Training neural networks using Metropolis Monte Carlo and an adaptive variant

IOP Publishing Mach. Learn.: Sci. Technol. 3 (2022) 045026 https://doi.org/10.1088/2632-2153/aca6cd

MACHINE LEARNING
Science and Technology

OPEN ACCESS

RECEIVED
11 August 2022

REVISED
31 October 2022

ACCEPTED FOR PUBLICATION
28 November 2022

PUBLISHED
19 December 2022

PAPER

Training neural networks using Metropolis Monte Carlo and an adaptive variant

Stephen Whitelam[1,*], **Viktor Selin**[2], **Ian Benlolo**[2], **Corneel Casert**[3] **and Isaac Tamblyn**[2,4,5,*]

[1] Molecular Foundry, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720, United States of America
[2] Department of Physics, University of Ottawa, Ottawa, ON K1N 6N5, Canada
[3] Department of Physics and Astronomy, Ghent University, 9000 Ghent, Belgium
[4] Cash App, Block, Toronto, ON M5A 1J7, Canada
[5] Vector Institute for Artificial Intelligence, Toronto, ON M5G 1M1, Canada
[*] Authors to whom any correspondence should be addressed.

E-mail: swhitelam@lbl.gov and itamblyn@cash.app

Keywords: adaptive, optimization, neural networks, Metropolis Monte Carlo, gradients

Abstract

We examine the zero-temperature Metropolis Monte Carlo (MC) algorithm as a tool for training a neural network by minimizing a loss function. We find that, as expected on theoretical grounds and shown empirically by other authors, Metropolis MC can train a neural net with an accuracy comparable to that of gradient descent (GD), if not necessarily as quickly. The Metropolis algorithm does not fail automatically when the number of parameters of a neural network is large. It can fail when a neural network's structure or neuron activations are strongly heterogenous, and we introduce an adaptive Monte Carlo algorithm (aMC) to overcome these limitations. The intrinsic stochasticity and numerical stability of the MC method allow aMC to train deep neural networks and recurrent neural networks in which the gradient is too small or too large to allow training by GD. MC methods offer a complement to gradient-based methods for training neural networks, allowing access to a distinct set of network architectures and principles.

1. Introduction

The Metropolis Monte Carlo (MC) algorithm was developed in the 1950s in order to simulate molecular systems [1–4]. The Metropolis algorithm consists of small, random moves of particles, accepted probabilistically. It is, along with other MC algorithms, widely used as a tool to equilibrate molecular systems [5]. Equilibrating a molecular system is similar in key respects to training a neural network: both involve optimizing quantities derived from many degrees of freedom that interact in a nonlinear way. Despite this similarity, the Metropolis algorithm and its variants are not widely used as a tool for training neural networks by minimizing a loss function (for exceptions, see e.g. [6–8]). Instead, this is usually done by gradient-based algorithms [9, 10], and sometimes by population-based evolutionary or genetic algorithms [11–13] to which MC methods are conceptually related.

In this paper we address the potential of the zero-temperature Metropolis MC algorithm and an adaptive variant thereof as tools for neural-network training [14]. The class of algorithm we consider consists of taking a neural network of fixed structure, adding random numbers to all weights and biases simultaneously, and accepting this change if the loss function does not increase. For uncorrelated Gaussian random numbers this procedure is equivalent, for small updates, to normalized or clipped gradient descent (GD) in the presence of Gaussian white noise [15–18], and so its ability to train a neural network should be similar to that of simple GD. We show in section 2.1 that, for a particular supervised-learning problem, this is the case, a finding consistent with results presented by other authors [6–8].

It is sometimes stated that the ability of stochastic algorithms to train neural networks diminishes sharply as the number of network parameters increases (particularly if all network parameters are updated simultaneously). However, population-based evolutionary algorithms have been used to train many-parameter networks [19], and in section 2.2 we show that the ability of Metropolis MC to train a

fully-connected neural network is similar for networks with of order a hundred parameters or of order a million: there is not necessarily a sharp decline of acceptance rate with increasing network size.

What *does* thwart the Metropolis MC algorithm is network heterogeneity. For instance, if the number of connections entering neurons differs markedly throughout the network (as is the case for networks with convolutional- and fully-connected layers) or if the outputs of neurons in different parts of a network differ markedly (as is the case for very deep networks) then stochastic weight changes of a fixed scale will saturate neurons in some parts of the network and scarcely effect change in other parts. The result is an inability to train. To address this problem we introduce a set of simple adaptive modifications of the Metropolis algorithm—a momentum-like term, an adaptive step-size scheduler, and a means of enacting heterogenous weight updates—that are borrowed from ideas commonly used with gradient-based methods. The resulting algorithm, which we call adaptive Monte Carlo or aMC, is substantially more efficient than the non-adaptive Metropolis algorithm in a variety of settings. In section 2.2 we show, for one particular problem, that the acceptance rate of aMC remains much higher than that of the Metropolis algorithm at low values of loss, and can be made almost insensitive to network width, depth, and size. In section 2.3 we show that its momentum-like term speeds the rate at which aMC can learn the high-frequency features of an objective function, much as adaptive methods such as Adam [20] can learn high-frequency features faster than regular GD. In section 2.4 we show that the MC method can train simple recurrent neural networks in the presence of small or large gradients, where gradient-based methods fail. In section 2.5 we show aMC can train deep neural networks in which gradients are too small for gradient-based methods to train. In section 2.6 we comment on the fact that best practices for training nets using MC methods await development. We introduce the elements of aMC throughout section 2, and summarize the algorithm in section 3.

Our conclusion is that the Metropolis MC algorithm and its adaptive variants such as aMC are viable ways of training neural networks, if less developed and optimized than modern gradient-based methods. In particular, MC algorithms can, for small updates, effectively sense the gradient, and they do not fail simply because the number of parameters of a neural network becomes large. MC algorithms should be considered a complement to gradient-based algorithms because they admit different design principles for neural networks. Given a network that permits gradient flow, modern gradient-based algorithms are fast and effective [9, 10, 21]. For large neural nets with tens of millions of parameters we find gradient-based methods to be considerably faster than MC (section 2.6). However, MC algorithms free us from the requirement of ensuring reliable gradient flow (and gradients can be unreliable even in differentiable systems [22]). As a result, we find that MC methods can train deep neural networks and simple recurrent neural networks in which gradients are too small (or too large) for gradient-based methods to work. There already exist solutions to these problems, namely the introduction of skip connections or more elaborate recurrent neural network architectures, but aMC requires neither of these things. One type of solution is architectural, the other algorithmic, and having both options offers more possibilities than having only one.

2. Results

2.1. Metropolis Monte Carlo and its connection to gradient descent

We start with the zero-temperature Metropolis MC algorithm. The zero-temperature limit is not often used in molecular simulation, but it and its variants are widely used (and sometimes called random-mutation hill climbing) for optimizing non-differentiable systems such as cellular automata [23, 24]. Consider a neural network with N parameters (weights and biases) $\boldsymbol{x} = \{x_1, \ldots, x_i, \ldots, x_N\}$, and associated loss function $U(\boldsymbol{x})$. If we propose the simultaneous change of each neural-network parameter by a Gaussian random number [25],

$$x_i \to x_i + \epsilon_i \quad \text{with} \quad \epsilon_i \sim \mathcal{N}(0, \sigma^2), \tag{1}$$

and accept the proposal if the loss does not increase, then, when the basic move scale σ is small, the values x_i of the neural-net parameters evolve according to the Langevin equation [17]

$$\frac{\mathrm{d}x_i}{\mathrm{d}n} = -\frac{\sigma}{\sqrt{2\pi}} \frac{1}{|\nabla U(\boldsymbol{x})|} \frac{\partial U(\boldsymbol{x})}{\partial x_i} + \eta_i(n), \tag{2}$$

where n is training time (epoch), and η is a Gaussian white noise with zero mean and variance $\langle \eta_i(n)\eta_j(n')\rangle = (\sigma^2/2)\delta_{ij}\delta(n-n')$. That is, small stochastic perturbations of a network's weights and biases, accepted if the loss function does not increase, is equivalent to noisy clipped or normalized GD on the loss function.

Given the success of gradient-based training methods, this correspondence shows the potential of the Metropolis algorithm to train neural networks. Consistent with this expectation, we show in figure 1 that the

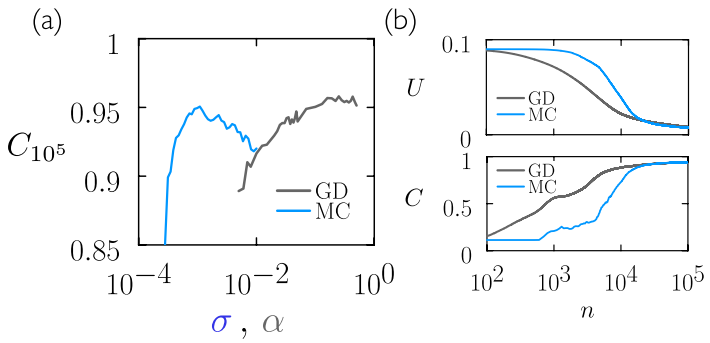

Figure 1. Comparison of zero-temperature Metropolis Monte Carlo (MC) and gradient descent (GD) used to train a neural network to minimize the mean-squared error U on the MNIST training set. (a) MNIST test-set accuracy C_{10^5} after 10^5 epochs of batch learning as a function of MC step size σ (blue) or GD learning rate α (gray). (b) Training loss U and test-set accuracy C versus epoch n for MC step size $\sigma = 2 \times 10^{-3}$ (blue) and GD learning rate $\alpha = 4.5 \times 10^{-2}$ (gray).

zero-temperature Metropolis algorithm can train a neural network. For comparison, we also train the network using simple GD,

$$\boldsymbol{x} \to \boldsymbol{x} - \alpha \nabla U(\boldsymbol{x}), \tag{3}$$

where α is the learning rate, $U(\boldsymbol{x})$ the loss function, and $\nabla \equiv (\partial/\partial x_1, \ldots, \partial/\partial x_N)$ the gradient operator with respect to the neural-network parameters $\boldsymbol{x}$.

We consider a standard supervised-learning problem, recognizing MNIST images [26, 27] using a fully-connected, two-layer neural net. The net has 784 input neurons, 16 neurons in each hidden layer, and one output layer of 10 neurons. The hidden neurons have hyperbolic tangent activation functions, and the output neurons comprise a softmax function. The net has in total 13 002 parameters [28]. We did batch learning, with the loss function U being the mean-squared error on the MNIST training set of size 6×10^4 (in the standard way we consider the ground truth for each training example to be a 1-hot encoding of the class label, and take the 10 outputs of the neural network as its prediction vector).

Figure 1(a) shows the classification accuracy C_{10^5} on the MNIST test set of size 10^4 after 10^5 epochs of training. We show results for MC (blue) and GD (gray), for a range of values of step size σ and learning rate α, respectively. The initial neural-net parameters for MC simulations were $x_i \sim \mathcal{N}(0, \sigma^2)$. The two algorithms behave in a similar manner: each has a range of its single parameter over which it is effective, and displays a maximum at a particular value of that parameter. The value of the maximum for GD is slightly higher than that for MC (about 96% compared to about 95%), and GD achieves near-maximal results over a broader range of its single parameter than does MC.

Figure 1(b) shows loss U and classification accuracy C as a function of epoch for two examples from panel (a). GD trains faster with n initially, but results are comparable near the end of the learning process. The learning dynamics of these algorithms is not the same: in the limit of small steps, the zero-temperature Metropolis algorithm approaches normalized or clipped GD, not standard GD (and its equivalence to the former would only be seen with an appropriately rescaled horizontal axis) [29]. Nonetheless, MC can in effect sense the gradient, as long as the step size is relatively small, and for this problem the range of appropriate step sizes is small compared to the effective GD step size. GD therefore trains faster, but MC has similar capacity for learning. The computational cost per epoch of the two algorithms is of similar order, with MC being cheaper per epoch for batch learning: each MC step requires a forward pass through the data, and each GD step a forward and a backward pass.

There are many things that could be done to improve the learning precision of these algorithms (no pre-processing of data was done, and a basic neural net was used [30]), but this comparison, given a neural net and a data set, confirms that Metropolis MC can achieve results roughly comparable to GD, even on a problem for which gradients are available. For this problem GD trains faster, but MC works. It is worth noting that this conclusion follows from considering a range of step sizes σ: for a single choice of step size it would be possible to conclude that MC does not work at all.

Similar findings have been noted previously: simulated annealing on the Metropolis algorithm [6, 7] and a variant of zero-temperature Metropolis MC (applied weight-by-weight) [8] were used to train neural nets with an accuracy comparable to that of gradient-based algorithms. These results, and the correspondence described in [17] establish both theoretically and empirically the ability of Metropolis MC to train neural nets.

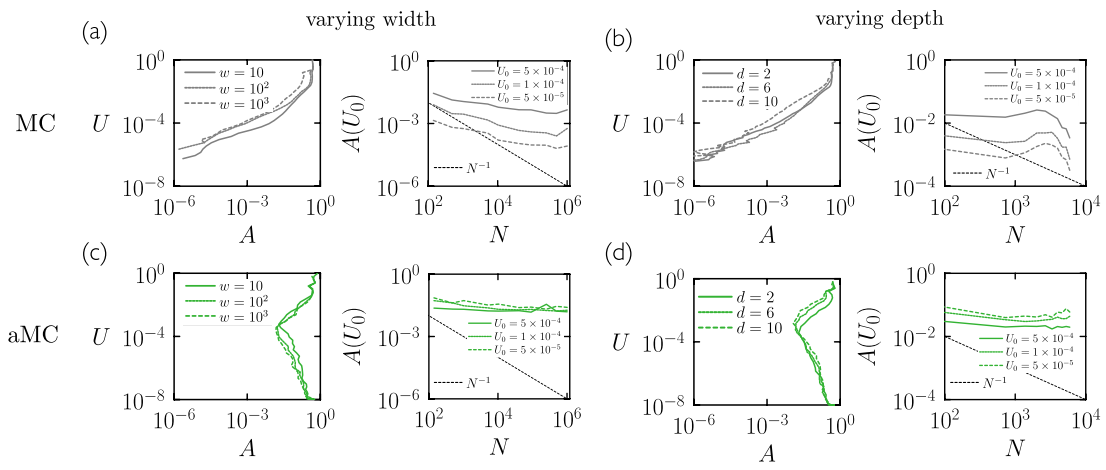

Figure 2. For a particular neural-network supervised-learning task, the acceptance rate of the Metropolis MC method decreases sharply with loss but not model size. (a) Left: Loss U versus acceptance rate A for Metropolis MC for a deep neural net of three different widths. Right: Acceptance rate $A(U_0)$ at fixed loss U_0 for models of different widths, as a function of the number of model parameters N. (b) Analog of (a) for models of increasing depth. (c) Analog of (a) for adaptive Monte Carlo, aMC. (d) Analog of (b) for aMC. MC step size: $\sigma = 10^{-3}$. aMC hyperparameters: $(\sigma_0, \epsilon, n_s, \text{signal norm}) = (10^{-2}, 10^{-2}, 10^2, \text{on})$. Each data point is the average of five independent simulations.

We next turn to the question of how the efficiency of MC training scales with net parameters, and how to improve this efficiency by introducing adaptivity to the algorithm.

2.2. Metropolis acceptance rate as a function of net size

To examine how the efficiency of the Metropolis algorithm changes with neural-net size and architecture, we consider in this section a supervised-learning problem in which a neural net is trained by zero-temperature Metropolis MC to express the sine function $f_0(\theta) = \sin(2\pi\theta)$ on the interval $\theta \in [0, 1]$. The loss U is the mean-squared error between $f_0(\theta)$ and the neural-net output $f_{\boldsymbol{x}}(\theta)$, evaluated at 10^3 evenly-spaced points on the interval. The neural net has one input neuron, which is fed the value θ, and one output, which returns $f_{\boldsymbol{x}}(\theta)$. Its internal structure is fully connected, with hyperbolic tangent nonlinearities. To explore the effect of varying width (panels (a) and (c) of figure 2) we set the depth to 2 and varied the width from 10 to 10^3, these choices corresponding to about 10^2 to about 10^6 parameters. To explore the effect of varying depth (panels (b) and (d) of figure 2) we set the width to 25 and varied the depth from 2 to 10, these choices corresponding to about 10^2 to about 10^4 parameters.

In figure 2((a), left) we show loss U as a function of Metropolis acceptance rate A for three different neural-net widths. The acceptance rate tells us, for fixed step size, the fraction of directions that point downhill in loss. It provides information similar to that shown in plots of the index of the critical points of a loss surface [31, 32], confirming that at large loss there are more downhill directions than at small loss. In figure 2((a), right) we plot the acceptance rate $A(U_0)$ at fixed loss U_0 as a function of the number of net parameters N (obtained by taking horizontal cuts across panel (a); note that more net sizes are shown in panel (b) than panel (a)).

The acceptance rate decreases with increasing net size, but relatively slowly. Upon increasing the size of the net by 4 orders of magnitude, the acceptance rate decreases by about 1 order of magnitude. We have indicated an N^{-1} scaling as a guide to the eye. In the extreme limit, if N simultaneous parameter updates each had to be individually productive, the acceptance rate would decrease exponentially with N, which is clearly not the case. The more dramatic decrease in acceptance rate is with loss: at small loss the acceptance rate becomes very small.

Similar trends are seen with depth in figure 2(b). The acceptance rate declines sharply with loss. It also declines with the number of parameters, slightly more rapidly than in panel (a) but not as rapidly as N^{-1}.

Empirically, therefore, we do not see evidence of a fundamental inability of MC to cope with large numbers of parameters. In section 2.5 we discuss how network heterogeneity can impair the Metropolis algorithm's ability to train a network. The solution, as we discuss there, is to introduce an adaptive variant of the Metropolis algorithm. To motivate the introduction of this algorithm we show in panels (c) and (d) the aMC analog of panels (a) and (b), respectively. The trends experienced by Metropolis have been annulled,

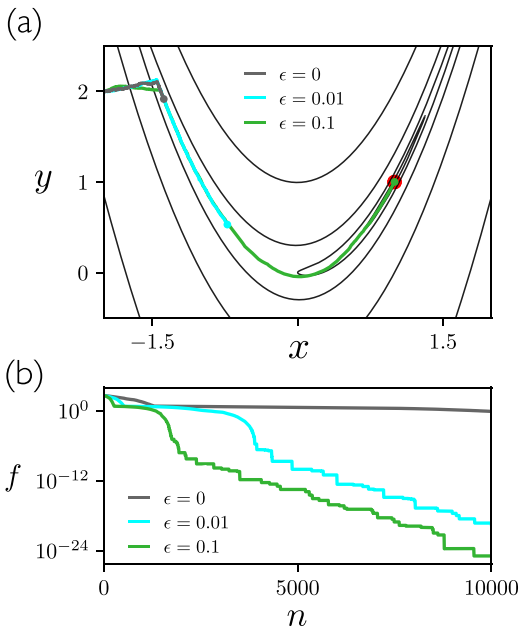

Figure 3. Adaptivity speeds convergence on a test function. We use aMC with three values of ϵ to simulate a particle moving on the Rosenbrock function $f(x, y)$, equation (6). Panel (a) shows a 2000-step simulation for each of three values of ϵ; the global minimum is indicated by the red dot at (1,1). Panel (b) shows the value of the function along each trajectory when continued to $n = 10^4$ steps. aMC hyperparameters: $\sigma_0 = 10^{-3}, n_s = 20$.

the acceptance rates of aMC remaining large and essentially constant with loss or model size over the range of parameters considered.

We now turn to a step-by-step introduction of the elements of aMC.

2.3. Adaptivity speeds learning, particularly of high-frequency features

Modern gradient-based methods are adaptive, allowing the learning rate for each neural-net parameter to differ and to change as a function of the gradients encountered during training [10] (adaptive learning is also used in evolutionary algorithms [33–35]). We can copy this general idea in a simple way within a zero-temperature Metropolis MC scheme by changing the proposed move of equation (1) to

$$\epsilon_i \sim \mathcal{N}(\mu_i, \sigma^2). \tag{4}$$

The parameters μ_i, set initially to zero, are updated after every accepted move according to

$$\mu_i \to \mu_i + \epsilon(\epsilon_i - \mu_i), \tag{5}$$

where ϵ is a hyperparameter of the method. This form of adaptation is similar to the inclusion of momentum in a gradient-based scheme: the center μ_i of each parameter's move-proposal distribution shifts in the direction of the last accepted move ϵ_i, with the aim of increasing the probability of generating moves that will be accepted. In addition, to remove the need for a search over the step-size parameter σ, we introduce a simple adaptive learning-rate schedule by setting $\sigma \to 0.95\,\sigma$ (and $\mu_i = 0$) after n_s consecutive rejected moves. We take the initial step size to be $\sigma = \sigma_0$. We refer to this adaptive Monte Carlo algorithm as aMC, specified in section 3.

The aMC parameter ϵ can be used to influence the rate of learning. In figure 3 we provide a simple illustration of this fact using the two-dimensional Rosenbrock function

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2, \tag{6}$$

often used as a test function for optimization methods [36–38]. The Rosenbrock function has a global minimum at $(x, y) = (1, 1)$ that is set within a long valley surrounded by steep slopes on either side. A particle on this function moving under pure GD takes a long time to reach the global minimum because gradients within the valley are small: the particle will quickly reach the valley and then move slowly along the

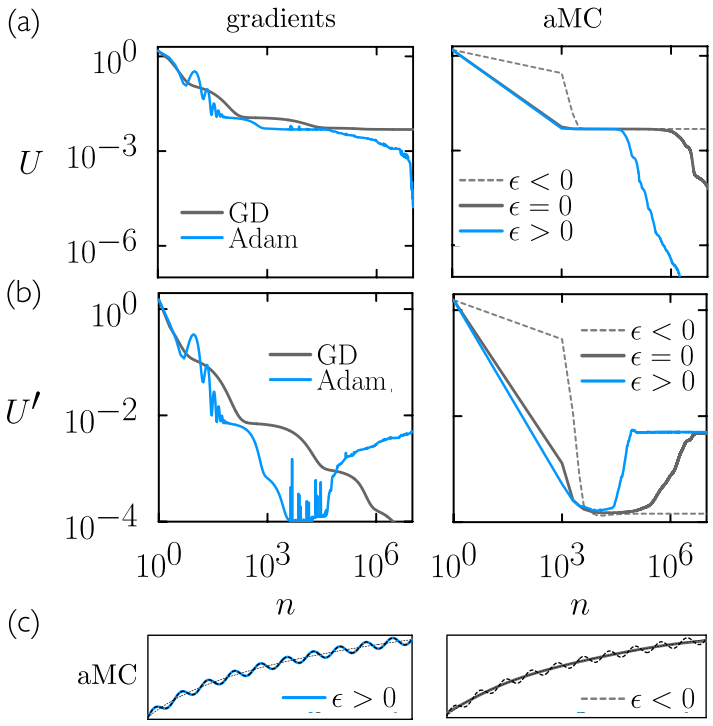

Figure 4. Adaptivity speeds learning of high-frequency features. We train a neural network to learn the function (7) using two gradient-based methods (GD and Adam) and aMC with three values of the parameter ϵ. (a) Loss U as a function of training time n. (b) Pseudo-loss U', which expresses the difference between the net function and the low-frequency term (the logarithm) of (7). (c) Output of neural nets trained by aMC (colored lines) at fixed training time. The thicker and thinner black lines are the function (7) and its low-frequency term, respectively. GD and Adam learning rate: $\alpha = 5 \times 10^{-3}$; aMC hyperparameters: $(\sigma_0, n_s) = (0.1, 50)$.

valley floor [36, 37]. Placing a particle at the point $(-2, 2)$, outside the valley, we evolve the position $\boldsymbol{x} = (x, y)$ of the particle using aMC. We used the initial step size $\sigma_0 = 10^{-3}$ and rescaling parameter $n_s = 100$, and carried out three simulations for three values of ϵ. As shown in the figure, the larger is ϵ the more rapidly is the global minimum attained, with the difference between zero and nonzero epsilon being considerable. This form of adaptivity has an effect similar to that of momentum with GD [39].

Returning to neural-network simulation, the aMC parameter ϵ can speed the rate at which a neural network can learn high-frequency features, much as adaptive methods do with gradient-based algorithms. The extent to which high-frequency features *should* be learned varies by application. For instance, if a training set contains high-frequency noise then we may wish to attenuate an algorithm's ability to learn this noise in order to enhance its ability to generalize. This is the idea expressed in figure 2 of [40]. Empirical studies show that non-adaptive versions of GD sometimes generalize better than their adaptive counterparts [41], in some cases because of the different abilities of these things to learn high-frequency features.

In figure 4 we consider a supervised-learning problem inspired by figure 2 of [40]. We use gradient-based methods (GD and Adam) and aMC to train a neural network to learn the function

$$f_0(\theta) = \ln(1 + 5\,\theta) + \frac{1}{10}\sin(20\,\pi\theta), \tag{7}$$

on $\theta \in [0, 1]$. This function contains a low-frequency term, the logarithm, and a high-frequency term, the sine. The neural network has one input neuron, which is fed the value θ, one output neuron, which returns $f_{\boldsymbol{x}}(\theta)$, and a single hidden layer of 100 neurons with tanh activations. The parameters $\boldsymbol{x}$ of the network were set initially to random values $x_i \sim \mathcal{N}(0, \sigma_0^2)$.

In figure 4(a) we show the training loss, the mean-squared difference U between $f_0(\theta)$ and the neural-net output $f_{\boldsymbol{x}}(\theta)$, evaluated at 1000 evenly-spaced points over the interval. At left we show results produced using GD and Adam, and at right we show results produced using aMC for three values of ϵ, one positive $(\epsilon = 5 \times 10^{-3})$, one negative $(\epsilon = -5 \times 10^{-3})$, and zero. (Note that the case $\epsilon = 0$ still involves adaptive adjustment of the step-size parameter σ.) Of the gradient-based methods Adam trains faster than GD, while for aMC the training loss U decreases fastest for positive ϵ and slowest for negative ϵ. (We did not carry out a systematic search of learning rates in order to compare directly the gradient-based and MC methods; our intent here is to illustrate how adaptivity matters within the two classes of algorithm.)

In figure 4(b) we show the pseudo-loss U' that expresses the mean-squared difference between the net function $f_{\boldsymbol{x}}(\theta)$ and the low-frequency logarithmic term of f_0. The net is not trained to minimize U', but it so happens during training that U' becomes small as the net first learns the low-frequency component of f_0. Subsequently, U' increases as the net also learns the high-frequency component of f_0.

Of the gradient-based methods Adam learns high-frequency features more rapidly than GD. As it does so, the value of the pseudo-loss U' increases. For aMC, the parameter ϵ controls the separation of timescales between the learning of the low- and high-frequency components of f_0. If we want to learn f_0 as quickly as possible then positive ϵ is the best choice. But if we consider the high-frequency component of f_0 to be noise, and regard U' as a measure of the network's generalization error, then negative ϵ is the best choice.

Panel (c) shows the aMC net functions at a time n such that the net trained using $\epsilon = 0$ has begun to learn the high-frequency features of f_0. At the same time the nets trained using positive and negative ϵ have learned these features completely or not at all, respectively.

2.4. Monte Carlo algorithms can train a neural network even when gradients are unreliable

Metropolis Monte Carlo and aMC can effectively sense the gradient [17], and so can train a neural network to similar levels of accuracy as GD. However, MC algorithms can also train a neural network when gradients become unreliable, such as when they vanish or explode.

Vanishing gradients can be overcome by the intrinsic stochasticity of the MC method. In the absence of gradients, pure gradient-based methods receive no signal [42, 43]. However, the Metropolis MC procedure (1) is equivalent, for vanishing gradients, to the diffusive dynamics $\dot{x}_i = \xi_i(n)$, where ξ is a Gaussian white noise with zero mean and variance $\langle \xi_i(n)\xi_j(n')\rangle = \sigma^2\delta_{ij}\delta(n-n')$. Thus Metropolis MC will, in the absence of gradients, enact diffusion in parameter space until nonvanishing gradients are encountered, at which point learning can resume.

MC algorithms can also cope with exploding gradients. Moves are proposed without reference to the gradient, and so can be made on a landscape for which the gradient varies rapidly or is numerically large. MC algorithms are also numerically stable, with moves that would induce large increases in loss being rejected but otherwise not harming the training process.

We show in this section that the ability to cope with vanishing and exploding gradients allows aMC to train recurrent neural networks that gradient-based methods cannot. Recurrent neural networks (RNNs) are designed to act on sequences, and have been applied to problems involving text, images, speech, and music [44–47]. An RNN possesses a vector known as its hidden state, which acts as a form of memory. This vector is updated each time the RNN views a position in a sequence. The size of the gradient scales in general exponentially with sequence length, and so when the sequence is long, and long-term dependencies exist, the gradient tends to vanish or explode. As a result, it can be difficult to train simple RNNs using first-order gradient-based methods in order to learn long-term dependencies in sequence data [9, 48–51]. One solution to this problem is the introduction of more elaborate and computationally expensive RNN architectures such as long short-term memory [42] and gated recurrent units [52, 53]. These architectures can be more reliably trained by gradient-based methods than can simple RNNs.

Here we demonstrate an algorithmic solution to the problem rather than an architectural one, by showing that MC methods can train an RNN in circumstances in which gradient-based methods cannot. We train a simple RNN with tanh activation functions to distinguish the digits of class 0 and 1 in the MNIST data set. We binarized the data, and used a vector of length 2 and a one-hot encoding to represent black or white pixels. The RNN was shown an unrolled version of each image, a sequence of 784 pixels. The RNN architecture is a two-layer stack in which the hidden state at each site in the first layer is used as input for the second layer. The dimension of the hidden state is set to 64, and the final hidden state is sent into a linear classifier. The loss function is the cross entropy between the neural network's prediction and the ground truth.

In figures 5(a) and (b) we show the results of training the RNN using aMC and gradient-based optimizers. We stochastically subsample the training data set for each update ('mini-batching'), using 1500 samples at each step. We used GD and the Adam optimizer, both with learning rate 10^{-3}, with gradient clipping turned on (this has been shown to solve the exploding-gradient problem for some data sets [54]), and aMC with hyperparameters $\sigma_0 = 10^{-3}, n_s = 100, \epsilon = 0$ (which is the Metropolis algorithm with an adaptive step-size scheduler). GD and Adam fail to train (a search over learning rates between the values 10^{-5} and 1 did not result in lower loss values), while aMC trains to small values of loss and a test-set accuracy of 99.8%.

In figure 5(c) we show the size of the gradient associated with the models produced by aMC. aMC does not calculate or make use of the gradient, but we can nonetheless evaluate it for the models produced by the training process. Two distinct regimes can be seen, with the size of the gradient changing by about three orders of magnitude between them. The gradient-based algorithms cannot escape from the small-gradient

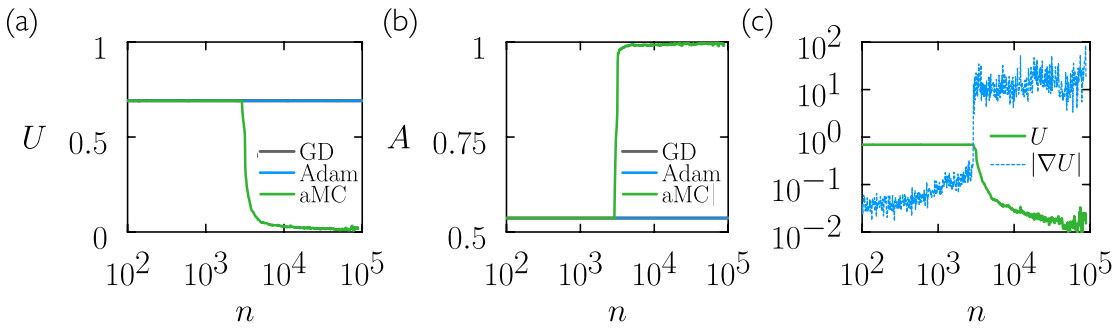

Figure 5. The intrinsic stochasticity and numerical stability of the Monte Carlo method allows it to learn in the presence of numerically small and large gradients. (a) Training-set loss U and (b) test-set classification accuracy C as a function of training epoch n for a simple RNN instructed to distinguish MNIST 0 s from 1 s. The RNN is presented with each image as an unrolled sequence of length 784. The gradient-based methods GD (gray) and Adam (blue) cannot train; aMC (green) trains to an accuracy of 99.8%. (c) Size of the gradient (blue) associated with the models produced by aMC. The algorithm trains productively in regimes in which $|\nabla U| \ll U$ and $|\nabla U| \gg U$. The loss (green) is reproduced from panel (a). Learning rate for GD and Adam: $\alpha = 10^{-3}$; aMC hyperparameters: $\sigma_0 = 10^{-3}, n_s = 100, \epsilon = 0$.

regime, and when initiated from the large-gradient regime that is encountered by aMC, in which $|\nabla U| \gg U$, the gradient-based integrators explode. Thus MC can train productively in the face of two of the classic obstacles to training by gradient-based methods, small and large gradients. Note also that the memory cost of doing GD with an RNN scales linearly with the sequence length, while for aMC the sequence length has no effect on memory cost.

Simple RNNs are known be harder to train by GD than more complex RNNs, but simple RNNs possess similar or greater *capacity* per parameter than more complex architectures [55]. Methods that can train simple RNNs may therefore allow more widespread use of those architectures.

2.5. For nets with heterogeneous structures or neuron activations, the weight-update scale must be made heterogenous

For some architectures, particularly those with structural heterogeneity or heterogenous neuron activations, it is necessary to scale the MC step-size parameter σ for each neural-net parameter individually. In this section we address this problem using deep neural networks for which, as in section 2.4, gradients are too small for gradient-based methods to train.

Choosing a set of heterogeneous MC step-size parameters can be done by adapting ideas used in the development of gradient-based methods [56]. Guided by that work we modify the proposal distribution of (4) to read

$$\epsilon_i \sim \mathcal{N}(\mu_i, \sigma_i^2), \tag{8}$$

where $\sigma_i = \lambda_i \sigma$. The λ_i are parameters that are either set to unity (a condition we call 'signal norm off') or according to equation (18) in section 3.2 ('signal norm on'). The parameters μ_i and σ are adjusted as in section 2.3. The parameters λ_i, which are straightforwardly calculated during a forward pass through the net, ensure that the scale of signal change to each neuron is roughly constant. The intent of signal norm is similar to that of layer norm [57], except that the latter is an architectural solution—it entails a modification of the net, and is present at test time—while the former is an algorithmic solution and plays no role once the neural net has been trained.

In figure 6 we show the results of neural networks of depth d trained by aMC and by gradient-based methods to express a step function $f_0(\theta)$ that is equal to 1/2 if $1/2 < \theta < 3/4$ and is zero otherwise. The neural nets have one input neuron, which is fed the value θ, and one output neuron, which returns $f_{\boldsymbol{x}}(\theta)$. They have 10 neurons in the penultimate hidden layer, and 4 neurons in each of the other $d-1$ hidden layers, the intent being to allow very deep nets with relatively few neurons. All neurons have tanh activation functions.

In figure 6(a) we show loss U as a function of epoch n for four algorithms: GD (gray); Adam (blue); and aMC with signal norm off (green dotted) and on (green). For each algorithm we ran 20 independent simulations, 10 using Kaiming initialization [58] and 10 initialized with Gaussian random numbers $x_i \sim \mathcal{N}(0, \sigma_0^2)$, where $\sigma_0 = 10^{-2}$. We plot the simulation having the smallest U after 10^6 epochs. As the depth of the network increases beyond four layers, GD and aMC with signal norm off stop learning on the timescales shown. Above 32 layers, Adam also stops learning on the timescale shown. (For depth 64 we tried a broad range of learning rates for GD and Adam, from 10 to 10^{-6}, none of which was successful. We also

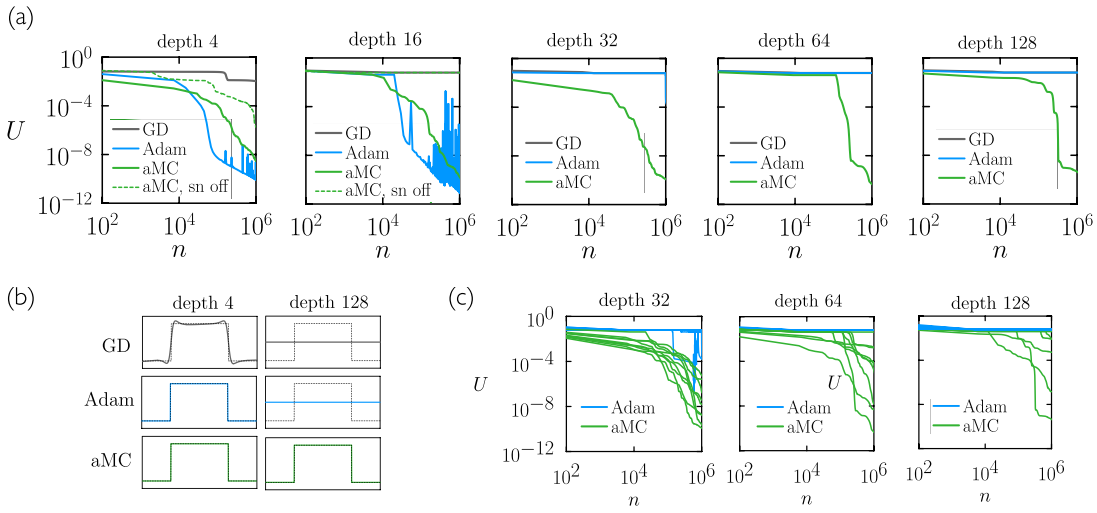

Figure 6. Intrinsic stochasticity and heterogenous weight-scale updates allow aMC to train deep nets in which the gradient is too small for gradient-based methods to work. (a) Loss function U versus epoch n for deep nets trained to express a step function by GD (gray), Adam (blue), and aMC with signal norm off (green dashed) and on (green). The latter is able to train for all depths shown. (b) Net outputs after 10^6 epochs (colored) against the target function (black) for two depths. (c) Loss U of 10 independent simulations for each of Adam and aMC for the deeper nets. GD learning rate: $\alpha = 10^{-3}$; Adam hyperparameters: $(\alpha, \beta_1, \beta_2) = (10^{-4}, 0.9, 0.999)$; aMC hyperparameters: $(\sigma_0, \epsilon, n_s) = (10^{-2}, 10^{-2}, 10^2)$.

varied the Adam hyperparameters β_1, β_2 over a small range of values, without success. It may be that hyperparameters that enable training do exist, but we were not able to find them.) aMC with signal norm on continues to learn up to a depth of 128 (we also verified that aMC trains nets of depth 256), and so can successfully train deep nets in which gradient-based algorithms receive too little signal to train.

In figure 6(b) we show net outputs at $n = 10^6$ epochs for three algorithms and two depths. As discussed in section 2.3, the adaptive algorithms Adam and aMC learn the sharp features of the target function more quickly than does GD. For the deeper net, the gradient-based algorithms GD and Adam do not receive sufficient signal to train.

In figure 6(c) we show 10 simulations for each of Adam and aMC for the deeper nets. The outcome of training is stochastic, resembling a nucleation dynamics with an induction time that increases with net depth. For some initial conditions both algorithms fail to train on the allotted timescale. In general, the rate of nucleation is higher for aMC than Adam, and remains measurable for all depths shown

We verified that introducing skip connections [59] or layer norm [57] to the neural nets enabled Adam to train at depth 64 (and enables aMC without signal norm to train at all depths). These architectural modifications allow gradient-based algorithms to train, but they are not required for MC. This comparison illustrates the fact that design principles for nets trained by MC methods differ from those trained by gradient-based methods.

2.6. Best practices for training neural nets using MC await development

In this section we first revisit figure 1 using aMC, and we present data indicating that numerical best practices for training nets using MC may differ from those developed for gradient-based algorithms.

In figure 7(a) we reproduce figure 1(b) with the addition of an aMC simulation (green) in which we use aMC's adaptive step-size attenuation (with $n_s = 20$) and signal norm. These features allow us to choose an initial step-size parameter $\sigma_0 = 10^{-2}$ larger than the optimum value for Metropolis MC (see figure 1(a)). As a result, training proceeds faster than for MC, at a rate comparable to the GD result shown. Training-set loss and test-set accuracy at long times are similar for all three algorithms.

In figure 7(b) we compare the aMC result of panel (a), which uses batch learning (green), with two additional aMC results. The first (cyan) uses conventional minibatching with a minibatch size of 2000. Minibatch learning proceeds faster than batch learning, as happens with gradient-based methods, but achieves slightly larger values of training-set loss and test-set error than does batch learning. We speculate that this happens because of competing sources of stochasticity, that of the minibatch and that intrinsic to the MC algorithm. The second (cyan) uses progressive batching: training begins with a minibatch of size 500, which doubles every time the classification error rate on the minibatch falls below 10% (aMC moves are conditioned against minibatch loss, in the usual way, not minibatch classification error, but the latter is the trigger for the doubling of the minibatch size). After doubling the minibatch size the aMC algorithm is reset

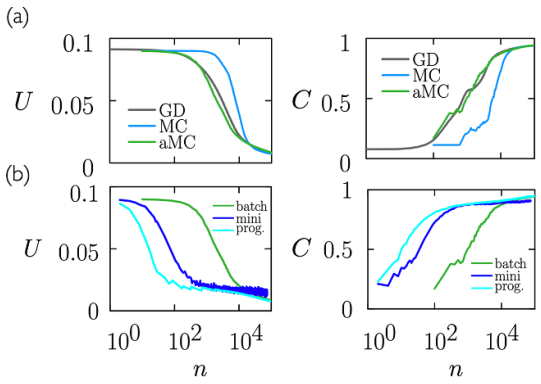

Figure 7. As for gradient-based methods, different forms of batching can speed neural-net training by MC. (a) As figure 1(b), with the addition of a simulation done using aMC (green) with hyperparameters $(\sigma_0, \epsilon, n_s, \text{s.n.}) = (10^{-2}, 0, 20, \text{on})$. (b) The aMC result from panel (a), which uses batch learning, compared with aMC using conventional minibatching (blue) and progressive batching (cyan).

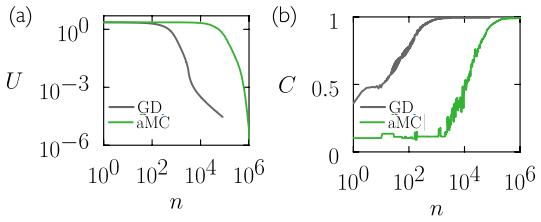

Figure 8. Training-set loss U (a) and test-set classification accuracy C (b) for MNIST classification done using ResNET-18, a deep neural network of $\approx 1.11 \times 10^7$ parameters. GD learning rate was 10^{-3}; aMC hyperparameters were $\sigma_0 = 10^{-3}$, $n_s = 100$, and $\epsilon = 10^{-3}$, with signal norm off.

($\sigma \to \sigma_0$ and $\mu_i \to 0$). Progressive-batch learning proceeds faster than batch learning, and reaches similar final values of training-set loss and test-set accuracy. This comparison suggests that MC may respond differently than GD to procedures such as minibatch training; best practices for MC training of neural networks await development.

We note that the noise intrinsic to the MC method provides a means of exploration even when the batch identity is kept fixed (noise also provides a way to effect change in the presence of vanishing gradients; see sections 2.5 and 2.4). For the problem discussed in this section there is a variation of about 1% in values of test-set accuracy at 10^5 epochs for 30 independent MC trajectories propagated with the same set of hyperparameters. Such fluctuations could provide the basis for an additional form of importance sampling that identified and propagated the best-performing networks in a population.

Finally, we show in figure 8 the analog of figure 1(b) for ResNET-18. ResNET-18 is a large, deep neural network with $\approx 1.11 \times 10^7$ parameters [59] (we changed the number of input channels for the first layer in order to apply it to MNIST). We applied layer normalization [57], which homogenizes neuron inputs and removes (or reduces) the need for signal norm when using aMC. The GD learning rate was set to 10^{-3} and we applied gradient clipping for $|\nabla U| > 1$; the aMC hyperparameters were $\sigma_0 = 10^{-3}$, $n_s = 100$, and $\epsilon = 10^{-3}$. Two points are apparent from the plot: GD trains faster than aMC, but aMC has similar ability to train in the long-time limit: both GD and aMC train to small loss and about 99.2% accuracy on the MNIST test set. Thus for large modern architectures such as ResNET-18 we find training with gradients faster than training by MC, but the latter has similar capacity for learning, suggesting that it is a promising tool for training neural networks when gradients are unreliable: see sections 2.4 and 2.5.

3. Summary of aMC

3.1. An adaptive version of the Metropolis algorithm for training neural networks

In this section we summarize the aMC algorithm used in this paper. It is based on the Metropolis MC algorithm, modified to allow the move-proposal distribution to adapt in response to accepted and rejected moves. The Metropolis acceptance criterion is $\min(1, e^{-\Delta U/T})$, where ΔU is the change of loss and T is temperature. For nonzero temperature the algorithm allows moves uphill in loss. We focus here on the limit of zero temperature, which allows no uphill moves in loss. This choice is motivated by the success of

Table 1. Clock times τ (in seconds) for 10^3 epochs of batch learning using a deep neural net of width 64 applied to MNIST, for GD, Adam, and aMC.

# hidden layers	$\tau_{\rm GD}$	$\tau_{\rm Adam}/\tau_{\rm GD}$	$\tau_{\rm aMC}/\tau_{\rm GD}$
2	8.5	1.01	0.56
4	10.6	1.01	0.54
8	14.9	1.01	0.51

gradient-descent algorithms and the intuition in deep learning (suggested by the structure of high-dimensional Gaussian random surfaces) that at large loss most stationary points on the loss surface are saddle points that can be escaped by moving downhill [31, 32].

aMC is specified by four hyperparameters: σ_0, the initial move scale; ϵ, the rate at which the mean of the move-proposal distribution is modified; $n_{\rm s}$, the number of consecutive rejected moves allowed before rescaling the parameters of the move-proposal distribution; and by the choice of signal norm being on or off.

We introduce a counter $n_{\rm cr} = 0$ to record the number of consecutive rejected moves. We initialize the parameters (weights and biases) $\boldsymbol{x} = \{x_1, \ldots, x_i, \ldots, x_N\}$ of the neural network (e.g. using Gaussian random numbers $x_i \sim \mathcal{N}(0, \sigma_0^2)$), and set the centers μ_i of each parameter's move-proposal distribution to zero. aMC proceeds as follows:

(a) *Current state.* Record the current neural-network parameter set $\boldsymbol{x}$. Select the data (defining the batch, episode, etc) and record the current value of the loss $U(\boldsymbol{x})$ on the data (for batch learning the value $U(\boldsymbol{x})$ is known from the previous step of the algorithm). If signal norm is on, calculate the values λ_i specified by equation (18), the required quantities having been calculated in the course of calculating $U(\boldsymbol{x})$.

(b) *Proposed move.* Propose a change

$$x_i \to x_i' = x_i + \epsilon_i \quad \text{with} \quad \epsilon_i \sim \mathcal{N}(\mu_i, \sigma_i^2), \tag{9}$$

of each neural-network parameter i, where $\sigma_i = \lambda_i \sigma$. Initially, $\mu_i = 0$ and $\sigma = \sigma_0$, where σ_0 is the initial move scale. The parameters λ_i are set either to unity ('signal norm off') or by equation (18) ('signal norm on'). Evaluate the loss $U(\boldsymbol{x}')$ at the set of coordinates $\boldsymbol{x}'$ resulting from the proposal (9). If $U(\boldsymbol{x}') \leqslant U(\boldsymbol{x})$ [60] then we accept the move and go to Step (c). Otherwise we reject the move and go to Step (d).

(c) *Accept move.* Make the proposed coordinates $\boldsymbol{x}'$ the current coordinates $\boldsymbol{x}$. Set $n_{\rm cr} = 0$. For each neural-network parameter i, set

$$\mu_i \to \mu_i + \epsilon(\epsilon_i - \mu_i), \tag{10}$$

using the values ϵ_i calculated in (9). Return to Step (a).

(d) *Reject move.* Retain the set of coordinates $\boldsymbol{x}$ recorded in Step (a). Set $n_{\rm cr} \to n_{\rm cr} + 1$. If $n_{\rm cr} = n_{\rm s}$ then set $n_{\rm cr} = 0$, $\sigma \to 0.95\,\sigma$, and (for all i) $\mu_i = 0$. Return to Step (a).

We refer to this algorithm as aMC, for adaptive Monte Carlo (the term 'adaptive Metropolis algorithm' has been used in a different context [61]). Standard zero-temperature Metropolis MC is recovered in the limit $\epsilon = 0, n_{\rm s} = \infty$, and $\lambda_i = 1$. Note that the algorithm requires calculation of the loss $U(\boldsymbol{x})$ only, and not of gradients of the loss with respect to the net parameters.

The computational cost of one aMC move is the cost to draw N Gaussian random numbers and to calculate the loss function twice (minibatch learning) or once (batch learning). Clock times per 10^3 epochs of batch learning using a deep neural net of width 64 applied to MNIST are shown in table 1 for GD, Adam, and aMC. Calculations were run on an NVIDIA Tesla T4 GPU.

The memory cost of aMC is the cost to calculate the loss, to store two versions of the model, and (if $\epsilon \neq 0$ and signal norm is on) to store the values μ_i and λ_i for each neural-net parameter.

3.2. Signal norm: enacting heterogenous weight updates in order to keep roughly constant the change of neuron inputs

The proposal step (9) contains the parameter step size $\sigma_i = \lambda_i \sigma$. For some applications, particularly involving deep or heterogeneous networks, it is useful to choose the λ_i in order to keep the scale of updates for each neuron approximately equal, following ideas applied to gradient-based methods [56]. We call this concept *signal norm*; when signal norm is off, all $\lambda_i = 1$. When it is on, we proceed as follows.

Consider the class of neural networks for which the input to neuron j (its pre-activation) is

$$I_j^\alpha = \sum_{i\to j}^{N_j} x_i S_i^\alpha, \tag{11}$$

where the sum runs over all weights x_i feeding into neuron j; N_j is the fan-in of j (the number of connections entering j); and S_i^α is the output of neuron i (the neuron that the weight x_i connects to neuron j) given one particular evaluation α of the neural network. Under the proposal (9) the change of input to neuron j is approximately [62]

$$\Delta_j^\alpha = \sum_{i\to j}^{N_j} \epsilon_i S_i^\alpha. \tag{12}$$

We therefore have

$$\langle \Delta_j^\alpha \rangle = \sum_{i\to j}^{N_j} \mu_i S_i^\alpha, \tag{13}$$

and

$$\langle (\Delta_j^\alpha)^2 \rangle = \sum_{i\to j}^{N_j}\sum_{k\to j}^{N_j} \left[\mu_i\mu_k(1-\delta_{ik}) + (\sigma_i^2+\mu_i^2)\delta_{ik}\right] S_i^\alpha S_k^\alpha, \tag{14}$$

where $\langle\cdot\rangle$ is the expectation over the move-proposal distribution (9), and δ_{ik} is the Kronecker delta. The expected approximate variance of the change of input to neuron j under the move (9) is therefore

$$\langle (\Delta_j^\alpha)^2 \rangle - \langle \Delta_j^\alpha \rangle^2 = \sigma^2 \sum_{i\to j}^{N_j} \lambda_i^2 (S_i^\alpha)^2. \tag{15}$$

This quantity, averaged over all $N_{\rm data}$ neural-net calls required to calculate the loss, is

$$[\langle (\Delta_j^\alpha)^2 \rangle - \langle \Delta_j^\alpha \rangle^2]_{\rm data} = \sigma^2 N_{\rm data}^{-1} \sum_{\alpha=1}^{N_{\rm data}} \sum_{i\to j}^{N_j} \lambda_i^2 (S_i^\alpha)^2. \tag{16}$$

We can choose the values of the λ_i in order to ensure that the right-hand side of (16) is always σ^2. A simple way to do so is to set equal the λ_i for all weights x_i feeding neuron j, in which case

$$\lambda_i = \left(N_{\rm data}^{-1} \sum_{\alpha=1}^{N_{\rm data}} \sum_{i'\to j}^{N_j} (S_{i'}^\alpha)^2 \right)^{-1/2}. \tag{17}$$

If all neuron outputs $S_{i'}^\alpha$ appearing in (17) vanish identically then the expression must be regularized; one option is to set $\lambda_i = 0$ for weights feeding a neuron whose input neurons are zero for a given pass through the data. Recall that the sum α runs over the input data; the sum $i' \to j$ runs over all neurons i' whose connections feed j; N_j is the fan-in of j (the number of connections entering j); and $S_{i'}^\alpha$ is the output of neuron i' given a particular evaluation α of the neural network. The values (17) can be calculated from the pass through the data immediately before the proposed move.

Under (17), weights on connections that feed into a neuron receiving many other connections will experience a smaller basic move scale than weights on connections that feed into a neuron receiving few connections. Similarly, weights on connections fed by active neurons will experience a smaller basic move scale than weights on connections fed by relatively inactive neurons.

Finally, if the parameter x_i is a bias we choose $\lambda_i = 1$.

To summarize, we consider two settings for the parameters λ_i that set the step-size parameters $\sigma_i = \lambda_i\sigma$ in the proposal (9). The first setting, 'signal norm off' (used in figures 1, 2(a), (b), 4, and 5), has $\lambda_i = 1$ for all parameters x_i.

The second setting, called 'signal norm on' (used in figures 2(c), (d), 6, and 7), has

$$\lambda_i = \begin{cases} 1 & \text{if } x_i \text{ is a bias;} \\ \text{Equation(17)} & \text{if } x_i \text{ is a weight into neuron } j. \end{cases} \tag{18}$$

4. Conclusions

We have examined the Metropolis MC algorithm as a tool for training neural networks, and have introduced aMC, an adaptive variant of it. MC methods are closely related to evolutionary algorithms, which are used to train neural networks [11–13, 19], but the latter are usually applied to populations of neural networks; the MC algorithms we have considered here are applied to populations of size 1, just as GD is. For sufficiently small moves the Metropolis algorithm is effectively GD in the presence of white noise [17]. Thus on theoretical grounds the Metropolis algorithm should possess the ability to train a neural network to values of a loss function similar to those achieved by GD; this is indeed what we (and others [6–8]) have observed empirically, both for simple neural nets and for large, modern architectures. This correspondence does not guarantee similar training times, however, and we have found gradient-based methods to be faster in general, particularly for large and heterogenous neural nets.

aMC is an adaptive version of the Metropolis algorithm. The efficiency of aMC diminishes less quickly with decreasing loss and increasing net size than does the efficiency of the Metropolis algorithm, and aMC can train faster than Metropolis, much as adaptive gradient-based methods can train faster than pure GD.

The Metropolis algorithm and aMC offer a complement to gradient-based methods in that they can sense the gradient when it exists but can work without it. In particular, aMC can train nets in which the gradient is too small (or too large) to allow gradient-based methods to train on the timescales simulated. We have shown here that aMC can train deep neural networks and recurrent neural networks that GD cannot train. In both cases there exist modifications to those networks that can be trained by gradient-based methods, but aMC does not require those modifications. The design principles of neural nets optimal for MC algorithms are largely unexplored but are likely distinct from those optimal for gradient-based methods, and having both sets of algorithms (together with hybrids of them [63]) offers more choices for net design than having only one.

Finally, we note that while Metropolis and aMC have a fundamental connection to gradient-based methods in the limit of small step size, MC algorithms more generally can enact large-scale nonlocal or collective changes that cannot be made by integrating gradient-based equations of motion [5, 64–68]. The analogy suggests that improved MC algorithms for training neural networks await development.

Data availability statement

The data that support the findings of this study are openly available at the following URL/DOI: https://github.com/reproducible-science/aMC.

Code availability

Calculations using gradient descent and Adam were done in PyTorch [58]. Monte Carlo calculations were done in C and in PyTorch. A PyTorch implementation of the aMC optimizer (with signal norm off) and an example RNN optimization are available from [69].

Acknowledgments

This work was performed as part of a user project at the Molecular Foundry, Lawrence Berkeley National Laboratory, supported by the Office of Science, Office of Basic Energy Sciences, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This work used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231. I T acknowledges funding from the National Science and Engineering Council of Canada. C C acknowledges a mobility grant from Research Foundation—Flanders (FWO).

ORCID iD

Isaac Tamblyn https://orcid.org/0000-0002-8146-6667

References

[1] Metropolis N, Rosenbluth A W, Rosenbluth M N, Teller A H and Teller E 1953 Equation of state calculations by fast computing machines *J. Chem. Phys.* **21** 1087–92
[2] Gubernatis J E 2005 Marshall rosenbluth and the metropolis algorithm *Phys. Plasmas* **12** 057303
[3] Rosenbluth M N 2003 Genesis of the Monte Carlo algorithm for statistical mechanics *AIP Conf. Proc.* **690** 22–30

[4] Whitacre M H 2021 Arianna wright rosenbluth (Los Alamos National Lab (LANL): Los Alamos, NM) *Technical Report*
[5] Frenkel D and Smit B 2001 *Understanding Molecular Simulation: From Algorithms to Applications* vol 1 (New York: Academic)
[6] Sexton R S, Dorsey R E and Johnson J D 1999 Beyond backpropagation: using simulated annealing for training neural networks *J. Organ. End User Comput.* **11** 3–10
[7] Rere L R, Fanany M I and Arymurthy A M 2015 Simulated annealing algorithm for deep learning *Proc. Comput. Sci.* **72** 137–44
[8] Tripathi R and Singh B 2020 Rso: a gradient free sampling based approach for training deep neural networks (arXiv:2005.05955)
[9] Schmidhuber J 2015 Deep learning in neural networks: an overview *Neural Netw.* **61** 85–117
[10] Goodfellow I, Bengio Y and Courville A 2016 *Deep Learning* (Cambridge, MA: MIT Press)
[11] Holland J H 1992 Genetic algorithms *Sci. Am.* **267** 66–73
[12] Fogel D B and Stayton L C 1994 On the effectiveness of crossover in simulated evolutionary optimization *Biosystems* **32** 171–82
[13] Montana D J and Davis L 1989 Training feedforward neural networks using genetic algorithms *IJCAI* **89** 762–7
[14] Zero temperature means that moves that increase the loss are not accepted. This choice is motivated by the empirical success in machine learning of gradient-descent methods, and by the intuition, derived from Gaussian random surfaces, that loss surfaces possess more downhill directions at large values of the loss [31, 32]
[15] Kikuchi K, Yoshida M, Maekawa T and Watanabe H 1991 Metropolis Monte Carlo method as a numerical technique to solve the Fokker–Planck equation *Chem. Phys. Lett.* **185** 335–8
[16] Kikuchi K, Yoshida M, Maekawa T and Watanabe H 1992 Metropolis Monte Carlo method for Brownian dynamics simulation generalized to include hydrodynamic interactions *Chem. Phys. Lett.* **196** 57–61
[17] Whitelam S, Selin V, Park S-W and Tamblyn I 2021 Correspondence between neuroevolution and gradient descent *Nat. Commun.* **12** 1–10
[18] Note that algorithms of this nature do not constitute random search. The proposal step is random (related conceptually to the idea of weight guessing, a method used in the presence of vanishing gradients [42]) but the acceptance criterion is a form of importance sampling, and leads to a dynamics equivalent to noisy gradient descent
[19] Salimans T, Ho J, Chen X, Sidor S and Sutskever I 2017 Evolution strategies as a scalable alternative to reinforcement learning (arXiv:1703.03864)
[20] Kingma D P and Ba J 2014 Adam: a method for stochastic optimization (arXiv:1412.6980)
[21] LeCun Y, Bengio Y and Hinton G 2015 Deep learning *Nature* **521** 436–44
[22] Metz L, Freeman C D, Schoenholz S S and Kachman T 2021 Gradients are not all you need (arXiv:2111.05803)
[23] Mitchell M, Holland J and Forrest S 1993 When will a genetic algorithm outperform hill climbing? *Adv. Neural Inf. Process. Syst.* **6**
[24] Mitchell M 1998 *An Introduction to Genetic Algorithms* (Cambridge, MA: MIT Press)
[25] In Metropolis Monte Carlo simulations of molecular systems it is usual to propose moves of one particle at a time. If we consider neural-net parameters to be akin to particle coordinates then the analog would be to make changes to one neural-net parameter at a time; see e.g. [8]. However, there is no formal mapping between particles and a neural network, and we could equally well consider the neural-net parameters to be akin to the coordinates of a single particle, in a high-dimensional space, in an external potential equal to the loss function. In the latter case the analog would be to propose a change of all neural-net parameters simultaneously, as we do here
[26] LeCun Y, Bottou L, Bengio Y and Haffner P 1998 Gradient-based learning applied to document recognition *Proc. IEEE* **86** 2278–323
[27] LeCun Y *et al* 1998 Gradient-based learning applied to document recognition *Proc. IEEE* **86** 2278–324
[28] 3Blue1Brown 2017 But what is a neural network? | Chapter 1, Deep learning (available at: https://www.youtube.com/watch?v=aircAruvnKk) (Accessed 2022)
[29] We have also found the GD-MC equivalence to break down in other circumstances: for certain learning rates α, the discrete-update equation (3) sometimes results in moves uphill in loss, in which case the discrete update is not equivalent to the equation $\dot{\boldsymbol{x}} = -(\alpha/\Delta t)\nabla U(\boldsymbol{x})$, while the latter *is* equivalent to the small-step-size limit of the finite-temperature Metropolis algorithm [15–17]
[30] In figure 8 we show that GD and MC can both train a large modern neural network to a classification accuracy in excess of 99% on the same problem
[31] Dauphin Y N, Pascanu R, Gulcehre C, Cho K, Ganguli S and Bengio Y 2014 Identifying and attacking the saddle point problem in high-dimensional non-convex optimization *Adv. Neural Inf. Process. Syst.* **27**
[32] Bahri Y, Kadmon J, Pennington J, Schoenholz S S, Sohl-Dickstein J and Ganguli S 2020 Statistical mechanics of deep learning *Annu. Rev. Condens. Matter Phys.* **11** 501–28
[33] Hansen N and Ostermeier A 2001 Completely derandomized self-adaptation in evolution strategies *Evol. Comput.* **9** 159–95
[34] Hansen N, Müller S D and Koumoutsakos P 2003 Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (cma-es) *Evol. Comput.* **11** 1–18
[35] Hansen N 2006 The CMA evolution strategy: a comparing review *Towards a New Evolutionary Computation* (Heidelberg: Springer) pp 75–102
[36] Rosenbrock H 1960 An automatic method for finding the greatest or least value of a function *Comput. J.* **3** 175–84
[37] Shang Y-W and Qiu Y-H 2006 A note on the extended Rosenbrock function *Evol. Comput.* **14** 119–26
[38] Emiola I and Adem R 2021 Comparison of minimization methods for Rosenbrock functions *2021 29th Mediterranean Conf. on Control and Automation (MED)* (IEEE) pp 837–42
[39] Goh G 2017 Why momentum really works *Distill* **2** e6
[40] Rumelhart D E, Durbin R, Golden R and Chauvin Y 1995 Backpropagation: the basic theory *Backpropagation: Theory, Architectures and Applications* (London: Psychology Press) pp 1–34
[41] Chen J, Zhou D, Tang Y, Yang Z, Cao Y and Gu Q 2018 Closing the generalization gap of adaptive gradient methods in training deep neural networks (arXiv:1806.06763)
[42] Hochreiter S and Schmidhuber J 1997 Long short-term memory *Neural Comput.* **9** 1735–80
[43] Hochreiter S 1998 The vanishing gradient problem during learning recurrent neural nets and problem solutions *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.* **6** 107–16
[44] Medsker L R and Jain L 2001 Recurrent neural networks *Des. Appl.* **5** 64–67
[45] Graves A, Mohamed A-r and Hinton G 2013 Speech recognition with deep recurrent neural networks *2013 IEEE Int. Conf. on Acoustics, Speech and Signal Processing* (IEEE) pp 6645–9
[46] Sutskever I, Vinyals O and Le Q V 2014 Sequence to sequence learning with neural networks *Adv. Neural Inf. Process. Syst.* **27**
[47] Graves A and Schmidhuber J 2008 Offline handwriting recognition with multidimensional recurrent neural networks *Adv. Neural Inf. Process. Syst.* **21**

[48] Wierstra D, Förster A, Peters J and Schmidhuber J 2010 Recurrent policy gradients *Logic J. IGPL* **18** 620–34
[49] Bengio Y, Simard P and Frasconi P 1994 Learning long-term dependencies with gradient descent is difficult *IEEE Trans. Neural Netw.* **5** 157–66
[50] Martens J and Sutskever I 2011 Learning recurrent neural networks with hessian-free optimization *Int. Conf. on Machine Learning* vol 28
[51] Bengio Y, Boulanger-Lewandowski N and Pascanu R 2013 Advances in optimizing recurrent networks *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (IEEE) pp 8624–8
[52] Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H and Bengio Y 2014 Learning phrase representations using RNN encoder–decoder for statistical machine translation (arXiv:1406.1078)
[53] Kanai S, Fujiwara Y and Iwamura S 2017 Preventing gradient explosions in gated recurrent units *Adv. Neural Inf. Process. Syst.* **30**
[54] Pascanu R, Mikolov T and Bengio Y 2013 On the difficulty of training recurrent neural networks *Int. Conf. on Machine Learning* (PMLR) pp 1310–8
[55] Collins J, Sohl-Dickstein J and Sussillo D 2016 Capacity and trainability in recurrent neural networks (arXiv:1611.09913)
[56] LeCun Y, Bottou L, Orr G B and Müller K-R 1996 Efficient backprop *Neural Networks: Tricks of the Trade* (Heidelberg: Springer) pp 9–50
[57] Ba J L, Kiros J R and Hinton G E 2016 Layer normalization (arXiv:1607.06450)
[58] Paszke A *et al* 2019 Pytorch: an imperative style, high-performance deep learning library *Adv. Neural Inf. Process. Syst* **32**
[59] He K, Zhang X, Ren S and Sun J 2016 Deep residual learning for image recognition *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition* pp 770–8
[60] For finite temperature T the move is accepted if $\xi < \mathrm{e}^{(U(\boldsymbol{x})-U(\boldsymbol{x}'))/T}$, where ξ is a random number drawn uniformly on $(0, 1]$
[61] Rosenthal J S *et al* 2011 Optimal proposal distributions and adaptive MCMC *Handbook of Markov Chain Monte Carlo* vol 4 (London: Chapman & Hall)
[62] This approximation assumes that the output neurons do not change under the move. This is not true, but the intent here is to set the basic move scale, and absolute precision is not necessary
[63] Leen T K and Moody J E 1997 Stochastic manhattan learning: Time-evolution operator for the ensemble dynamics *Phys. Rev.* E **56** 1262
[64] Swendsen R H and Wang J-S 1987 Nonuniversal critical dynamics in Monte Carlo simulations *Phys. Rev. Lett.* **58** 86
[65] Wolff U 1989 Collective Monte Carlo updating for spin systems *Phys. Rev. Lett.* **62** 361
[66] Chen B and Siepmann J I 2001 Improving the efficiency of the aggregation-volume-bias Monte Carlo algorithm *J. Phys. Chem.* B **105** 11275–82
[67] Liu J and Luijten E 2004 Rejection-free geometric cluster algorithm for complex fluids *Phys. Rev. Lett.* **92** 035504
[68] Whitelam S and Geissler P L 2007 Avoiding unphysical kinetic traps in Monte Carlo simulations of strongly attractive particles *J. Chem. Phys.* **127** 154101
[69] Whitelam S, Selin V, Benlolo I, Casert C and Tamblyn I 2022 *Adaptive MC* (available at: https://github.com/reproducible-science/aMC) (Accessed 2022)