A general approach to fast online training of modern datasets on real neuromorphic systems without backpropagation

Sonia Buckley*
Adam N. McCaughan*
sonia.buckley@nist.gov
adam.mccaughan@nist.gov
National Institute of Standards and Technology
Boulder, Colorado, USA

ABSTRACT

We present parameter-multiplexed gradient descent (PMGD), a perturbative gradient descent framework designed to easily train emergent neuromorphic hardware platforms. We show its applicability to both analog and digital systems. We demonstrate how to use it to train networks with modern machine learning datasets, including Fashion-MNIST and CIFAR-10. Assuming realistic timescales and hardware parameters, our results indicate that PMGD could train a network on emerging hardware platforms orders of magnitude faster than the wall-clock time of training via backpropagation on a standard GPU/CPU.

CCS CONCEPTS

• **Hardware → Emerging technologies**; • **Computing methodologies → Machine learning algorithms**.

KEYWORDS

machine learning, neural networks, neuromorphic computing, emerging hardware

ACM Reference Format:
Sonia Buckley and Adam N. McCaughan. 2022. A general approach to fast online training of modern datasets on real neuromorphic systems without backpropagation. In *International Conference on Neuromorphic Systems (ICONS 2022), July 27–29, 2022, Knoxville, TN, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3546790.3546810

1 INTRODUCTION

Backpropagation is by far the most commonly used method of computing the gradient for gradient descent in multi-layer neural networks, but has proved to be challenging to implement in new hardware platforms [20]. However, training via the gradient descent algorithm 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 than backpropagation in software and so are rarely used in machine learning. 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, which 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 [9], which has been employed for chip-in-the-loop training [17]. 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 [5, 18].

In this paper, we describe and demonstrate parameter-multiplexed gradient descent (PMGD), a framework for implementing model-free perturbative methods in hardware. PMGD can be used to efficiently train modern neural network architectures in a way that can be directly applied to modern neuromorphic hardware. Model-free perturbative methods were investigated for training VLSI neural networks beginning in the 1990s [1, 4, 8, 10–13] but were extremely limited in scale, comprising small datasets with only a few neurons. Here, we take a new look at these techniques in the context of modern machine learning datasets and new neuromorphic hardware platforms. We show that under realistic assumptions for analog and digital neuromorphic hardware platforms, PMGD should be able to train modern datasets such as CIFAR-10 significantly faster than the wall-clock time of a GPU.

2 DESCRIPTION OF MODEL-FREE METHODS

We begin by assuming a hardware neural network that takes inputs x and parameters (e.g. weights and biases) θ and uses them to compute the output $y = f(x;\theta)$. The goal is to train the network to produce outputs y_{target} via gradient descent on a cost function $C(y, y_{target})$. To perform gradient descent, the gradient $dC/d\theta$ must be calculated and the parameters adjusted in order to minimize C. In software machine learning, the gradient $dC/d\theta$ is commonly calculated using backpropagation. However, backpropagation is not the only way to compute $dC/d\theta$. For instance, an alternative way to estimate the gradient is via finite difference which perturbs individual parameters and observes their effect on the cost output.

In finite-difference, one parameter θ_i is perturbed at a time by $\Delta\theta_i$ while holding the remaining parameters constant, and the corresponding change in the cost ΔC_i is recorded. By repeating this process for each parameter, a gradient estimate can be generated

*Both authors contributed equally to this research.

ICONS '22, July 27–29, 2022, Knoxville, TN

ACM ISBN 978-1-4503-9789-6/22/07…$15.00
https://doi.org/10.1145/3546790.3546810

component-by-component as $\partial C/\partial\theta_i = \Delta C_i/\Delta\theta_i$. This has the advantage of being "model-free": the gradient computation can be performed without any knowledge of the system's model such as its layer structure or the form of its activation functions. There are several disadvantages to using finite-difference in practice. Before any parameter can be updated using the gradient estimate, all parameters must be sequentially perturbed and their corresponding effect on the cost (ΔC_i) recorded, which may cause a significant delay between parameter updates. During the update process, each value of ΔC_i must be stored, requiring extra memory as compared to the inference phase. A global sequence of perturbations must also be implemented in hardware, requiring synchronization signals to be sent to each parameter. Fortunately, other model-free perturbative methods have been developed that avoid all of these disadvantages. For example, if the update to the weight θ_i is applied immediately after computing $\Delta C_i/\Delta\theta_i$ (coordinate descent) the requirement for the extra memory is eliminated.

Below we describe the PMGD framework for applying these model-free techniques to neuromorphic hardware with an emphasis on creating simple, highly localized neuronal circuits. With a single hardware instantiation, this framework allows the designer to implement a wide variety of model-free gradient descent techniques – all the way from finite-difference to SPSA [18] – by only modifying three time constants (Sections 5-6) and the perturbation type (Section 4). We then analyze these techniques' training performance in both analog and digital configurations (Section 7) and on modern machine learning datasets (Section 8). A major advantage of this framework is that it can be used to perform online training on any hardware platform with programmable weights–including many hardware platforms originally designed only for inference–while making minimal hardware modifications.

3 PARAMETER-MULTIPLEXED GRADIENT DESCENT

Assume that we have the network shown in Fig 1 with time-varying inputs $x(t)$, $y_{target}(t)$, outputs $y(t)$ and parameters θ. Our goal is to train the hardware parameters θ such that the inputs x produce the desired outputs $y = y_{target}$. We begin by perturbing the parameters θ from their static values θ^0 by $\tilde{\theta}(t)$ (Fig 1a)[1]. Depending on the hardware (e.g. digital versus analog), the perturbations $\tilde{\theta}(t)$ could be either discrete or continuous, and can take a variety of patterns as shown in Fig 1c.

As the parameters are perturbed in time, the hardware continuously computes[2] the output $y(t)$ and the cost $C(t)$. To compute the gradient, the hardware must extract the variation in the cost due to the perturbation $\tilde{C}(t)$. In the discrete case, this is calculated by subtraction of the unperturbed cost C_0 from the perturbed cost $C[t]$ at timestep t ($\tilde{C}[t] = C[t] - C_0$). In the continuous case this is calculated via a high pass filter with time constant τ_{hp} on the cost that extracts the time-varying (perturbative) component $\tilde{C}(t)$. In

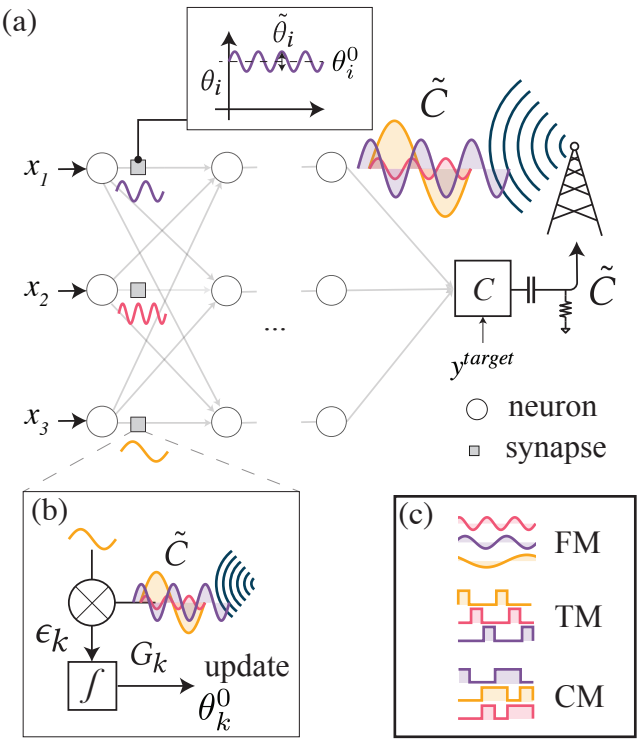

Figure 1: The PMGD framework for training neuromorphic hardware. (a) A perturbation $\tilde{\theta}_i$ is applied to the static value of every synapse/parameter θ_i^0 in a network. Perturbations are propagated through the network to influence the cost C. The time varying component of the cost $\tilde{C}$ due to perturbations to the parameters is extracted and broadcast such that it is available globally to every parameter. (b) The global parameter $\tilde{C}$ is multiplied locally with the time varying parameter perturbation $\tilde{\theta}_k$ and integrated to generate an estimate of the gradient G_k with respect to that parameter. The static value of the parameter θ_k^0 is then updated by $\theta_k^0 \to \theta_k^0 - \eta G_k$. (c) $\tilde{\theta}$ can take different forms, including both analog and discrete cases.

both cases the $\tilde{C}(t)$ signal is then broadcast globally such that it is available to all synapses.

The key idea is that using only this global $\tilde{C}(t)$ signal and the local perturbation signal $\tilde{\theta}_i(t)$, each parameter θ_i can compute its own partial derivative $\partial C/\partial\theta_i$ and autonomously update itself. This is possible as long as the perturbations $\tilde{\theta}(t)$ are small in amplitude and orthogonal, which guarantees that when the product $\tilde{C}(t)\tilde{\theta}_i(t)$ is integrated over time, contributions due to other parameters will cancel out. We denote the product $\tilde{C}(t)\tilde{\theta}_i(t)$ as ϵ_i, the error signal for parameter i. To ensure the magnitude of the perturbation does not affect the magnitude of the error, this value is normalized by A^2, the square of the amplitude of the perturbation $\tilde{\theta}_i$.

Fig 1b shows how the error signal ϵ_k is continuously computed and integrated over time, building up an approximation of the

[1]We note that in principle the "static" value of θ^0 is also time varying, as it changes due to gradient descent updates. However, for reasonable learning rates η this is a much smaller change, and so we denote this θ^0 rather than $\theta^0(t)$. Similarly for C_0.
[2]For the sake of simplicity, we assume $y(t)$ and $C(t)$ are computed instantaneously, however this is not a strict requirement [5]

partial gradient $G_k(t)$ for the parameter θ_k. In general, the longer the parameter integration time τ_θ, the better the gradient approximation $G(t)$ will be when the parameters are updated. Although individual updates generally do not follow the exact direction of the gradient, over time the updates accumulate such that the system is following the gradient with respect to the cost.

While the gradient approximation is continuously accumulated in both the analog and digital cases, the details differ. In the digital case, this accumulation is just a summation of the error signal $\epsilon_i = \tilde{C}[t]\tilde{\theta}_i[t]/A^2$ in each discrete timestep t as follows:

$$G_i[t] = G_i[t-1] + \epsilon_i \tag{1}$$

where A is the amplitude of the perturbation signals $\tilde{\theta}$. After time τ_θ, the accumulated gradient approximation $G_i[t]$ is used to perform the update $\theta_i^0 \rightarrow \theta_i^0 - \eta G_i[t]$, and $G[t]$ is reset to zero. Here, η is the learning rate.

In the analog case, the gradient approximation is accumulated using a leaky integrator circuit applied to the error signal $\epsilon_i = \tilde{C}(t)\tilde{\theta}_i(t)/A^2$, taking the form:

$$G_i(t) = \int_0^t (\epsilon_i - G_i(t)/\tau_\theta)\, dt \tag{2}$$

and $G(t)$ is never reset to zero. For this case, the integration can be done via a low pass filter with time constant τ_θ, and the update is performed continuously, in effect operating as an optimizer with momentum. The value of the time constant τ_θ approximately determines the amount of momentum.

The parameter update is always given by $\theta^0 \rightarrow \theta^0 - \eta G(t)$. In the discrete case, if $G[t]$ is not reset to zero, it can be used to implement momentum.

This technique allows hardware to be very localized – for instance, each weight i in the network could have accompanying circuitry nearby that generates its own perturbation (e.g. a unique oscillator) and stores $G_i(t)$ (e.g. in a capacitor). Each parameter would then use only one non-local signal (the globally-broadcast $\tilde{C}(t)$) in addition to local information to update itself, while the system as a whole evolves in the direction of gradient descent. This configuration represents a truly non-Von Neumann architecture that avoids the "tyranny of wires" and bandwidth issues common in bussed architectures. In a system in which such local circuits cannot or have not yet been implemented, this can also be used in a chip-in-the-loop technique for training, by computing $\tilde{C}(t)$ and the update to θ off-chip and applying only the perturbations and final updates on-chip.

4 PERTURBATION TYPES

The perturbation signals $\tilde{\theta}(t)$ can take many forms, but ideally they are small-amplitude, zero-mean, and orthogonal to each other [5]. During operation, $\tilde{\theta}(t)$ is temporarily added to the values of the parameters θ as a means of estimating the gradient. These perturbations are distinct from the gradient descent updates to the parameters. Here we discuss several types of discrete perturbations suitable for digital hardware, and a single type of analog perturbation.

Conceptually, the simplest perturbation is time-multiplexing (TM), where one parameter per timestep τ_p is perturbed and the remaining parameters have zero perturbation (coordinate descent). In this scenario, the error signal $\epsilon_i = \tilde{C}(t)\tilde{\theta}_i(t)/A^2$ is only non-zero for parameter θ_i once per period, where the period length is determined by the number of parameters. Time multiplexing has several practical disadvantages for implementation on hardware, including the fact that the parameter perturbations are not simultaneous, requiring a global clock to synchronize timings. This is a result of the fact that parameters must track when it's "their turn" to be perturbed.

A second discrete example is code-multiplexing, which refers to simultaneous discrete perturbations on $\{-A, +A\}$ for every parameter θ_i at every timestep τ_p. These can be either predetermined, fully-orthogonal sequences (CM) or randomly generated sequences of $\{-A, +A\}$ that are statistically orthogonal (R-CM). Unlike time-multiplexing, all parameters are updated simultaneously rather than one at a time. In the R-CM case, since the perturbations are only statistically orthogonal, it may take longer to converge to the true gradient — however, in our tests it is not significantly slower, and random perturbations may be easier to implement in hardware.

A third example which is more applicable for certain types of analog hardware is frequency-multiplexing (FM) using sinusoidal perturbations. For frequency multiplexing, each parameter gets a unique frequency of perturbation, which in hardware may be implemented with local oscillators. On the output of the network, the cost will have components at each perturbative frequency relative to each parameter's impact. The individual contribution of each parameter to the cost can be extracted via frequency analysis, which can be implemented locally with simple analog circuits, for example using homodyne detection or with a lock-in technique. Similar to code-multiplexing, during operation gradient information is generated simultaneously for each parameter. Frequency multiplexing is totally asynchronous, and parameters don't have to keep track of any global synchronization.

5 PARAMETER UPDATES/GRADIENT INTEGRATION TIME

As noted in the previous section, the longer the system integrates the error signal $\epsilon = \tilde{C}(t)\tilde{\theta}(t)/A^2$ before applying an update to the parameters, the better the gradient approximation $G(t)$ becomes. We examined the the amount of time it took the gradient to converge for the various multiplexing techniques. Fig 2a shows the angle between the true gradient and the gradient approximation $G(t)$ versus time when used in a feedforward neural network with three input neurons, three hidden neurons and one output neuron (3-3-1 network) solving a single 3-bit parity problem using various perturbation types. The time axis is in units of τ_p (discrete cases) or $1/2$ bandwidth^{-1} (analog case), where the bandwidth is defined here as the difference between the maximum and minimum perturbation frequencies.

All the multiplexing techniques follow approximately the same trajectory with time, but there are minor differences. Time multiplexing on average takes longer to decrease than the simultaneous techniques at short time scales. R-CM takes slightly longer to converge to zero because it is only statistically orthogonal, and thus a given parameter cannot fully filter out its neighbors' unwanted signals. The oscillations occur in all of the periodic perturbations,

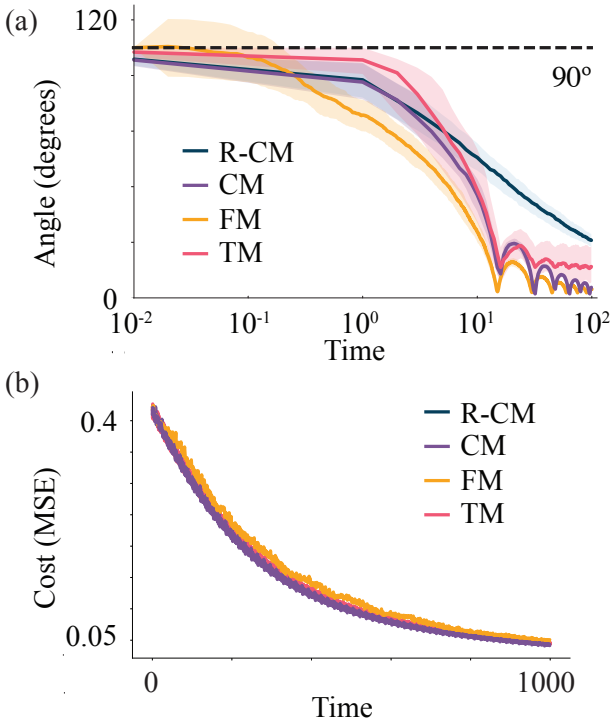

Figure 2: (a) Angle between the gradient approximation and the true gradient versus time for a single 3-bit parity problem with 9 parameters in a 3-3-1 network. The solid lines show the median cost value for randomly initialized networks using the different perturbation types, while the shaded regions show the third quartile values. (b) Basic demonstration of PMGD training a 3-3-1 network using various perturbation types.

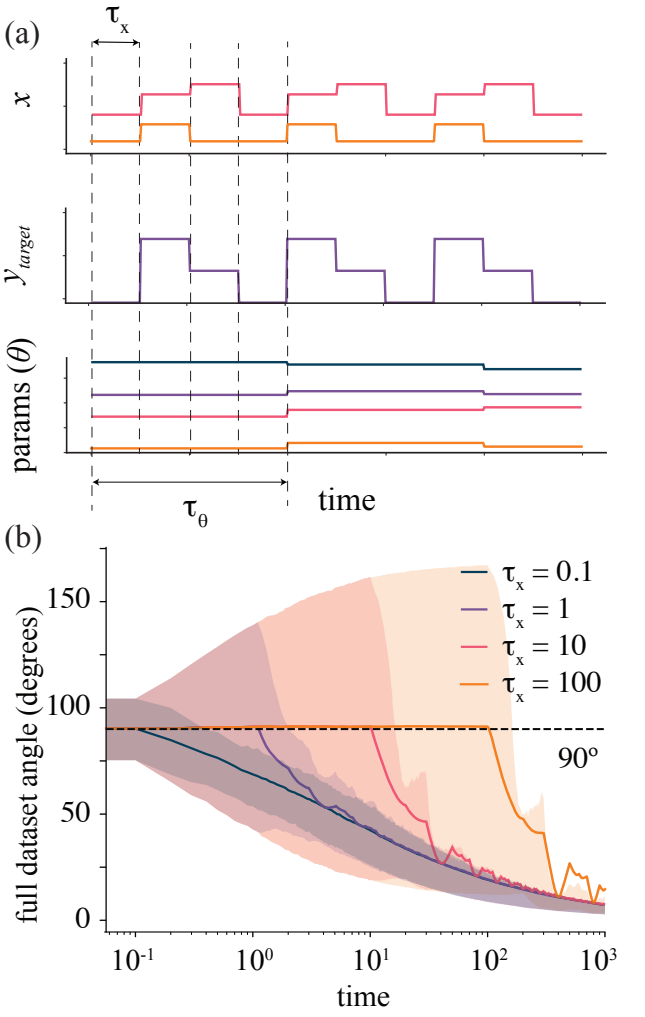

Figure 3: Batching τ_x and τ_θ. (a) x, y_{target} and θ versus time for a toy problem with four parameters, showing τ_x and τ_θ for a batch size of four. (b) Angle between the integrated gradient approximation $G(t)$ and the true gradient with respect to the full dataset versus time for various τ_x values for the 2-bit parity problem implemented on a 2-2-1 network. The solid lines are the median values of 1000 random initializations and the shaded regions represent the third quartile values

as generally after a full period the integrated gradient is approximately equal to the true gradient of the system – for example, in the time multiplexing scheme the completion of a full period corresponds to having accumulated error signals from every parameter. Fig 2b highlights that despite minor differences in gradient convergence time, all of these techniques have very similar performance at minimizing the cost for a fixed bandwidth.

6 MINI-BATCHING AND TIME CONSTANTS

In real applications, one generally needs to to minimize the objective function over an entire training dataset. This can be difficult due to the large sizes of these datasets, so software implementations typically break these datasets into "mini-batches" that are used to perform stochastic gradient descent. Similarly for hardware, there may be a restriction in the number of input samples that can be presented to the network at a time–in many cases for emerging hardware, there may only be one sample at a time. Fortunately, by integrating through time, we can perform a mathematical equivalent to mini-batching even for hardware that can only accept one sample at a time. During training, we need to introduce a new x, y_{target} sample periodically – we define this period as τ_x. As the sample changes, the integrated gradient approximation $G(t)$ will accumulate the error signal from each sample it is shown. After time τ_θ, the parameters will be updated with this accumulated gradient. The mini-batch size is determined by how many samples the network is shown before the parameter update is applied. In the discrete case, the mini-batch size is equal to the ratio τ_θ/τ_x. For example, if $\tau_\theta = \tau_x$ the mini-batch size is 1, if $\tau_\theta = 10\tau_x$, the mini-batch size is 10.

Fig 3a shows an illustration of how the parameter update process is affected by τ_x and τ_θ for the discrete case with two inputs, one

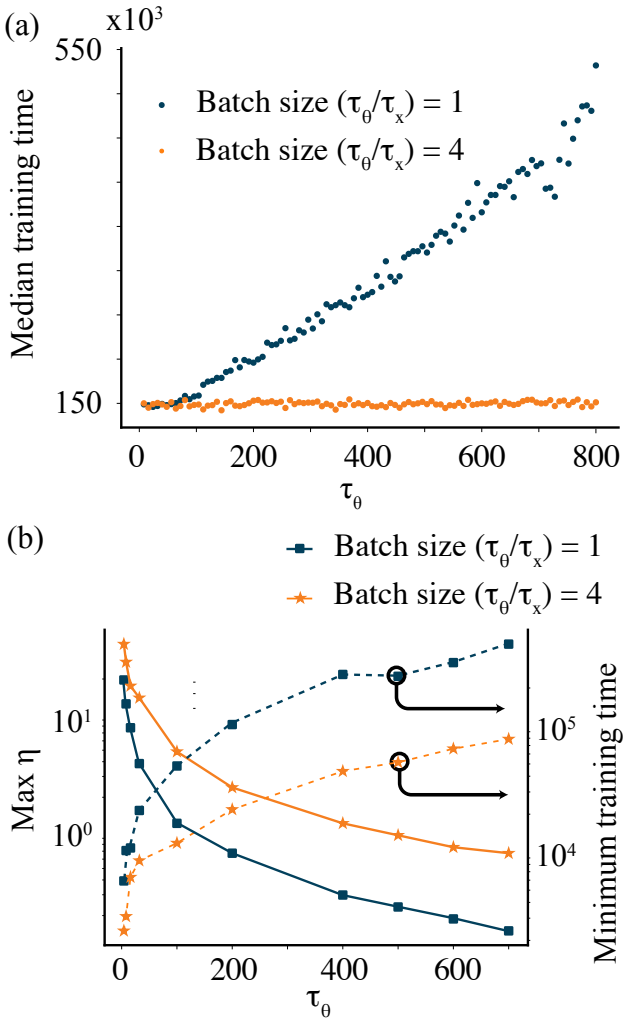

Figure 4: Effect of τ_θ on the solution time of the 2-bit parity (XOR) problem using R-CM perturbations. (a) Solution time as a function of τ_θ and batch size. (b) Effect of τ_θ on the learning rate (η) and minimum training time. For a given τ_θ, "max η" corresponds to the maximum η where the problem converged for at least 50 out of 100 random initializations.

output, and four parameters. Fig 3b shows how long the integrated gradient approximation $G(t)$ takes to align with the gradient of the full dataset for various τ_x values for the 2-bit parity problem implemented on a 2-2-1 network. We see that it takes on the order of τ_x to drop below 90° with respect to the true gradient. In Fig 3b, for a given integration time, $G(t)$ is always more accurate for shorter τ_x values. Therefore we can conclude that for the problem explored here, a shorter τ_x is always the best choice.

To investigate the effect of varying τ_θ and τ_x on solution time, we empirically analyzed the effect of changing τ_x and τ_θ for a particular problem. Fig 4a shows the effect of τ_θ on the solution time of the 2-bit parity (XOR) problem using R-CM perturbations. We define solution time in units of τ_p. Each datapoint represents the median solution of 100 examples at each value of gradient integration/parameter update time (τ_θ). As we varied τ_θ, we kept the mini-batch size constant by making τ_x a multiple of τ_θ. Since the 2-bit parity dataset is composed of 4 x/y_{target} pairs, when τ_x was $4\tau_\theta$, this was equivalent to full gradient descent – all samples were integrated into the $G(t)$ gradient estimation before performing a parameter update. When $\tau_x = \tau_\theta$, this was equivalent to stochastic gradient descent with a mini-batch size of 1. As previously mentioned, τ_θ is a proxy for how accurately the gradient is measured before performing each parameter update.

In the case where mini-batch size = 1, we observed that increasing τ_θ increased the solution time. This is due to the fact that getting *some* gradient information with respect to all training examples is more important than accumulating an accurate gradient measurement with respect to a single example. The conclusion is that if doing stochastic gradient descent (i.e. the batch size is less than the number of training examples), then waiting for a more accurate gradient (larger τ_θ) before updating hurts your overall solution time, as was shown in Ref. [18].

In the case where batch size = 4, we are always getting the gradient with respect to the entire dataset. Since in our implementation the size of the update to the parameters is proportional to τ_θ, if we make τ_θ larger the effective step in the direction of the gradient is larger and the time to solution therefore remains constant. Essentially in the batch size = 4 case, the accuracy with which we measure the gradient does not matter for a given learning rate η.

Even though Fig 4a appears to have a training time independent of τ_θ for the larger batch size, this is not the full story because it only shows results for a single low, fixed learning rate. To investigate how changing τ_θ affects the convergence of training, we repeated the simulation while varying both τ_θ and η. As we increase τ_θ, we find that training does not converge at higher learning rates. The left axis of Fig. 4b shows how the maximum learning rate for which the problem converged versus the gradient integration/parameter update time (τ_θ) for the two different batch sizes. The right axis shows the corresponding median solution time for this maximum η value, i.e. the "minimum" expected training time for a given value of τ_θ. We define convergence as at least 50% of the randomly-initialized training problems converged to a mean cost of 0.01. Fig. 4b also shows the median solution time when trained at that maximal η value. The solution time was calculated by taking the median of the converged solutions at the corresponding maximum η for a given τ_θ. We observed that with larger batch sizes the learning rate can be set higher and therefore train faster. Additionally, smaller values of τ_θ also train faster. We note that in a hardware system, smaller values of τ_θ mean more frequent updates to the parameters.

7 ANALOG AND DIGITAL IMPLEMENTATIONS

Fig 5a illustrates the operational differences between the discrete and analog algorithms for a network with two parameters. The discrete version is shown with R-CM perturbations and the analog version uses FM perturbations. To show how the training performance for the different perturbation types compare, we also trained XOR

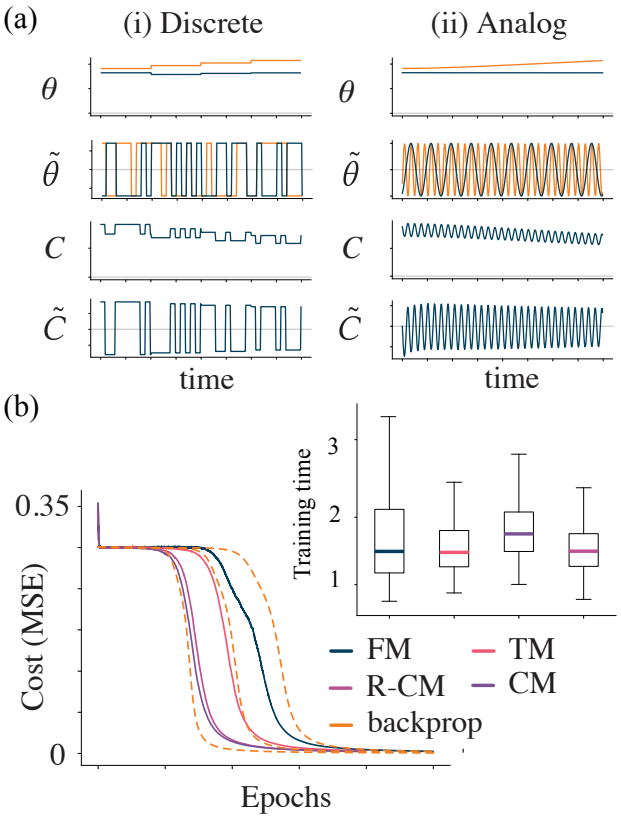

Figure 5: Comparing analog versus digital implementations. (a) Parameter ($\theta(t)$), parameter perturbation ($\tilde{\theta}(t)$), cost ($C(t)$) and cost variation ($\tilde{C}(t)$) for (i) discrete R-CM and (ii) analog FM perturbations on a toy optimization problem with two parameters. (b) Cost versus epochs for four different perturbation types solving the 2-bit parity problem using the same bandwidth. The dashed lines show the same problem trained using backpropagation with three different learning rates. Inset shows statistics for each of the perturbation types with 100 randomly initialized samples.

(2-bit parity) for four different perturbation types with different randomly initialized parameters. Fig. 5b shows the cost versus epochs for one randomly initialized configuration for each perturbation type, along with cost versus epochs calculated via backpropagation with three different learning rates shown in orange dashed lines. The inset shows the solution time distribution for 10 randomly initialized 2-bit parity problems using the different perturbation types. The bandwidth for FM 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.

There are a few notable differences for their hardware implementation. The analog case requires a highpass filter (e.g. RC circuit) at the network output that continuously approximates the perturbative components of the cost $C(t)$. As we have implemented it, the analog case requires a lowpass filter at every parameter element, and a single highpass filter on the network output to convert $C(t)$ to $\tilde{C}(t)$. 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 $\tilde{C}[t] = C[t] - C_0$. 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 accumulate gradient information locally.

8 MODERN DATASET RESULTS

To see if this technique could be useful in real world hardware, we ran a variety of standard machine learning tasks for different network architectures and hyperparameters. The results are shown in Table 1 using R-CM. We also timed (wall-clock time) how long it took to reach the same accuracy on either a CPU (AMD Ryzen 1950X) or GPU (Nvidia 1080 Ti) using both backpropagation and PMGD.

Based on literature, some realistic estimates of parameters that could be implemented in hardware are below. The time below is assumed to be the speed of perturbation (τ_p) and sample change (τ_x). The weight update speed τ_θ shown in the table is in units of τ_p. We are assuming that the inference time τ_{inf} is not the limiting factor, i.e. $\tau_p >> \tau_{inf}$.

- (HW1) Timescale 1 millisecond. Examples: hardware-in-the-loop using an external computer to perform parameter updates (limiting the overall speed) for new neuromorphic hardware under test, or custom electro-optic hardware [17, 19] using thermo-optic weights.
- (HW2) Timescale 1 microsecond. Examples: hardware-in-the-loop using an external custom FPGA to perform parameter updates [7] or custom electro-optic hardware using micro-electro mechanical weights [15].
- (HW3) Timescale 10 nanoseconds. Examples: memristive hardware with custom circuits [2] implemented for PMGD, optical networks using piezo-optomechanical tuning [6]
- (HW4) Timescale 1 nanosecond. Examples: superconducting electronic implementations [16] or hardware using optical high speed modulators [14] or all-optical weights and inputs [3].

The final rows in Table 1 show approximate estimates of how long these problems would take to solve in real time informed by the realistic estimates of time constants above[3]. Based on the results of the table, we see that using realistic estimates for emerging hardware, online training could be significantly faster than current implementations using backpropagation, and with potentially much lower energy costs.

Additionally, for many hardware platforms, training via backpropagation is only done on an imperfect simulation of the hardware, leading to significant degradation in performance when the simulated parameters are deployed to the hardware platforms [20]. However, since PMGD uses the actual hardware for gradient estimation and parameter updates this degradation may be avoided.

[3]Times include computation of C_0

Table 1: Modern machine learning datasets trained with gradient descent by PMGD and backpropagation

Task	2-bit parity	CIFAR10	Fashion-MNIST
Network	2-2-1	3-layer CNN + FC layer	3-layer CNN + FC layer
Total number of parameters	9	26154	14378
τ_θ	1	1	1
Number of τ_p executed	10^4	5×10^6	10^6
batch size	1	1000	1000
Accuracy level	100%	60%	82.5%
Wall-clock time backprop CPU (HW0)	70 ms	-	-
Wall-clock time PMGD CPU (HW0)	200 ms	-	-
Wall-clock time backprop GPU (HW0)	-	480 s	54 s
Wall-clock time PMGD GPU (HW0)	-	14 hours	2.3 hours
Solution time HW1 (1 ms)	20 s	2.8 hours	0.55 hours
Solution time HW2 (1 μs)	20 ms	10 s	2 s
Solution time HW3 (10 ns)	0.2 ms	100 ms	20 ms
Solution time HW4 (1 ns)	0.02 ms	10 ms	2 ms

All the code was written in the Julia language. The n-bit parity problems were solved on the CPU, while the larger problems (CIFAR10 and Fashion-MNIST) were performed on the GPU. The CIFAR10 network is a 3-layer convolutional structure. Each filter has a size 3 ×3 with 16, 32 and 32 output channels respectively, stride 1 and relu activation function. Each convolution is followed by 2×2 maxpooling. There is a final fully-connected layer at the output. The median maximum accuracy obtained using backpropagation for batch size 1000 over a range of learning rates and random initializations was 68.0%. This was limited by our choice of a relatively small network and simple optimizer–to improve the accuracy further, a more complex network architecture (e.g. more layers) or optimizer (e.g. adding momentum) would be needed. The Fashion-MNIST network used a 3-layer convolutional structure with each filter of size 3 ×3 with 16, 32 and 64 output channels respectively, stride of 1 and relu activation function. Each convolution is followed by 2×2 maxpooling. There is a final fully-connected layer at the output. The median maximum accuracy obtained using backpropagation for a batch size of 1000 over a range of learning rates and random initializations was 84.5%.

9 DISCUSSION AND CONCLUSIONS

We show that with realistic timescales for emerging hardware, PMGD could be orders of magnitude faster than backpropagation in terms of wall-clock time-to-solution on a standard GPU/CPU. Although not examined in detail here, many of the hardware platforms examined may also have several orders of magnitude improvement in terms of energy usage as well. Our analysis has looked only at the training speed as a performance metric, but there may be time/energy tradeoffs for particular hardware platforms that lead to a different optimization.

Current emerging hardware for machine learning show great promise for increased speed and energy efficiency, but are often limited by the lack of viable training algorithms. PMGD overcomes this barrier – since PMGD is a perturbative technique, it is applicable to a wide range of systems. It can be applied to both analog and digital hardware platforms, and should be resilient to the presence of noise and device imperfections. While implementing PMGD fully in hardware may require significant redesign of existing hardware implementations, it should be readily testable on almost any hardware platform by using standard chip-in-the-loop techniques. In this case, the speed will most likely be limited by communication with an external computer. For example, perturbations can be injected directly to the hardware from a computer, and that same computer could capture the fluctuations in cost and perform the gradient approximation and calculate the parameter updates. This would allow testing of the algorithm without any changes to the hardware.

ACKNOWLEDGMENTS

This is a contribution of NIST, an agency of the US government, not subject to copyright. Certain commercial equipment, instruments, or materials are identified in this paper to foster understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

REFERENCES

[1] Shyam Prasad Adhikari, Hyongsuk Kim, Ram Kaji Budhathoki, Changju Yang, and Leon O. Chua. 2015. A circuit-based learning architecture for multilayer neural networks with memristor bridge synapses. *IEEE Transactions on Circuits and Systems I: Regular Papers* 62, 1 (jan 2015), 215–223.

[2] Stefano Ambrogio, Pritish Narayanan, Hsinyu Tsai, Robert M. Shelby, Irem Boybat, Carmelo di Nolfo, Severin Sidler, Massimo Giordano, Martina Bodini, Nathan C. P. Farinha, Benjamin Killeen, Christina Cheng, Yassine Jaoudi, and Geoffrey W. Burr. 2018. Equivalent-accuracy accelerated neural-network training using analogue memory. *Nature 2018 558:7708* 558, 7708 (jun 2018), 60–67. https://doi.org/10.1038/s41586-018-0180-5

[3] Liane Bernstein, Alexander Sludds, Ryan Hamerly, Vivienne Sze, Joel Emer, and Dirk Englund. 2021. Freely scalable and reconfigurable optical hardware for deep learning. *Scientific Reports 2021 11:1* 11 (2 2021), 1–12. Issue 1. https://doi.org/10.1038/s41598-021-82543-3

[4] Gert Cauwenberghs. 1996. An analog VLSI recurrent neural network learning a continuous-time trajectory. *IEEE Transactions on Neural Networks* 7, 2 (1996), 346–361. https://doi.org/10.1109/72.485671

[5] Amir Dembo and Thomas Kailath. 1990. Model-Free Distributed Learning. *IEEE Transactions on Neural Networks* 1, 1 (1990), 58–70. https://doi.org/10.1109/72.80205

[6] Mark Dong, Genevieve Clark, Andrew J. Leenheer, Matthew Zimmermann, Daniel Dominguez, Adrian J. Menssen, David Heim, Gerald Gilbert, Dirk Englund, and

Matt Eichenfield. 2021. High-speed programmable photonic circuits in a cryogenically compatible, visible–near-infrared 200 mm CMOS architecture. *Nature Photonics* 16 (12 2021), 59–65. Issue 1. https://doi.org/10.1038/s41566-021-00903-x

[7] Brian Hoskins, Mitchell Fream, Matthew Daniels, Jonathan Goodwill, Advait Madhavan, Jabez Mcclelland, Osama Yousuf, Gina Adam, Wen Ma, Tung Hoang, Mark Branstad, Muqing Liu, Rasmus Madsen, Martin Lueker-Boden, and Martin Lueker. 2021. A System for Validating Resistive Neural Network Prototypes. *International Conference on Neuromorphic Systems 2021* 5 (2021), 1–5.

[8] Marwan Jabri and Barry Flower. 1992. Weight Perturbation: An Optimal Architecture and Learning Technique for Analog VLSI Feedforward and Recurrent Multilayer Networks. *IEEE Transactions on Neural Networks* 3, 1 (1992), 154–157.

[9] J. Kiefer and J. Wolfowitz. 1952. Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics* 23, 3 (1952), 462–466.

[10] Yutaka Maeda, Hiroaki Hirano, and Yakichi Kanata. 1995. A learning rule of neural networks via simultaneous perturbation and its hardware implementation. *Neural Networks* 8, 2 (jan 1995), 251–259.

[11] Yutaka Maeda and Toshiki Tada. 2003. FPGA implementation of a pulse density neural network with learning ability using simultaneous perturbation. *IEEE Transactions on Neural Networks* 14, 3 (may 2003), 688–695. https://doi.org/10.1109/TNN.2003.811357

[12] T. Matsumoto and M. Koga. 1990. Novel learning method for analogue neural networks. *ElL* 26, 15 (sep 1990), 1136.

[13] Hiroshi Miyao, Kazuhiro Noguchi, Masafumi Koga, and Takao Matsumoto. 1997. Multifrequency oscillation learning method for analog neural network: Its implementation in a learning LSI. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)* 80, 5 (1997), 52–64.

[14] Abdul Rahim, Artur Hermans, Benjamin Wohlfeil, Despoina Petousi, Bart Kuyken, Dries Van Thourhout, and Roel Baets. 2021. Taking silicon photonics modulators to a higher performance level: state-of-the-art and a review of new technologies. *https://doi.org/10.1117/1.AP.3.2.024003* 3 (4 2021), 024003. Issue 2.

[15] Carl Ramey, Darius Bunandar, Michael Gould, Mykhailo Tymchenko, Nicholas C. Harris, Reza Baghdadi, and Shashank Gupta. 2021. Dual slot-mode NOEM phase shifter. *Optics Express, Vol. 29, Issue 12, pp. 19113-19119* 29 (6 2021), 19113–19119. Issue 12. https://doi.org/10.1364/OE.423949

[16] Michael Schneider, Emily Toomey, Graham Rowlands, Jeff Shainline, Paul Tschirhart, and Ken Segall. 2022. SuperMind: a survey of the potential of superconducting electronics for neuromorphic computing. *Superconductor Science and Technology* 35 (3 2022), 053001. Issue 5. https://doi.org/10.1088/1361-6668/AC4CD2

[17] Yichen Shen, Nicholas C. Harris, Scott Skirlo, Mihika Prabhu, Tom Baehr-Jones, Michael Hochberg, Xin Sun, Shijie Zhao, Hugo Larochelle, Dirk Englund, and Marin Soljačić. 2017. Deep learning with coherent nanophotonic circuits. *Nature Photonics* 11, 7 (jun 2017), 441–446.

[18] James C. Spall. 1992. Multivariate Stochastic Approximation Using a Simultaneous Perturbation Gradient Approximation. *IEEE Trans. Automat. Control* 37, 3 (1992), 332–341. https://doi.org/10.1109/9.119632

[19] Alexander N. Tait, Thomas Ferreira De Lima, Ellen Zhou, Allie X. Wu, Mitchell A. Nahmias, Bhavin J. Shastri, and Paul R. Prucnal. 2017. Neuromorphic photonic networks using silicon photonic weight banks. *Scientific Reports* 7 (2017), 10 pages. Issue 1. https://doi.org/10.1038/s41598-017-07754-z

[20] Logan G. Wright, Tatsuhiro Onodera, Martin M. Stein, Tianyu Wang, Darren T. Schachter, Zoey Hu, and Peter L. McMahon. 2022. Deep physical neural networks trained with backpropagation. *Nature 2022 601:7894* 601 (1 2022), 549–555. Issue 7894. https://doi.org/10.1038/s41586-021-04223-6