Gradient-descent-based learning in memristive crossbar arrays

Manu V Nair and Piotr Dudek
School of Electrical and Electronic Engineering,
The University of Manchester, UK
manu.nair@postgrad.manchester.ac.uk;
p.dudek@manchester.ac.uk

***Abstract*—This paper describes techniques to implement gradient-descent-based machine learning algorithms on crossbar arrays made of memristors or other analog memory devices. We introduce the Unregulated Step Descent (USD) algorithm, which is an approximation of the steepest descent algorithm, and discuss how it addresses various hardware implementation issues. We discuss the effect of device parameters and their variability on performance of the algorithm by using artificially generated and real-world datasets. In addition to providing insights on the effect of device parameters on learning, we illustrate how the USD algorithm partially offsets the effect of device variability. Finally, we discuss how the USD algorithm can be implemented in crossbar arrays using a simple 4-phase training scheme. The method allows parallel update of crossbar memory elements and reduces the hardware cost and complexity of the training architecture significantly.**

Keywords—memristor; crossbar arrays; machine learning; gradient descent; back propagation; neural networks; deep learning;

I. INTRODUCTION

Memristive crossbar arrays consist of two perpendicular wire layers acting as contacts to nanoscale memristor devices (passive 2-terminal components that display a hysteretic resistance-switching behavior) formed at the cross-over points of the two layers [1,2]. Crossbar arrays are attractive for several reasons. They can be made very dense, enabling construction of high-capacity memories and computational hardware. Furthermore, they could be layered on top of traditional CMOS ICs. In such a configuration, the CMOS circuits can be used for complex and precise signal conditioning and processing tasks, leaving the crossbar array to perform simpler, but parallel, operations [3]. Such systems hold great promise for hardware acceleration of computation, by enabling ultra-fast, low-power implementations of large-scale neural networks and other machine learning algorithms.

However, several issues need to be addressed before memristive crossbar arrays can be widely adopted, one of which is high device variability [4]. This problem is unlikely to go away given the desire for very high device densities and physical limitations. Another issue is the absence of precise models. The physics of practical memristor devices is not yet fully understood. They display a complex dependence on internal state parameters, which makes it difficult to program them to a desired state without resorting to elaborate feedback schemes [5,6]. In addition, stochastic behavior has been observed [7,15]. While several models describe general memristive behavior, they are often not accurate enough for reliable circuit design.

Despite these difficulties, there is an increasing interest in developing practical memristor-based computational circuits [2, 8]. In particular, neuromorphic systems with dynamic circuits employing memristors to mimic the plasticity rules seen in biological synapses have been proposed [9-13]. These systems are trained using STDP (Spike Timing Dependent Plasticity) learning schemes and have been demonstrated to work in spite of high device variability. However, the learning capability of spike-timing based systems is still rather limited when compared against state-of-the-art deep learning systems used in practice today. Simple artificial neural networks implementing binary pattern classification have been already demonstrated in memristive hardware [14]. More complex neural networks such as Restricted Boltzmann Machines (RBM), Convolutional Neural Networks (CNNs), etc. perform even better and use much fewer parameters or weights than spiking systems [23]. Therefore, it is of practical interest to investigate how memristive crossbar arrays can be used to implement such networks.

In this paper, we investigate methods for implementing machine-learning algorithms based on gradient-descent in memristive crossbar arrays. Gradient descent is a first order optimization algorithm for finding a local minimum of a function. For convex functions, it converges to the global minimum. It can be used for a variety of optimization problems such as inverting matrices, regression analysis, and to minimize the objective function in several machine learning algorithms. It is at the core of the back-propagation and similar training algorithms used on several artificial neural networks such as RBMs, CNNs, Auto-encoders, etc. However, as we discuss in this paper, the steepest gradient descent cannot be efficiently implemented in memristive crossbar arrays, because of device limitations. In this paper, we propose an approximate gradient descent rule to train memristive crossbar arrays, and investigate the impact of device parameters and variability on the algorithm's performance.

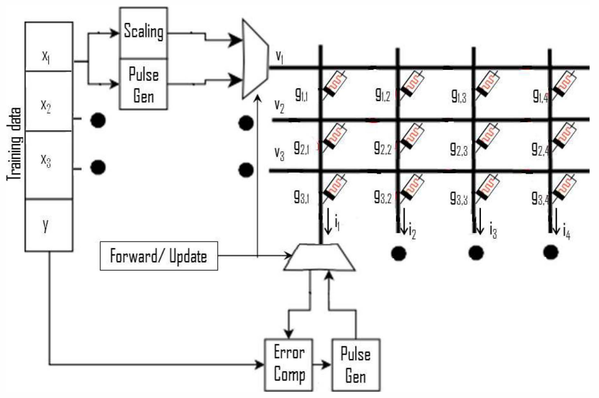


Figure 1: Block diagram of the training module

II. Gradient descent in crossbar arrays

For a given learning rate α, an N-dimensional parameter vector $\mathbf{w} = [w_1, \dots, w_N]^{\mathrm{T}}$, and an objective function $F(\mathbf{w})$, the gradient-descent training rule for updating $\mathbf{w}$ is:

$$\mathbf{w} := \mathbf{w} - \alpha \nabla F(\mathbf{w}) \tag{1}$$

For least-square minimization the objective function measures the mean square error when a predictive model $h_w(\mathbf{x})$ is used to fit the training samples $(y_i, \mathbf{x}_i)$, $i \in \{1, \dots, M\}$. Each training sample is a 2-tuple of observed output y_i and the input vector $\mathbf{x}_i = [x_{i,1}, \dots, x_{i,N}]^{\mathrm{T}}$. The objective function is:

$$F(\mathbf{w}) = \frac{1}{M} \Sigma_{i=1}^{M} \big(y_i - h_w(\mathbf{x}_i)\big)^2 \tag{2}$$

For linear predictive models of the form $h_w(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x}$, the steepest descent update rule for $\mathbf{w}$ in each iteration is given by:

$$w_k := w_k - \frac{\alpha}{M} \Sigma_{i=1}^{M} \big(y_i - h_w(\mathbf{x}_i)\big) x_{i,k} \tag{3}$$

The online variant of this algorithm is called the stochastic gradient descent, where the prediction error from each training sample is immediately used to update the weights. This algorithm converges faster than the steepest descent technique for large datasets. The update rule for stochastic gradient descent is:

$$w_k := w_k - \alpha\, \delta\, x_{i,k} \tag{4}$$

Where, prediction error

$$\delta = y_i - h_w(\mathbf{x}_i) \tag{5}$$

The algorithm involves two passes per iteration, a forward predictive pass, and a reverse update pass. The forward pass is essentially a matrix multiplication. As shown in Fig 1, this is easy to implement with crossbar arrays by treating the weights w as device conductances g, the inputs x as voltages v, and outputs $h_w(\mathbf{x})$ as currents i with suitable scaling to ensure that data falls within the dynamic range of the devices.

In the reverse pass, the weights of the predictive model are updated using a chosen training rule. In order to do this with a crossbar array, techniques to modify the device conductance are required. In addition, an understanding of design issues, such the memristive device characteristics, dynamic range of the input, and variability, on algorithm performance is needed.

In memristive devices, a change in conductance is observed when it is subjected to a non-zero flux. Therefore, the desired change in memristor conductance can be achieved by modulating the amplitude and/or duration of the applied training pulse β. The gradient training rule would then be implemented as:

$$w_k := w_k - \lambda_k(\beta, \boldsymbol{\omega}_k) \tag{6}$$

where, $\lambda_k(\beta, \boldsymbol{\omega}_k)$ models the magnitude of change in device conductance when the training pulse β is applied to the device with state parameters denoted by $\boldsymbol{\omega}_k$. In theory, any weight update rule can be implemented in this manner. For example, for (6) to be equivalent to (4), $\lambda_k(\beta, \boldsymbol{\omega}_k)$ must be equal to $\alpha\, \delta\, x_{i,k}$. However, that is impractical. The function λ_k is typically monotonically increasing with respect to β, and positive (negative) pulses lead to increase (decrease) of memristor conductance, but the magnitude of change in conductance depends on the device state. For example, a greater change in conductance is observed when the device is at a higher conductance state than at a lower conductance state. Moreover, the internal device state parameters $\boldsymbol{\omega}_k$ might not be directly observable and device models are either too simplistic or computationally expensive to compute λ_k to desired precision. The memristor behavior might even be stochastic [22]. This makes it impossible to stimulate a precisely controlled change in device conductance by simple schemes.

Some designers have proposed look-up tables (LUT) and/or feedback-based schemes in order to solve this problem [5,6,24]. These techniques generally use some form of read-monitored write method to program the memristors to the desired states. However, they have several practical limitations. Memristor device parameters exhibit high variability (spread across orders of magnitude) which makes it impossible to use a common LUT for the entire array. Maintaining a dedicated LUT for each device is clearly impractical. The feedback schemes typically require integration of CMOS devices within the crossbar array, making fabrication of crossbars significantly more complex. Such schemes also make the control circuitry complex and, most importantly, require different training pulses for each device, i.e. all the devices in the array cannot be simultaneously trained. Sequential training is unattractive because high-dimensional inputs result in a large number of devices, making the scheme unfeasibly slow.

Therefore, in order to use memristive crossbar arrays effectively, a more practical training scheme is essential. When training memristive arrays for pattern recognition tasks, it has to be remembered that the goal is not to precisely regulate the device states, but to minimize/maximize the objective function. Therefore, we propose a training procedure to allow the devices to settle to a state that minimizes the least square error in the training data without actively trying to regulate each device state to a predetermined value at every step.

III. UNREGULATED STEP DESCENT

In order to tackle the problems described in the previous section, we propose that the training pulse be approximated as:

$$\beta \propto \alpha \cdot sgn(\delta\, x_{i,k}). \tag{7}$$

By using β from equation (7) in (6), we get the Unregulated Step Descent (USD) algorithm. Here, the programmer only chooses α without concerning himself with the magnitude of $\delta\, x_{i,k}$. The intuition behind Equation (7) is to stop trying to regulate the update step size, which is impractical in memristive crossbar arrays anyway. Instead, the training rule moves the weights in the general direction that results in reduction of the objective function. The error introduced by making this approximation is compensated by taking small steps and recalibrating the direction of descent in each iteration. One of the limitations of the USD approach is that the magnitude of weight changes must be kept small. This slows down the rate of descent when the gradients are large. In addition, when the value approaches a local minima, the ability of the training rule to converge is limited by α. This effect is seen as noise or settling error that can be improved by adaptively reducing α as the solution converges.

It should be noted that stochastic gradient descent and other inexact gradient methods are based on a similar idea, in that the gradient calculated for each iteration only approximates the actual gradient of the objective function. The parallel weight perturbation (PWP) algorithm [25,26] has been proposed as an analogue hardware friendly approximation of the gradient-descent algorithm. However, some crucial differences make USD better suited for crossbar arrays. In the PWP algorithm, the devices are uniformly perturbed and the change in the objective function is computed. This change is used to approximate the gradient and update the weights. Note that the uniform distribution requirement is central to the PWP algorithm. To improve the approximation accuracy, multiple perturbations can be performed. The first implementation issue

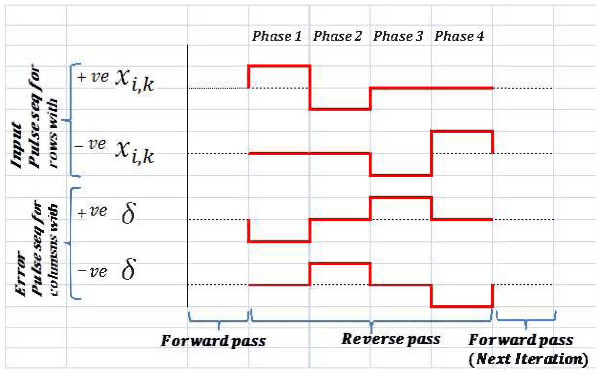


Figure 2: 4-phase Training Scheme. The voltage levels in the pulsing scheme shown here can only take three levels: $+V_{drive}$, 0, and $-V_{drive}$

Table 1: Device parameters for the BCM model [12] used for simulations

Property	Description	Value
G_{off}	Mean off state conductance	6.25e-4 S
G_{on}	Mean on state conductance	$G_{off} \times G_{ratio}$
μ_V	Mobility of oxygen ions	1e-14 $m^2 s^{-1} V^{-1}$
D	Length of the thin-oxide film	10 nm
η	Mean value of polarity coefficient	1
v_{max}	Maximum voltage applied to the device	1 V
t_0	Time normalization factor	10 ms

with PWP is that each update involves programming the devices twice, a perturbation step followed by a correction step. If multiple perturbations are required, it further increases the number of programming events. On the other hand, the USD implementation always involves a single memristor update in each step. Second, it is impossible to perturb all the devices in parallel and expect the perturbations to be uniformly distributed (because of the state-dependent response of memristors). The alternative is to perturb the devices sequentially where the pulse widths can be controlled finely. However, it increases training time exponentially and, as discussed in section II, it is practically impossible to control the state of each device precisely. The USD algorithm is not subject to such complications. Finally, the USD algorithm does not need a uniform perturbation signal generator, hence simplifying the circuit design.

IV. IMPLEMENTATION

Following commonly used memristive device models [15-17], we assume that memristors display a threshold voltage v_{th}, below which no or minimal change in conductance occurs. In the forward pass of the training process, the voltages driven into the array should be much lower than v_{th}. This restriction is to minimize unwanted changes in the device state. The gradient of the objective function is approximately computed using the sign of the input and the resultant prediction error. The computed values are then used to train the devices in the update pass using the 4-phase scheme shown in Fig 2. A similar scheme was used in [14] where it was used to train a binary perceptron. In this scheme, four-phased pulse patterns are driven into the input (rows) and output (columns) sides of the crossbar array. The choice of pattern at the input/output crossbar terminals is based on the sign of the input feature and output error as shown in Fig 2. In each phase, only a subset of the devices is subjected to a voltage difference that is large enough to cause a change in their internal states. For example, in the first phase, memristors with positive inputs and positive error outputs see a voltage of $2V_{drive}$ while the other devices see a voltage of either zero or V_{drive}. The value of V_{drive} should be chosen such that $V_{drive} < v_{th} < 2V_{drive}$. This ensures that the devices that are not targeted are not subjected to a voltage difference greater than the threshold. In addition, keeping the widths of the pulse constant for all the devices in all the phases is necessary to eliminate spurious $2V_{drive}$ paths that can corrupt the untargeted devices. If the simplifications given by equation (7) were not made, the 4-phase scheme would not work because each device would require a unique training pulse.

The pulse amplitudes could be scaled in proportion to δ and $x_{i,k}$. Such a scheme would potentially reduce the settling error and settling time. This study does not pursue this idea because it expects significant variability in v_{th} which is not adequately modeled by the memristor model used here. Experimental evidence also suggests that threshold voltage is a soft value below which there could be small changes in conductance [1]. This can result in some change/drift in the weights of the non-targeted devices in the array at each phase. It should be noted that the 4-phase scheme tries to compensate for this by ensuring that all the devices effectively see a net flux only in the desired update direction.

V. Simulations

Simulations to test the algorithm were run using the Boundary Condition Model (BCM) model for memristors; this paper uses the same notation and equations used in [16], not repeated here for brevity. The physical parameters used for the device are listed in Table 1. The BCM model was chosen because it has been reported to provide a fair approximation to memristor behavior [17] even though it appears to be imprecise close to v_{th}. This is not an issue when using the USD training rule because the pulse amplitudes are fixed and much larger than v_{th}. The model provides closed form equations for all the device parameters making simulations faster. Additionally, it is easy to control various device parameters in this model. The parameters used to study the effect of variability are the on-state conductance G_{on}, off-state conductance G_{off}, and polarity coefficient η. Variability was controlled by σ. For any device parameter p, the samples were generated such that the standard deviation of its distribution was $\sigma \cdot p$. Based on empirical observations [4][18], the lognormal distribution was used for G_{on} and G_{off}, and the normal distribution was used for η. Idealized components were used for the peripheral circuit elements such as sigmoid function, scaling, logic, pulse generation, etc. Note that the model implicitly assumes that the memristive conductances are continuous. Consequently, we do not deal with the effect of finite resolution here. Note that this is a realistic assumption as can be seen in [19].

To study the effect of device parameters on learning, tests were run on several datasets with dataset size $M = 1000$ and dimension $N = 20$. A linear classifier algorithm was chosen to test the USD rule because the simplicity of the algorithm makes it easy to develop intuition for the performance of the design. Additionally, the crossbar architecture for the linear classifier extends naturally to more complicated machine learning algorithms such as polynomial regression or classification, MLPs, RBMs, etc. Negative weights are implemented by driving input features of same magnitude but opposite polarities into the array. A bias term is learnt by adding a constant input and treating it as an added dimension to the input vector. For example a ten dimensional training set using a single linear classifier could consist of a column of 21 memristors. Alternatively, the positive and negative weights can be split into two columns of 10 devices each with a summation circuit that also includes a bias term. In this paper, we simulate the devices in the array and assume ideal peripheral circuitry. Under these simulation conditions, both structures are equivalent.

The parameters whose effects on the learning capability of the array were studied are $G_{ratio} (= G_{on}/G_{off})$, σ, and α. The training pulses in each phase had amplitude V_{drive}=1V and duration αt_0. Initial conditions of the devices at the start of the algorithm affect convergence behavior and performance. Hardware aspects in addition to the well-studied algorithmic

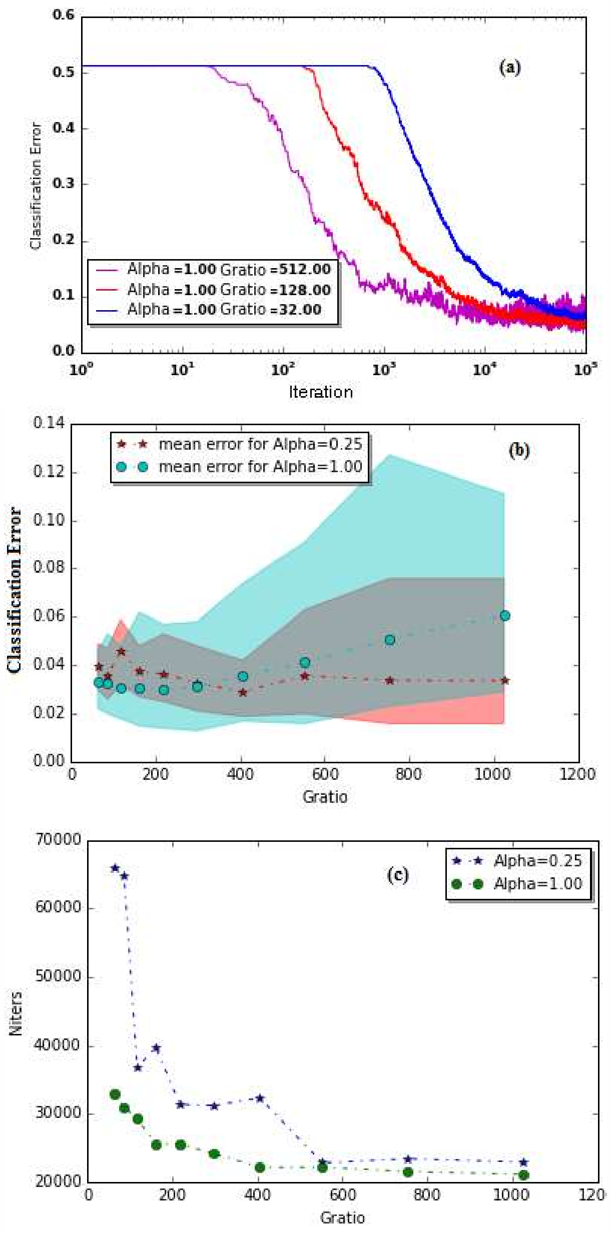


Figure 3(a) Evolution of Classification Error with Iterations
3(b) Classification error vs G_{ratio}. The bands straddle the maximum and minimum classification error because of settling error at convergence.
3(c) Settling time (Niters) vs G_{ratio} for different values of α (Alpha).

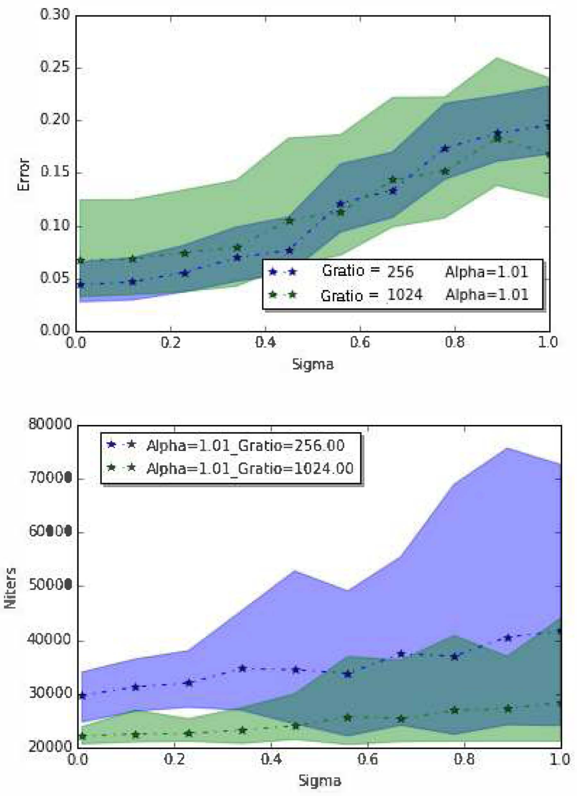
ones have to be considered when discussing initialization. A detailed discussion of this is beyond the scope of this paper. In all the experiments described in this paper, the device conductances were initialized to $G_{ratio}/2$.

The learning capability of the array was studied in terms of settling time and settling error. Classification error was computed on the entire training dataset at each iteration. Settling time is defined as the number of iterations after which there is no improvement in error performance. Settling error is the asymptotic mean error rate observed on the training dataset. The training sets used to obtain the plots in Fig 3 and Fig 4 were generated by sampling a uniformly distributed N-dimensional random variable and bisecting the training sample space by a randomly chosen hyperplane. We allowed a random subset of the training samples to lie in the wrong side of the hyperplane with a probability p_e, which unless stated otherwise is set at 5%.

The effects of G_{ratio} and α on settling error and settling time are shown in Fig 3a, 3b, and 3c. The curves in Fig 3a show evolution of classification error as the algorithm converges. Fig 3b and 3c plot the effect of varying G_{ratio} and α on settling behavior.

The value of G_{ratio} affects the settling time, settling error, and design of peripheral circuitry. It can be seen that the algorithm converges faster, but shows higher settling error as G_{ratio} increases. This can be seen in Fig 3b where the colored bands straddle the maximum and minimum settling error for corresponding values of G_{ratio} and α. As G_{ratio} and α increase, the bands get wider (Fig 3b). This effect is explained by looking at the BCM model [16]. A quick analysis shows that that greater G_{ratio} and α values result in a greater change in device conductance. On the other hand, from a design point of view, it is useful to have a greater G_{ratio}. A greater G_{ratio} would provide a greater dynamic range in the output current because the difference between the max and min-state currents would be larger. This makes the design of sigmoid or other threshold function simpler and less susceptible to variability and noise. The trade-off between larger G_{ratio} or α values and learning performance is a constant feature in our simulations. It should also be noted that the mean error performance of the algorithm only modestly increases with respect to G_{ratio} and α. The noisy behavior resulting from larger G_{ratio} values can be compensated by averaging over multiple predictors.

There is plenty of literature on how the learning rate α affects learning performance in standard gradient-descent implementations. Most of those ideas are directly applicable here. The figures show that a larger α results in faster convergence. However, larger values of α also make settling noisier (Fig 3b). A simple technique to improve settling time and error is to start with a large α and progressively shrink it. It should be noted that although the number of iterations required for convergence increases with decrease in α, the time taken in each iteration decreases (because α determines the duration of the training pulse). Therefore, the net training time does not necessarily increase when using smaller values of α. The optimal choice of α should take into account factors such as training time, G_{ratio} and the dataset.



| **Figure 4(a)** Classification Error (Error) vs σ (sigma). **4(b)** Number of training iterations (Niters) vs σ (Sigma). Training time increases as variability increases.

Fig 4a and 4b show how settling error and settling time are affected by device variability. The lognormal distribution causes a wide variation in the device properties (Fig 5). As device variability increases, the range of input and output currents increasingly deviates from the expected mean value. This can result in saturation of the peripheral circuitry. Note that the gradient-descent rule partially compensates for such behavior. The compensation mechanism does not appear to affect the mean convergence time significantly. However, the

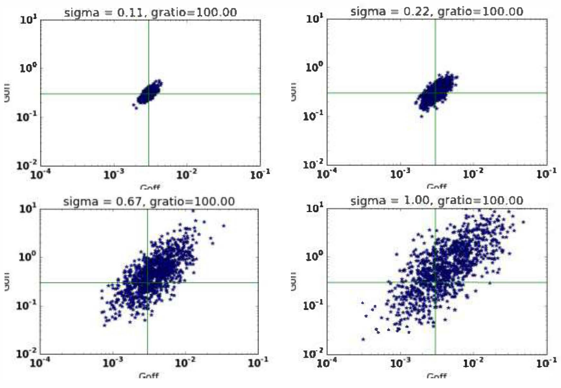


Figure 5: Effect of σ in the spread of G_{on} (y-axis) and G_{off} (x-axis) values shown using 1000 memristive device samples with mean G_{off} = $\mathbf{0.003}\ S$, mean G_{ratio}= 100. Note that both x and y-axis are in log-scale.

variance in settling time increases with σ. Increasing the dynamic range of the peripheral circuits will help mitigate the effect of variability. Additionally, arrays with larger G_{ratio} values find it easier to compensate for variability because such arrays have more room to adjust. This can be seen in a smaller spread of settling time (Niters) in Fig 4b.

A method to minimize the effect of variability is to have several columns train on the same database and take a weighted average of all the predictions. This can be achieved by boosting schemes that have been successfully used in several machine learning algorithms. An important requirement for such techniques is to have uncorrelated predictors [20]. This is satisfied by the implementation described in this paper because the columns are trained independently of each other.

Fig 6 compares the performance of the crossbar array trained by the USD algorithm with that of the floating-point software implementation of the logistic regression for various values of p_e. It can be seen that a smaller value of α results in a better mean error rate. This is expected because all the devices are updated in each iteration even when they are very close to their ideal states (unless the error is precisely zero, which is practically impossible). Therefore, we can get closer to the minima by using smaller values of α.

To test the USD rule on a more realistic problem, it was used to train a $(784 \times 2)\ rows \times 10\ columns$ crossbar array on the complete MNIST handwritten digits database [21], where the soft-max (multinomial) logistic regression [22] was used as the classification rule. The implementation of the algorithm was straightforward with only some changes in the peripheral circuitry. We used devices with a range of G_{ratio} from 32 to 1000 and σ of 0.2. Interestingly, in all our experiments the validation-set error hovered around 7.5% and the test-set error was around 10%. Using a floating-point software implementation (softmax regression using steepest-gradient descent) gives a test-set error of about 5%.

VI. Conclusion

In this paper, we presented the USD rule, which approximates the standard gradient-descent algorithm. This approximation can be applied to several algorithms based on gradient descent such as linear/polynomial classification, soft-max regression, back-propagation algorithm for MLPs, contrastive divergence for RBMs, matrix inversion, etc.

If one were to attempt to train a crossbar array using the standard gradient descent technique, it would require fine control over the internal states of all the memristive devices. This can only be done using complex feedback schemes or sequentially training each device individually. This is unattractive for larger arrays. The approximation presented in this paper solves these design problems, making it feasible to use memristive crossbar arrays in applications where power and speed are important considerations. The USD gradient-descent rule iteratively adapts to the imprecision in device updates. It also partially cancels the effect of device variability. These benefits accrue from the self-correcting nature of the gradient-descent rule. Additionally, the USD rule allows use of simple 4-phase schemes to train all the devices in crossbar arrays simultaneously. Such schemes are hardware-efficient, and reduce array-training signal-generation complexity. Enabling such techniques make fabrication of crossbar-array-based machine learning systems easier to integrate with the current VLSI fabrication techniques because of the absence CMOS devices embedded within the crossbar array.

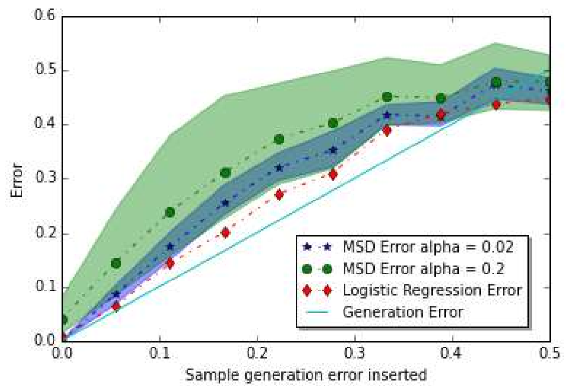


Figure 6: Performance of the stochastic USD implementation in comparison to floating-point logistic regression for various values of p_e.

We applied the USD rule to the logistic regression and soft-max algorithm to train the crossbar array. These algorithms were chosen because the lessons from these experiments can be applied to more complex neural networks. Our experiments with artificially constructed and the MNIST databases suggest that training memristive crossbar arrays using techniques based on gradient-descent USD approximation could be more attractive than STDP, LUT, feedback, or PWP-based mechanisms. Even with a simple linear-classifier, the performance obtained using the proposed USD approximation compares well with spiking neural networks [11,23]. More importantly the performance was attained with a much smaller number of devices than the spiking neural network implementations [23]. The USD approximation also results in a faster and simpler training architecture in comparison to LUT or feedback-based schemes.

Finally, our experiments provide an understanding of how crossbar array device and training parameters such as α, G_{ratio}, and σ affect the learning capability of the crossbar array. These insights can also be useful when developing methods to program memristive crossbar arrays for non-learning applications such as memories.

The experimental results presented in this paper indicate that using the USD approximation to implement machine learning algorithms in memristive crossbar arrays might result in practical, low-power, and fast designs. This is encouraging and motivates further analysis to test the performance of USD training rule in larger and deeper crossbar array networks based on auto-encoders, RBMs, CNNs, etc.

References

[1]. D B Strukov, et al. "The missing memristor found." Nature 453.7191 (2008)

[2]. J. Joshua Yang, Dmitri B. Strukov, and Duncan R. Stewart. "Memristive devices for computing." *Nature nanotechnology* 8.1 (2013): 13-24.

[3]. E Lehtonen, J. H. Poikonen, and Mika Laiho. "Two memristors suffice to compute all Boolean functions." *Electronics letters* 46.3 (2010): 230.

[4]. Medeiros-Ribeiro G et al. "Lognormal switching times for titanium dioxide bipolar memristors: origin and resolution", Nanotechnology. 2011 Mar 4

[5]. W Yi, et al. "Feedback write scheme for memristive switching devices." Applied Physics A 102.4 (2011): 973-982.

[6]. Manem, Harika, and Garrett S. Rose. "A read-monitored write circuit for 1T1M multi-level memristor memories." ISCAS, 2011.

[7]. R. Berdan, T. Prodromakis, F.P. Diaz, E. Vasilaki, A. Khiat, I. Salaoru and C. Toumazou, "Temporal Processing with Volatile Memristors", IEEE International Symposium on Circuits and Systems, May 2013

[8]. G. Snider, "Self-organized computation with unreliable, memristive nanodevices," Nanotechnology 18, 10 August 2007

[9]. C. Zamarreno-Ramos et al. "On spike-timing-dependent-plasticity, memristive devices, and building a self-learning visual cortex", *Front. Neurosci.*, vol. 5, no. 26, pp.1 -36 2011

[10]. SH Jo et al. "Nanoscale memristor device as synapse in neuromorphic systems." Nano letters 10.4 (2010): 1297-1301.

[11]. D Querlioz, O Bichler, and Christian Gamrat. "Simulation of a memristor-based spiking neural network immune to device variations." Neural Networks (IJCNN), 2011.

[12]. Snider, Greg S. "Spike-timing-dependent learning in memristive nanodevices."*Nanoscale Architectures, 2008. NANOARCH 2008. IEEE International Symposium on*. IEEE, 2008.

[13]. Linares-Barranco, Bernabé, and Teresa Serrano-Gotarredona. "Memristance can explain spike-time-dependent-plasticity in neural synapses." *Nature precedings* (2009): 1-4.

[14]. Alibart, Fabien, Elham Zamanidoost, and Dmitri B. Strukov. "Pattern classification by memristive crossbar circuits using ex situ and in situ training."*Nature communications* 4 (2013).

[15]. Guan, Ximeng, Shimeng Yu, and H-SP Wong. "On the switching parameter variation of metal-oxide RRAM—Part I: Physical modeling and simulation methodology." *Electron Devices, IEEE Transactions on* 59.4 (2012): 1172-1182.

[16]. F Corinto;A Ascoli, "A Boundary Condition-Based Approach to the Modeling of Memristor Nanostructures," *Circuits and Systems I: Regular Papers, IEEE Transactions on* , vol59, no.11, pp.2713,2726, Nov. 2012

[17]. A Ascoli et al. "Memristor model comparison." *Circuits and Systems Magazine, IEEE* 13.2 (2013): 89-105

[18]. T Serrano-Gotarredona et al, "STDP and STDP Variations with Memristors for Spiking Neuromorphic Learning Systems, "Front. in Neuroscience", Vol7, 2013,2

[19]. Abu Sebastian, Manuel Le Gallo, and Daniel Krebs. "Crystal growth within a phase change memory cell." *Nature communications* 5 (2014).

[20]. Freund, Yoav, Robert Schapire, and N. Abe. "A short introduction to boosting."*Journal-Japanese Society For Artificial Intelligence* 14.771-780 (1999): 1612.

[21]. MNIST handwritten digits database. http://yann.lecun.com/exdb/mnist/

[22]. Y So "A tutorial on logistic regression." SAS White Papers (1995).

[23]. Peter U. Diehl, Matthew Cook, "Unsupervised Learning of Digit Recognition Using Spike-Timing-Dependent Plasticity", *Neural networks and learning systems, IEEE Transactions on* 59.4 (2012)

[24]. Merkel, Cory E., et al. "Reconfigurable n-level memristor memory design." *Neural Networks (IJCNN), The 2011 International Joint Conference on. IEEE, 2011.*

[25]. Cauwenberghs, Gert. "A fast stochastic error-descent algorithm for supervised learning and optimization." *Advances in neural information processing systems (1993): 244-244.*

[26]. Alspector, Joshua, et al. "A parallel gradient descent method for learning in analog VLSI neural networks." *Advances in neural information processing systems. 1993.* .