RESEARCH ARTICLE | APRIL 17 2025

Scaling of hardware-compatible perturbative training algorithms

Special Collection: Neuromorphic Technologies for Novel Hardware AI

B. G. Oripov ; A. Dienstfrey ; A. N. McCaughan ; S. M. Buckley

APL Mach. Learn. 3, 026107 (2025)
https://doi.org/10.1063/5.0258271

Scaling of hardware-compatible perturbative training algorithms

Cite as: APL Mach. Learn. 3, 026107 (2025); doi: 10.1063/5.0258271
Submitted: 15 January 2025 • Accepted: 4 April 2025 •
Published Online: 17 April 2025

B. G. Oripov,[1] A. Dienstfrey,[2] A. N. McCaughan,[2] and S. M. Buckley[2,a)]

AFFILIATIONS

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

Note: This paper is part of the APL Machine Learning Special Topic on Neuromorphic Technologies for Novel Hardware AI.
[a)] **Author to whom correspondence should be addressed:** sonia.buckley@nist.gov

ABSTRACT

In this work, we explore the capabilities of multiplexed gradient descent (MGD), a scalable and efficient perturbative zeroth-order training method for estimating the gradient of a loss function in hardware and training it via stochastic gradient descent. We extend the framework to include both weight and node perturbation and discuss the advantages and disadvantages of each approach. We investigate the time to train networks using MGD as a function of network size and task complexity. Previous research has suggested that perturbative training methods do not scale well to large problems since in these methods, the time to estimate the gradient scales linearly with the number of network parameters. However, in this work, we show that the time to reach a target accuracy—that is, actually solve the problem of interest—does not follow this undesirable linear scaling and in fact often decreases with network size. Furthermore, we demonstrate that MGD can be used to calculate a drop-in replacement for the gradient in stochastic gradient descent, and therefore, optimization accelerators such as momentum can be used alongside MGD, ensuring compatibility with existing machine learning practices. Our results indicate that MGD can efficiently train large networks on hardware, achieving accuracy comparable with backpropagation, thus presenting a practical solution for future neuromorphic computing systems.

I. INTRODUCTION

Machine learning (ML) algorithms are fundamentally altering our interactions with technology, driven predominantly by artificial neural networks. However, the significant costs associated with training and deploying these algorithms—largely stemming from energy expenditures—pose a substantial barrier to their scalability and broader adoption. Industry leaders have voiced concerns over the energy demands of conventional complementary metal–oxide–semiconductor (CMOS) hardware used in ML, advocating for substantial investments in innovative hardware solutions.[1] In contrast to this, the human brain achieves similar computational feats at a fraction of the energy cost, suggesting that brain-inspired hardware represents a promising direction. In this context, analog neuromorphic hardware offers a promising solution to the energy challenges being faced by current hardware technologies. However, training on analog hardware has proved more difficult than on its digital counterparts. In this article, we demonstrate the effectiveness of multiplexed gradient descent (MGD), a general perturbative training framework, in matching the accuracy of the backpropagation algorithm in identical network architectures, even for large networks ($>10^6$ parameters). We also address common misconceptions regarding perturbative training methods and show that they can indeed scale, contrary to prevailing sentiment in the field.[2,3]

Designing dedicated neuromorphic hardware introduces several challenges, particularly when considering the methods for training such systems. Training an ML model amounts to minimizing a specified loss function. In ML contexts, this minimization proceeds by gradient descent and its various extensions. In nearly all standard computing frameworks, this gradient is computed by reverse-mode automatic differentiation and is referred to as backpropagation. While this algorithm is efficient in software, implementing backpropagation in hardware poses significant difficulties including the requirement that the in-hardware computational path

can be reversed, substantial memory at each neuron, and the necessity of computing the derivative of the activation function. Due to these challenges, implementation of backpropagation in analog hardware to date[4–6] has typically involved a computer in the loop to implement part of the computation, or been limited to relatively small networks.[7] To avoid the difficulties of a full hardware implementation of backpropagation, the field has explored several alternative training approaches.

One approach is to conduct training in a traditional computer using a model of the hardware, transfer the resulting weights into devices, and restrict the in-hardware computation to inference tasks only. This solution can reduce deployment costs in many cases, as a single training simulation can determine parameters for many inference instantiations. However, discrepancies between the simulated hardware model and the actual hardware can result in diminished accuracy. Developing training algorithms that produce networks that are more robust to device-to-device variations is a significant line of research.[8] In addition, this approach is less viable for situations that require *in situ* adaptability. Various "computer-in-the-loop" strategies[9] can help with these issues, for example, by implementing the forward pass in hardware and calculating the individual weight updates via simulation.[10]

Another strategy employs Hebbian learning, a simpler learning paradigm based on the empirical observation that the connections between neurons strengthen when they fire synchronously. Although straightforward to implement in hardware,[11,12] Hebbian learning lacks the general applicability and mathematical rigor of gradient-based methods and does not guarantee convergence to a solution. Recent insights suggest that the brain's learning mechanisms might involve more complex three-factor rules,[13] indicating that Hebbian learning might not fully capture the neural learning processes.

In contrast, perturbative methods offer a model-free[14] stochastic gradient-descent approach, treating the hardware as a black box, and applying small perturbations to estimate the gradient and, therefore, minimize network cost using the same optimization algorithm as traditional ML. This approach does not rely on a model of the network's operation, allowing it to be applied across various hardware platforms. Despite early interest[14–25] and ease of implementation, perturbative techniques fell out of favor due to the poor scaling of time to estimate the gradient. Critically, gradient estimation time was assumed to be a good proxy for training time.[2,3,26] In this paper, we test this assumption and find that the connection between gradient and training accuracy is much more nuanced. Our findings are bolstered by a number of recent papers that have also found perturbative techniques to be more effective than assumed[27–30] and similarly are convenient to implement in hardware.[31]

Multiplexed gradient descent (MGD) is a model-free gradient descent framework that attempts to generalize earlier perturbative approaches[14,32] in a hardware-friendly way by defining a set of three time constants for the perturbation process that correspond to the time between weight updates, time between sample updates, and time between perturbation updates. By varying these time constants in a given hardware system, a wide variety of numerical gradient descent techniques (e.g., coordinate descent, simultaneous perturbation stochastic approximation (SPSA), etc.) can be achieved. In previous work,[27] we introduced MGD and examined the speed of gradient estimation and training time of different perturbative techniques given particular hardware parameters. For modest-sized networks and limited architectures, the results indicated that in-hardware training with MGD could be performed faster than backpropagation on a standard graphics processing unit (GPU).

In this study, we extend the application of MGD to train larger and deep feedforward neural networks. We show that perturbative techniques can match the accuracy of backpropagation for networks of up to one million parameters. We evaluate the scaling of both the time to estimate the gradient and the time to train to a given accuracy as a function of network size. As expected, we find that the speed of gradient convergence can be estimated analytically as a function of network size. However, the time to attain a converged approximation of the gradient is not representative of the time to train a network. This result agrees with previous work showing that an accurate gradient estimate is not necessarily required for online gradient descent.[33] Furthermore, in this study, we perform a more systematic analysis of different perturbative approaches—weight perturbation[14] and node perturbation[17]—and discuss the trade-offs associated with their implementations. For example, while the time to estimate the gradient does scale better for dense networks with node perturbation compared to weight perturbation, this is not true for arbitrary architectures[34]—as one example, convolutional networks may have more nodes than weights. Finally, we show that standard optimization algorithms such as momentum and the Adam optimizer can be implemented in the MGD framework. This is important, as in practice, most ML implementations rely on more than vanilla backpropagation, and therefore, it is likely that more advanced optimization techniques will be required in the MGD framework as well. This is supported by recent results.29 Together, these results suggest that MGD represents a promising class of algorithms for training new emerging neuromorphic hardware.

II. MULTIPLEXED GRADIENT DESCENT

Multiplexed gradient descent provides an approach for model-free, *in situ* training of hardware-based implementations of neural networks. In this section, we expand upon the ideas originally presented in Ref. 27. We first generalize the connection between perturbation parameters and neural network weights. In Ref. 27, these two variable classes were the same—that is, every individual weight had its own perturbation—resulting in what is typically called the "weight perturbation" model. This model was investigated exclusively in Ref. 27. By relieving this constraint, we are able to implement a "node perturbation" model. In the following, we define node perturbation and study its impact on training efficiency. Other perturbation models are also possible although not investigated here. Next, we recast the MGD algorithm in a more formal framework. As a result, we prove a theorem stating that for a linear cost function or, equivalently, for sufficiently small perturbation magnitudes, the principal random variable defined in the MGD formalism is an unbiased estimator of the true parameter gradient. We also compute its second-order statistics. Finally, we introduce the distinction between gradient convergence and network accuracy. While the former has generally been the focal point for prior studies in perturbative training of neural networks,[3,26] our numerical studies in the following sections call into question whether this attention is fully warranted.

A. Weight and node perturbation

To present the MGD idea in its most general form, we distinguish two classes of parameters. At the hardware level, one considers the set of *perturbation parameters,*

$$\Theta = (\theta_1, \theta_2, \ldots, \theta_K).$$

These are the physical quantities—for example, voltages, currents, conductances, and optical power—that can be modified most readily by the hardware engineer. The precise definition depends on the physics and interface of the hardware-based model implementing the neural network and can change from one design to the next. For the neural network, one considers the usual mathematical model of a parameterized non-linear function. The parameters are referred to as "weights" and "biases" that we combine into a large vector, which we will simply refer to as the weights,

$$\mathbf{W} = (w_1, \ldots, w_N).$$

The hardware-based implementation of the neural network takes input $\mathbf{x}$ to output $\mathbf{y}$. The map depends on both vectors,

$$\mathbf{y} = f(\mathbf{x}; \mathbf{W}, \Theta).$$

We consider a supervised learning problem in which we are provided with labeled training data $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_T, \mathbf{y}_T)\}$. For a given input $\mathbf{x}_t$, we use $\hat{\mathbf{y}}_t = f(\mathbf{x}_t; \mathbf{W}; \Theta)$ to denote the associated inference. We assume a hardware implementation of a differentiable residual function that evaluates the quality of the network inference in comparison with the target, $\rho(\hat{\mathbf{y}}_t, \mathbf{y}_t)$. We also assume a hardware-based accumulator that computes the empirical loss as the mean of the residual over some collection of training data,[35]

$$\mathrm{C}(\mathbf{W}, \Theta) := \frac{1}{T}\sum_{t=1}^{T} \rho(f(\mathbf{x}_t; \mathbf{W}, \Theta), \mathbf{y}_t). \tag{1}$$

Here, the sum could be over the full set of training data or some minibatch. The training problem consists of minimizing the loss as a function of $\mathbf{W}$. Within the ML community, variations of gradient descent are nearly universally used for this task. In all cases, one requires an estimate of the gradient,

$$\nabla_{\mathbf{W}}\mathrm{C} := \left(\frac{\partial \mathrm{C}}{\partial w_1}, \frac{\partial \mathrm{C}}{\partial w_2}, \ldots, \frac{\partial \mathrm{C}}{\partial w_N}\right). \tag{2}$$

In computational networks, backpropagation is generally used to evaluate the above-mentioned gradient.

The analytic details of backpropagation do not readily lend themselves to hardware-based implementation. By contrast, the premise of hardware-based ML entails that there are parameters whose variation have measurable impact on network performance. MGD amounts to a strategy for organizing these perturbative degrees of freedom, so as to estimate

$$\nabla_{\Theta}\mathrm{C} := \left(\frac{\partial \mathrm{C}}{\partial \theta_1}, \frac{\partial \mathrm{C}}{\partial \theta_2}, \ldots, \frac{\partial \mathrm{C}}{\partial \theta_K}\right). \tag{3}$$

The central assumption is that in a neighborhood of a fixed parameter vector, $\|\mathbf{W} - \mathbf{W}_0\| < \delta$, variation in $f(\mathbf{x}; \mathbf{W}, \mathbf{0})$ as a function of $\mathbf{W}$ can be inferred from variation in $f(\mathbf{x}; \mathbf{W}_0, \Theta)$ as a function of Θ. More specifically, we assume a relationship between gradients,

$$\nabla_{\Theta}\,\mathrm{C} \leftrightarrow \nabla_{\mathbf{W}}\,\mathrm{C}.$$

This connection is used to estimate the gradient of a cost function with respect to the weights after which training proceeds by gradient descent and its variants.

In the simplest setting, the weights themselves are directly perturbable,

$$\mathbf{W} = \mathbf{W}_0 + \Theta. \tag{4}$$

In this case, the two gradients (2) and (3) are one and the same. This is referred to as *weight perturbation.* A different strategy relies on the typical construction of neural network functions as compositions of an affine transformation, followed by a non-linear activation function. In this case, one can consider a perturbative degree of freedom added to the output of the affine transformation,[36]

$$\mathbf{x}^{(\ell+1)} = \sigma\left(\mathbf{W}^{(\ell)}\mathbf{x}^{(\ell)} + \mathbf{b}^{(\ell)} + \Theta\right), \quad \text{for } \Theta = \mathbf{0}. \tag{5}$$

For any fixed training instance, we abbreviate the loss as $\rho_t = \rho(f(\mathbf{x}_t; \mathbf{W}, \Theta), \mathbf{y}_t)$. The chain rule then implies that

$$\frac{\partial \rho_t}{\partial b_k} = \frac{\partial \rho_t}{\partial \theta_k}, \tag{6a}$$

$$\frac{\partial \rho_t}{\partial w_{k,j}} = \frac{\partial \rho_t}{\partial \theta_k} x_j^{(\ell)}. \tag{6b}$$

Summing over all training instances in (1) gives the gradient of the cost. As the perturbation parameter in (5) varies the input into an activation function and the latter are depicted as nodes in the computational graph, this is referred to as *node perturbation*. Note that a fully hardware-based implementation of node perturbation requires a multiplication circuit to infer $\nabla_{\mathbf{W}}\mathrm{C}$ from $\nabla_{\Theta}\mathrm{C}$, as shown in Eq. (6b).

These two perturbation strategies are represented graphically in Fig. 1. Figure 1(a) depicts the weight perturbation algorithm, where individual weights are perturbed and then correlated with the corresponding change in the cost. As we will see in the following, perturbation gradient calculation time scales linearly with

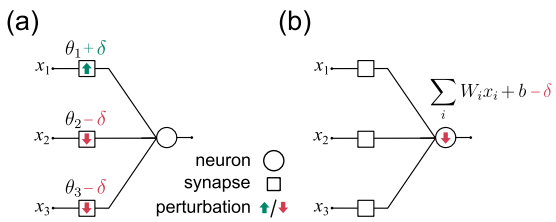

FIG. 1. Illustration of weight perturbation vs node perturbation. (a) Weight perturbation, where each weight is perturbed individually. Weights can be updated entirely within the synapse by combining the global cost-feedback signal with the local perturbation. (b) Node perturbation, where only the summed input to each neuron is perturbed. Here, weight updates must be performed through a one-step backpropagation process that computes the error at the neuron input and then passes it backward (through a multiplication) to the synapse.

TABLE I. Number of parameters (weights) and activations (nodes) for each layer to be trained in a medium sized network evaluated in this work. In this specific example, the depth of the input convolutional layer is $d = 48$.

Layer	Layer type	Kernel	Num. parameters	Num. activations
1	Convolutional	$3 \times 3 \times 1 \times d$	480	37 632
2	Convolutional	$3 \times 3 \times d \times d$	20 784	37 632
3	MaxPool	2×2	⋯	⋯
4	Convolutional	$3 \times 3 \times d \times 2d$	41 568	18 816
5	Convolutional	$3 \times 3 \times 2d \times 2d$	83 040	18 816
6	MaxPool	2×2	⋯	⋯
7	Convolutional	$3 \times 3 \times 2d \times 4d$	166 080	9408
8	Convolutional	$3 \times 3 \times 4d \times 4d$	331 968	9408
9	MaxPool	2×2	⋯	⋯
10	Dense	$36d \times 4d$	331 968	192
11	Dense	$4d \times 4d$	37 056	192
12	Dense	$4d \times 10$	1930	10
Total			1 014 874	132 106

$\dim(\Theta) = K$. In the weight perturbation model, $K = N$. In Ref. 27, we showed in detail how weight perturbation could be implemented in hardware and can be used for emergent learning in a neuromorphic hardware system. However, there has long been concern that the scaling of this algorithm is too slow for larger modern datasets. Figure 1(b) depicts the node perturbation algorithm,[17] where the perturbation is instead applied to the input to the activation function and a single-layer backpropagation is performed to compute the weight update. In this case, the gradient calculation time scales with the number of nodes in the network. This is significant for dense layers: where the number of weights scales as N and the number of activations or nodes scale as $K = \sqrt{N}$. However, for different layer types, this may not be the case. For example, in convolutional layers, there may be more activations than weights. This can be seen in Table I, where, for example, layers 1 and 2 have more activations than weights. The ubiquity of convolutional layers in modern machine learning motivates us to compare the two approaches in terms of speed and then their implementation in hardware. We note that single-layer backpropagation is still a completely local learning rule—signals still do not need to propagate backward through weights or activations.

B. Perturbative gradient estimation

We introduce perturbations as a function of time,

$$\Theta(t) := (\theta_1(t), \theta_2(t), \ldots, \theta_K(t)). \tag{7}$$

Many different types of perturbations can be implemented.[14,27] For example, components of Θ can be assigned distinct frequencies or they can be associated with elements of an orthogonal code. In this work, we used the later, Bernoulli perturbations where simultaneous discrete perturbations of $\theta_i(t) = \pm\delta$ for every parameter every time step. We consider time to be discrete, $t = 1, \ldots, \tau_\theta$, and each parameter is associated with an independent random variable. Thus, (7) represents a collection of $K \cdot \tau_\theta$ random variables in total. We assume that all random variables are independent, and that the distributions are fixed over time and are mean zero.

Consequently, the covariance matrix is diagonal and constant over time,

$$\mathbf{\Lambda} := \mathrm{E}\left(\Theta(t)\Theta(t)^T\right) = \mathrm{E}\left(\Theta(t')\Theta(t')^T\right) = \mathrm{diag}\,(\sigma_1^2, \ldots, \sigma_K^2). \tag{8}$$

Multiplexed gradient descent uses a correlation analysis of the perturbed loss function to provide an estimate of its gradient. By Taylor expansion, we have (the dependence on $\mathbf{W}$ is omitted for readability)

$$\Delta C(\Theta(t)) := C(\Theta(t)) - C(\mathbf{0}) = \nabla_{\mathbf{\Theta}} C \cdot \Theta(t) + \mathcal{O}(\theta_i(t)\theta_j(t)). \tag{9}$$

If we assume that the perturbations are characterized by a small scale, $\max_{i,t} \mathrm{E}(\theta_i^2(t)) \leq \epsilon$, then for ϵ sufficiently small, one may approximate ΔC by

$$\Delta C(t) \approx \nabla_{\mathbf{\Theta}} C \cdot \Theta = \sum_{i=1}^{K} \frac{\partial C}{\partial \theta_i} \theta_i(t). \tag{10}$$

Finally, we define the MGD random vector $\mathbf{G}$ as

$$\mathbf{G} := \frac{1}{\tau_\theta} \mathbf{\Lambda}^{-1} \sum_{t=1}^{\tau_\theta} \Delta C(t) \Theta(t), \tag{11}$$

which in component form is,

$$G_m = \frac{1}{\tau_\theta \sigma_m^2} \sum_{t=1}^{\tau_\theta} \sum_{\mu=1}^{K} \frac{\partial C}{\partial \theta_\mu} \theta_\mu(t) \theta_m(t), \tag{12}$$

where we recall the covariance matrix of Θ defined in Eq. (8). We claim that the random vector $\mathbf{G}$ can be used as an approximation to $\nabla_{\mathbf{\Theta}} C$ in the optimization routines that underpin machine learning. In Appendix B, we show that $\mathbf{G}$ is an unbiased estimator of the gradient, $\mathrm{E}(\mathbf{G}) = \nabla_{\mathbf{\Theta}} C$, and we compute its full covariance matrix, $\mathrm{Cov}(\mathbf{G})$. The key result is that this $\mathrm{Cov}(\mathbf{G})$ tends to zero as $\tau_\theta \to \infty$. In other words, averaging over a long time period yields an increasingly deterministic approximation to $\nabla_{\mathbf{\Theta}} C$. These technical points motivate the following numerical experiments.

C. Network accuracy vs gradient accuracy

Figure 2 presents a schematic representation of a cost landscape and an illustrative comparison of descent curves using MGD vs backpropagation. The black arrow in Fig. 2(a) represents the direction of the true gradient as calculated by backpropagation ($\nabla_W C$), while the black dashed line in both panels (a) and (b) represents the steepest descent path that the backpropagation algorithm would follow in the cost landscape. For comparison, the possible directions for the gradient estimate from MGD (**G**) are represented by the red shading around the arrow, while the red solid line shown in panel (b) illustrates the path taken by the MGD algorithm. It can be seen from the figure that the MGD algorithm tends to follow the general direction of gradient descent backpropagation, with additional stochasticity.

The level of stochasticity is determined by the gradient-integration time constant, τ_θ, and has been discussed in detail in Ref. 27. For short gradient integration time τ_θ [e.g., Fig. 2(a-i)], the MGD algorithm computes a directional derivative along a random direction, and the descent is highly stochastic. For longer gradient integration times τ_θ [e.g., Figs. 2(a-ii) and 2(a-iii)], the gradient estimate **G** calculated by MGD converges to the direction of $\nabla_W C$. In this case, the MGD algorithm replicates the behavior of the backpropagation. Note that no matter what the value of τ_θ, **G** is always pointing "downhill," although it may not be pointed in the direction of steepest descent.

In Sec. III, we demonstrate that a network can be trained to the same accuracy as backpropagation even without the need for averaging (large τ_θ) at each step. We will also discuss the scaling of the gradient estimation time and the training time with network size. These two convergences are illustrated, respectively, across the top and bottom rows of Fig. 2. We define the "gradient estimation time" as the time it would take for the cosine of the angle between **G** and $\nabla_W C$ to be >0.95. The "training time" is the time taken from initialization to reach a particular cost or accuracy value.

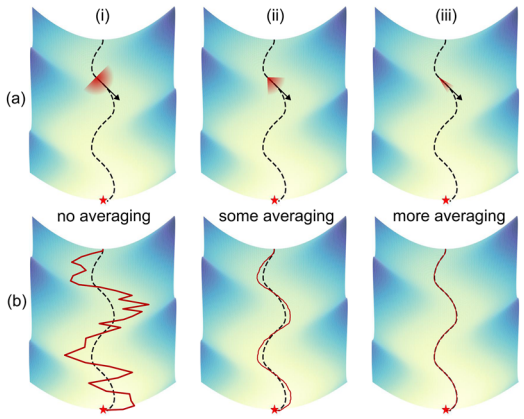

FIG. 2. Illustration of gradient descent down a cost landscape using MGD, for varying amounts of averaging. (a) In MGD, the true gradient (black dashed line) is not presumed to be known analytically and is instead estimated. The gradient estimation (red-shaded regions) starts out inaccurate but can be refined to arbitrary precision with additional averaging toward the true gradient. (b) The resulting weight updates and trajectory down the cost-landscape (red lines) do not strictly follow the gradient, but instead deviate to a degree that depends on the amount of averaging.

III. SCALING OF THE MGD ALGORITHM TRAINING SPEED

In the following simulations, we compare the performance of the MGD algorithm in terms of training time for networks of different sizes using both weight and node perturbation and compare it to the gradient estimation time. The learning rates for weight and node perturbation were kept approximately equivalent using the normalization calculations in Appendix B. Using the MGD time constants, we can perform quantitative comparisons of the speed of training these networks. The network we used throughout this paper consisted of six convolution neural network layers, followed by three dense neural network layers with three max-pool layers in between (see Table I). The network was trained to classify images in the FashionMNIST dataset. We kept this architecture constant but we varied the size of the layers (d = 1, 2, 4, 8, 16, 32, 48, 64) to change the overall network size. Table I shows an example of such a network with ~1×10^6 parameters (d = 48). The algorithm used for network training is described in Appendix A.

A. Gradient estimation

We first investigate the time to accurately estimate the gradient for the architecture described at the beginning of Sec. III as a function of the number of network parameters. To measure the gradient estimation accuracy as a function of time, we begin by randomly initializing the network. Then, for each iteration, we generate a new gradient estimate in the MGD fashion by randomly perturbing the network, measuring the resulting change in cost, and multiplying them together to get **G** as in (11). Each iterative estimation of **G** is summed together and the total is compared to the true local gradient $\nabla_W C$ (computed by backpropagation), generating the plots shown in Fig. 3(a). This was performed for both weight perturbation (red lines) and node perturbation (green lines). It is clear in both cases that when averaged for a sufficiently large number of iterations, **G** becomes a perfect proxy for $\nabla_W C$ (matching both direction and amplitude of $\nabla_W C$ as shown in Appendix B). As evident from this figure, node perturbation can estimate $\nabla_W C$ much faster than weight perturbation due to the smaller number of independent perturbations involved. This simulation was repeated for multiple networks with varying number of parameters, where the depths of the network, hence number of layers was kept constant, and number of parameters in each layer was proportionally scaled. Figure 3(b) shows the number of iterations required for the estimated gradient **G** to be approximately aligned with the true gradient ($\cos\alpha = 0.95$) as a function of network size. It is clear that, as expected, the time required to accurately estimate $\nabla_W C$ using MGD scales with the number of total parameters trained in the case of weight perturbation and with the square root of the number of total parameters trained in case of node perturbation. We additionally performed a similar set of simulations varying task complexity for a fixed network size of 4.52×10^5 parameters. To vary task complexity, we simply used the subset of data in the FashionMNIST dataset corresponding to 2, 4, 6, 8, or 10 classes. Here, a less complex task means that there are fewer classes to classify images into. As expected and

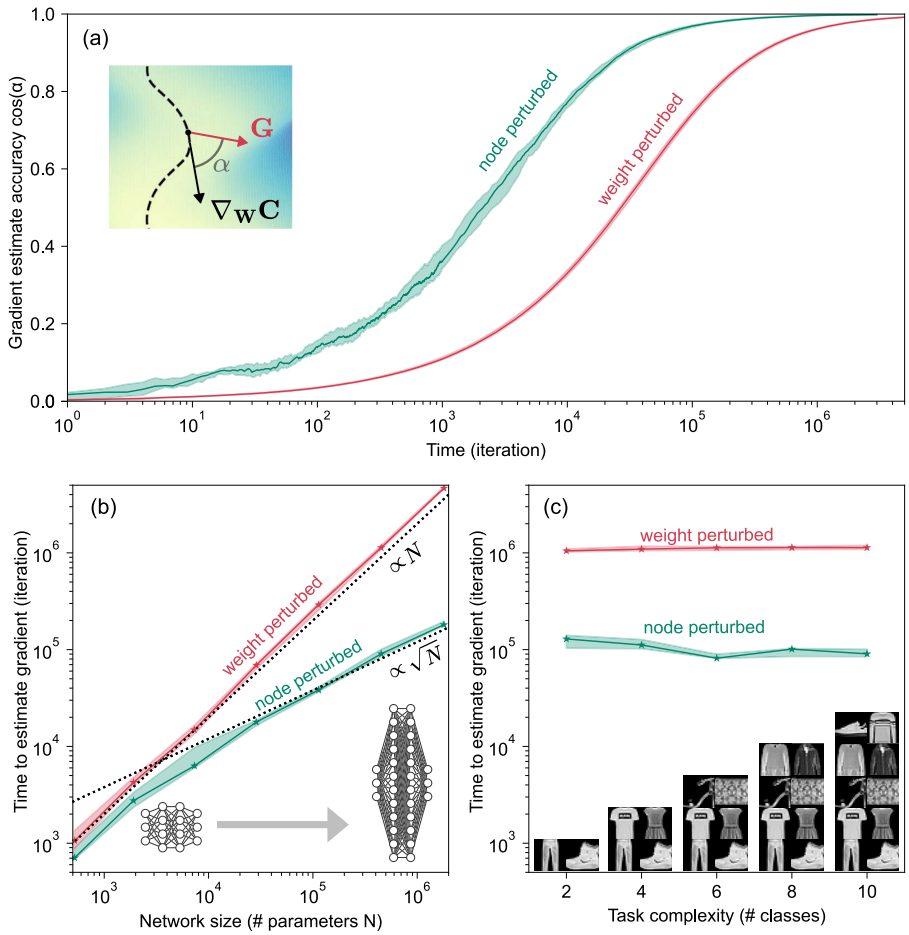

FIG. 3. Analyzing the behavior of gradient estimation in MGD. (a) The accuracy of the gradient estimation vs time. Each iteration refines the accuracy of the gradient estimation, defined as the cosine of the angle (α) between the true local gradient $\nabla_{\mathbf{W}}C$ and local gradient estimated by MGD **G** for a network with $N = 4.52 \times 10^5$ parameters. (b) Number of iterations it takes for $\cos(\alpha)$ to reach 0.95 vs total number of trained parameters in the network (N). The dashed lines represent the expected N and $\sqrt{N}$ scaling for weight and node perturbation, respectively. (c) Number of iterations it takes for $\cos(\alpha)$ to reach 0.95 (or 95% gradient estimation accuracy) vs the task complexity. The shaded regions represent the upper and lower quartile bounds of ten random initializations, and the solid line corresponds to the median.

is evident from Fig. 3(c), task complexity has no effect on gradient estimation. These results are in keeping with previous results using perturbative techniques.

B. Image classification

While Fig. 3 shows that the scaling of the gradient follows the expected trends, we note that this is not the important figure of merit for ML tasks—ultimately, computing the gradient is only a means toward training the network. We next investigated whether the same scaling applies to the time to train a network as to the time to estimate the gradient. This scaling cannot be easily predicted analytically. However, in real-world applications, peak accuracy and time to reach that peak accuracy are typically the relevant performance metrics. This has not been previously investigated in detail for perturbative algorithms. Figure 4(a) shows testing error vs number of iterations for a network with 2.55×10^5 parameters trained to classify all ten classes of the FashionMNIST dataset for weight perturbation (red) and node perturbation (green) and backpropagation (black). As before, an iteration on the x axis represents a single perturbation and subsequent gradient estimation of the network. To perform a comparison on the scale with backpropagation (black line in Fig. 4), we perform the same simulation but use the true gradient instead of **G** for the weight update. In this demonstration, we set the gradient integration time τ_θ to 1, hence weights are updated after each perturbation with no gradient averaging. This means that we are moving along a direction that is always downhill (directional derivative), but a poor gradient estimate [see Fig. 2(a-i)]. We can see that both weight and node perturbation can achieve the same accuracy as backpropagation on the same network architecture.

This simulation was then repeated for the same set of network sizes as in Fig. 3(b). Figure 4(b) shows the number of iterations required for the network to reach 80% accuracy vs the total number of parameters in the network. The time to reach 80% accuracy varies by less than one order of magnitude over >3 orders of magnitude change in network size for both weight and node perturbation. This is markedly different from the gradient estimation time in Fig. 3(b), which, in the case of weight perturbation, scales proportionally to network size. This result is in contrast to some of the scaling arguments that have been previously made about perturbative techniques.[2,3]

In addition, while node perturbation reduces the time required to reach a given accuracy when compared to weight perturbation, the performance enhancement depends on the required accuracy,

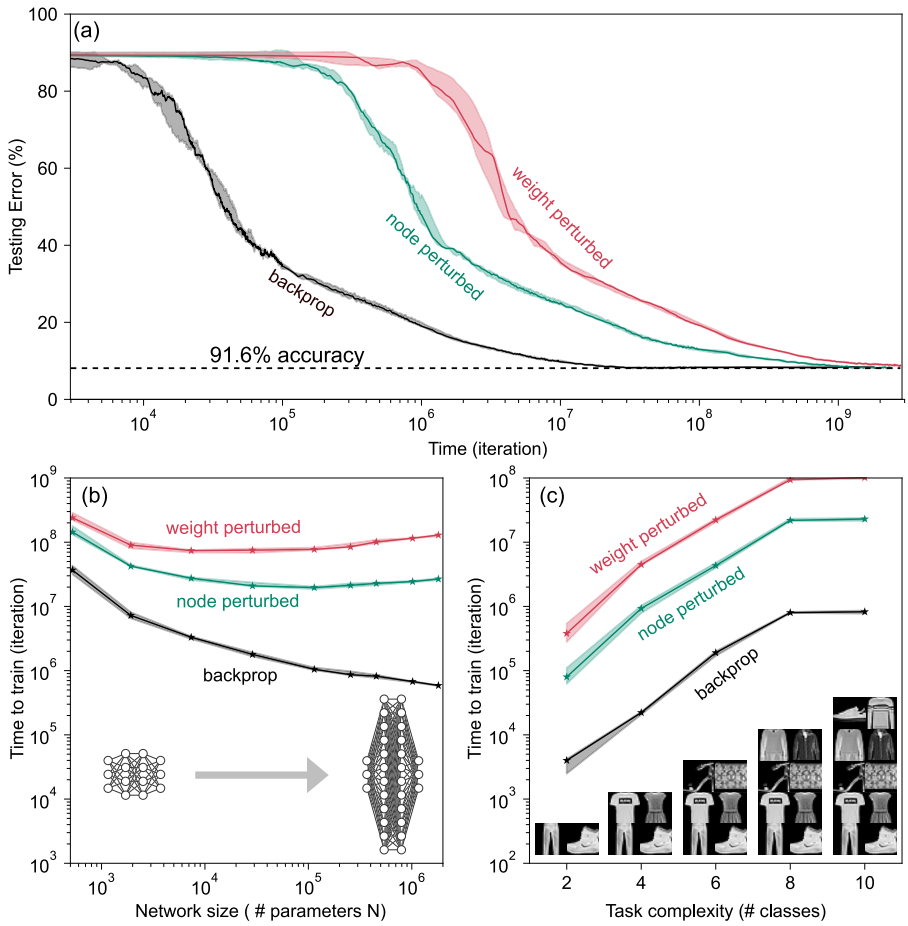

FIG. 4. Time needed to successfully train networks with MGD as a function of network size and task complexity. (a) Testing error vs number of iterations for network with $N = 2.55 \times 10^4$ parameters, showing that the MGD algorithm can match the same final accuracy of 91.6% as backpropagation. Although backpropagation is generally not available in hardware, here in simulation, its analytical nature provides the fastest time-to-train as expected. (b) Number of iterations it takes for the network to be trained to 80% testing accuracy vs total number of trained parameters in the network N. (c) Number of iterations it takes for the network to be trained to 80% testing accuracy vs the task complexity. The learning rate $\eta = 0.001$ was kept constant for all simulations shown here.

and is not a simple relationship as for the gradient estimation time. For example, in Fig. 4(b), we observe that the time to reach 80% accuracy using node perturbation was reduced by approximately a factor of 2 over node perturbation for most network sizes. However, we can see from the individual cost vs training time curve in Fig. 4(a) that this scaling factor was different depending on the desired final accuracy. For example, node perturbation is 10× faster to get to 50%. As in Sec. III A, the same comparison was then performed for varying task complexities. Figure 4(c) shows that task complexity strongly affects the time required to reach 80% accuracy. Task complexity is clearly very important to the training time and can supersede network size in affecting training time.

These results demonstrate conclusively that a network with more than 1×10^6 parameters can be trained to the same testing accuracy as backpropagation using perturbative zero-order optimization techniques. In addition, the results indicate that while the time required to accurately estimate $\nabla_W C$ is indeed proportional to the number of trained parameters in the network, time required to reach a set accuracy target is a more complicated function of network size. This result is in keeping with previous observations that an accurate gradient estimate is not required for machine learning. While MGD is slower when compared to backpropagation when simulated on a digital computer, iterations could be implemented very quickly on a dedicated analog hardware. To get an intuitive sense of this, consider that 10^9 MGD time iterations would take 16 min on hardware with a modest speed of 1 MHz for perturbations, inference, and updates. Another interesting observation is that although there is a speedup associated with node perturbation, in general, we find that node perturbation does not perform as well as expected compared to weight perturbation. For example, node perturbation is only 2× faster than weight perturbation to get to 80% accuracy for a network of 1×10^6 parameters. This indicates that learning rate and other hyperparameter factors may be as or more important than node vs weight perturbation in determining time to solution. In addition, convolutional layers can sometimes have fewer weights than activations, invalidating any scaling advantage to node perturbation.

IV. TAILORING MGD TO SPECIFIC HARDWARE

MGD allows training to be optimized for specific hardware platforms. For example, some hardware platforms are based on non-volatile memory technologies with slow update speeds and a limited number of write–erase cycles before the memory elements

begin to deteriorate. This type of hardware can be accommodated by adjusting the MGD time constants to reduce the number of weight updates. All the examples that we described above were calculated for $\tau_\theta = 1$, meaning the gradient was estimated for a single time step and the network was updated immediately afterward. Increasing this integration time such that $\tau_\theta \gg 1$ can give similar performance with fewer overall weight updates. In a practical implementation, this could be accomplished if perturbations are implemented separately and faster than weight updates, for instance, by placing a fast, volatile, perturbation element (such as a transistor) in series with a the slower-to-update and less-durable weight.

Figure 5 shows the total number of weight updates that it takes for the network to be trained to 80% testing accuracy as a function of the gradient integration time τ_θ. The total number of weight updates required to train the network can be seen to decrease as a function of integration time, approaching the number of updates required for backpropagation. This is due to the accuracy of **G** increasing with τ_θ. As can be seen from the figure, the number of weight updates can be reduced by several orders of magnitude using this technique. However, in the absence of weight update constraints, setting $\tau_\theta = 1$ still results in the fastest overall training time (as would be measured in terms of wall-clock time on a real piece of hardware). In general, the MGD framework has the flexibility to match the details of the optimization algorithm to the hardware constraints.

A second example of the flexibility of the MGD platform is the implementation of different types of perturbations, including both weight and node perturbations. Any set of mean-zero orthogonal perturbations can be used to yield equivalent results.[14,27] However, the effect of bias or non-orthogonal perturbations has not yet been explored fully. Node perturbation provides modest improvements in training time over weight perturbation. However, implementing these two different training approaches on hardware presents distinct challenges. To implement node perturbation in the form described in this paper, each synapse requires a circuit capable of performing one multiplication operation to compute the product of the global change in cost C and the perturbation θ_w. The algorithm can, in principle, be simplified further to implement only additions or Boolean multiplications at the synapse. In the case that a longer integration time $\tau_\theta > 1$ is desired to reduce the number of weight updates, an additional memory element is required per synapse to store intermediate gradient values.

Node perturbation requires single-layer backward data transfer, which reintroduces some of the same challenges that we face when trying to implement backprop in a hardware. For example, to compute the gradient estimate (**G**) in node perturbation, each neuron must have circuitry capable of performing $2N$ multiplication operations [the cost change ΔC multiplied by the perturbation θ_k multiplied by the synapse input x_j $(\Delta C \theta_k x_j)$]. Each neuron must also store inputs to a layer in a memory buffer, hence requiring larger local memory on hardware. Another requirement of node perturbation is that weights must be linear to facilitate the backpropagation of the multiply accumulate (MAC) process. Conversely, weight perturbation makes no assumptions about the network architecture, including linearity of synaptic operations. Weight perturbation does, however, require one perturbation per weight, while node perturbation only requires one perturbation per node. This may offer significant advantages in hardware with limited write cycles and a high cost to perturbations. Overall, the choice between these two approaches depends on the specific hardware constraints and requirements.

Finally, additional hardware or architecture specific tricks beyond those described in this paper can be used to reduce the number of parameters to be perturbed in a network. For example, an approach combining perturbative training with the tensor train method[30] was proposed recently.

V. OPTIMIZERS

Modern problems require more than just basic stochastic gradient descent due to its relatively slow convergence rate. A wide variety of specialized optimizers and other tricks are used to improve the training process as networks become larger and deeper. Some common examples of specialized optimizers are Momentum, AdaGrad, and Adam. Other "tricks" for training large and deep networks include well-designed network initializations, dropout, local competition strategies, and architecture changes such as skip-layer connections. Given the ubiquity of these tricks in modern machine learning, it would be unfair to compare the performance of basic MGD to the performance of backpropagation with decades worth of optimization built on it.

However, since MGD can be used to compute the gradient to an accuracy close to that of backpropagation, all these standard tricks may still be applicable and directly translatable into the framework, although the hardware requirements will inevitably increase. In particular, we expect that Momentum, Adam, and Dropout will require additional local memory and multiplications to implement. Other techniques such as batch normalization are likely to prove significantly more challenging to implement due to the requirement that

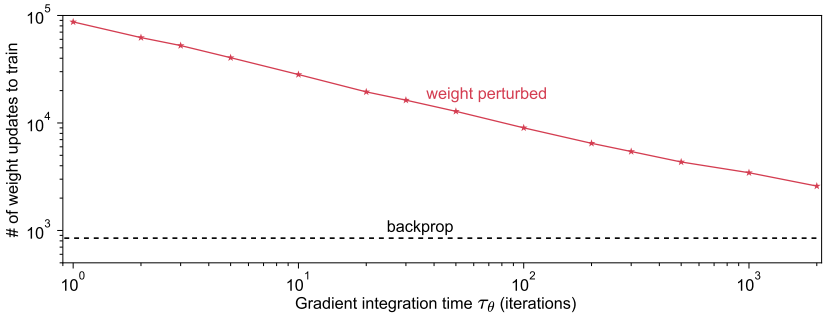

FIG. 5. Number of weight updates required to train Fashion-MNIST on a 2.55×10^5 parameter network to 80% testing accuracy vs τ_θ.

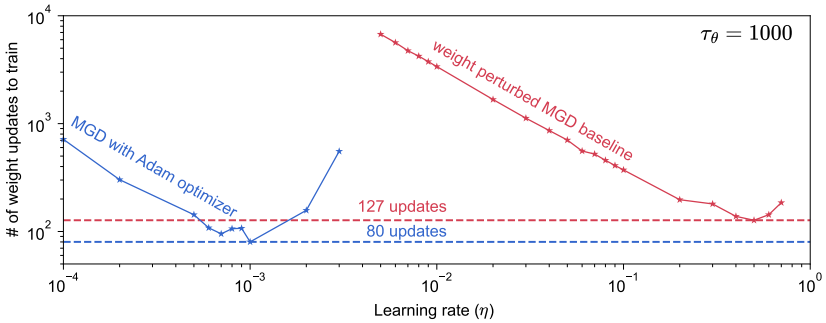

FIG. 6. Number of weight updates required to train Fashion-MNIST to 80% accuracy on a 255 000-parameter network using vanilla MGD (red) and with the Adam optimizer (blue).

many network parameters throughout the network be accessed to compute the normalization required for a single parameter update during training. Figure 6 is a preliminary result demonstrating that MGD can take advantage of more modern optimizers.

The red line shows the number of updates it takes for the network to reach 80% testing accuracy using weight perturbation and vanilla MGD. The update rule for vanilla MGD is

$$\mathbf{W}_t = \mathbf{W}_{t-1} - \eta \mathbf{G}, \tag{13}$$

where $\mathbf{W}$ represents the network weights, $\mathbf{G}$ is the MGD gradient, and η is the learning rate. We then implemented the Adam optimizer update rule instead of vanilla gradient descent, as shown in Eq. (14) (arithmetic operations on vectors are interpreted component-wise),

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{G}, \tag{14a}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\mathbf{G}^2, \tag{14b}$$

$$\mathbf{W}_t = \mathbf{W}_{t-1} - \eta \left(\frac{\mathbf{m}_t}{\sqrt{\mathbf{v}_t} + \epsilon} \right). \tag{14c}$$

The vectors $\mathbf{m}_t$ and $\mathbf{v}_t$ represent exponentially decaying moving averages of the gradient and its square, while β_1, β_2, and ϵ are user-selected hyperparameters. Following standard guidance, we set $\beta_1 = 0.9, \beta_2 = 0.999$, and $\epsilon = 10^{-8}$.[37] To avoid any numerical instabilities in the simulation, the learning rate is set to $\eta = 0$ for the initial 1000 iterations.

The simulations were run using the MGD gradient estimate (with $\tau_\theta = 1000$) within the Adam update rule. The pink line in Fig. 6 shows the number of updates it takes for the network to reach 80% testing accuracy as a function of the learning rate. We find that there is a 37% decrease in the number of updates required when using the Adam optimizer. This result demonstrates that MGD can serve as a drop-in replacement for the gradient computed by backpropagation in modern optimization algorithms. This opens the possibility for significantly improving the speed of MGD on different hardware platforms. Other results in the literature indicate similar advantages to augmenting perturbative gradient descent with modern machine learning. For example, in Refs. 29 and 38, it was shown that adding greedy local learning significantly improved the training speed for a network of 13×10^6 parameters classifying MNIST and CIFAR10.

VI. DISCUSSION

MGD is one of a growing number of hardware-compatible or brain-inspired algorithms proposed in the past several years. For example, one class of these are fixed-point or energy-based networks and learning algorithms such as Hopfield networks,[38] equilibrium propagation,[39] or contrastive local learning rules,[40] which dynamically evolve to local energy minimum states that correspond to stored patterns. We note that while these algorithms are also very well suited to implementation on certain types of hardware, they require specific architectures that differ from the standard architectures commonly used to generate modern machine learning results. These restricted architectures prevent the direct transferring of known results to hardware and have not yet been proven work for the same applications. In contrast, MGD can be used to exactly reproduce the results of backpropagation, allowing direct transfer of successful architectures to hardware. Other non-energy based algorithms have also been proposed, such as feedback alignment,[33,41] the forward-forward algorithm,[2] and many others.[42,43] In some cases, these have not yet been fully explored,[2] or have been shown to have applicability only in limited circumstances.[44] A comprehensive comparison of the speed, accuracy, and scaling of all these techniques is beyond the scope of this work. However, we note that in most cases, MGD has the advantage that by setting the time constants appropriately, the hardware can train in an equivalent manner to backpropagation. While this may not be the most efficient setting for fast training, it is a very useful feature for ensuring that the hardware will work as designed.

The unexpected effectiveness of MGD fits into a growing body of literature, showing that an accurate gradient estimate is not required for training neural networks.[33] The scaling results are consistent with the current theoretical understanding of how network size affects training difficulty. This can be seen in investigations of architectures and learning algorithms based on hidden layers with fixed parameters, such as reservoir computers[45] and extreme learning machines.[46] As layers become wide enough, training of the inner layers becomes unnecessary altogether, and high accuracy can be obtained with only a linear solve on the output layer (a much faster and simpler algorithm). This paper, therefore, adds to the body of the literature, indicating that effective training of neural networks can been done with surprisingly simple algorithms.

Perturbative learning was implemented in CMOS hardware in the 1990s,[15,16,18–25] but hardware networks only reached a scale of ~10 weights before the community as a whole shifted toward digital communication and address-event representation.[47] In the past decade, there has been a resurgence of interest in analog and analog digital neuromorphic hardware. This includes research into novel

memristive, optical, and superconducting implementations, among others. Training with conventional backpropagation based algorithms has proved challenging for these types of analog hardware. Perturbative algorithms have been gaining in popularity within this community[31,48,49] due to simplicity of implementation. MGD provides a framework for evaluating the training speed of these perturbative algorithms given particular hardware constraints. We hope showing the scalability of these algorithms will prove useful to the development of new hardware.

It has been speculated that perturbative learning takes place in the brain,[50,51] but there is skepticism that it can play a major role due to the poor scaling of the time to estimate the gradient[3] and the results based on training of linear networks.[52] The result of this and other recent papers[53]—that the time to train does not scale directly with network size—will hopefully prompt some re-examination of this conclusion. In this context, another important feature of the MGD framework is that weight updates require only information that is spatially local to the synapse and a single globally broadcast cost-change signal. Unlike backpropagation, signal propagation in the MGD framework is always in the forward direction; the derivative of the activation function is not required to be known or used; and there are no separate forward and backward phases in the learning algorithm. In addition, small scale demonstrations of training of spiking networks by these techniques were implemented in the early 2000s[50,51] and have undergone a recent resurgence in interest.[54,55] This offers up the intriguing possibility that this is a general training technique that can apply equally to spiking and non-spiking networks. Further work on this is required to reach conclusions about spiking networks.

In this paper, we showed modest improvements to the scaling of the time to train a neural network by using node perturbation instead of weight perturbation. The choice of weight or node will likely be influenced strongly by the specifics of the hardware implementation. We also note as a qualitative observation that we found node perturbation to be more challenging to optimize than weight perturbation at almost every stage of its implementation. Other recent work on node perturbation has shown that the scaling of node perturbation is worse when recurrent networks are considered[34] and there can be issues with stability.[53] We note that Ref. 53 reached a similar conclusion with respect to the fact that training time does not scale linearly with network size for node perturbation. They also derived the covariance of weight updates under node perturbation for arbitrary multilayer perceptrons.

In summary, the trade-offs between node and weight perturbation in hardware implementation highlight the need for a careful evaluation of the specific requirements and capabilities of the hardware platform. This evaluation ensures that the chosen method aligns with the overall goals of efficiency, speed, and resource utilization.

Finally, we note that perturbative techniques have taken off recently in other contexts beyond the training of neuromorphic hardware. Recent studies have investigated similar algorithms for fine-tuning of large language models.[56–58] This is because less memory is required to implement than is required for backpropagation. Therefore, large language models can be fine-tuned on smaller machines than required for the original training.

VII. CONCLUSION

Our investigation into multiplexed gradient descent (MGD) demonstrates its potential as a scalable and efficient training method for neuromorphic hardware. Despite common misconceptions, we have shown that perturbative methods such as MGD can scale to large networks. Although the time required for the stochastic variable $\mathbf{G}$ to accurately estimate the true gradient $\nabla_{\mathbf{W}}C$ scales with the number of parameters, it is nevertheless possible that the time to reach a fixed accuracy target can be independent of network size.

Furthermore, our results demonstrate how MGD can be swapped for backpropagation algorithmically, allowing the application of standard gradient descent enhancements such as momentum and Adam. This flexibility enables MGD to achieve comparable testing accuracy as backpropagation, even for networks with over one million parameters. Although MGD may appear slower than backpropagation in simulations, the speed on physical hardware depends on the time to execute iterations of the algorithm. MGD may be much simpler and faster to implement than backpropagation on emerging analog hardware, and speeds competitive with backpropagation training on conventional hardware are plausible given our simulation results. Overall, MGD provides a practical and adaptable solution for on-hardware training, capable of leveraging advanced optimizers and handling large-scale networks within reasonable time frames. Our findings underscore the importance of considering both the hardware architecture and the specific needs of the training process when selecting perturbation methods, paving the way for more efficient and scalable neuromorphic computing systems.

ACKNOWLEDGMENTS

The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. Analysis performed in part on the NIST Enki HPC cluster. This research was funded by 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

A.N.M. and S.M.B. conceptualized these experiments. B.G.O. run the simulations. Analysis and interpretation of the data was done by B.G.O., A.N.M., S.M.B., and A.D. All authors co-wrote the manuscript.

B. G. Oripov: Conceptualization (equal); Formal analysis (equal); Methodology (equal); Validation (equal); Visualization (equal); Writing – original draft (equal); Writing – review & editing (equal). **A. Dienstfrey**: Formal analysis (lead); Methodology (supporting);

Writing – review & editing (supporting). **A. N. McCaughan**: Conceptualization (equal); Formal analysis (supporting); Methodology (supporting); Software (lead); Supervision (equal); Writing – original draft (supporting); Writing – review & editing (supporting). **S. M. Buckley**: Conceptualization (equal); Formal analysis (equal); Methodology (equal); Supervision (equal); Writing – original draft (equal); Writing – review & editing (equal).

DATA AVAILABILITY

The data used for all the plots in this work are available from the corresponding author upon reasonable request.

The MGD library and code used to perform these simulations are available at github.com/bakhromtjk/mgd_scaling.

APPENDIX A: MGD PSEUDOCODE

The MGD algorithm pseudocode is shown in Algorithm 1. The definitions of the variables are given in Table II.

ALGORITHM 1. Weight and node-perturbed MGD algorithm.

Initialize parameters θ
for n **in** num_iterations **do**
  Input new training sample x, y
  if $(n \bmod \tau_\theta = 0)$ **then**
    Set perturbations to zero $\theta \leftarrow 0$
    if node_perturbed **then**
      Compute input to every layer $\hat{x}_l \leftarrow \hat{y}_{l-1}$
    end if
    Update baseline cost $C_0 \leftarrow C(f(x;\Theta), y)$
  end if
  Update perturbations θ
  for l **in** num_layers **do**
    Compute output
    $\hat{y}_l \leftarrow f(x; \Theta + \theta_l)$
    Compute cost $C_l \leftarrow C(\hat{y}_l, y_l)$
    Compute change in cost $\Delta C_l \leftarrow C_l - C_0$
    Compute error signal
    if weight_perturbed **then**
      $e_l \leftarrow \Delta C_l \theta_l / \Gamma$
    end if
    if node_perturbed **then**
      $e_l \leftarrow \Delta C_l \theta_l x_l / \Gamma$
    end if

    Accumulate gradient approximation $G_l \leftarrow G_l + e_l$
  end for
  if $(n \bmod \tau_\theta = 0)$ **then**
    Update parameters $\Theta \leftarrow \Theta - \eta G$
    if $(n \bmod \tau_\theta \neq \infty)$ **then**
      Reset gradient approximation $G \leftarrow 0$
    end if
  end if
end for

APPENDIX B: MEAN AND COVARIANCE OF G

In this section, we compute the first two moments—i.e., the mean and covariance—of **G** defined in (11). Note that the diagonal of the covariance matrix can be used to compute the expectation of the squared norm, $\mathrm{E}(\|\mathbf{G}\|^2)$.

1. Random perturbations

In the stochastic perturbation model of MGD, the perturbations are a collection of random variables,

$$\{\theta_i(t) | \quad \text{for } i = 1, \ldots, K \text{ and } t = 1, \ldots, \tau_\theta\}.$$

Here, K is the dimension of the parameter vector and τ_θ is the total number of time steps. We assume the following:

1. $\theta_i(t)$ are "symmetric" meaning that $\mathrm{E}(\theta_i^{2k+1}(t)) = 0$ for $k \geq 0$.
2. $\theta_i(t)$ can be distinct over i, but are identical over t.
3. $\theta_i(t)$ values are independent over i and t.

These assumptions are reasonably general and lead to significant simplifications in the following computations. For reference, we define the variance and fourth moment for each index,

$$\mathrm{E}(\theta_i^2(t)) = \mathrm{E}(\theta_i^2(t')) = \sigma_i^2 \quad \mathrm{E}(\theta_i^4(t)) = \mathrm{E}(\theta_i^4(t')) = \alpha_i^4.$$

Note that these quantities depend on index but not time.

2. Moments of G

We assume that magnitudes of the perturbations are sufficiently small enough that the cost function can be approximated by its linear expansion (10). Thus, for a perturbation vector at time t, the change in cost is given by

$$\Delta C(t) = \sum_i \frac{\partial C}{\partial \theta_i} \theta_i(t).$$

From (11), we write component MGD vector as

$$G_m = \frac{1}{\tau_\theta \sigma_m^2} \sum_t \sum_i \frac{\partial C}{\partial \theta_i} \theta_i(t) \theta_m(t).$$

We prove the following theorem.

Theorem 1. *The MGD random variable,* **G**, *is an unbiased estimator of* $\nabla_\Theta C$,

$$\mathrm{E}(\mathbf{G}) = \nabla_\Theta C. \tag{B1}$$

The covariance matrix is given by

TABLE II. Description of the variables used in the algorithm.

Symbol	Description
x	Input to the network
y	Target output from the network
$\hat{y}$	Inferred output from the network
N	Number of trainable parameters
K	Number of perturbed parameters
Θ	Trainable network parameters
θ	Perturbations applied to Θ
$\delta = \lvert\theta_k\rvert$	Amplitude of perturbations applied to Θ
τ_θ	The period with which Θ is updated
η	Learning rate
C_0	Baseline cost before perturbing
C	Final cost after perturbing
ΔC_l	Difference in cost due to perturbation
$\mathbf{G}$	Estimated gradient
$\Gamma = \delta^2\sqrt{K}\sqrt{1+(K-1)/\tau_\theta}$	A normalization factor [see Eq. (B7)]
n	Iteration step number
Num_iterations	Total number of iterations
Num_layers	Total number of layers
Weight_perturbed	Boolean; is the network trained using weight perturbation
Node_perturbed	Boolean; is the network trained using node perturbation

$$\mathrm{Cov}(\mathbf{G})_{m,n} = \begin{cases} \frac{1}{\tau_\theta}\left(\left(\frac{\partial \mathrm{C}}{\partial \theta_m}\right)^2\left(\frac{\alpha_m^4}{\sigma_m^4}-1\right)+\sum_{\mu\neq m}\left(\frac{\partial \mathrm{C}}{\partial \theta_\mu}\right)^2\frac{\sigma_\mu^2}{\sigma_m^2}\right), & m=n, \\ \frac{1}{\tau_\theta}\frac{\partial \mathrm{C}}{\partial \theta_m}\frac{\partial \mathrm{C}}{\partial \theta_n}, & m\neq n. \end{cases} \tag{B2}$$

Proof. The mean of G_m is computed,

$$\begin{aligned} \mathrm{E}(G_m) &= \mathrm{E}\left(\frac{1}{\tau_\theta\sigma_m^2}\sum_t\sum_i\frac{\partial \mathrm{C}}{\partial \theta_i}\theta_i(t)\theta_m(t)\right) \\ &= \frac{1}{\tau_\theta\sigma_m^2}\sum_t\sum_i\frac{\partial \mathrm{C}}{\partial \theta_i}\mathrm{E}(\theta_i(t)\theta_m(t)) \\ &= \frac{1}{\tau_\theta\sigma_m^2}\sum_t\frac{\partial \mathrm{C}}{\partial \theta_m}\sigma_m^2 = \frac{\partial \mathrm{C}}{\partial \theta_m}. \end{aligned}$$

The second equality follows from linearity of expectation. In going from the second to third line, the independence of $\theta_i(t)$ and $\theta_m(t)$ implies that the expectation of the product is zero for $i \neq m$, and thus, these terms in the summation over i drop out. The only non-zero contribution comes from $i = m$, in which case, the expected product is σ_m^2. Finally, as σ_m^2 does not depend on t and there are τ_θ terms in the sum over t, the result follows. Thus, we have Eq. (B1).

We next compute the covariance matrix,

$$\mathrm{Cov}(\mathbf{G}) = \mathrm{E}\left(\mathbf{G}^T\mathbf{G}\right) - \mathrm{E}(\mathbf{G})^T\mathrm{E}(\mathbf{G}) = \mathrm{E}\left(\mathbf{G}^T\mathbf{G}\right) - \nabla_\Theta \mathrm{C}^T\nabla_\Theta \mathrm{C}. \tag{B3}$$

First, we compute the off-diagonal terms,

$$\begin{aligned} \mathrm{E}(G_mG_n) &= \mathrm{E}\left(\frac{1}{\tau_\theta^2\sigma_m^2\sigma_n^2}\sum_t\sum_i\frac{\partial \mathrm{C}}{\partial \theta_i}\theta_i(t)\theta_m(t)\right. \\ &\quad\left.\times\sum_{t'}\sum_j\frac{\partial \mathrm{C}}{\partial \theta_j}\theta_j(t')\theta_n(t')\right) \\ &= \frac{1}{\tau_\theta^2\sigma_m^2\sigma_n^2}\sum_{t,t'}\sum_{i,j}\frac{\partial \mathrm{C}}{\partial \theta_i}\frac{\partial \mathrm{C}}{\partial \theta_j}\mathrm{E}\big(\theta_i(t)\theta_m(t)\theta_j(t')\theta_n(t')\big). \end{aligned}$$

The last sum is split into two cases: $t = t'$ and $t \neq t'$. In either case, any non-zero expectations will be constant as a function of t resulting in τ_θ summands in the first case and $\tau_\theta(\tau_\theta - 1)$ in the second. Thus, we have

$$\begin{aligned} &\sum_{t,t'}\sum_{i,j}\frac{\partial \mathrm{C}}{\partial \theta_i}\frac{\partial \mathrm{C}}{\partial \theta_j}\mathrm{E}\big(\theta_i(t)\theta_m(t)\theta_j(t')\theta_n(t')\big) \\ &\quad= \tau_\theta\sum_{i,j}\frac{\partial \mathrm{C}}{\partial \theta_i}\frac{\partial \mathrm{C}}{\partial \theta_j}\mathrm{E}(\theta_i\theta_m\theta_j\theta_n) \\ &\qquad+ \tau_\theta(\tau_\theta-1)\sum_{i,j}\frac{\partial \mathrm{C}}{\partial \theta_i}\frac{\partial \mathrm{C}}{\partial \theta_j}\mathrm{E}(\theta_i\theta_m)\mathrm{E}(\theta_j\theta_n). \end{aligned}$$

Observe that the first sum over i,j has only two non-zero contributions—$i = m$ and $j = n$ or $i = n$ and $j = m$—and the expectation is $\sigma_m^2\sigma_n^2$ in both cases. Similarly, the second sum can only contribute when $i = m$ and $j = n$. The result is

$$\begin{aligned}\mathrm{E}(G_mG_n) &= \frac{1}{\tau_\theta^2\sigma_m^2\sigma_n^2}\Big(2\tau_\theta\frac{\partial \mathrm{C}}{\partial\theta_m}\frac{\partial \mathrm{C}}{\partial\theta_n}\sigma_m^2\sigma_n^2 \\ &\quad + \tau_\theta(\tau_\theta-1)\frac{\partial \mathrm{C}}{\partial\theta_m}\frac{\partial \mathrm{C}}{\partial\theta_n}\sigma_m^2\sigma_n^2\Big) \\ &= \frac{\partial \mathrm{C}}{\partial\theta_m}\frac{\partial \mathrm{C}}{\partial\theta_n} + \frac{1}{\tau_\theta}\frac{\partial \mathrm{C}}{\partial\theta_m}\frac{\partial \mathrm{C}}{\partial\theta_n}. \end{aligned} \tag{B4}$$

Note that the first term on the right-hand side is the m,n component of $\nabla_\Theta \mathrm{C}^T\nabla_\Theta \mathrm{C}$. Subtracting this from both sides results in the $m \neq n$ case in Eq. (B2).

The computation for the diagonal term is slightly more complicated,

$$\begin{aligned}\mathrm{E}\big(G_m^2\big) &= \mathrm{E}\left(\frac{1}{\tau_\theta^2\sigma_m^4}\sum_t\sum_i\frac{\partial \mathrm{C}}{\partial\theta_i}\theta_i(t)\theta_m(t)\sum_{t'}\sum_j\frac{\partial \mathrm{C}}{\partial\theta_j}\theta_j(t')\theta_m(t')\right) \\ &= \frac{1}{\tau_\theta^2\sigma_m^4}\sum_{t,t'}\sum_{i,j}\frac{\partial \mathrm{C}}{\partial\theta_i}\frac{\partial \mathrm{C}}{\partial\theta_j}\mathrm{E}\big(\theta_i(t)\theta_m(t)\theta_j(t')\theta_m(t')\big).\end{aligned}$$

Again, we split the sum over t,t' into a diagonal and off-diagonal part,

$$\begin{aligned}&\sum_{t,t'}\sum_{i,j}\frac{\partial \mathrm{C}}{\partial\theta_i}\frac{\partial \mathrm{C}}{\partial\theta_j}\mathrm{E}\big(\theta_i(t)\theta_m(t)\theta_j(t')\theta_m(t')\big) \\ &\quad = \tau_\theta\sum_{i,j}\frac{\partial \mathrm{C}}{\partial\theta_i}\frac{\partial \mathrm{C}}{\partial\theta_j}\mathrm{E}\big(\theta_i\theta_j\theta_m^2\big) \\ &\qquad + \tau_\theta(\tau_\theta-1)\sum_{i,j}\frac{\partial \mathrm{C}}{\partial\theta_i}\frac{\partial \mathrm{C}}{\partial\theta_j}\mathrm{E}(\theta_i\theta_m)\mathrm{E}(\theta_j\theta_m).\end{aligned}$$

As before, the second sum has a non-zero contribution only for $i = m$ and $j = m$, in which case the expectations result in σ_m^4. The first sum has a second-moment contributions when $i = j = \mu \neq m$ and a single fourth-moment when $i = j = m$,

$$\sum_{i,j}\frac{\partial \mathrm{C}}{\partial\theta_i}\frac{\partial \mathrm{C}}{\partial\theta_j}\mathrm{E}\big(\theta_i\theta_j\theta_m^2\big) = \sum_{\mu\neq m}\frac{\partial \mathrm{C}}{\partial\theta_\mu}^2\sigma_\mu^2\sigma_m^2 + \frac{\partial \mathrm{C}}{\partial\theta_m}^2\alpha_m^4.$$

Using these simplifications, we have

$$\begin{aligned}\mathrm{E}\big(G_m^2\big) &= \frac{1}{\tau_\theta^2\sigma_m^4}\left(\tau_\theta\left(\sum_{\mu\neq m}\left(\frac{\partial \mathrm{C}}{\partial\theta_\mu}\right)^2\sigma_\mu^2\sigma_m^2 + \left(\frac{\partial \mathrm{C}}{\partial\theta_m}\right)^2\alpha_m^4\right)\right. \\ &\quad \left. + \tau_\theta(\tau_\theta-1)\left(\frac{\partial \mathrm{C}}{\partial\theta_m}\right)^2\sigma_m^4\right) \\ &= \left(\frac{\partial \mathrm{C}}{\partial\theta_m}\right)^2 + \frac{1}{\tau_\theta}\left(\left(\frac{\partial \mathrm{C}}{\partial\theta_m}\right)^2\left(\frac{\alpha_m^4}{\sigma_m^4}-1\right)\right. \\ &\quad \left. + \sum_{\mu\neq m}\left(\frac{\partial \mathrm{C}}{\partial\theta_\mu}\right)^2\frac{\sigma_\mu^2}{\sigma_m^2}\right).\end{aligned} \tag{B5}$$

Again, subtracting $(\partial \mathrm{C}/\partial\theta_m)^2$ from both sides results in the diagonal case of Eq. (B2). □

Note that the diagonal terms of the covariance matrix can be reorganized and summed resulting in the following corollary:

Corollary 1.1. The norm of the MGD random variable $\|\mathbf{G}\|^2$ is a biased estimator of $\|\nabla_\Theta \mathrm{C}\|^2$,

$$\begin{aligned}\mathrm{E}\big(\|\mathbf{G}\|^2\big) &= \|\nabla_\Theta \mathrm{C}\|^2 + \frac{1}{\tau_\theta}\sum_{m=1}^{K}\left(\left(\frac{\partial \mathrm{C}}{\partial\theta_m}\right)^2\left(\frac{\alpha_m^4}{\sigma_m^4}-1\right)\right. \\ &\quad \left. + \sum_{\mu\neq m}\left(\frac{\partial \mathrm{C}}{\partial\theta_\mu}\right)^2\frac{\sigma_\mu^2}{\sigma_m^2}\right).\end{aligned} \tag{B6}$$

Proof. As $\mathbf{G}$ is an unbiased estimator of $\nabla_\Theta \mathrm{C}$ [see (B1)], from the diagonal terms of (B3), we see that

$$\mathrm{E}\big(G_m^2\big) = \left(\frac{\partial \mathrm{C}}{\partial\theta_m}\right)^2 + \mathrm{Cov}(\mathbf{G})_{m,m}.$$

Summing over components. m gives Eq. (B6). □

Finally, assuming that $\theta_m(t)$ are independent, identically distributed Bernoulli random variables taking values $\pm\epsilon$ with probability 1/2 simplifies these expressions. In this case, the variance and forth moments are

$$\theta_m(t) \sim \mathrm{Bernoulli}\left(\pm\epsilon, \frac{1}{2}\right) \Rightarrow \sigma_m = \alpha_m = \epsilon.$$

Therefore, the first term in the sum (B6) vanishes. Furthermore, as the ratio of variances in the inner sum becomes one, we evaluate this sum as

$$\sum_{\mu\neq m}\left(\frac{\partial \mathrm{C}}{\partial\theta_\mu}\right)^2 = \|\nabla_\Theta \mathrm{C}\|^2 - \left(\frac{\partial \mathrm{C}}{\partial\theta_m}\right)^2.$$

Thus, for the Bernoulli perturbation model, we have

$$\mathrm{E}\big(\|\mathbf{G}\|^2\big) = \left(1 + \frac{K-1}{\tau_\theta}\right)\|\nabla_\Theta \mathrm{C}\|^2. \tag{B7}$$

Consequently, the norm of the MGD-derived gradient vector is biased with respect to the true norm of the gradient, and furthermore, this norm will be a relatively poor estimator unless $\tau_\theta \gg K$.

REFERENCES

[1] K. Hagey and A. Fitch, "Sam Altman seeks trillions of dollars to reshape business of chips and AI," Wall Street Journal, Feb. 8, 2024.
[2] G. Hinton, "The forward-forward algorithm: Some preliminary investigations," arXiv:2212.13345 [cs.LG] (2022).
[3] T. P. Lillicrap, A. Santoro, L. Marris, C. J. Akerman, and G. Hinton, "Backpropagation and the brain," Nat. Rev. Neurosci. **21**, 335–346 (2020).
[4] C. Li, D. Belkin, Y. Li, P. Yan, M. Hu, N. Ge, H. Jiang, E. Montgomery, P. Lin, Z. Wang, W. Song, J. P. Strachan, M. Barnell, Q. Wu, R. S. Williams, J. J. Yang, and Q. Xia, "Efficient and self-adaptive in-situ learning in multilayer memristor neural networks," Nat. Commun. **9**(1), 2385 (2018).
[5] S. Pai, Z. Sun, T. W. Hughes, T. Park, B. Bartlett, I. A. D. Williamson, M. Minkov, M. Milanizadeh, N. Abebe, F. Morichetti, A. Melloni, S. Fan, O. Solgaard, and D. A. B. Miller, "Experimentally realized in situ backpropagation for deep learning in photonic neural networks," Science **380**, 398–404 (2023).

[6]S. R. Nandakumar, M. Le Gallo, C. Piveteau, V. Joshi, G. Mariani, I. Boybat, G. Karunaratne, R. Khaddam-Aljameh, U. Egger, A. Petropoulos, T. Antonakopoulos, B. Rajendran, A. Sebastian, and E. Eleftheriou, "Mixed-precision deep learning based on computational memory," Front. Neurosci. **14**, 519263 (2020).
[7]E. R. W. van Doremaele, T. Stevens, S. Ringeling, S. Spolaor, M. Fattori, and Y. van de Burgt, "Hardware implementation of backpropagation using progressive gradient descent for in situ training of multilayer neural networks," Sci. Adv. **10**, eado8999 (2024).
[8]M. J. Rasch, C. Mackin, M. L. Gallo, A. Chen, A. Fasoli, F. Odermatt, N. Li, S. R. Nandakumar, P. Narayanan, H. Tsai, G. W. Burr, A. Sebastian, and V. Narayanan, "Hardware-aware training for large-scale and diverse deep learning inference workloads using in-memory computing-based accelerators," Nat. Commun. **14**(1), 5282 (2023).
[9]S. M. Buckley, A. N. Tait, A. N. McCaughan, and B. J. Shastri, "Photonic online learning: A perspective," Nanophotonics **12**, 833–845 (2023).
[10]L. G. Wright, T. Onodera, M. M. Stein, T. Wang, D. T. Schachter, Z. Hu, and P. L. McMahon, "Deep physical neural networks trained with backpropagation," Nature **601**, 549–555 (2022).
[11]O. Bichler, M. Suri, D. Querlioz, D. Vuillaume, B. DeSalvo, and C. Gamrat, "Visual pattern extraction using energy-efficient '2-PCM synapse' neuromorphic architecture," IEEE Trans. Electron Devices **59**, 2206–2214 (2012).
[12]S. Friedmann, J. Schemmel, A. Grubl, A. Hartel, M. Hock, and K. Meier, "Demonstrating hybrid learning in a flexible neuromorphic hardware system," IEEE Trans. Biomed. Circuits Syst. **11**, 128–142 (2017).
[13]W. Gerstner, M. Lehmann, V. Liakoni, D. Corneil, and J. Brea, "Eligibility traces and plasticity on behavioral time scales: Experimental support of neoHebbian three-factor learning rules," Front. Neural Circuits **12**, 53 (2018).
[14]A. Dembo and T. Kailath, "Model-free distributed learning," IEEE Trans. Neural Networks **1**, 58–70 (1990).
[15]T. Matsumoto and M. Koga, "Novel learning method for analogue neural networks," Electron. Lett. **26**, 1136 (1990).
[16]G. Cauwenberghs, "A fast stochastic error-descent algorithm for supervised learning and optimization," in *Advances in Neural Information Processing Systems (NIPS 1992)* (Morgan Kaufmann, 1992), Vol. 5, pp. 244–251.
[17]B. Flower and M. Jabri, "Summed weight neuron perturbation," in *NIPS'92: Proceedings of the 5th International Conference on Neural Information Processing Systems* (Morgan Kaufmann, 1992), pp. 212–219.
[18]J. Alspector, R. Meir, B. Yuhas, A. Jayakumar, and D. Lippe, "A parallel gradient descent method for learning in analog VLSI neural networks," in *NIPS'92: Proceedings of the 5th International Conference on Neural Information Processing Systems* (Morgan Kaufmann, 1992), pp. 836–844.
[19]D. B. Kirk and D. Kerns, "Analog VLSI implementation of multi-dimensional gradient descent," in *Advances in Neural Information Processing Systems (NIPS 1992)* (Morgan Kaufmann, 1992), Vol. 5, pp. 789–796.
[20]Y. Maeda, H. Hirano, and Y. Kanata, "A learning rule of neural networks via simultaneous perturbation and its hardware implementation," Neural Networks **8**, 251–259 (1995).
[21]G. Cauwenberghs, "An analog VLSI recurrent neural network learning a continuous-time trajectory," IEEE Trans. Neural Networks **7**, 346–361 (1996).
[22]P. Moerland and E. Fiesler, "Hardware-friendly learning algorithms for neural networks: An overview," in *Proceedings of Fifth International Conference on Microelectronics for Neural Networks* (IEEE, 1996), pp. 117–124.
[23]A. J. Montalvo, R. S. Gyurcsik, and J. J. Paulos, "Toward a general-purpose analog VLSI neural network with on-chip learning," IEEE Trans. Neural Networks **8**, 413–423 (1997).
[24]H. Miyao, K. Noguchi, M. Koga, and T. Matsumoto, "Multifrequency oscillation learning method for analog neural network: Its implementation in a learning LSI," Electron. Commun. Jpn. **80**, 52 (1997).
[25]S. Draghici, "Neural networks in analog hardware—Design and implementation issues," Int. J. Neural Syst. **10**, 19–42 (2000).
[26]G. Cauwenberghs, "Analog VLSI autonomous systems for learning and optimization," Ph.D. thesis, Caltech, 1994.
[27]A. N. McCaughan, B. G. Oripov, N. Ganesh, S. W. Nam, A. Dienstfrey, and S. M. Buckley, "Multiplexed gradient descent: Fast online training of modern datasets on hardware neural networks without backpropagation," APL Mach. Learn. **1**, 026118 (2023).
[28]S. Dalm, M. van Gerven, and N. Ahmad, "Effective learning with node perturbation in deep neural networks," arXiv:2310.00965 (2023).
[29]M. Ren, S. Kornblith, R. Liao, and G. Hinton, "Scaling forward gradient with local losses," arXiv:2210.03310 [cs.LG] (2023).
[30]Y. Zhao, X. Yu, Z. Chen, Z. Liu, S. Liu, and Z. Zhang, "Tensor-compressed back-propagation-free training for (physics-informed) neural networks," arXiv:2308.09858 (2023).
[31]S. Bandyopadhyay, A. Sludds, S. Krastanov *et al.*, "Single-chip photonic deep neural network with forward-only training," Nat. Photon. **18**, 1335–1343 (2024).
[32]J. C. Spall, "Multivariate stochastic approximation using a simultaneous perturbation gradient approximation," IEEE Trans. Autom. Control **37**, 332–341 (1992).
[33]T. P. Lillicrap, D. Cownden, D. B. Tweed, and C. J. Akerman, "Random synaptic feedback weights support error backpropagation for deep learning," Nat. Commun. **7**, 13276 (2016).
[34]P. Züge, C. Klos, and R. M. Memmesheimer, "Weight versus node perturbation learning in temporally extended tasks: Weight perturbation often performs similarly or better," Phys. Rev. X **13**, 021006 (2023).
[35]Note, whereas details of normalization factors are important to make mathematical derivations precise, in practice there is often flexibility that can be accommodated by a well-placed scale factor. For example, loss function scaling can be passed into gradient scaling both of which, operationally, can be absorbed into the setting of the learning rate.
[36]The non-linear activation function is assumed to act componentwise on each of its inputs.
[37]D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," arXiv:1412.6980 [cs.LG] (2014).
[38]J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," Proc. Natl. Acad. Sci. U. S. A. **79**, 2554 (1982).
[39]B. Scellier and Y. Bengio, "Equilibrium propagation: Bridging the gap between energy-based models and backpropagation," Front. Comput. Neurosci. **11**, 24 (2017).
[40]S. Dillavou, B. D. Beyer, M. Stern, A. J. Liu, M. Z. Miskin, and D. J. Durian, "Machine learning without a processor: Emergent learning in a nonlinear analog network," Proc. Natl. Acad. Sci. U. S. A. **121**, e2319718121 (2024).
[41]A. Nøkland, "Direct feedback alignment provides learning in deep neural networks," in *30th Conference on Neural Information Processing Systems* (Curran Associates, 2016), pp. 1037–1045.
[42]A. Momeni, B. Rahmani, B. Scellier, L. G. Wright, P. L. McMahon, C. C. Wanjura, Y. Li, A. Skalli, N. G. Berloff, T. Onodera, I. Oguz, F. Morichetti, P. del Hougne, M. L. Gallo, A. Sebastian, A. Mirhoseini, C. Zhang, D. Marković, D. Brunner, C. Moser, S. Gigan, F. Marquardt, A. Ozcan, J. Grollier, A. J. Liu, D. Psaltis, A. Alù, and R. Fleury, "Training of physical neural networks," arXiv:2406.03372 (2024).
[43]S. Schmidgall, R. Ziaei, J. Achterberg, L. Kirsch, S. P. Hajiseyedrazi, and J. Eshraghian, "Brain-inspired learning in artificial neural networks: A review," APL Mach. Learn. **2**, 21501 (2024).
[44]S. Bartunov, A. Santoro, B. Richards, L. Marris, G. E. Hinton, and T. Lillicrap, "Assessing the scalability of biologically-motivated deep learning algorithms and architectures," in *Advances in Neural Information Processing Systems (NeurIPS)*, 31 (Curran Associates, 2018).
[45]M. Yan, C. Huang, P. Bienstman, P. Tino, W. Lin, and J. Sun, "Emerging opportunities and challenges for the future of reservoir computing," Nat. Commun. **15**(1), 2056 (2024).
[46]G. B. Huang, Q. Y. Zhu, and C. K. Siew, "Extreme learning machine: Theory and applications," Neurocomputing **70**, 489–501 (2006).
[47]C. Mead, "How we created neuromorphic engineering," Nat. Electron. **3**(7), 434–435 (2020).
[48]S. P. Adhikari, H. Kim, R. K. Budhathoki, C. Yang, and L. O. Chua, "A circuit-based learning architecture for multilayer neural networks with memristor bridge synapses," IEEE Trans. Circuits Syst. **62**, 215–223 (2015).
[49]C. Wang, L. Xiong, J. Sun, and W. Yao, "Memristor-based neural networks with weight simultaneous perturbation training," Nonlinear Dyn. **95**, 2893–2906 (2019).

[50]H. S. Seung, "Learning in spiking neural networks by reinforcement of stochastic synaptic transmission," Neuron **40**, 1063–1073 (2003).
[51]I. R. Fiete and H. S. Seung, "Gradient learning in spiking neural networks by dynamic perturbation of conductances," Phys. Rev. Lett. **97**, 048104 (2006).
[52]J. Werfel, X. Xie, and H. S. Seung, "Learning curves for stochastic gradient descent in linear feedforward networks," Neural Comput. **17**, 2699–2718 (2005).
[53]N. Hiratani, Y. Mehta, T. P. Lillicrap, and P. E. Latham, "On the stability and scalability of node perturbation learning," in *Advances in Neural Information Processing Systems* (Curran Associates, 2022), Vol. 35, pp. 31929–31941.
[54]B. Mukhoty, V. Bojkovic, W. D. Vazelhes, X. Zhao, G. D. Masi, H. Xiong, and B. Gu, "Direct training of SNN using local zeroth order method," in *Advances in Neural Information Processing Systems* (Curran Associates, 2023), Vol. 36, pp. 18994–19014.
[55]M. Xiao, Q. Meng, Z. Zhang, D. He, and Z. Lin, "Online pseudo-zeroth-order training of neuromorphic spiking neural networks," arXiv:2407.12516 (2024).
[56]S. Malladi *et al.*, "Fine-tuning language models with just forward passes," in *Advances in Neural Information Processing Systems* (Curran Associates, 2023), Vol. 36, pp. 53038–53075.
[57]A. Chen, Y. Zhang, J. Jia, J. Diffenderfer, J. Liu, K. Parasyris, Y. Zhang, Z. Zhang, B. Kailkhura, S. Liu, and S. Barbara, "DeepZero: Scaling up zeroth-order optimization for deep model training," arXiv:2310.02025 (2023).
[58]N. Ding, Y. Qin, G. Yang *et al.*, "Parameter-efficient fine-tuning of large-scale pre-trained language models," Nat. Mach. Intell. **5**, 220–235 (2023).