RESEARCH ARTICLE | JUNE 26 2023

Multiplexed gradient descent: Fast online training of modern datasets on hardware neural networks without backpropagation

Adam N. McCaughan ; Bakhrom G. Oripov ; Natesh Ganesh; Sae Woo Nam; Andrew Dienstfrey; Sonia M. Buckley

APL Mach. Learn. 1, 026118 (2023)
https://doi.org/10.1063/5.0157645

Multiplexed gradient descent: Fast online training of modern datasets on hardware neural networks without backpropagation

Cite as: APL Mach. Learn. 1, 026118 (2023); doi: 10.1063/5.0157645
Submitted: 9 May 2023 • Accepted: 1 June 2023 •
Published Online: 26 June 2023

Adam N. McCaughan,[1,a] Bakhrom G. Oripov,[1,2] Natesh Ganesh[1,2] Sae Woo Nam,[1] Andrew Dienstfrey,[1] and Sonia M. Buckley[1,b]

AFFILIATIONS

[1] National Institute of Standards and Technology, Boulder, Colorado 80305, USA
[2] University of Colorado Boulder, Boulder, Colorado 80309, USA

[a] Author to whom correspondence should be addressed: adam.mccaughan@nist.gov
[b] sonia.buckley@nist.gov

ABSTRACT

We present multiplexed gradient descent (MGD), a gradient descent framework designed to easily train analog or digital neural networks in hardware. MGD utilizes zero-order optimization techniques for online training of hardware neural networks. We demonstrate its ability to train neural networks on modern machine learning datasets, including CIFAR-10 and Fashion-MNIST, and compare its performance to backpropagation. Assuming realistic timescales and hardware parameters, our results indicate that these optimization techniques can train a network on emerging hardware platforms orders of magnitude faster than the wall-clock time of training via backpropagation on a standard GPU, even in the presence of imperfect weight updates or device-to-device variations in the hardware. We additionally describe how it can be applied to existing hardware as part of chip-in-the-loop training or integrated directly at the hardware level. Crucially, because the MGD framework is model-free it can be applied to nearly any hardware platform with tunable parameters, and its gradient descent process can be optimized to compensate for specific hardware limitations, such as slow parameter-update speeds or limited input bandwidth.

I. INTRODUCTION

As the applications of machine learning and artificial intelligence have grown in size and power, so have their implementation costs and energy usage, leading to a significant effort toward building custom hardware that can perform these tasks at high speeds with lower energy costs.[1] A number of hardware platforms have emerged using analog,[2] digital,[3,4] or mixed-signal processing[5] that will potentially offer increased operational speeds and/or reduced energy costs.[6] However, many of the most promising hardware neural networks only perform inference, as integrating *in situ* training on these platforms has proved insurmountable. At the same time, the largest portion of energy costs in modern applications is spent on the training process,[7] usually via gradient descent, with the gradient calculated via backpropagation. While backpropagation is by far the most computationally efficient method of computing the gradient in software, this does not mean that it is the fastest, most robust, or most energy efficient technique in every hardware platform,[8] as evidenced by learning in the brain.[9,10]

Although often conflated, training via gradient descent does not require backpropagation—backpropagation is only used to calculate the gradient. Other methods for computing the gradient in neural networks exist, but are much less efficient in software than backpropagation and so are rarely used in today's machine learning applications. This is not generally true in hardware, where backpropagation may not only be challenging to implement but also may not be the most efficient way to compute the gradient.

Of particular interest in hardware are model-free methods, in which we require no knowledge of the internal structure of the network (e.g., topology, activation function, derivatives, etc.), only the ability to perturb the network's parameters and measure the network's response. The simplest example of such a method is finite-difference,[11] which has been employed for chip-in-the-loop training.[12] However, finite-difference has several other disadvantages

that prevent its widespread implementation in hardware, including the requirements for extra memory at every synapse and global synchronization. Fortunately, there are a variety of other model-free methods that overcome some of the issues associated with finite-difference.[13,14]

In this paper, we show that model-free perturbative methods can be used to efficiently train modern neural network architectures in a way that can be implemented natively within emerging hardware. These methods were investigated for training VLSI neural networks beginning in the 1990s,[15–24] and more recently on memristive crossbars[25] and photonic hardware,[26] but all these demonstrations have been very limited in scale, comprising small datasets with only a few neurons. Below we describe a framework for applying these techniques to existing hardware at much larger scales, with an emphasis on creating simple, highly localized circuits that could be implemented on-chip if desired. The framework is also extensible to training existing hardware systems via a chip-in-the-loop technique. We note that these methods have also been adapted in forward gradient approaches using auto-differentiation, which have attracted recent interest in the machine learning literature.[27–29]

We show that under realistic assumptions of the operating timescales of analog and digital hardware neural networks, one can train hardware to solve modern datasets such as CIFAR-10 orders of magnitude faster than training a software network on a graphics processing unit (GPU), even in the presence of signal noise and device-to-device variations in the hardware. A major advantage of this framework is that it can be used to perform online training of hardware platforms originally designed only for inference while making minimal hardware modifications.

II. MULTIPLEXED GRADIENT DESCENT

A. Computing the gradient with perturbations

We begin with the basic assumption that we have some hardware with programmable parameters (e.g., weights and biases) that can perform inference. Our goal is to augment the hardware minimally such that it can also be trained via gradient descent. We will show how to configure the hardware such that the network as a whole automatically performs gradient descent, without backpropagation. As an example, assume we have a hardware instantiation of a feedforward multi-layer neural network as shown in Fig. 1. The hardware takes time-varying inputs $x(t)$, training target $\hat{y}(t)$, has variable parameters θ, outputs the inference $y(t)$, and computes a cost $C(y(t), \hat{y}(t))$. To allow us to compute the gradient of such a system, we first add a small time-varying perturbation $\tilde{\theta}_i(t)$ to each parameter base value θ_i [Fig. 1(a), inset]. This perturbation will slightly modulate the cost C, and that modulation will be fed back to the parameters. This process will ultimately allow us to extract the gradient of the system.

Although the perturbations can take a variety of different forms, we will first describe this process by using sinusoidal perturbations as they are conceptually straightforward to understand. In this scenario, each parameter θ_i is slightly modulated at a unique frequency ω_i and amplitude $\Delta\theta$, giving the perturbation $\tilde{\theta}_i(t) = \Delta\theta \sin(\omega_i t)$. As each parameter is modulated, it slightly changes $y(t)$, which in turn changes the cost. Thus, if the parameters are modulated by frequencies $\omega_1, \omega_2, \omega_3$, etc., those same frequencies will necessarily appear as small modulations in the cost $\tilde{C}(t)$ on top of the baseline (unperturbed) cost value C_0 such that

$$C(t) = C_0 + \tilde{C}(t) = C_0 + \sum_i \Delta C_i \sin(\omega_i t). \tag{1}$$

If we remove C_0, we are left with a time varying signal $\tilde{C}(t) = \sum_i \Delta C_i \sin(\omega_i t)$ corresponding only to the effects of our parameter perturbations. The amplitude ΔC_i is simply the amplitude of change in the cost due to $\tilde{\theta}_i(t)$, the perturbation of parameter i.

Since the gradient with respect to the cost $dC/d\theta$ is composed solely from the partial gradients $dC/d\theta = (\partial C/\partial\theta_1, \partial C/\partial\theta_2, \ldots)$, if we can extract ΔC_i for each parameter, we can produce an estimate of the complete gradient $G = (\Delta C_1/\Delta\theta_1, \Delta C_2/\Delta\theta_2, \ldots)$. Now

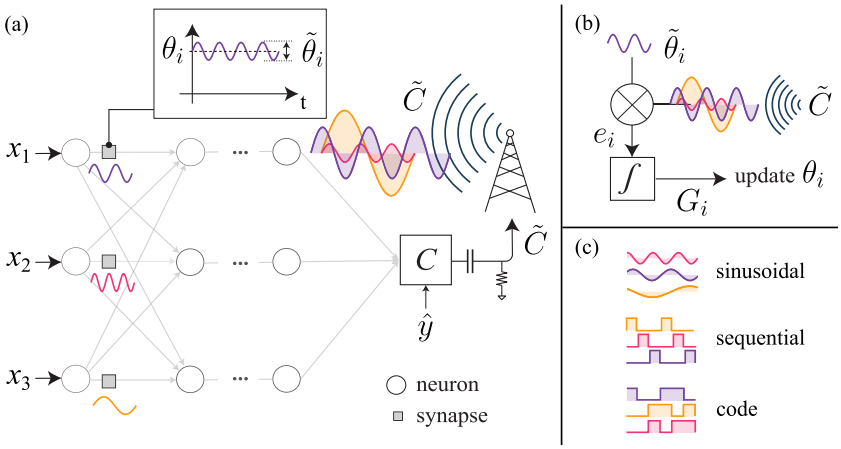

FIG. 1. (a) Schematic diagram showing the operation of the MGD framework in a feedforward neural network using example sinusoidal perturbations. [(a), inset] Each parameter i is modulated slightly from its base value θ_i by the perturbation $\tilde{\theta}_i$. The result of these perturbations causes a modulation in the cost $\tilde{C}$, which is globally broadcast back to all the parameters. (b) A homodyne detection process is used to compute the partial gradient approximations G_i from the integrated product of θ_i and $\tilde{C}$. This partial gradient is then used to update θ_i in the approximate direction of the gradient. (c) Example perturbation types that can be used with this process.

the task becomes to extract individual ΔC_i out of the summed signal $\tilde{C}(t)$. Fortunately, to extract a given ΔC_i, all we need to do is integrate the product of the input perturbation $\tilde{\theta}_i(t)$ with $\tilde{C}(t)$. The integration takes the form of a homodyne detection, where unwanted perturbations (frequencies) from other parameters are eliminated via integration,

$$G_i = \frac{1}{\Delta\theta_i^2}\frac{1}{T}\int_{t=0}^{T}\sum_k \Delta C_k \sin(\omega_k t)\Delta\theta_i \sin(\omega_i t)dt = \frac{\Delta C_i}{\Delta\theta_i} \quad \text{as} \quad T \to \infty, \tag{2}$$

where $1/\Delta\theta_i^2$ is a normalization constant.

The value G_i is the approximation for the partial gradient for parameter i. G approaches the exact gradient when both $T \to \infty$ and the amplitude of the perturbation $\Delta\theta_i$ approaches zero and is only an approximation otherwise. Fortunately, even at realistic timescales and amplitudes, G contains meaningful information and can be used to perform gradient descent.[13]

For illustrative purposes we have described the algorithm using sinusoidal parameter perturbations. However, any collection of orthogonal, mean zero perturbations can be used,[14] including a variety of analog and discrete perturbations as shown in Fig. 1(c). In general, we will be integrating the product $e_i(t) = \tilde{C}(t)\tilde{\theta}_i(t)/\Delta\theta_i^2$, which we refer to as the error signal, and G_i will be given by

$$G_i = \int_{t=0}^{T}\frac{\tilde{C}(t)\tilde{\theta}_i(t)}{\Delta\theta_i^2}dt. \tag{3}$$

Note that here and in the simulation results, G_i is being accumulated with time and is not normalized by $1/T$, unlike Eq. (2). As described later, this allows us to vary the integration time without greatly impacting the rate of training—equivalently, one can integrate for a long time resulting in a large step down the gradient, or one can take a series of shorter steps instead and travel approximately the same distance along the gradient.

We discuss the effects of changing the perturbation type in Sec. III D. We also note that although many of the quantities described here are time-varying, in the following sections, we will drop the explicit time dependence notation for the sake of brevity.

B. Gradient descent in the MGD framework

Here, we describe the practical implementation of a model-free gradient descent framework in hardware, which we term multiplexed gradient descent (MGD). To better understand the algorithm from a hardware perspective, we will run through the same computation previously described, but from the viewpoint of a single parameter (e.g., a synapse weight in a hardware neural network). The process begins with the application of a local perturbation $\tilde{\theta}_i$ that slightly modifies the base value of the parameter θ_i [Fig. 1(a), inset]. As previously described, this perturbation—and any other perturbations from other parameters—induce a change in the cost $\tilde{C}$ on top of the baseline cost C_0 such that the cost at the output is $C = C_0 + \tilde{C}$. $\tilde{C}$ may be extracted from C either by direct subtraction of C_0 or, in some analog scenarios, by a simple highpass filter. The resulting $\tilde{C}$ signal is broadcast globally to all parameters, so our parameter θ_i has access to it. (Note that although Fig. 1 shows a wireless broadcast tower for illustrative purposes, in most hardware platforms, this will be a wired connection.) However, we must assume that parameters other than the ith are also causing modulations in the cost as well. To our parameter θ_i, these other modulations are unwanted and must be filtered out. As shown in Fig. 1(b), for the parameter i to extract only its own effect on the cost, it can just integrate the product of its local perturbation $\tilde{\theta}_i$ and the global cost signal $\tilde{C}$ it receives. This has the effect of isolating the contribution from θ_i due to the pairwise orthogonality of the perturbation signals. From Eq. (3), this integration produces the partial gradient approximation $G_i \propto \Delta C_i/\Delta\theta_i$. The parameter can then use the G_i value directly to reduce the cost by updating itself according to a gradient descent step

$$\theta_i \to \theta_i - \eta G_i, \tag{4}$$

where η is the learning rate. When all the parameters of a system perform this operation in parallel, the resulting weight update corresponds to gradient descent training of the entire network. Because all the parameters are being perturbed and updated simultaneously, we call the framework multiplexed.

Looking at this process from the hardware perspective, one must also examine several practical considerations such as when to perform the weight update, how long to integrate the gradient, and so forth. We introduce the following time constants to provide a framework for managing these considerations in the MGD context.

- τ_p is the timescale over which perturbations occur. In a digital system, the perturbations of each parameter would be updated to new values every τ_p. In an analog (continuous) system, τ_p corresponds to the characteristic timescale of the perturbations. For instance, if using sinusoidal perturbations, it corresponds approximately to the inverse of the total bandwidth the frequencies occupy.
- τ_θ is the gradient-integration time [i.e., T in Eq. (3)]. It sets how often parameter updates occur and also determines how accurate the gradient approximation will be. For each time period τ_θ, the gradient approximation G is integrated, and at the end of that period, the parameters are updated according to Eq. (4).
- τ_x controls how often new training samples x, $\hat{y}$ are shown to the hardware. After each τ_x period, the old sample is discarded and a new one is applied, generating new outputs y and cost C.

The values of the time constants τ_θ and τ_p have a large impact on the particulars of the training, and particular choices of τ_θ and τ_p allow the implementation of certain conventional algorithms in numerical analysis. For instance, consider the implementation of the forward finite-difference algorithm within this framework. To do this, we start with discrete perturbations, such that every τ_p only a single parameter is perturbed by $\Delta\theta$, and the parameters are perturbed sequentially as shown in Fig. 1(c). When parameter i is perturbed by $\Delta\theta_i$, the cost changes by ΔC and the resulting partial gradient $\Delta C/\Delta\theta_i \approx \partial C/\partial\theta_i$ is stored in G_i. Now if we set $\tau_\theta = P\tau_p$, where P is the number of parameters in the network, this is exactly equivalent to computing a finite-difference approximation of the gradient: each τ_p, one element of the gradient is approximated, and after $P\tau_p$, every partial gradient G_i has been measured and stored. Since $\tau_\theta = P\tau_p$, the weight update only comes after all the partials

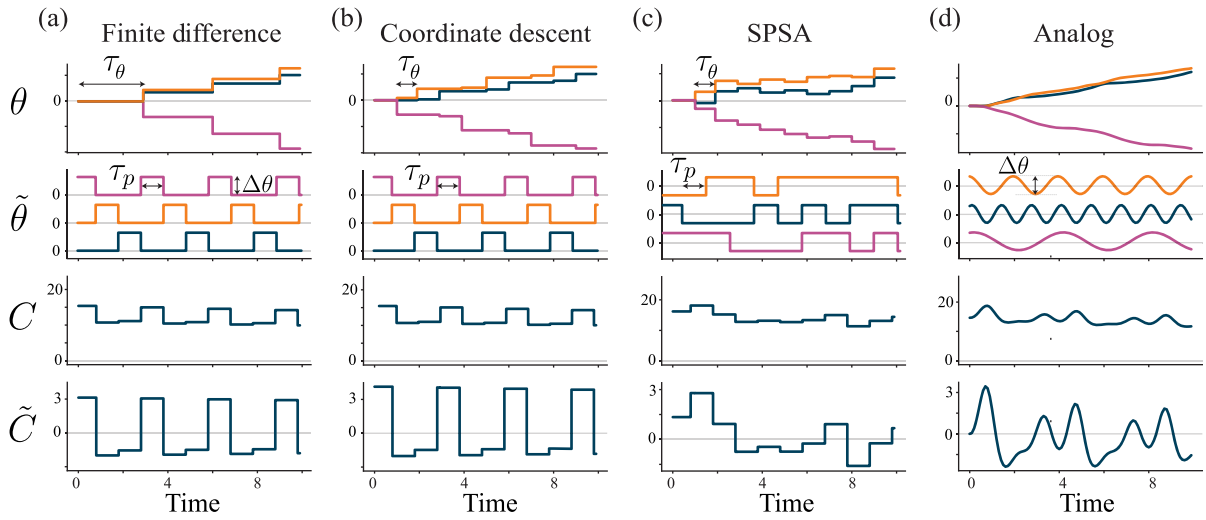

FIG. 2. Different optimization algorithms can be implemented in the MGD framework by changing the values the time constants, τ_p, τ_θ, and τ_x, and varying the perturbation type. Shown here are the parameter values θ that are updated every τ_θ, perturbation $\tilde{\theta}$ with characteristic time constant τ_p, computed cost C, and cost-modulation $\tilde{C}$. Varying τ_θ and τ_x and perturbation type allows MGD to implement a variety of optimization algorithms including (a) finite-difference, (b) coordinate descent, (c) simultaneous perturbation stochastic approximation (SPSA), and (d) an analog implementation. Each line color in the upper two plots corresponds to one of the three parameters in this simulation.

of G has been collected, just as in finite-difference. This process is shown schematically in Fig. 2(a).

Similarly, other optimization algorithms can be implemented simply by modifying the values of the time constants. For example, using same procedure as above but with the integration time reduced to a single timestep, i.e., $\tau_\theta = \tau_p$, corresponds exactly to coordinate descent. In this case, rather than storing each G_i until all the partials of the gradient are fully assembled, the weight update is applied immediately after each G_i is computed. This may be more appealing from a hardware perspective than a finite-difference approach, as G_i can be used for the weight update and immediately discarded—unlike finite-difference, it does not require a per-parameter memory to store G_i until the weight update is executed. This coordinate-descent process is shown schematically in Fig. 2(b).

As a third example, it is possible to implement simultaneous-perturbation stochastic approximation (SPSA)[13] by only changing the values of the time constants and the form of the perturbation. In this case, $\tau_\theta = \tau_p$ and a random, discrete $\{+\Delta\theta, -\Delta\theta\}$ perturbation is applied to every parameter every τ_p, as shown in Fig. 2(c). Similar to coordinate descent, this configuration avoids the need for additional per-parameter memories, as G_i values do not need to be stored. This method avoids the need for global synchronization of the parameters—the perturbations do not need to be sequential, and instead can be generated locally and randomly at the parameter.

In addition to being able to choose these specific optimization algorithms, by varying the time constants τ_p and τ_θ one can also implement entirely new optimization algorithms. For instance, in Fig. 2(d), we show an MGD implementation on an analog system using sinusoidal perturbations that does not correspond to any of the aforementioned methods. In this case, τ_p corresponds to the timescale $1/\Delta f$, where Δf is the perturbation bandwidth, the difference between the maximum and minimum perturbation frequency. Additionally, in the analog case, there is no discrete update of the parameters, and instead, τ_θ is an integration time constant. Unlike the discrete case, which accumulates G for τ_θ time then resets it to zero, θ is continuously updated with the output of an lowpass filter with time constant τ_θ (see Algorithm 2).

Because modern machine learning datasets are composed of many training examples (often tens of thousands), τ_x, the time constant that controls how often training samples are changed, is critical. In fact, by setting τ_x appropriately, mini-batching can even be implemented in hardware that only allows 1 sample input at a time. The batch size is determined by τ_θ/τ_x, the ratio of the gradient integration time τ_θ, and the sample update time τ_x. When τ_x is shorter than τ_θ, multiple samples are shown to the network during a single gradient integration period. As the sample changes, the gradient approximation G will then include gradient information from each of those samples. This integration-in-time process is arithmetically identical to summing multiple examples in parallel (as is often done on a GPU). As a concrete example, for hardware that accepts one input example at a time, if $\tau_\theta = \tau_x$, the batch size is 1, but if $\tau_\theta = 4\tau_x$, the batch size is 4. This is shown in Fig. 3 for the same network as in Fig. 2 and using simultaneous discrete perturbations. In Sec. III, we demonstrate that batch sizes as large as 1000 function correctly in MGD.

In summary, setting just three time constants and the perturbation type allows the MGD framework to implement a wide range of variations of gradient descent and, depending on the desired approach, one can tailor their training to the needs of the problem and the hardware. For example, MGD can match backpropagation by making τ_θ arbitrarily long, as we show in Sec. III B—the MGD

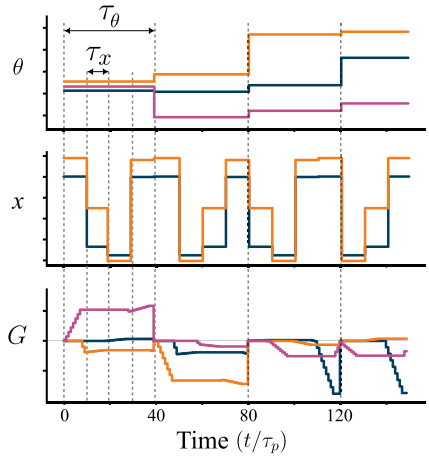

FIG. 3. Illustration of batching in a small network with three parameters and two x inputs, training on a dataset with four samples. The parameters θ are updated every τ_θ, and during that time, all four training samples are shown to the network and integrated into the gradient approximation G (batch size $\tau_\theta/\tau_x = 4$). G accumulates each timestep and is reset during the weight-update process after each τ_θ period. Note that the updates to θ occur in the opposite direction of G, per Eq. (4).

approximation to a partial derivative converges to the true partial derivative as τ_θ becomes arbitrarily long. Thus, the various parameter update steps presented above can be made arbitrarily close to their analytical definitions as implemented using digital arithmetic and backpropagation. At the same time, we show that short τ_θ values can work equally well. In practice, hardware considerations, such as write-speed limitations, may require an intermediate τ_θ value, as we discuss in Sec. IV. These features allow the MGD framework to leverage the same training techniques that have been so successful for training machine learning models, while also being compatible with new and emergent hardware.

ALGORITHM 1. Discrete algorithm.

Initialize parameters θ
for n **in** num_iterations **do**
  if $(n \bmod \tau_x = 0)$ **then**
    Input new training sample $x, \hat{y}$
  if $(n \bmod \tau_x = 0)$ or $(n \bmod \tau_\theta = 0)$ **then**
    Set perturbations to zero $\tilde{\theta} \leftarrow 0$
    Update baseline cost $C_0 \leftarrow C(f(x;\theta),\hat{y})$
  if $(n \bmod \tau_p = 0)$ **then**
    Update perturbations $\tilde{\theta}$
  Compute output $y \leftarrow f(x;\theta+\tilde{\theta})$
  Compute cost $C \leftarrow C(y,\hat{y})$
  Compute change in cost $\tilde{C} \leftarrow C - C_0$
  Compute instantaneous error signal $e \leftarrow \tilde{C}\tilde{\theta}/\Delta\theta^2$
  Accumulate gradient approximation $G \leftarrow G + e$
  if $(n \bmod \tau_\theta = 0)$ **then**
    Update parameters $\theta \leftarrow \theta - \eta G$
    Reset gradient approximation $G \leftarrow 0$

ALGORITHM 2. Analog algorithm.

Initialize parameters θ
for $t = 0$ **to** T **step** dt **do**
  if $(t \bmod \tau_x = 0)$ **then**
    Input new training sample $x, \hat{y}$
  Update perturbations $\tilde{\theta}$
  Compute output $y \leftarrow f(x;\theta+\tilde{\theta})$
  Compute cost $C(t) \leftarrow C(y,\hat{y})$
  Compute discretized highpass
    $\tilde{C}(t) \leftarrow \frac{\tau_{hp}}{\tau_{hp}+dt}\left(\tilde{C}(t-dt) + C(t) - C(t-dt)\right)$
  Compute instantaneous error signal $e(t) \leftarrow \tilde{C}\tilde{\theta}dt/\Delta\theta^2$
  Update gradient approximation
    $G(t) \leftarrow \frac{dt}{\tau_\theta+dt}\left(e(t) + \frac{\tau_\theta}{dt}G(t-dt)\right)$
  Update parameters $\theta \leftarrow \theta - \eta G$

III. SIMULATION OF MGD

A. Introduction

To characterize the utility of the MGD framework, we first simulated its performance on modern machine learning datasets. The goal of the simulator was not to perform gradient descent as fast as possible on a central processing unit (CPU) or GPU, but rather to emulate hardware implementing MGD and evaluate its potential performance in a hardware context. In particular, we used the simulator to estimate the speed, accuracy, and resilience to noise and fabrication imperfections. The simulator was written in the Julia language and can be run on a CPU or GPU and is available online.[30] The algorithms used in the simulation are shown in the Algorithms 1 and 2 boxes, with the parameters and their types in software shown in Table I.

TABLE I. Algorithm parameters and variables used in the simulations.

Description	Symbol	Analog or digital
Change in the cost due to perturbation	$\tilde{C}$	Both
Perturbation to parameters	$\tilde{\theta}$	Both
Parameters	θ	Both
Input sample	x	Both
Target output	$\hat{y}$	Both
Network output	y	Both
Cost	C	Both
Unperturbed baseline cost	C_0	Digital
Gradient approximation	G	Both
Instantaneous error signal	e	Both
Learning rate	η	Both
Perturbation amplitude	$\Delta\theta$	Both
Input-sample change time constant	τ_x	Both
Parameter update time constant	τ_θ	Both
Perturbation time constant	τ_p	Digital
Highpass filter time constant	τ_{hp}	Analog

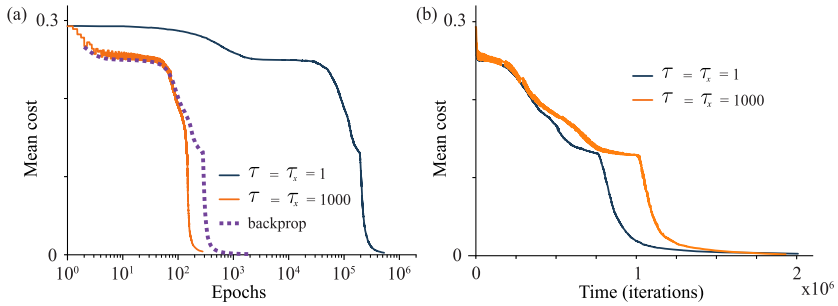

FIG. 4. Solving the 2-bit parity (XOR) problem using a 2-2-1 network with nine parameters, $\tau_p = 1$, and batchsize $\tau_\theta/\tau_x = 1$. (a) Mean cost over dataset vs epochs the on a 2-2-1 feedforward network averaged over 1000 random initializations and trained with MGD with different gradient integration times τ_θ (solid lines). The dashed line shows the same network and dataset trained using backpropagation. (b) The same MGD training data as in part (a) plotted vs time (number of perturbations τ_p) instead of epochs.

B. Equivalence to backpropagation

We first verified that the simulation was able to minimize the cost for a sample problem, and that it was equivalent to gradient descent via backpropagation with appropriate parameter choices. For this initial comparison, we chose to solve the 2-bit parity problem by training a 2-2-1 feedforward network with nine parameters (six weights, three biases). We simulated the problem with a very large value for τ_θ and $\tau_\theta = \tau_x$ such that we achieved a very good approximation of the gradient in G for each training sample. We then ran the same problem using a τ_θ value of 1 so that the gradient approximation G for each sample was relatively poor. We measured both the number of epochs and the amount of time (number of iterations of the simulation) for the two experiments, and the results are shown in Fig. 4. Here, an epoch was defined in the typical way—equivalent to the network being shown all training examples of a dataset.

Comparing the two scenarios in terms of epochs in Fig. 4(a), one can observe that a value of $\tau_\theta = \tau_x = 1000$ resulted in the system following a nearly identical training trajectory as backpropagation. This is as expected—for each sample shown to the network, the gradient approximation G has 1000 timesteps to integrate a very accurate estimate that should be very close to the true gradient computed by backpropagation. When $\tau_\theta = \tau_x = 1$, however, each sample only has a single timestep to estimate the gradient before moving on to the next sample. As a result, the samples have to be shown to the network many more times to minimize the cost, resulting in a much larger number of epochs.

However, while the $\tau_\theta = \tau_x = 1$ case uses the sample data less efficiently (requiring more epochs), there is actually a trade-off for data efficiency and run time. If we plot the cost vs iterations rather than vs epoch, we get a more accurate estimate of how long it will take hardware to train in terms of real time. As shown in Fig. 4(b), one can see that the shorter τ_θ and τ_x values take approximately half the time to minimize the cost as the longer values. These examples serve to highlight that while longer integration times produce a more accurate gradient approximation, integration times as short as τ_p may also be used to train a network, consistent with the findings of Ref. 13. In fact, shorter integration times may even be faster to train in terms of operational time.

To quantify the effect of longer integration times on the accuracy of the gradient approximation, we measured how the gradient approximation G converged to the true gradient $\partial C/\partial\theta$ (as computed by backpropagation) as a function of time. For this experiment, we ran the simulation with $\tau_\theta = \infty$ and $\tau_x = \tau_p = 1$ so that it continuously integrated G without ever resetting or updating the parameters. As the simulation ran, we repeatedly computed the angle between the true gradient $\partial C/\partial\theta$ and the approximation G. The results are shown in Fig. 5, showing the solution to the 2-bit parity, 4-bit parity, and NIST7x7 problems. The NIST7x7 dataset is a small image recognition problem based on identifying the letters N, I, S, and T on a 7×7 pixel plane. The dataset has the property that it cannot be solved to greater than 93% with a linear solve for a 49-4-4 feedforward network with sigmoidal activation functions. We also chose this network and dataset because it was small enough to perform many different simulations to acquire statistics, and the problem space is large enough that random solutions are unlikely.

As expected, the angle decreased with time as G aligned with the true gradient. The time axis is in units of τ_p, which is the minimum discrete timestep in this system. For a real hardware platform, this timestep is approximately the inference time of the system. In general, the more parameters the network has, the longer it takes to converge to the true gradient.

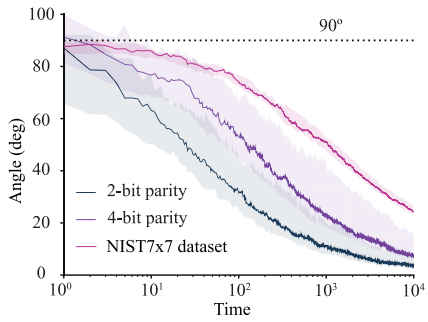

FIG. 5. Angle between the gradient approximation G and the true gradient vs time. The n-bit parity networks used n-n-1 networks, while the NIST7x7 networks used a 49-4-4 network. The networks have 9, 25, and 220 parameters, respectively, and the datasets have 4, 16, and 44 136 training examples, respectively. The solid line shows the median angle value for 100 random initializations for the n-bit parity and 15 random initializations for the NIST7x7 dataset. The shaded regions show the first to third quartile values.

C. Details of mini-batching

We next investigated in more detail how τ_θ and τ_x affect the training time. Longer τ_θ values result in a more accurate gradient approximation, but reduce the frequency of parameter updates. Using a fixed, low η value, we trained a 2-2-1 network to solve 2-bit parity (XOR) for 100 different random parameter initializations, varying τ_θ but keeping the batch size τ_θ/τ_x constant at either 4 or 1. Since the 2-bit parity dataset is composed of four $(x, \hat{y})$ pairs, $\tau_x = 4\tau_\theta$ is analogous to gradient descent—all four samples are integrated into the gradient approximation G before performing a weight update. When $\tau_x = \tau_\theta$, the network performs stochastic gradient descent (SGD) with a batch size of 1. Figure 6(a) shows the training time as a function of τ_θ for these two cases. Here, training time corresponds to the time at which the total cost C dropped below 0.04, indicating the problem was solved successfully. In the case where the batch size was 1, we observed that increasing τ_θ increased the training time. However, when the batch size was 4, increasing τ_θ had little effect on the training time.

As with any training process, the training can become unstable at higher η values and fail to solve the task. Since the results in Fig. 6(a) were only for a fixed learning rate, we also wanted to examine the effect of τ_θ on the maximum achievable η. Here, we defined the "max η" as the maximum learning rate where the network successfully solved the 2-bit parity problem for at least 50 out of 100 random initializations. As seen in Fig. 6(b), as τ_θ is increased, the max η decreases, resulting in longer minimum training times. This was true whether the batch size was large or small, although the larger batch sizes had higher achievable η values overall.

From these results, we infer that a poor gradient approximation taken with respect to *all* training examples is more useful than collecting an accurate gradient with respect to a single example. Stated another way, waiting a long time for an extremely accurate gradient then taking a large step is less productive than taking a series of short (but less accurate) steps. This is an important conclusion for hardware, as it shows that implementing an effective gradient descent process in MGD does not necessarily require additional memory to store accurate, high-bit-depth gradient values. Note that in our implementation G accumulates with time and so the size of the parameter update ηG from Eq. (4) grows proportionally to the integration time. This meant that when τ_θ was larger the effective step in the direction of the gradient was also larger, and so for fixed η the rate of training therefore remains approximately constant. If this was not the case, whenever τ_θ was doubled, we would also need to reduce η by half to maintain the same approximate rate of training.

D. Analog and digital perturbations

The parameter perturbations can take many different forms, as long as they are low-amplitude and their time averages are pairwise orthogonal or, in a statistical setting, are uncorrelated.[14] We have implemented four types of perturbations: sinusoidal perturbations, sequential discrete perturbations, discrete code perturbations, and random code perturbations. Sinusoidal perturbations are like those shown in Fig. 1(a), where each parameter is assigned a unique frequency. Sequential discrete perturbations refer to the case where parameters are sequentially perturbed, one at a time, by $+\Delta\theta$, as described in the finite-difference implementation of Sec. II B. "Code" perturbations are simultaneous discrete perturbations of $\{-\Delta\theta, +\Delta\theta\}$ for every parameter every τ_p timesteps. We call them code perturbations due to their similarity to code-multiplexing (spread-spectrum) techniques used in wireless communication technologies. There are two flavors of code-perturbations: the first consists of a predefined set of pairwise-orthogonal square wave functions that take the values of $\{-\Delta\theta, +\Delta\theta\}$. Each of these perturbation patterns is a deterministic sequence, and no two parameters have the same sequence. One example of these is the Walsh codes used in modern cell-phone communications. The second consists of randomly generated sequences of $\{-\Delta\theta, +\Delta\theta\}$ that are pairwise uncorrelated. We call these "statistically orthogonal." The statistically orthogonal case is slightly less efficient than the deterministic orthogonal codes since perturbations from multiple parameters interfere with each other more in $\tilde{C}$—any finite sample of the perturbations will have a non-zero correlation that decreases to zero with sample size. However, the use of the statistically orthogonal version allows the perturbations to be generated locally and randomly. These perturbations may be very useful in hardware implementations, as they are spread-spectrum and single-frequency noise from external sources is unlikely to corrupt the feedback signals.

To compare the training performance between different perturbation types, we applied four different perturbation types to the same 2-2-1 network described in Sec. III C in order to solve the

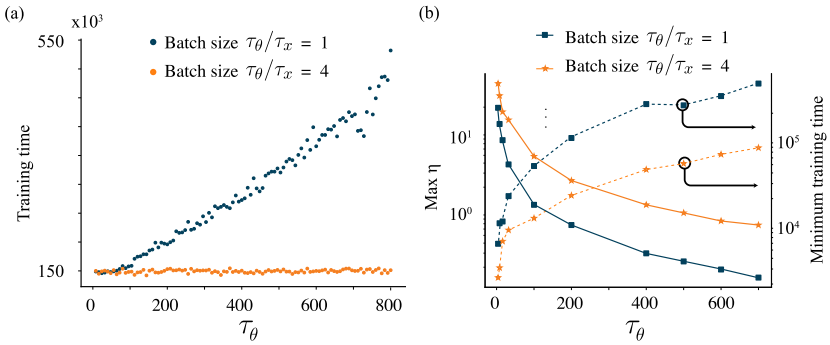

FIG. 6. Effect of τ_θ on the training time of the 2-bit parity (XOR) problem. (a) Training time as a function of τ_θ and batch size. (b) Effect of τ_θ on the maximum achievable learning rate η and corresponding minimum training time.

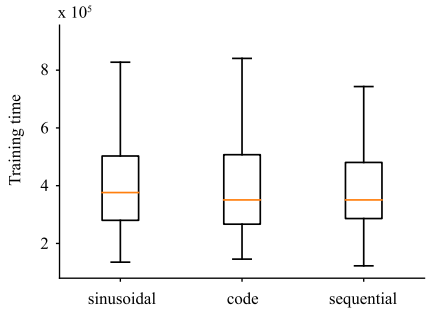

FIG. 7. Demonstrating equivalence between the various perturbation types by measuring their time to train a 2-2-1 network on the 2-bit parity (XOR) problem. Each box plot represents the results of 100 random initializations. The hyperparameters were $\tau_x = 250$, $\tau_\theta = 1$, $\eta = 0.05$, $\tau_p = 1$ (discrete), and $\Delta f = 0.3$ (analog).

2-bit parity problem. In particular, we aimed to show that training can happen in both a purely analog or purely digital way. In practice, many systems have hybrid analog–digital features, and a combination approach may be used.

Figure 7 shows the training time distributions for the 2-bit parity problem using the different perturbation types. The bandwidth for sinusoidal perturbations was set to be $1/2\tau_p$. As expected, we found that the different perturbation types are approximately equivalent in terms of speed of training. This equivalence makes sense when one considers that the feedback from $\tilde{C}$ has a finite bandwidth that must be shared between all the parameters—no matter the encoding (perturbation) scheme, the information carried in that feedback to the parameters will be limited by that finite bandwidth.

E. Operation on noisy or imperfect hardware

The fabrication defects and signal noise present in emerging hardware platforms pose significant challenges for current training techniques in hardware. For example, weight parameters may differ from design or update non-deterministically, causing a discrepancy between the model and actual hardware that can negatively affect training.[8,31,32] For example, in Ref. 32, a simulated network achieves 97.6% accuracy for the MNIST dataset, but the hardware diffractive optical network implementation only obtains 63.9% accuracy after direct transfer of the pre-trained model to their hardware without some *in situ* training, while Ref. 33 shows that a 3% component variation can lead to a 7% drop in classification accuracy without error correction. Reference 8 shows how small discrepancies in modeled parameters can blow-up in general hierarchical problems. In their toy problem, a 0.5% error in a model parameter leads to a 30% error in output after 20 layers. The solutions to these non-idealities can be cumbersome for offline training. For example, the actual value of the weights may have to be regularly measured during training,[31] imperfections may be carefully measured and accounted for in the models,[33,34] or weights may be quantized with low bit-depth, where the bit depth is chosen such that the system can be modeled and controlled accurately despite device-to-device variations. This is common in hardware platforms, such as memristive crossbar arrays[35] or phase change materials,[36] where bit depths of 6-8 can be achieved.

Here, we investigate the effects of three different types of imperfections and noise that could affect hardware systems: (1) stochastic noise on the output cost C_{noise}, (2) stochastic noise on the parameter update θ_{noise}, and (3) per-neuron defects in the activation function, where each neuron has a randomly scaled and offset sigmoidal activation function that is static in time. These tests were performed on the NIST7x7 dataset using the previously described 49-4-4 network with 220 parameters and with $\tau_x = \tau_\theta = 1$ unless otherwise stated.

In the first test, we added Gaussian noise with mean zero and standard-deviation σ_C to the cost, applied every timestep such that noise $C(t) = C_{ideal}(t) + C_{noise}(t;\ \sigma_C)$. For example, in optical hardware, this could be noise due to laser fluctuations. Noise in the cost is also equivalent to any other Gaussian mean-zero noise affecting the accumulated gradient G. Figure 8(a) shows the effect on the

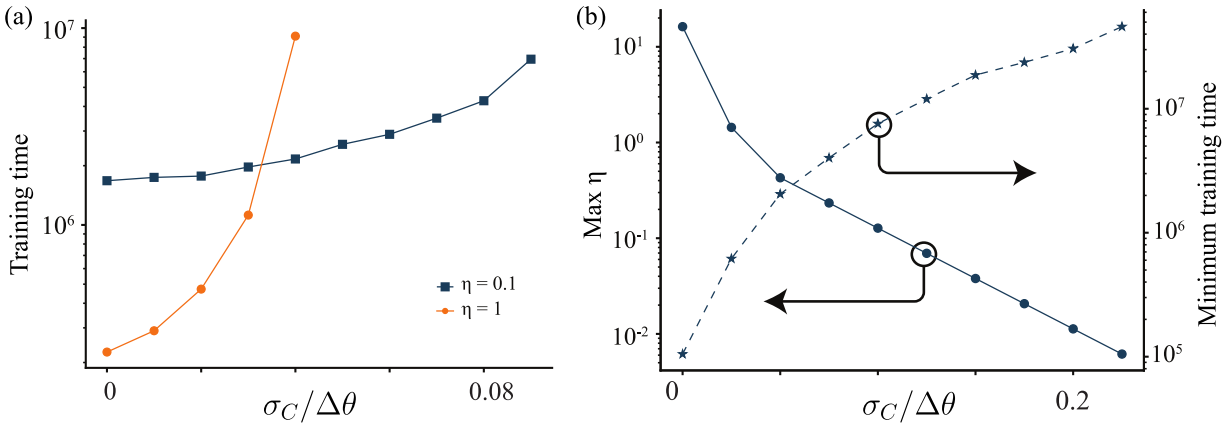

FIG. 8. Effect of noise applied to the cost signal. (a) Training time vs σ_C, the amplitude of the Gaussian noise added to the cost signal C (normalized to the perturbation magnitude $|\tilde{\theta}|$). The training time is the median time to >80% accuracy for 10 random parameter initializations. (b) Effect of noise on maximum η and training time. The maximum η is the largest learning rate for which >80% of the 10 random initializations converged; the corresponding training time at that maximum learning rate is shown on the right axis.

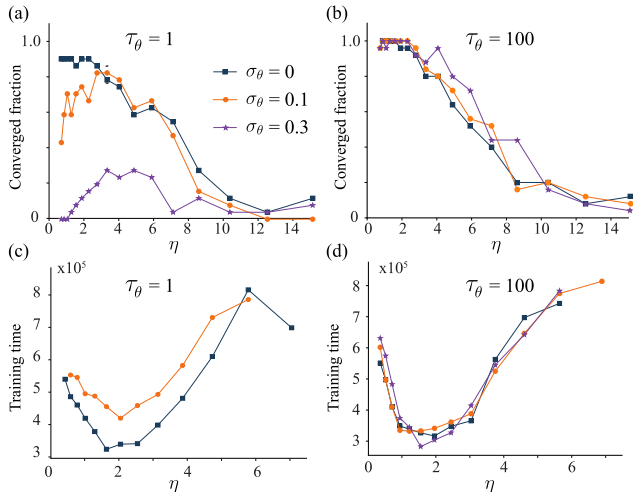

FIG. 9. Effect of noisy (stochastic) parameter updates on solving XOR in a 2-2-1 feedforward network, measured for various noise amplitudes σ_θ. [(a) and (b)] Effect of parameter update noise on the probability of the training converging, as a function of learning rate η. Noise was much more likely to prevent convergence for (a) $\tau_\theta = 1$ than (b) $\tau_\theta = 100$. Convergence was defined as reaching 93% accuracy within 5×10^7 iterations. Statistics are from 25 random parameter initializations. [(c) and (d)] Effect of parameter update noise on training time as a function of the learning rate η for integration times (c) $\tau_\theta = 1$ and (d) $\tau_\theta = 100$. Only η values sets with >50% convergence are shown—in (c), $\sigma_\theta = 0.3$ did not have η values that met this criteria.

training time for different learning rates as σ_C was increased. For a given learning rate, there is a threshold amount of noise below which the training time is minimally changed. However, as cost noise increases, the training time eventually increases and ultimately stops converging.

To see how this noise would affect the minimum training time for optimized learning rates, we also measured the maximum achievable η value for a range of σ_C. Figure 8(b) shows this maximum η value vs cost noise, and corresponding minimum training time. The trend indicates that the lower the cost noise σ_C, the higher we can make η and the faster we can train. Stated in reverse, one can compensate for large amounts of cost noise in a system simply by reducing the learning rate.

In the next test, we analyzed the effect of noisy parameter updates on the training process. For this experiment, whenever any parameter was updated, the update included a randomly applied deviation. Thus, the update rule became

$$\theta \leftarrow \theta - \eta G + \theta_{\text{noise}}, \tag{5}$$

where θ_{noise} was Gaussian with mean zero and standard deviation σ_θ, normalized by $\Delta\theta$, such that $\theta_{\text{noise}} \sim \mathcal{N}(0, \sigma_\theta/\Delta\theta)$. This kind of noise is often seen in analog memories without closed-loop feedback.[37,38]

We found that larger values of σ_θ can prevent convergence entirely [Fig. 9(a)]. Curiously, in the presence of this noise, increasing η can actually improve the convergence of the problem, as highlighted by the $\sigma_\theta = 0.1$ and $\sigma_\theta = 0.3$ lines in Fig. 9(a). We believe this is likely because at very small η values θ_{noise} will overwhelm the very small ηG in Eq. (5). By making the η term larger, one can prevent ηG from being drowned out by θ_{noise}. Obviously, at very large η values, the usual gradient-descent instability starts to dominate and the convergence approaches zero. For a given η, we found that small values of σ_θ marginally increase the training time, but the effect is less significant than simply changing the learning rate η [Fig. 9(c)].

Another way to reduce the impact of θ_{noise} is to increase the integration time of the gradient. When τ_θ is increased, the value of G is accumulated for a longer time and becomes proportionally larger. For instance, when τ_θ is 100 times larger, the value of ηG becomes 100 times larger, meaning the effect of θ_{noise} in Eq. (5) is relatively 100 times smaller. The result can be seen in Figs. 9(b) and 9(d), where even the largest σ_θ value has little effect on the result.

In the final test, we analyzed the effect of including "defects" in the neuronal activation functions. Here, neuronal activation functions were no longer identical sigmoid functions, but had fixed random offsets and scaling that were static in time. These variations were meant to emulate device-to-device variations that may be found in hardware, for instance, in analog VLSI neurons.[39] The sigmoid activation function for each neuron k was modified to a general logistic function $f_k(a) = \alpha_k(1 - e^{-\beta_k(a-a_k)})^{-1} + b_k$. The variations were all Gaussian, and the scaling factors α_k, β_k had a standard deviation σ_a and a mean of 1, while the offset factors a_k, b_k also had a standard deviation of σ_a but were mean-zero.

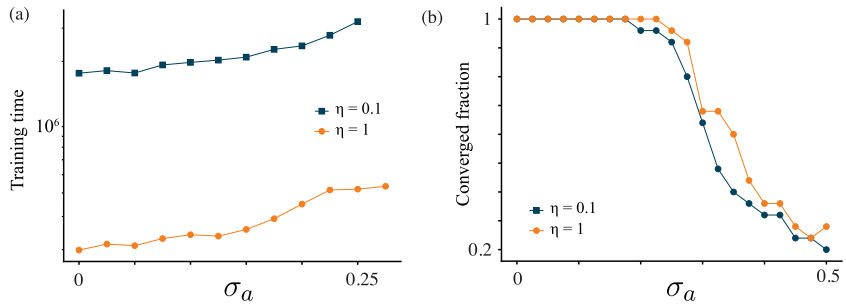

FIG. 10. Effect of adding random offsets and scaling to each neuron's sigmoid activation function. (a) Training time vs σ_a, the standard deviation of activation-function offsets and scaling. (b) Converged fraction vs the standard deviation of the logistic function parameters. The results are statistics of 25 different random network initializations and activation-function randomizations. Convergence is defined as reaching 80% accuracy within the time of the simulation.

As seen in Fig. 10(a), adding defects to the network's activation functions had a relatively small effect on the training time. Even with relatively large variations in the activation functions ($\sigma_a = 0.25$), the network only took about twice as much time to fully train the NIST7x7 dataset. However, deviations that were too large ($\sigma_a > 0.25$) caused issues with the training being able to converge. This is likely due to the output neurons being so scaled and offset that it was no longer possible for them to produce the correct output.

Overall, we found that MGD is robust against different types of noise and fabrication imperfections of a reasonable scale as those that would be seen in a real hardware system. In general, these nonidealities affected the training speed of the network, but did not prevent the desired problem from being solved.

F. Modern dataset results

To assess the scalability and relevance of MGD for larger machine learning problems, we compared MGD and backpropagation on a variety of tasks for different network architectures and hyperparameters. Table II compares the accuracies obtained with MGD and backpropagation for different datasets and various hyperparameter choices (τ_θ, τ_p, η, batch size), with τ_x fixed at 1. To make an effective comparison, for the backpropagation results, we used a basic stochastic gradient descent (SGD) optimizer without momentum. Both strategies used a mean squared error (MSE) cost function. These choices allowed us to avoid confounding effects from more complex strategies, although we note that MGD is capable of implementing several of these more complicated features (e.g., momentum, dropout, etc.). In all cases we used the same network architectures for both sets of results. The networks architectures used were not selected to reach state-of-the-art accuracy values, but instead smaller networks were chosen for purposes of rapid testing and statistical analysis. To improve the accuracy further, more advanced network architectures (e.g., more layers) or optimizers (e.g., adding momentum) would be needed. For the purposes of meaningful comparison, we measured the accuracy obtained in the MGD process after fixed numbers of timesteps. This allowed us to estimate actual training time for various hardware configurations in Sec. IV A.

The CIFAR-10 network was composed of three convolution and max-pool layers followed by a fully connected layer. The convolutional filters were 3×3 with 16, 32, and 64 output channels, respectively, stride 1, and relu activation function outputs. Each convolution was followed by a 2×2 max-pool layer. The final fully connected layer converted the 256 outputs of the convolutional layers to the 10 classes, and no softmax was used. The Fashion-MNIST network was similarly composed, comprising two convolution and max-pool layers followed by a (32×10) fully connected layer and no softmax. The networks used to solve NIST7x7 and XOR were fully connected feedforward networks with sigmoidal activation functions. To make the comparisons in Table II as fair as possible, the backpropagation accuracies of each row were maximized by training the networks with the listed batch size and a variety of η values. The backpropagation accuracy reported is the highest median accuracy obtained for all the η values tested, with the median taken over five random initializations that each ran for 2500 epochs (long after the accuracy had converged).

By training networks on the 2-bit parity, NIST7x7, Fashion-MNIST, and CIFAR-10 datasets, we found that MGD approached the solution-accuracy of backpropagation. However, even after 10^7 steps, we observed the MGD-trained networks still had slightly lower accuracy than could be achieved by backpropagation. This is likely related to convergence criteria of the gradient approximation not being met, due to fixed η values—for instance, the theory underlying the SPSA process only guarantees convergence with a learning rate that asymptotically approaches zero.[13] In general, we observed that while larger learning rates achieved higher accuracies earlier on, lower learning rates achieved higher accuracies at longer times (see, e.g., NIST7x7 result). Therefore, custom learning rates are likely to achieve more optimal training time and accuracy.

The simulations also indicate that changing τ_θ had a marginal effect on the maximum accuracy of MGD for a fixed η value for the larger datasets shown here, with increased values of τ_θ leading to slightly higher final accuracies. This may be because longer τ_θ values produce more-accurate gradient estimates and allow the network to better optimize the weights as the system approaches a local minima in the cost, although more experiments and statistics are needed to verify if this effect is occurs, in general. Fortunately, our results show this effect is relatively small, allowing the hardware-designer flexibility in how often weight updates need to be performed.

TABLE II. Performance of MGD training on four different datasets. The setup, MGD parameters and achieved test accuracies are shown. The best accuracy achieved with training via backpropagation for the same network is shown in the final column for comparison.

Setup		Parameters					Accuracy				
Task	Network	$\lvert\theta\rvert$	τ_θ	τ_p	η	Batch size	10^4 steps	10^5 steps	10^6 steps	10^7 steps	Backprop
2-Bit parity	2-2-1	9	1	1	5	1	100%	100%	100%	100%	100%
N-I-S-T	49-4-4	220	1	1	3	1	38%	81%	94%	97.7%	99.8%
N-I-S-T	49-4-4	220	1	1	0.5	1	22%	45%	93%	98.7%	99.8%
Fashion-MNIST	2-layer CNN	14 378	1	1	9	1000	34.2%	66.3%	79.3%	83.5%	88.6%
Fashion-MNIST	2-layer CNN	14 378	10	1	9	1000	34.3%	66.3%	79.2%	83.4%	88.6%
Fashion-MNIST	2-layer CNN	14 378	100	1	9	1000	35.3%	66.3%	77.7%	84.7%	88.6%
Fashion-MNIST	2-layer CNN	14 378	1000	1	9	1000	35.3%	59.6%	79.1%	86.1%	88.6%
CIFAR-10	3-layer CNN	26 154	1	1	9	1000	12%	23%	43.8%	60.7%	68%

These results show explicitly that MGD is a viable technique for training emerging hardware platforms on real machine learning datasets. Previous theoretical studies have focused on convergence proofs and computation of gradient approximations to varying degrees of accuracy[13,14] or on simplified problems, such as linear activation functions.[40] While these theoretical results are clearly important and necessary, they are not sufficient to give insight into the applicability of these optimization techniques to real machine learning problems on experimental hardware platforms. Meanwhile, previous experimental and simulation studies have for the most part focused on small scale problems for which 100% accuracy is trivial, such as XOR,[15,16] few-bit parity,[17,21,22] the iris dataset,[25] or other similar toy problems.[18,19,26] These are not representative of larger machine learning datasets, which typically have on the order of tens of thousands of training examples and are trained on networks with even more parameters.

These results go further than any of these previous studies, by elucidating the utility of these techniques for larger, modern, machine learning datasets, and implementing a framework that can connect different optimization techniques (SPSA, finite-difference, approximate gradient descent) via specific hardware parameters.

It is also worth noting that the random weight change (RWC) algorithm has been implemented in memristor networks[24,41] previously, and while superficially similar to MGD is functionally very different. RWC is not an approximate gradient descent technique, since the weight update is not scaled by the magnitude of the change in the cost, but rather random perturbations are either kept or discarded based on whether or not they improve the cost. Because of this, it scales more poorly with number of parameters than the optimization techniques we have implemented in the MGD framework.

IV. HARDWARE CONSIDERATIONS

A. Time constants

While the MGD framework is general, individual hardware platforms may have different practical considerations that dictate the optimal choices for the different time constants. Here, we discuss potential issues for implementation on hardware and point out trade-offs that may exist.

Perturbation speed (τ_p): In many hardware platforms, perturbing parameters directly may be difficult or undesired either due to memory-endurance or update-speed concerns. However, it is not necessary to perturb parameters directly—instead, one may perturb a separate element that is in series with the parameter. For instance, in a photonic MZI array, the weights may be implemented with slow (~ms) thermal or mechanical phase shifters, but the small perturbation could be implemented with a fast (~ns) electro-optic modulator, such as a depletion[46] or opto-mechanical[47] modulator. Alternatively, in a memristor system, the perturbation could be implemented with a single sub-threshold transistor in series with the memristor.

For perturbative-style techniques, such as MGD, to function properly, τ_p should be slower than the system's inference time,[14] which includes the cost calculation and broadcast processes. If τ_p is too short, it can introduce misalignment between the input perturbation and resulting cost feedback.

Parameter update (τ_θ): When performing gradient descent on hardware, parameters must be updated multiple times. For a training period of length T, the weights will be updated T/τ_θ times. For some hardware platforms, it may be advantageous to reduce the number of parameter updates, for instance, if the parameters take a long time to update, the update process requires a large amount of energy, or the underlying parameter values are stored in memory elements with a limited lifetime. Fortunately, Table II shows that for networks with large numbers of parameters, τ_θ can be increased without greatly impacting the overall training speed or final accuracy.

Changing training examples (τ_x): The value of τ_x is limited by the speed with which training examples can be input to the network. For most technologies, τ_θ or τ_x will not be limiting factors, as x and $\hat{y}$ samples can be provided by a conventional computer or FPGA at nanosecond timescales. However, it should be noted that MGD will not function correctly if τ_x is shorter than the inference time of the hardware, and this will likely be the practical limit for τ_x. τ_x is also used as a way of controlling the batch size (given by τ_θ/τ_x). The particulars of the task and dataset may determine the desired batch size, and this may have some effect on τ_x, for example, if the minimum allowable value of τ_θ is very long.

Calculation of cost: The calculation of the cost is performed only at the output of the system, and therefore it is acceptable that the hardware used in its computation be more expensive and complex. In integrated electronic systems, the cost can be straightforward to implement, although it may occupy more relatively more chip real-estate than other elements. However, for very exotic hardware platforms, this cost computation may still be difficult to perform on-chip. This can be addressed by using a non-standard cost function or by using a computer to calculate the cost off-chip. However, the input and output to an off-chip computation may limit the speed of the cost computation and also limit the speed of the global broadcast of $\tilde{C}$ and may be another speed bottleneck in some hardware systems.

Based on the literature, some estimates of plausible time constants that could be implemented in hardware are shown in Table III with the corresponding times to solve benchmark tasks, based on the results from Table II. Examples from the literature of hardware platforms that could implement these sets of parameters are shown in the final row. By comparison of these results with the final column in Table III, we see that using realistic estimates for emerging hardware, MGD could be significantly faster than current implementations using backpropagation.

B. Analog and digital implementations

There are a few notable differences for an analog vs a digital hardware implementation of the MGD framework. The discrete case requires one memory element at the network output to store C_0 (sample-and-hold), and a simple subtraction operation to compute $\tilde{C} = C - C_0$. The network also requires some timesteps to be devoted to the measurement of C_0. This means that for the simple case of $\tau_\theta = \tau_p$, only a single additional memory element is required for training the entire hardware system, located at the network cost output.

However, in the case where $\tau_\theta > \tau_p$, an additional memory element is required for every parameter (for instance, an analog capacitor or discrete memory) to perform the gradient integration. The analog case requires a lowpass filter at every parameter element, and a single highpass filter on the network output to convert C to

TABLE III. Approximate training time using MGD for estimated hardware time constants. Data shown are estimated using the same dataset and network architectures used in Table II. The final column shows the time it takes to get to that same accuracy using backpropagation on a standard GPU or CPU.

	HW1	HW2	HW3	Backprop
τ_x (input-sample update time)	100 ns	1 ns	10 ps	n/a
τ_p (perturbation time)	1 ms	10 ns	200 ps	n/a
τ_θ (parameter-update time)	1 ms	1 μs	200 ps	n/a
2-Bit parity training time (10^4 steps)	20 s	200 μs	4 μs	70 ms†
Fashion-MNIST training time (10^6 steps)	33 min	20 ms	400 μs	54 s*
CIFAR-10 training time (10^7 steps)	5.6 h	200 ms	4 ms	480 s*
Examples of hardware	Chip-in-the-loop, integrated photonics with thermo-optic tuning[12,42]	Mem-compute devices,[43] analog VLSI[44]	Superconducting devices,[45] athermal resonant silicon photonic modulator[46]	†CPU/*GPU

$\tilde{C}$. These could be created with simple analog circuits such as RC or LR filters. However, we note the use of a continuous highpass filter can cause implementation issues is some cases. For instance, if η is too large, rapid changes in θ can generate unwanted frequency components that mix with the perturbation input and may negatively affect the gradient approximation. Moreover, if the dataset is discrete, jumps in x can propagate high frequency noise through C and $\tilde{C}$, again negatively affecting the gradient approximation. The simulations demonstrate that, in principle, MGD can be used on different hardware platforms of either an analog or digital nature for network training.

V. DISCUSSION

An interesting analogy can be drawn between the MGD training framework and wireless communication systems. In wireless communications, cell phone users must be able to extract the signal that is relevant to their own communication from a received broadcast that contains multiple users' signals. MGD operates similarly—each parameter is analogous to an individual cell phone user and the global cost broadcast $\tilde{C}$ is analogous to the cell tower transmitter. The broadcasted signal $\tilde{C}$ is available to all parameters, and each parameter extracts its relevant (gradient) information—analogous to voice data—from that broadcasted signal.

The encoding and multiplexing techniques used in wireless communications are broadly termed "multiple access" techniques in communications, and there are several varieties of them, including frequency, time, and code multiplexing. Similarly, in MGD, these multiplexing techniques are directly analogous to the different perturbation types discussed in Sec. III D. For instance, sinusoidal perturbations can be considered a type of frequency-multiplexing: each "user" (parameter) has a unique frequency containing information relevant to it, and must perform a time-integration to extract that information from the $\tilde{C}$ signal that contains many other, irrelevant signals. Likewise, as mentioned in Sec. III D, the discrete $\{-\Delta\theta, +\Delta\theta\}$ perturbation type is analogous to the code-multiplexing techniques used in modern cell phones.

It is important to note that for a fixed bandwidth broadcast, all multiplexing techniques have the same nominal spectral efficiency. However, the particulars of real systems can greatly affect which multiplexing techniques are most effective. For example, because code-multiplexing encodes a user's signal out over a wide time window and a spread spectrum, it has been shown to have an improved robustness to multipath distortion and timing errors compared to time-multiplexing.

Similarly, the various perturbation types described earlier may operate better or worse depending on the particulars of the hardware network and environment. Newly developed communications techniques may also be directly applicable to MGD training in real hardware systems. Key to its usage in hardware, each parameter can operate like a cell phone—as an independent device with a receiver, processor, and memory using only information available locally and from the global broadcast. This contrasts with most other training algorithms, where error signals must propagate through the entire network.[10] While trivial in software, this type of upstream information-flow means that hardware neurons must (1) have additional memory to process the error signal, (2) perform different operations for forward-inference and reverse error propagation, and (3) operate in a synchronized fashion to propagate the error correctly.

In systolic arrays, "broadcast" is often discouraged because the time and energy of communication is proportional to the number of inputs by CV^2 (with C growing with the number of inputs). Furthermore, as the capacitance is larger, then a larger amplifier is needed for the communication that also requires energy and area. However, in other types of analog or emergent hardware, broadcast may be a more energy and space efficient process. For example, in optical hardware, a global optical signal could be accessible in a simple slab waveguide mode. Alternatively, in analog electrical hardware, a single electrical plane could be used to carry the $\tilde{C}$ signal back to the individual parameters.

The MGD framework may also be much more biologically plausible than other algorithms. There are a host of algorithms being developed to address the mystery of how the brain learns and to use biology as inspiration to find more hardware-friendly approaches.[10,48,49] One of the major issues with backpropagation from both a practical and bio-plausibility standpoint is the need for a separate type of operation in the forward (inference) and backward (learning) phases. This is not the case for MGD: in MGD, the training is done in the same configuration used for inference. In a biological analogy, a global cost signal $\tilde{C}$ could be modeled as some type of quasi-global chemical signal in the brain.

In the future, the MGD algorithm could be extended such that the cost is not global to every synapse, but rather to specific clusters of neurons/synapses. There are chemical signals in the brain with these types of properties. The time scales needed for perturbation may line up with biologically observed effects, such as short-term plasticity (min)[50,51] and synaptic noise.[52] MGD is also consistent with "three-factor" learning rules for which there is mounting experimental evidence.[51] Additionally, although not explored in this work, MGD should work in a spiking model. Similar algorithms that use perturbations in neuron conductance,[53] probabilistic synapse transmission,[54] or the stochastic nature of spike-timing arrival[55] for reward-based learning have been explored in small spiking networks.

VI. CONCLUSIONS

In this paper, we show that with realistic timescales for emerging hardware, training via MGD could be orders of magnitude faster than backpropagation in terms of wall-clock time-to-solution on a standard GPU/CPU. The MGD framework allows the implementation of multiple optimization algorithms using a single, global, cost signal and local parameter updates. The style of algorithm used (e.g., finite-difference, coordinate-descent, and SPSA) can be adjusted via the tuning of the MGD time-constants and can even be adjusted during training if desired. Because it is a model-free perturbative technique (sometimes called zeroth order optimization), it is applicable to a wide range of systems—it can be applied to both analog and digital hardware platforms, and it can be used in the presence of noise and device imperfections. This overcomes a major barrier to using hardware platforms based on emerging technologies, which are often difficult to train.

Going forward, there are many opportunities to explore the application of MGD on both large and small hardware systems. The most direct next step will be to test the training performance in existing hardware, for example, on photonic[42] or memristive crossbar hardware using a chip-in-the-loop process. This can occur without even redesigning the hardware, as MGD only requires the ability to input samples, capture inference output, and modify parameters when in a chip-in-the-loop configuration. When used in this fashion, the speed will most likely be limited by system I/O. For example, perturbations can be injected directly to the hardware from an external computer, and that same computer could capture the changes in cost and perform the gradient approximation and calculate the weight updates. This would allow testing of the algorithm without any changes to the hardware. Ultimately, to overcome I/O limitations local circuits can be designed be implemented for autonomous online training. Although not examined in detail here, in this case, many of the hardware platforms examined would likely have several orders of magnitude improvement in terms of energy usage as well.

There is also lots of room for improvement on tuning the technique and its hyper-parameters. In this paper, we performed very little hyperparameter optimization or regularization, and so there are likely opportunities for further optimization by examining techniques, such as dropout, momentum, and regularization. Also of interest is examining the node-perturbation version of the algorithm[56] on large datasets, which could significantly reduce the number of perturbative elements and speed up the training process.

While in this paper we only examined feed forward networks, it has already been demonstrated that is possible to use perturbative techniques to train recurrent neural networks,[57] spiking networks,[54] and other non-standard networks at small scale; future work remains to demonstrate their utility on problems of modern interest. This will have applicability to a wide range of neuromorphic hardware and other physical neural networks.

Ultimately, MGD is well-suited to implementation directly on-chip with local, autonomous circuits. Although significant work remains before autonomous learning can be truly implemented without the participation of an external computer, such a device would be truly enabling for remote and edge-computing applications, allowing training in-the-field on real-time data.

ACKNOWLEDGMENTS

The authors acknowledge helpful discussions with the NIST NCSG community. This research was funded by the NIST (https://ror.org/05xpvk416) and the University of Colorado Boulder (https://ror.org/02ttsq026).

AUTHOR DECLARATIONS

Conflict of Interest

The authors have no conflicts to disclose.

Author Contributions

Adam N. McCaughan: Conceptualization (equal); Formal analysis (equal); Investigation (equal); Software (equal); Writing – original draft (equal). **Bakhrom G. Oripov**: Formal analysis (equal); Software (equal); Writing – review & editing (equal). **Natesh Ganesh**: Formal analysis (supporting). **Sae Woo Nam**: Formal analysis (equal); Funding acquisition (equal); Project administration (equal). **Andrew Dienstfrey**: Conceptualization (equal); Formal analysis (equal); Project administration (equal). **Sonia M. Buckley**: Conceptualization (equal); Formal analysis (equal); Methodology (equal); Project administration (equal); Writing – original draft (equal).

DATA AVAILABILITY

The data that support the findings of this study are openly available in multiplexed-gradient-descent-paper at https://github.com/amccaugh/multiplexed-gradient-descent-paper.[30]

REFERENCES

[1] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, arXiv:1705.06963 (2017).
[2] W. Haensch, T. Gokmen, and R. Puri, Proc. IEEE **107**, 108 (2019).
[3] P. A. Merolla, J. V. Arthur, R. Alvarez-icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, Science **345**, 668 (2014).
[4] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty,

S. Mccoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang, IEEE Micro **38**, 82 (2018).
[5]K. Meier, in 2015 IEEE International Electron Devices Meeting (IEDM), 2015.
[6]A. Mehonic and A. J. Kenyon, Nature **604**, 255 (2022).
[7]N. C. Thompson, K. Greenewald, K. Lee, and G. F. Manso, "The computational limits of deep learning," arXiv:2007.05558 (2020).
[8]L. G. Wright, T. Onodera, M. M. Stein, T. Wang, D. T. Schachter, Z. Hu, and P. L. McMahon, Nature **601**, 549 (2022).
[9]F. Crick, Nature **337**, 129 (1989).
[10]T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton, Nat. Rev. Neurosci. **21**(6), 335 (2020).
[11]J. Kiefer and J. Wolfowitz, Ann. Math. Stat. **23**, 462 (1952).
[12]Y. Shen, N. C. Harris, S. Skirlo, M. Prabhu, T. Baehr-Jones, M. Hochberg, X. Sun, S. Zhao, H. Larochelle, D. Englund, and M. Soljačić, Nat. Photonics **11**, 441 (2017).
[13]J. C. Spall, IEEE Trans. Autom. Control **37**, 332 (1992).
[14]A. Dembo and T. Kailath, IEEE Trans. Neural Networks **1**, 58 (1990).
[15]T. Matsumoto and M. Koga, Electron. Lett. **26**, 1136 (1990).
[16]M. Jabri and B. Flower, IEEE Trans. Neural Networks **3**, 154 (1992).
[17]J. Alspector, R. Meir, B. Yuhas, A. Jayakumar, and D. Lippe, *in NIPS '92: Proceedings of the 5th International Conference on Neural Information Processing Systems* (Morgan-Kaufmann, 1992), pp. 836–844.
[18]D. B. Kirk and D. Kerns, *Advances in Neural Information Processing Systems*, 5 (Morgan-Kaufmann, 1992), pp. 789–796.
[19]Y. Maeda, H. Hirano, and Y. Kanata, Neural Networks **8**, 251 (1995).
[20]G. Cauwenberghs, IEEE Trans. Neural Networks **7**, 346 (1996).
[21]H. Miyao, K. Noguchi, M. Koga, and T. Matsumoto, *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 80 (Elsevier Science, 1997).
[22]A. J. Montalvo, R. S. Gyurcsik, and J. J. Paulos, IEEE Trans. Neural Networks **8**, 413 (1997).
[23]Y. Maeda and T. Tada, IEEE Trans. Neural Networks **14**, 688 (2003).
[24]S. P. Adhikari, H. Kim, R. K. Budhathoki, C. Yang, and L. O. Chua, IEEE Trans. Circuits Syst. **62**, 215 (2015).
[25]C. Wang, L. Xiong, J. Sun, and W. Yao, Nonlinear Dyn. **95**, 2893 (2019).
[26]S. Bandyopadhyay, A. Sludds, S. Krastanov, R. Hamerly, N. Harris, D. Bunandar, M. Streshinsky, M. Hochberg, and D. Englund, arXiv:2208.01623 (2022).
[27]A. G. Baydin, B. A. Pearlmutter, D. Syme, F. Wood, and P. Torr, "Gradients without backpropagation," arXiv:2202.08587 (2022).
[28]D. Silver, A. Goyal, I. Danihelka, M. Hessel, and H. van Hasselt, in Proceedings of the International Conference on Learning Representations (ICLR), 2022.
[29]M. Ren, S. Kornblith, R. Liao, and G. Hinton, "Scaling forward gradient with local losses," arXiv:2210.03310 (2022).
[30]A. N. McCaughan, B. G. Oripov, and S. M. *Buckley* (2023). "Multiplexed gradient descent code repository," Github. https://github.com/amccaugh/multiplexed-gradient-descent-paper
[31]B. Hoskins, M. Fream, M. Daniels, J. Goodwill, A. Madhavan, J. Mcclelland, O. Yousuf, G. Adam, W. Ma, T. Hoang, M. Branstad, M. Liu, R. Madsen, M. Lueker-Boden, and M. Lueker, in *International Conference on Neuromorphic Systems* (Association for Computing Machinery, 2021), Vol. 5.
[32]T. Zhou, X. Lin, J. Wu, Y. Chen, H. Xie, Y. Li, J. Fan, H. Wu, L. Fang, and Q. Dai, Nat. Photonics **15**, 367 (2021).
[33]S. Bandyopadhyay, R. Hamerly, and D. Englund, Optica **8**(10), 1247–1255 (2021).
[34]M. Y-S. Fang, S. Manipatruni, C. Wierzynski, A. Khosrowshahi, and M. R. DeWeese, Opt. Express **27**(10), 14009–14029 (2019).
[35]M. Hu, C. E. Graves, C. Li, Y. Li, N. Ge, E. Montgomery, N. Davila, H. Jiang, R. S. Williams, J. J. Yang, Q. Xia, and J. P. Strachan, Adv. Mater. **30**, 1705914 (2018).
[36]S. Ambrogio, P. Narayanan, H. Tsai, R. M. Shelby, I. Boybat, C. di Nolfo, S. Sidler, M. Giordano, M. Bodini, N. C. P. Farinha, B. Killeen, C. Cheng, Y. Jaoudi, and G. W. Burr, Nature **558**, 60 (2018).
[37]S. Ambrogio, S. Balatti, A. Cubeta, A. Calderoni, N. Ramaswamy, and D. Ielmini, IEEE Trans. Electron Devices **61**, 2912 (2014).
[38]J. Hazra, M. Liehr, K. Beckmann, S. Rafiq, and N. Cady, *2019 IEEE International Integrated Reliability Workshop (IIRW)* (Institute of Electrical and Electronics Engineers, Inc., 2019), pp. 1–4.
[39]C. Merkel and D. Kudithipudi, *2015 28th International Conference on VLSI Design* (IEEE Computer Society, 2015), pp. 99–104.
[40]J. Werfel, X. Xie, and H. S. Seung, Neural Comput. **17**, 2699 (2005).
[41]C. Yang, H. Kim, S. P. Adhikari, and L. O. Chua, Sensors **17**, 16 (2017).
[42]A. N. Tait, T. F. de Lima, E. Zhou, A. X. Wu, M. A. Nahmias, B. J. Shastri, and P. R. Prucnal, Sci. Rep. **7**, 7430 (2017).
[43]G. W. Burr, R. M. Shelby, A. Sebastian, S. Kim, S. Kim, S. Sidler, K. Virwani, M. Ishii, P. Narayanan, A. Fumarola, L. L. Sanches, I. Boybat, M. Le Gallo, K. Moon, J. Woo, H. Hwang, and Y. Leblebici, Adv. Phys.: X **2**, 89 (2017).
[44]Y. Kohda, Y. Li, K. Hosokawa, S. Kim, R. Khaddam-Aljameh, Z. Ren, P. Solomon, T. Gokmen, S. Rajalingam, C. Baks, W. Haensch, and E. Leobandung, in *Technical Digest - International Electron Devices Meeting, IEDM* (IEEE, 2020), Vol. **36**.2.1.
[45]M. Schneider, E. Toomey, G. Rowlands, J. Shainline, P. Tschirhart, and K. Segall, Supercond. Sci. Technol. **35**, 053001 (2022).
[46]E. Timurdogan, C. M. Sorace-Agaskar, J. Sun, E. Shah Hosseini, A. Biberman, and M. R. Watts, Nat. Commun. **5**, 4008 (2014).
[47]M. Dong, G. Clark, A. J. Leenheer, M. Zimmermann, D. Dominguez, A. J. Menssen, D. Heim, G. Gilbert, D. Englund, and M. Eichenfield, Nat. Photonics **16**, 59–65 (2022).
[48]F. Zenke and E. O. Neftci, Proc. IEEE **109**, 935 (2021).
[49]B. Scellier and Y. Bengio, Front. Comput. Neurosci. **11**, 24 (2017).
[50]A. Soltoggio, Biol. Cybern. **109**, 75 (2015).
[51]W. Gerstner, M. Lehmann, V. Liakoni, D. Corneil, and J. Brea, Front. Neural Circuits **12**, 53 (2018).
[52]B. A. Rowland, A. S. Maida, and I. S. N. Berkeley, Connect. Sci. **18**, 69 (2007).
[53]I. R. Fiete and H. S. Seung, Phys. Rev. Lett. **97**, 048104 (2006).
[54]H. S. Seung, Neuron **40**, 1063 (2003).
[55]X. Xie and H. S. Seung, Phys. Rev. E **69**, 10 (2004).
[56]B. Flower and M. Jabri, *in NIPS '92: Proceedings of the 5th International Conference on Neural Information Processing Systems* (Morgan-Kaufmann, 1992), pp. 212–219.
[57]G. Cauwenberghs, *Advances in Neural Information Processing Systems*, 5 (Morgan-Kaufmann, 1992), pp. 244–251.