, as does PHAX [1].

No matter how a neural network is implemented in hardware, the training of a neural network can be done using a separate off-chip system called ex-situ training [1], [6], [12], [13], [14] or upon the hardware with in-situ training [7], [8], [9], [15], [16]. The main challenge facing the in-situ training approach is its complex implementation and long runtime; on the other hand, the quality of the ex-situ training approach is highly dependent on the accuracy of the modeling of the neural network.

Since our work attempts to train the circuit model off-chip, the training algorithm of our network must accommodate the mismatch between the ideal neural network and its hardware implementation. The modified backpropagation algorithm proposed in PHAX [1] accounts for the mapping of unconstrained hardware weights to constrained model network weights, enhancing the quality and accuracy of the ex-situ training. The added computations involved in this modified ex-

situ training required significant code modifications to an already complex tool built in MATLAB, which is a closed source software. In consideration of other researchers attempting to build further upon the PHAX training framework, our work attempts to reduce the code complexity and create a more intuitive software implementation of the proposed ex-situ training phase of a memristor crossbar, CMOS inverter-based neuromorphic circuit.

With the recent popularity in structuring deep learning architectures as graphical models, our work attempts to model the aforementioned ex-situ training phase in TensorFlow. This dataflow programming library allows high scalability of computations across machines and can be executed on GPU or CPU architectures [17]. The contributions of this paper are to incorporate a graphical computation framework in the ex-situ training phase of neuromorphic DNNs with the hope of providing a base model and expose hardware researchers to the rich DNN community working with TensorFlow.

II. Model and Algorithms

A. Inverter-Based Memristive Crossbar Modeling

In this paper, we focus on improving the physical characteristics aware ex-situ (PHAX) training framework [1], continuing with the inverter-based memristive crossbar implementation of a neuron. A neural network implemented in this manner is depicted in Fig. 1.

Implementing the neurons of our neuromorphic circuit using CMOS inverters avoids the sneak path problem experienced by op-amp implementations because of the virtual ground at the inputs of such op-amps. The inverter-based neurons utilize voltage instead of current as the weighted sum of inputs of each neuron, defined in (1). As a tradeoff, the relationship between input voltages and weighted sum is more complex than that of an op-amp implementation.

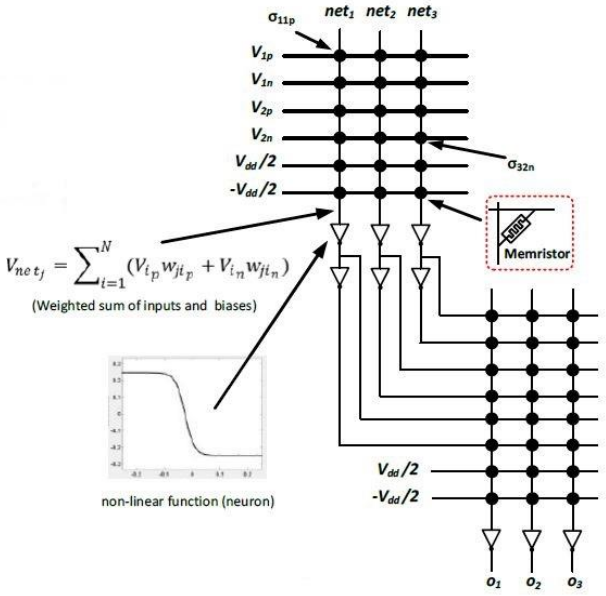

Fig. 1. Crossbar implementation of neuromorphic circuit.

$$V_{net_j} = \sum_{i=1}^{N} \left(V_{i_p} w_{ji_p} + V_{i_n} w_{ji_n} \right) \quad (1)$$

B. Internal Structure of Inverter Based Neuron

Zeroing in on a single inverter-based neuron, its synaptic weights are implemented with a column of the memristive crossbar and two CMOS inverters, as depicted in Fig. 2.

Each input is differential, containing inverted (V_{i_n}) and non-inverted (V_{i_p}) voltage signals. In order to account for the negative and positive voltage bias of the hardware, we have two weight components as defined below in (2).

$$\begin{cases} w_{ji_p} = \dfrac{\sigma_{ji_p}}{\sum_{m=1}^{N} \left(\sigma_{jm_p} + \sigma_{jm_n} \right)} \\ w_{ji_n} = \dfrac{\sigma_{ji_n}}{\sum_{m=1}^{N} \left(\sigma_{jm_p} + \sigma_{jm_n} \right)} \end{cases} \quad (2)$$

These are the weights that will be modified in the training algorithm. However, we need to define functions that will be able to map the neural network weights to the conductance values of the memristors conductance. Fig. 3 shows a pictorial representation of the constrained network model given one input for simplicity.

The function f in Fig. 3 is the activation function, which was chosen to be the hyperbolic tangent function because it closely models the voltage transfer characteristics of the CMOS inverter used in the neuromorphic circuit hardware implementation. The box in Fig. 3 represents the weight mapping operation that is in our TensorFlow computational graph. Since the network weights determined from training will inevitably be written to the fabricated memristors, there needs to be some physical constraints integrated into the normally

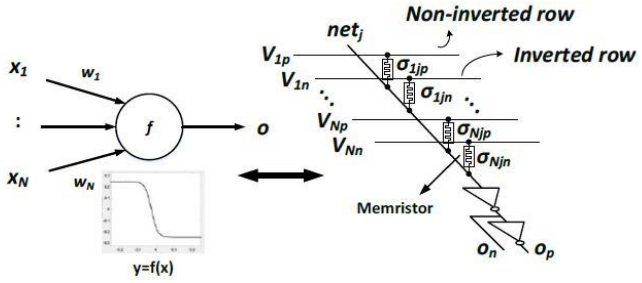

Fig. 2. Relationship between synaptic weights for each neuron.

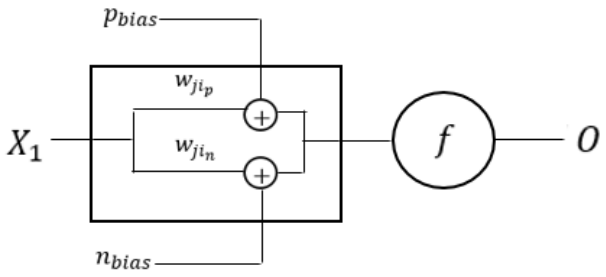

Fig. 3. Diagram of weight-mapping functionality.

unconstrained ANN weights, which will be denoted as θ. The linear mapping equations used in the backpropagation algorithm are defined below.

$$w_{ji_p} = g_2\left(\sigma_{ji_p}\right) = g_2\left(g_1\left(\theta_{ji_p}\right)\right) \quad (3)$$

$$g_1\left(\theta_{ji_p}\right) = \frac{K}{1+e^{-\theta_{ji_p}}} \quad (4)$$

$$g_2\left(\sigma_{ji_p}\right) = \frac{\sigma_{ji_p}}{\sum_{m=1}^{N}\left(\sigma_{jm_p}+\sigma_{jm_n}\right)} \quad (5)$$

Accurate mapping requires a linear function that can transform network weights to conductance values that can be written to the fabricated memristor. The bounded maximum (σ_{max}) and minimum (σ_{min}) conductance values are 7.9μ℧ and 0.12μ℧, respectively and were taken from prior work in this field [18]. The value of K in (4) is the difference between maximum and minimum conductance values, which thereby bounds the output of the sigmoid function within the required range. Meanwhile, (3) and (5) describe the weight mapping relation that is crucial to the modified backpropagation algorithm. Another important characteristic of the mapping functions is that they are continuously differentiable, which ensures the gradients can be calculated.

C. Backpropogation Algorithm with TensorFlow

Modeling DNN architecture in TensorFlow is desirable because of the built-in support for the common task of gradient computation [17]. Gradient Descent is basically a general function for minimizing a cost function, which in our case is the mean squared error seen from the output of our network compared to the real values. The methodology is to modify the parameters (weights) slightly during each training epoch of the network to hopefully find the minimum error. The general formula is shown in (6), where θ represents the weights of the training network.

$$\nabla = \frac{\partial J}{\partial \theta} \quad (6)$$

TensorFlow has a set of pre-built optimizers that can minimize a defined cost function with respect to the change in network weights through a method called automatic differentiation. The theory behind automatic differentiation is that all numerical computations are composed of a finite set of elementary operations for which the gradient is well defined [20]. In TensorFlow, this is done by first backtracking from the output of the network to the input of the network while adding partial derivatives to the graph along the backwards path using the chain rule. These nodes then compute the gradient function for the corresponding operation in the forward path [17].

Since TensorFlow has pre-defined operations for activation and cost functions, the partial derivative calculation of these functions with respect to input tensors is a functionality already built in. However, our cost function, as defined in (7), looks a little different because the weights of the training network are also a function of the hardware parameters associated with the memristor circuit. In addition, the activation function in our network is a function of the voltage transfer characteristics of the inverter hardware, adding another level of complexity. These modified operations make the previously non-trivial task of computing gradients difficult for our ex-situ training network because of the hardware dependencies affecting weight mapping constraints.

$$\nabla = \frac{\partial J}{\partial \theta} = \frac{\partial J}{\partial g(\theta)} \times \frac{\partial g(\theta)}{\partial \theta} \quad (7)$$

D. Analysis of Modified Algorithm

The time complexity of the backpropagation algorithm used in training the neural network is $T(n) = (2nmo)^3$, where n is the number of inputs, m is the number of hidden neurons, and o is the number of outputs. Thus, our time complexity is $O(n^3)$ in big-O time complexity because it takes three passes through each neuron crossbar to update its weight. The first pass computes the error between the inputs of the memristive crossbar and the output, the second pass is the backward propagation of the error to the lowest weights, and the third pass is the updating of each weight to reduce the output error.

The auxiliary space complexity of the memristive crossbar analog implementation of an Artificial Neural Network (ANN) is O(1) because we are just updating the weight values at already allocated spaces in memory. The total space complexity includes the input of the memristive crossbar which is $M(n) = 2nmo$, where n is the number of inputs, m is the number of hidden neurons, and o is the number of outputs.

III. Experimental Results

A. Graphical TensorFlow Model

The computational graph shown in Fig. 4 was used to implement the modified backpropagation algorithm. The following operations are captured in the Cost_Fx block:

- Weight mapping functions (forward and backward pass)
- Modified sigmoid activation functions (forward pass)
- Differentiated sigmoid activation function (backward pass)

B. Simulation Environment

The simulation study was performed using the MNIST data set [19]. This data set consisted of 1000 14×14 pixel images that were compressed forms of the 28 by 28 images found in the MNIST database. During the training phase of the network, 80% of the input data was randomly selected and used while the remaining 20% was kept for testing the accuracy and generality of the trained network. Our network consisted of 1 hidden layer to simplify the equations in the modified backpropagation algorithm. For the memristor device used in this work, the write threshold voltage was 4 V and the minimum (maximum) resistance of the memristors was about 125 kΩ (8.3 MΩ). In all simulations, the supply voltage level was 0.5 V. Also, the computer system used for the simulations utilized an Intel Core i7-2640M CPU with nominal clock frequency of 2.8 GHz and 8 GB of RAM.

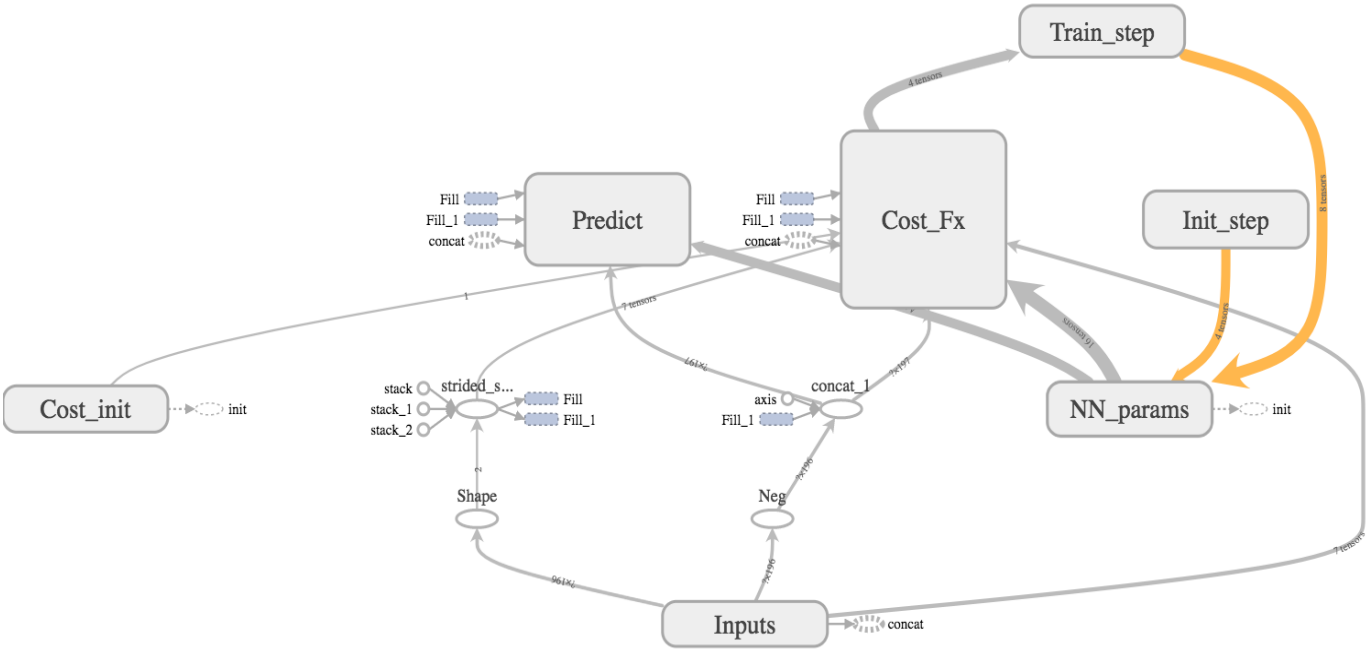

Fig. 4. Data-flow Model of Ex-Situ Training Phase.

C. Simulation Steps

Referring to the graphical model in Fig. 4, the initial step (Init_step) of the training phase was to randomly initialize the unconstrained weights to small values and set the cost function to 0. Then for each layer in the network, the mapped weights were calculated, the activations of those constrained weights were fed through the network and the cost was calculated. The errors were then propagated back through the network and weights were updated based on the updating rule defined in the original PHAX paper [1]. All these operations signified a training step in the computational graph. The training of the network was run for 100 epochs with a learning rate equal to 0.01.

D. Simulation Results

The graph in Fig. 5 shows that the cost of our network decreases as the number of epochs increases, which is desirable.

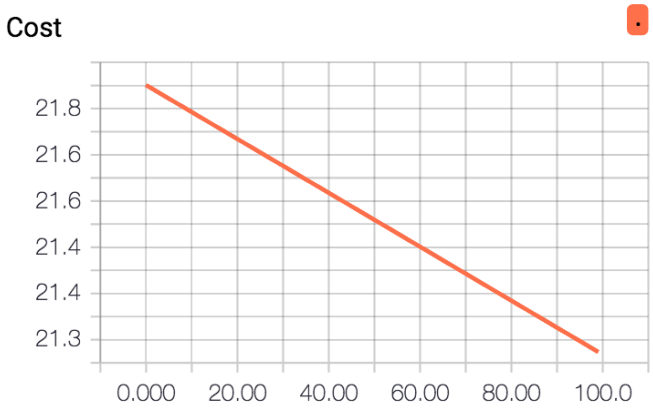

Fig. 5. Training Network Cost vs. Number of Epochs.

However, the decrease is linear, where the steepness of the slope is close to the learning rate of our network. This is a result of our network not being able to utilize the built-in optimizers that simplify gradient computation. For automatic differentiation to work properly for our modified training scheme, the gradient behavior of our existing forward pass operations need to be modified. Due to time limitations of our research, this functionality was unable to be implemented into our graphical model.

IV. Conclusion

In this work, the PHAX enabled training of the inverter-based neuromorphic circuit was implemented using TensorFlow. The results of our simulation proved that the weight mapping and inverter activation operations in our training framework led to a reduction in mean squared error by learning desired network weights. However, the gradient behavior of the backward pass of the network was not properly modeled to learn the weights by overriding TensorFlow automatic differentiation. Further research into defining custom gradient operations is needed to accurately incorporate conductance value mapping of memristors to off-chip training. In structuring our DNN as a computational graph, our framework provides researchers with the ability to better visualize network training schemes and accelerate code adoption to GPU and TPU processor architectures. Introducing hardware executed DNNs to the rich TensorFlow community equips academics with the necessary tools to explore new training paradigms on neuromorphic circuits.

REFERENCES

[1] M. Ansari, A. Fayyazi, A. Banagozar, M. Maleki, M. Kamal, A. Afzali-Kusha and M. Pedram, "PHAX: Physical Characteristics Aware Ex-Situ Training Framework for Inverter-Based Memristive Neuromorphic Circuits," in *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst*, Oct. 2017.

[2] Z. Du *et al*., "Neuromorphic accelerators: A comparison between neuroscience and machine-learning approaches," *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Waikiki, HI, 2015, pp. 494-507.

[3] M. Sharad, D. Fan and K. Roy, "Ultra low power associative computing with spin neurons and resistive crossbar memory," *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Austin, TX, 2013, pp. 1-6.

[4] S. H. Jo *et al*., "Nanoscale memristor device as synapse in neuromorphic systems," Nano Lett., vol. 10, no. 4, pp. 1297–1301, Oct. 2010.

[5] L. Chua, "Memristor-The missing circuit element," in *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507-519, September 1971.

[6] R. Hasan, C. Yakopcic and T. M. Taha, "Ex-situ training of dense memristor crossbar for neuromorphic applications," *Proceedings of the 2015 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, Boston, MA, 2015, pp. 75-81.

[7] Yakopcic *et al*., "Efficacy of memristive crossbars for neuromorphic processors," in *Proc. Int. Joint Conf. Neural Networks (IJCNN)*, Beijing, China, Jul. 2014, pp. 15–20.

[8] R. Hasan and T. M. Taha, "Enabling back propagation training of memristor crossbar neuromorphic processors," *2014 International Joint Conference on Neural Networks (IJCNN)*, Beijing, 2014, pp. 21-28.

[9] B. Feinberg, S. Wang, and E. Ipek, "Making Memristive Neural Network Accelerators Reliable," *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Vienna, Austria, Feb. 2018, pp. 52-65.

[10] S. P. Adhikari, C. Yang, H. Kim and L. O. Chua, "Memristor Bridge Synapse-Based Neural Network and Its Learning," in *IEEE Transactions on Neural Networks and Learning Systems*, vol. 23, no. 9, pp. 1426-1435, Sept. 2012.

[11] M. P. Sah, C. Yang, H. Kim and L. O. Chua, "Memristor circuit for artificial synaptic weighting of pulse inputs," *2012 IEEE International Symposium on Circuits and Systems*, Seoul, Korea (South), 2012, pp. 1604-1607.

[12] R. S. Amant *et al*., "General-purpose code acceleration with limited-precision analog computation," *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, Minneapolis, MN, 2014, pp. 505-516.

[13] X. Liu *et al*., "RENO: A high-efficient reconfigurable neuromorphic computing accelerator design," *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, San Francisco, CA, 2015, pp. 1-6.

[14] P. a Merolla *et al*., "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science* (80-.), vol. 345, no. 6197, pp. 668–673, Aug. 2014.

[15] R. Hasan, T. Taha, and Z. Alom, "A reconfigurable low power high throughput streaming architecture for big data processing," *arXiv Prepr. arXiv1603.07400*, Mar. 2016.

[16] B. Li, Y. Wang, Y. Wang, Y. Chen and H. Yang, "Training itself: Mixed-signal training acceleration for memristor-based neural network," *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Singapore, 2014, pp. 361-366.

[17] M. Abadi *et. al*., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[18] W. Lu, K. H. Kim, T. Chang and S. Gaba, "Two-terminal resistive switches (memristors) for memory and logic applications," *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*, Yokohama, 2011, pp. 217-223.

[19] Y. LeCun, C. Cortes and C. J. C. Burges, "The MNIST database of handwritten digits." 1998.

[20] "Custom Gradients in TensorFlow," UofG Machine Learning Research Group, 2018. [Online]. Available: https://uoguelph-mlrg.github.io/tensorflow_gradients/.