An Overview of Stochastic Gradient Descent in Machine Learning *

Mason del Rosario
University of California, Davis
mdelrosa@ucdavis.edu

June 11, 2019

Abstract

With the proliferation of available data in recent decades, online approaches to optimization need to be designed with computational complexity in mind. For deep learning applications, optimization algorithms must strike a balance between minimizing noise, accounting for non-convexity, and dealing with ill-conditioning while keeping the compute time tractable. Stochastic gradient descent and extensions thereof have proven effective in addressing these issues, and modern deep learning libraries, such as Tensorflow and Keras, come with built-in optimizers based on stochastic gradient descent. This study presents the foundations for stochastic gradient descent, describes variations, such as minibatch and quasi-Newton methods, which have the potential to deal with issues of computational complexity and non-convexity, and describes optimizers, including Nesterov accelerated gradient (NAG) and Adam, popular in the machine learning community. Finally, additional avenues of research into SGD are discussed.

Keywords: Stochastic gradient descent, machine learning, optimizers

*This paper was written to fulfill the term paper requirement for the Fall 2019 offering of the graduate course, *EEC263 Optimal and Adaptive Filtering*, taught by Professor Bernard Levy.

Contents

1 Introduction **3**

2 Optimization Problems **3**

3 Stochastic Gradient and Batch Gradient **4**
3.1 Trade-offs 4

4 Extensions/Variations **5**
4.1 Noise-reduction methods 6
4.1.1 Dynamic sampling 6
4.1.2 Gradient aggregation 6
4.1.3 Iterate averaging 7
4.2 Second-order methods 7
4.2.1 Newton's method 7
4.2.2 Inexact Newton methods 7
4.2.3 Quasi-Newton methods 8
4.2.4 Gauss-Newton methods 8
4.2.5 Diagonal scaling methods 9

5 Optimizers **9**
5.1 Momentum 9
5.2 Acceleration 10
5.3 Adaptive optimizers 10
5.3.1 Adagrad 11
5.3.2 Adadelta/RMSProp 11
5.3.3 Adam 12
5.4 Comparison of Optimizers 12

6 Frontiers **13**
6.1 Distributed gradient descent 13
6.2 Batch normalization 13
6.3 Shuffling/curriculum learning 14

1 Introduction

Stochastic gradient descent is a method of numerical optimization which has found popularity in the fields of machine learning and neural networks [2]. Section 2 of this paper provides a framework for numerical optimization problems. Section 3 formulates stochastic and batch gradient descent and summarizes the trade-offs between them. Section 4 describes extensions to and variations on SGD which deal with noise, ill-conditioning, and non-convexity. Section 5 highlights popular optimizers for SGD methods which are frequently used in neural networks and deep learning. Section 6 concludes the paper and discusses open challenges and areas of investigation regarding SGD.

2 Optimization Problems

The goal in optimization is to minimize the loss incurred by a given parameterized system model. Generally speaking, a gradient algorithm seeks to find the optimum parameters for the model. The algorithm may begin with an initial parameterization, and updates to the parameters are made in the 'direction' that reduces the loss. The direction of descent is taken in the opposite direction of the loss function's *gradient* with respect to the parameters. If steps of proper magnitude are taken and if the function is convex, then the method will arrive at the loss function's *global minimum*, which correpsonds to an optimal parametrization.

Before introducing gradient methods formally, we consider the general form of an arbitrary loss function based on Bottou's notation [1]. To formulate numerical optimization problems, the key ingredients include:

- **Prediction Function**: The prediction function maps system inputs and function parameters to possible outputs. This function can be thought of as a system model which we seek to optimize such that we can produce accurate system outputs for arbitrary inputs. The set of prediction functions, $\mathcal{H}$, is written as

$$\mathcal{H} := \{h(\cdot; w) : w \in \mathbb{R}^d\} \tag{1}$$

  where $h(\cdot; w) : \mathbb{R}^{d_x} \times \mathbb{R}^d \to \mathbb{R}$ (assuming a non-variational prediction function).

- **Loss Function**: The loss function measures the discrepancy between system observations and the output of the prediction function. A general form for such a loss function is

$$\ell\left(h(\cdot; w), y\right) : \mathbb{R}^{d_y} \times \mathbb{R}^{d_y} \to \mathbb{R} \tag{2}$$

  such that the loss function compares a single prediction, $h(\cdot; w)$, to a single true output, y, and yields a scalar value in $\mathbb{R}$.

- **Risk Function**: The risk function denotes the loss incurred across all possible input and output combinations for a given parameterization. Analytically, this is equivalent to integrating $\ell(\cdot, \cdot)$ over $\mathbb{R}^{d_x} \times \mathbb{R}^{d_y}$,

$$R(w) = \int_{\mathbb{R}^{d_x} \times \mathbb{R}^{d_y}} \ell\left(h(x; w), y\right) dP(x, y) \tag{3}$$

  - **Expected Risk**: Assuming a closed-form, differentiable expression of $h(\cdot; w)$ is avaiable, then (3) evaluates to the expectation of the loss,

$$R(w) = \mathrm{E}\left[\ell\left(h(x; w), y\right)\right] dP(x, y) \tag{4}$$

  - **Empirical Risk**: In practice, the prediction function does not admit a readily differentiable closed-form solution, and the expectation cannot be taken. Instead, it is possible to evaluate the loss based on a finite number of samples, $\{(x_i, y_i)\}_i^{n=1} \subseteq \mathbb{R}^{d_x} \times \mathbb{R}^{d_y}$.

$$R_n(w) = \frac{1}{n} \sum_{i=1}^{n} \ell\left(h(x_n; w), y_n\right) \tag{5}$$

(5) is the object of interest in practical optimization problems. For compactness, a composite function for the i-th realiztaion of the ℓ and h, $f_i(w) := f(w; \xi_{[i]}) = \ell\left(h(x_n; w), y_n\right)$, for a sequence of random input-output realizations, $\xi_{[i]} := (x_i, y_i)$, can be used to rewrite (5) as follows:

$$R_n(w) = \frac{1}{n}\sum_{i=1}^{n} f_i(w) \tag{6}$$

3 Stochastic Gradient and Batch Gradient

With the optimization problem defined as the minimization of $R_n(w)$, we move on to a particular family of optimization methods. The canonical stochastic approach is known as the *stochastic gradient method* (SGD), with the form

$$w_{k+1} = w_k - \alpha_k \nabla f_{i_k}(w_k), \tag{7}$$

where the parameters, w_i, are updated by a single realization of the random process, $\{\xi_{[i]}\} = \{(x_{i_k}, y_{i_k})\}$, leading to a non-deterministic parameterization. By extension, $\{w_k\}$ is a random process determined by the index $\{i_k\}$. Individual realizations of the negative term, $-\nabla f_{i_k}(w_k)$, may not yield negative updates to w_k (i.e., they may not individually minimize R_n), but given a sufficient number of samples, SGD will minimize R_n in expectation.

While SGD is a stochastic method which updates w_k based on individual samples, *batch methods* use the average value of several individual gradients due to multiple realizations of $\xi_{[i]}$. For example, the *steepest descent algorithm* is written as

$$w_{k+1} = w_k - \frac{\alpha_k}{n}\sum_{i=1}^{n} \nabla f_{i_k}(w_k). \tag{8}$$

3.1 Trade-offs

Stochastic and batch optimization methods can be contrasted by their convergence rates. In the machine learning literature, the type of convergence which batch optimization exhibits is called *linear convergence*, which means that at sufficiently high values of k, the training error is bounded by

$$E_n = R_n(w_k) - R_n^* \leq \mathcal{O}(\rho^k) \tag{9}$$

with constant $\rho \in (0, 1)$. R_n^* is the empirical risk under optimal w_k, meaning E_n is the error in risk. In comparison, the error-bound in stochastic optimization is *sub-linear*, which is described by

$$E_n = R_n(w_k) - R_n^* \leq \mathcal{O}\left(\frac{1}{k}\right). \tag{10}$$

Thus batch optimization schemes might appear to assume lower error at convergence than stochastic schemes. However, consider the *big data* case (which is often encountered in the context of deep learning) where an

	Batch	Stochastic
$\mathcal{T}(n, \epsilon)$	$n \log\left(\frac{1}{\varepsilon}\right)$	$\frac{1}{\varepsilon}$
$\mathcal{E}^*$	$\frac{\log(\mathcal{T}_{\max})+1}{\mathcal{T}_{\max}}$	$\frac{1}{\mathcal{T}_{\max}}$

Table 1: Summary of compute time and minimized asymptotic error based on given compute time-limit, $\mathcal{T}$, batch size n, and error bound ϵ.

effectively infinite number of observations are available and a time constraint, $\mathcal{T}_{max}$, is placed on the iterative algorithm. Under the assumption of *strong convexity*, the compute times, τ, and minimal optimization/estimation error, $\mathcal{E}^*$, are summarized in Table 1.

We note that for batch methods, $\mathcal{T}$ becomes varies linearly with n such that in the case of infinite data, batch methods will take infinitely long. In contrast, stochastic methods merely depend on the acceptable error-bound, ϵ.

Thus stochastic gradient descent serves as a starting point for methods which can perform model optimization in a computationally tractable manner for deep learning, and the following sections discuss how we can alter classic SGD to account for its inherent limitations.

4 Extensions/Variations

Roughly speaking, there are two axes along which extensions to SGD can be categorized, noise-reduction methods and second-order methods (see Figure 1).

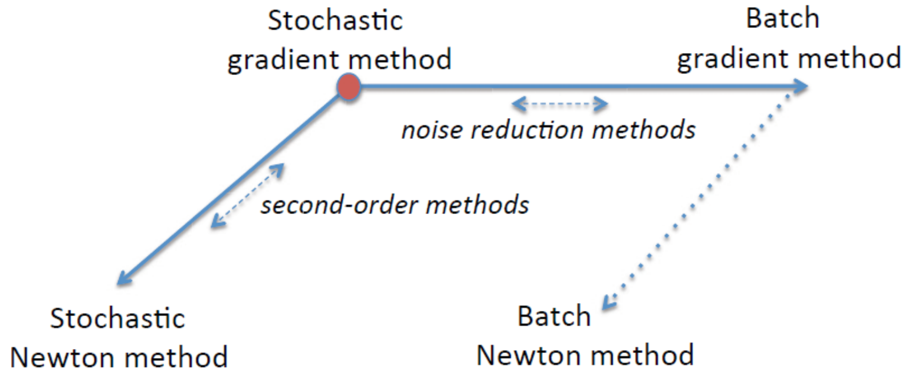

Figure 1: Illustration of stochastic gradient descent and the axes along which its variants can be categorized. Methods which incorporate more gradient samples per iteration are known as *noise reduction methods* while methods which use the Hessian or estimates of the Hessian are known as *second-order methods*.

4.1 Noise-reduction methods

As SGD involves random observations, $\xi_{[i]}$, noise is inherent in the parameter update iterations. This noise can prolong the time to convergence. Thus the goal of *noise-reduction methods* is to reduce the variance in either the gradient estimates or in the parameter iterates themselves. At least three categories of noise-reduction methods exist, including dynamic sampling, gradient aggregation, and iterate averaging.

4.1.1 Dynamic sampling

Dynamic sampling involves changing the number of random samples, ξ_i, to incorporate in the parameter update. First, we introduce the notion of the **mini-batch gradient**, which rather than evaluating the gradient at individual realizations of $\xi_{[i]}$, takes an average contribution from a subsect of all observations, $\mathcal{S}_i$.

$$w_{k+1} = w_k - \frac{\alpha_k}{|\mathcal{S}_k|} \sum_{i \in \mathcal{S}_k} \nabla f_i(w_k). \tag{11}$$

The size of the minibatch can be adjusted at different values of k to achieve gradient estimates with progressively lower variance. For example, one realization of dynamic sampling with fixed step size, $\bar{\alpha}$, is written as

$$w_{k+1} = w_k - \bar{\alpha} g(w_k, \xi_k) \tag{12}$$

$$g(w_k, \xi_k) := \frac{1}{n_k} \sum_{i \in \mathcal{S}_k} \nabla f(w_k; \xi_{k,i}) \tag{13}$$

where $n_k := |\mathcal{S}_k| = \lceil \tau^{k-1} \rceil$ for $\tau > 1$. Thus, the cardinality of sampled observations, $\mathcal{S}_k$, increases per-iteration. Minibatch methods have been shown to exhibit linear convergence, and while they are not widely used in ML applications, they could serve to combine the low computational complexity of stochastic gradient at low k while limiting the variance at high k [1].

4.1.2 Gradient aggregation

Gradient aggregation methods incorporate gradient estimates from previous iterations and update those estimates upon re-sampling. Variants of gradient aggregation include:

- **SVRG**: *Stochastic variance reduced gradient method* operates in between the iterates, w_k, by iteratively updating an intermediate unbiased parameter estimate, $\tilde{w}_k$.

  $$\tilde{w}_{j+1} = \tilde{w}_j - \alpha \tilde{g}_j \tag{14}$$

  $$\tilde{g}_j = \nabla f_{i_j}(\tilde{w}_j) - (\nabla f_{i_j}(w_k) - \nabla R_n(w_k)) \tag{15}$$

  where the batch gradient is defined as $\nabla R_n = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w_k)$ and $i_j \in \{1, \ldots, n\}$ is a random variable. After completion of the inner loop for $\tilde{w}_j$, the SG parameter w_k can be updated in one of three ways:

  1. Set $w_{k+1} = \tilde{w}_{n+1}$
  2. Set $w_{k+1} = \frac{1}{n}\sum_{j=1}^{n} \tilde{w}_{j+1}$
  3. Choose j uniformly from $\{1, \ldots, n\}$, set $w_{k+1} = \tilde{w}_{j+1}$

  As SVRG requires computation of the batch gradient at each iteration its outer SGD loop, its computational complexity is greater than that of vanilla SGD. In practice, SGD exhibits faster initial convergence than SVRG, but SVRG can achieve higher training accuracy.

- **SAGA**: The *stochastic average gradient* algorithm is written as

$$w_{k+1} = w_k - \alpha g_k \tag{16}$$

$$g_k = \nabla f_j(w_k) - \nabla f_j(w_{[j]}) + \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(w_{[i]}) \tag{17}$$

  where $i \in \{1, \dots, n\}$, $j \in \{1, \dots, n\}$ and $w_{[i]}$ represents the latest iterate at which ∇f_i was evaluated. The algorithm is initialized by computing and storing the sampled gradient values, $\nabla f_i(w_{[i]}) \leftarrow \nabla f_i(w_1)$, and they are updated at each iteration of k. Thus, the computational complexity of the batch gradient is high at initialization but relatively cheap during subsequent iterations as it requires n additions and a single multiplication.

In short, gradient aggregation methods are able to integrate previous gradient information to iteratively update an estimate of the batch gradient, and such methods can achieve better convergence time than SGD alone. However, the applicability of gradient aggregation in large-scale machine learning is still not certain, as its compute time scales linearly with the sample size, n.

4.1.3 Iterate averaging

Another method of noise reduction involves taking the average of *parameter iterates*

$$w_{k+1} = w_k - \alpha_k g(w_k, \xi_k) \tag{18}$$

$$\tilde{w}_{k+1} = \frac{1}{k+1}\sum_{j=1}^{k+1} w_j, \tag{19}$$

The performance of iteratve averaging is tightly linked to the selection of step sizes, α_k. Combined with a diminishing step size sequence, iterate averaging has been shown to possess asymptotic excess MSE of $\mathcal{O}(\frac{1}{k})$ for step sizes larger than a pure SGD method [1].

4.2 Second-order methods

Second-order methods, such as Newton methods, quasi-Newton methods, and Gauss-Newton methods, have the potential to deal with ill-conditioning and non-linearities in the objective function.

4.2.1 Newton's method

The based on full-gradient update,

$$w_{k+1} = w_k - \alpha_k B^2 \nabla F(w_k) \tag{20}$$

with $B = (\nabla^2 F(w_k))^{-1/2}$. This equates to a regularization by the Hessian at each iteration of the algorithm.

Due to matrix factorization, exact calculation of the Hessian is computationally expensive, so inexact methods which can leverage previously calculated values of $F(w_k)$ and $\nabla F(w_k)$ are appealing. This is the premise behind *conjugate-gradient (CG) methods* and motivates the *Newton-CG* method.

4.2.2 Inexact Newton methods

The same stochastic treatment the gradient can be applied to the Hessian. As before, we can define a set of realizations of the random variable $\mathcal{S}_k^H := \{\xi_{[i]}^H\} = \{(x_{i_k}, y_{i_k})\}$ where the samples $\mathcal{S}_k$ and $\mathcal{S}_k^H$ are conditionally uncorrelated. This sub-sampling allows for the stochastic Hessian

$$\nabla^2 F(w_k; \xi_k^H) = \frac{\alpha_k}{|\mathcal{S}_k^H|}\sum_{i \in \mathcal{S}_k^H} \nabla^2 F(w_k; \xi_{k,i}^H) \tag{21}$$

In selecting $|\mathcal{S}_k^H|$, the sample size should be large enough to provide adequate information about the local curvature of $F(w_k)$ while keeping the sample size small enough to reduce computational cost.

Fortunately, the iterations of SGD are less sensitive to noise from the Hessian they are to noise arising from the gradient, meaning that $|\mathcal{S}_k^H| \leq |\mathcal{S}_k|$ is a typical heuristic for sample size.

For practical machine learning applications, subsampling for Hessian approximation is sensible when $|\mathcal{S}_k|$ is sufficiently large such that a small $|\mathcal{S}_k^H|$ does not introduce an undue computational burden. If a stochastic gradient is used, then a sample size of $|\mathcal{S}_k^H| > |\mathcal{S}_k|$ may be necessary, negating the computational savings of the stochastic gradient altogether.

4.2.3 Quasi-Newton methods

Rather than explicitly calculating the Hessian at each iteration, one can approximate the Hessian based on samples of the gradient. The most prevalant Quasi-Newton method is called *BFGS*, named after its authors Broyden-Fletcher-Goldfarb-Shanno, which takes the form

$$w_{k+1} = w_k - \alpha_k H_k \nabla F(w_k) \tag{22}$$

Where H_k is an approximation of $(\nabla^2 F(w_k))^{-1}$ computed iteratively based on samples of the gradient. The Hessian approximation is based on the parameter update and gradient differences

$$s_{k+1} = w_{k+1} - w_k \tag{23}$$
$$v_{k+1} = \nabla F(w_{k+1}) - \nabla F(w_k) \tag{24}$$

yielding the update rule

$$H_{k+1} = \left(I - \frac{v_k s_k^T}{s_k^T v_k}\right) H_k \left(I - \frac{v_k s_k^T}{s_k^T v_k}\right) + \frac{s_k s_k^T}{s_k^T v_k}. \tag{25}$$

A stochastic variant, called limited-memory BFGS (L-BFGS), is written as

$$w_{k+1} = w_k - \alpha_k H_k \nabla g(w_k, \xi_k) \tag{26}$$

which depends on the random variable ξ_k.

4.2.4 Gauss-Newton methods

Rather than using the subsampled Hessian (21) in the parameter update, a Jacobian-based approximation can be used

$$G_{\mathcal{S}_k^H}(w_k; \xi_k^H) = \frac{\alpha_k}{|\mathcal{S}_k^H|} \sum_{i \in \mathcal{S}_k^H} J_h(w_k; \xi_{k,i}^H) H_\ell(w_k; \xi_{k,i}^H) J_h(w_k; \xi_{k,i}^H) \tag{27}$$

where $J_h(\cdot; \xi)$ is the Jacobian of $h(x_\xi; \cdot)$ w.r.t. w and where $H_\ell = \frac{\partial^2 \ell}{\partial h^2}(h(x_\xi, w_k), y_\xi)$. (27) is known as the *generalized Gauss-Newton method*, which is resembles the classical Gauss-Newton method, which solves the least-squares loss function, $\ell(h(x_\xi, w), y_\xi) = \frac{1}{2}||h(x_\xi, w) - y_\xi||_2^2$, yielding $H_\ell = I$.

4.2.5 Diagonal scaling methods

In the case of quasi-Newton methods, the scaling matrix, H_k, is potentially a dense matrix of size $\mathbb{R}^d \times \mathbb{R}^d$, the size of which can outweigh the benefits of curvature information. To counteract this, methods which incorporate diagonal or block diagonal scaling matrices have been developed.

A running estimate of the diagonal entries of (27) can be used to perform the parameter iterations. This estimate is written as

$$[G_k]_i = (1-\lambda)\,[G_{k-1}]_i - \lambda\,[J_h(w_k;\xi_k)J_h(w_k;\xi_k)]_{ii} \tag{28}$$

$$[w_{k+1}]_i = [w_k]_i - \left(\frac{\alpha}{[G_k]_i + \mu}\right)[g(w_k,\xi_k)]_i \tag{29}$$

where $[\cdot]_i$ denots the ith element of a vector. μ is a smoothing term which prevents division by zero at initialization, λ is an exponential decay term, and $g(w_k, \xi_k)$ is the stochastic gradient at iteration k. Whereas the methods until now have used Hessian or Hessian aprrroximations of size $n \times n$, G_k is a vector of size n, which affords diagonal scaling methods substantial computational savins when the parameter space is large.

5 Optimizers

Practitioners of deep learning have access to libraries which allow for fast implementation of neural networks. In the notation developed in Section 2, a neural network takes inputs, x, and produces predictions, $h(w, \xi_k)$, which are meant to approximate desired outputs, y. Thus, the optimization of a neural network is amenable to stochastic gradient descent and the variations described in Section 4.

In this section, we define and characterize the variants of SGD which are commonly implemented in popular deep learning libraries [5].

5.1 Momentum

A momentum term with β_k may be added to (7), yielding

$$w_{k+1} = w_k - \alpha_k \nabla F(w_k) + \beta_k(w_k - w_{k-1}). \tag{30}$$

With sequences $\{\alpha_k\}$ and $\{\beta_k\}$ and initial condition $w_0 := w_1$. For $\alpha_k = \alpha$ and $\beta_k = \beta$ (i.e., constant step size and momentum coefficients), (30) is known as the *heavy ball method*. The heavy ball method can be viewed as a sum of exponentially decaying previous samples of the gradient.

$$w_{k+1} = w_k - \alpha \sum_{j=1}^{i} \beta^{k-j} \nabla F(w_j) \tag{31}$$

Thus, the heavy ball method tends to favor directions which have occurred multiple times in past realizations while discounting directions which occur infrequently. Figure 2 demonstrates how such momentum-based methods have the ability to overcome ill-conditioned prediction functions. In the context of deep neural networks (DNN), stochastic methods incorporating momentum have shown experimental promise given well-chosen parameter initialization [3].

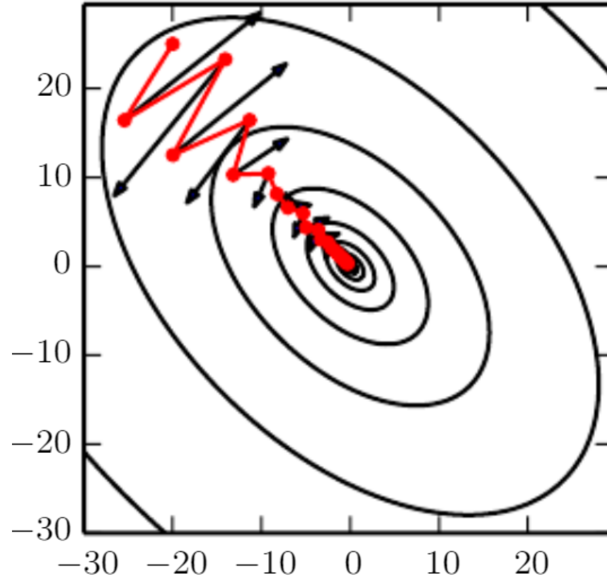

Figure 2: Illustration of the effect of momentum on SGD for a function with a poorly conditioned Hessian [9]. The black lines indicate the direction of steepest descent while the red lines indicate SGD with momentum. Momentum helps damp the oscillations which steepest descent would induce in the parameter iterates and increases the rate of converge.

5.2 Acceleration

Similar to the momentum formulation, another variant of (7) incorporating previous descent directions can be written as

$$\tilde{w}_{k+1} = w_k + \beta_k (w_k - w_{k-1}) \tag{32}$$

$$w_{k+1} = \tilde{w}_k - \alpha_k \nabla F(\tilde{w}_{k+1}) + \beta_k (w_k - w_{k-1}) \tag{33}$$

$$= w_k - \alpha_k \nabla F(w_k + \beta_k (w_k - w_{k-1})) + \beta_k (w_k - w_{k-1}) \tag{34}$$

Where the gradient samples are taken after following the parameter momentum, $\tilde{w}_k$, rather than at w_k. This *Nesterov accelerated gradient (NAG) method* is known to have $\mathcal{O}\left(\frac{1}{k^2}\right)$ convergence rate, which has been empirically shown to exhibit faster convergencce than previously developed gradient methods [4]. Figure 3 provides a geometric description of the difference between a simple momentum update and a Nesterov-supplemented momentum update.

5.3 Adaptive optimizers

While previously described algorithms have updated learning rates uniformly across all parameters, the following methods incorporate parameter-wise adaptation of learning rates. The motivating intuition here is that depending on the update history, more emphasis should be placed on certain parameters over others. Discussion and formulation of these adaptive methods was adopted from [5].

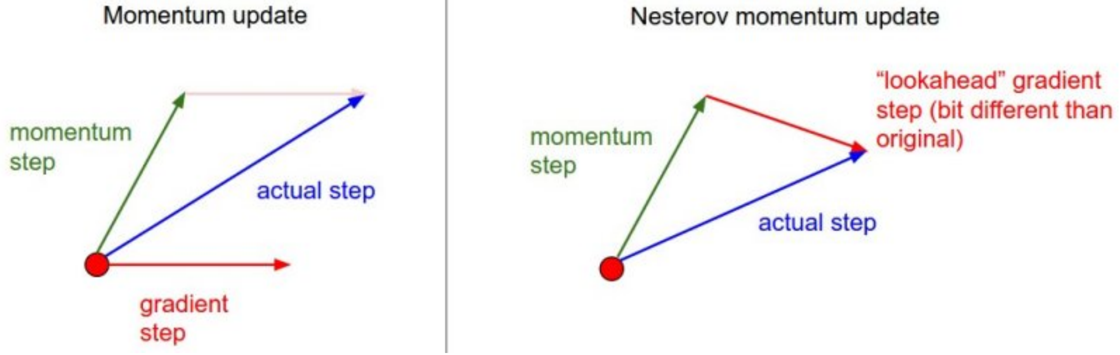

Figure 3: Illustration of difference between momentum and Nesterov accerated gradient (NAG) parameter update [6]. In the momentum update, the gradient is taken at the previous parameter value (red dot) while in NAG, the gradient is evaluated after following the momentum step.

5.3.1 Adagrad

Reminiscent of the normalized least-mean squares (NLMS) algorithm, Adagrad is diagonal scaling quasi-Newton variant of SGD with an adaptive learning rate.

$$w_{k+1} = w_k - \frac{\alpha}{\sqrt{G_t + \epsilon}} \nabla F(w_k) \tag{35}$$

$$G_t = \mathrm{diag}\left(\sum_{j=1}^{k} (\nabla_{w_i} F(w_k))^2 \right) \tag{36}$$

Where G_t is a diagonal matrix with elements corresponding to the sum of the squares of past gradients and ϵ is a smoothing term which prevents division by zero at initialization. Inspecting (35), we see that parameters $w_{k,i}$ which have a large influence on the gradient a given time t will have their future influence decay. From a machine learning practitioner's perspective, Adagrad eliminates the need to choose a particular learning since the rate is effectively controlled by G_t.

5.3.2 Adadelta/RMSProp

As the elements of G_t monotonically increase, the effective learning rate of Adagrad will always tend towards 0. To correct for this, Adadelta uses a decaying average of past squared gradients instead of the full sum of squared gradients, G_t.

$$w_{k+1} = w_k - \frac{\sqrt{\mathrm{E}\left[\Delta w_{k-1}\right] + \epsilon}}{\sqrt{\mathrm{E}\left[g\right]_k + \epsilon}} g_k \tag{37}$$

$$= w_k - \frac{\mathrm{RMS}\left[\Delta w_{k-1}\right]}{\mathrm{RMS}\left[g_k\right]} g_k \tag{38}$$

Where the RMS error terms in the numerator and denominator track the average of previous squared parameter updates and of previous squared gradients, respectively.

$$\mathrm{E}\left[\Delta w^2\right]_k = \gamma \mathrm{E}\left[\Delta w^2\right]_{k-1} + (1-\gamma)\Delta w_k^2 \tag{39}$$

$$\mathrm{E}\left[g^2\right]_k = \gamma \mathrm{E}\left[g^2\right]_{k-1} + (1-\gamma) g_t^2 \tag{40}$$

(38) does not depend on a step size coefficient, α. Similar to Adagrad, *RMSprop* adapts its step size by a decaying average of previous squared gradients without accounting for previous squared parameter

updates.

$$w_{k+1} = w_k - \frac{\alpha}{\sqrt{\mathrm{E}\left[g\right]_k + \epsilon}} g_k \tag{41}$$

$$\mathrm{E}\left[g^2\right]_{k+1} = \gamma \mathrm{E}\left[g^2\right]_k + (1-\gamma) g_k^2 \tag{42}$$

Where the author of RMSprop suggests decay rate of $\gamma = 0.9$ [7].

5.3.3 Adam

Adaptive moment estimation (Adam) was suggested by Kingma and Lei Ba in order to combine attractive features of Adagrad and RMSprop [8]. The zero-biased estimates of the first- and second-order moments (i.e., mean and variance) of the gradient are written as

$$m_k = \beta_1 m_{k-1} + (1-\beta_1) g_k \tag{43}$$

$$v_k = \beta_2 v_{k-1} + (1-\beta_2) g_k^2. \tag{44}$$

With exponential decay rates β_1, β_2. As m_0 and v_0 are initialized as zero-vectors, m_k and v_k are zero-biased. Bias-corrected versions of these moments are written as

$$\hat{m}_k = \frac{m_k}{1-\beta_1^k} \tag{45}$$

$$\hat{v}_k = \frac{v_k}{1-\beta_2^k}. \tag{46}$$

With the bias-corrected moments, the update rule for Adam can be written as

$$w_{k+1} = w_k - \frac{\alpha}{\sqrt{\hat{v}_k} + \epsilon} \hat{m}_k \tag{47}$$

Thus, parameter selection involves step size α as well as decay rates β_1, β_2. Good default values have been listed as $\alpha = 0.002$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$ [5]. Adam has generally proven robust to variation of these parameters with the caveat that learning rate, α, sometimes needs to be adjusted for the given application [9].

5.4 Comparison of Optimizers

Figure 4 compares the performance of non-adaptive optimizers (SGD, Momentum, NAG) to adaptive optimizers (Adagrad, Adadelta, RMSProp). SGD's salient shortcoming is convergence time, which both Momentum and NAG improve. However, both Momentum and NAG begin heading in the wrong direction and must wait until their momentum and acceleration terms accumulate until they course-correct. In contrast, the adaptive methods do a much better job of traversing the loss function, quickly finding a convergent trajectory.

The particular graphic in Figure 4 does not list the respective learning/decay rates, α and β. While this forces us to take the non-adaptive results with a grain of salt, the adaptive algorithms' performance only depend on these parameters near initialization.

Which optimizer should one use is largely dependent on the appliciation. If the designer cares about minimizing time to convergence, then one of the adaptive methods is ideal. Recent papers using deep learning in wireless communications have simply opted to use Adam with parameter values close to the previously described defaults [13]. However, consensus as to which algorithm enjoys the best performance overall remains elusive [9].

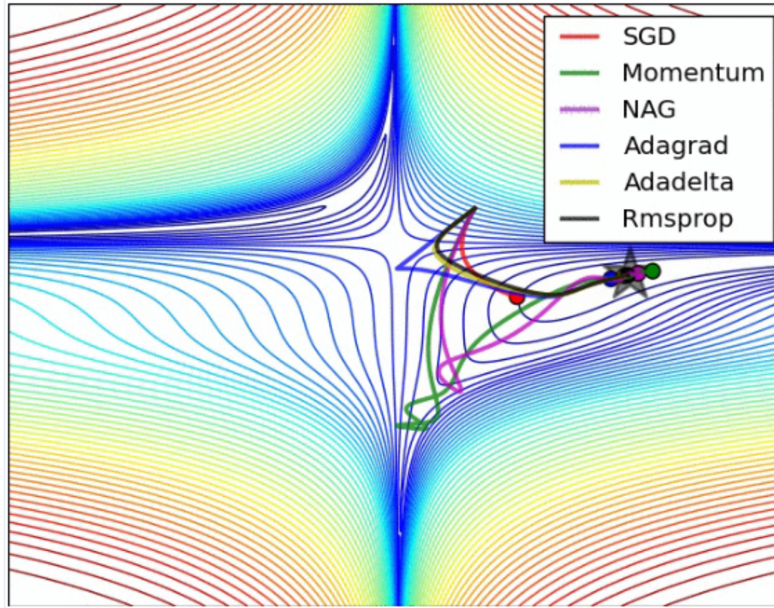

Figure 4: Last frame of an animation showcasing the parameter trajectories of the above optimizers [5]. The adaptive algorithms take more direct paths to the local minimum than the non-adaptive methods.

6 Frontiers

Stochastic gradient descent and extensions thereof are the predominant method of parameter optimization in neural networks. The mini-batch and second-order variations discussed in the last two sections have bolstered SGD's ability to deal with noise reduction, ill-conditioning, and non-convexity, all of which are salient challenges in the optimization of deep neural networks.

Beyond the methods discussed, several active areas of research have the potential to further bolster the applicability of SGD in the context of deep learning. A brief survey of these research areas follows.

6.1 Distributed gradient descent

Given the size of the datasets on which SGD operates, parallelized algorithms which calculate multiple gradients for multiple iterations concurrently are highly appealing for their potential to reduce compute time. Such parallel algorithms may serve to make SGD amenable to execution on GPUs or on multiple CPUs. Multiple efforts to implement distributed algorithms with shared memory [10], asynchronous worker nodes [11], and delay-tolerant capabilities [12].

6.2 Batch normalization

In the methods discussed, we have allowed the first- and second-order statistics of parameters to evolve such they may lose normalization (i.e., non-zero mean, non-unit variance). This might cause bias or ill-conditioning in our parameterized model, and it would be useful to renormalize parameters on a scheduled basis.

To address this, *batch normalization* serves to renormalize parameters, allows the designer to use a higher learning rate and achieve faster convergence. Batch normalization has shown a greater than 10-fold decrease in steps while achieving the same accuracy as comparable image classification networks [15].

6.3 Shuffling/curriculum learning

Consideration must be given to the order in which we present training examples to our algorithm. If we wish to prevent our neural network from learning a particular function, then we may *shuffle* the training data on subsequent iterations to prevent bias.

In contrast, we may want to incrementally 'increase' the 'difficulty' of training examples which we present to our neural network. In this case, a *curriculum* of training examples can be constructed. Such a sequence of training data has the potential to improve convergence and performance of a network [14].

References

[1] L. Bottou, "Optimization methods for large-scale machine learning," *SIAM Review*, vol. 60, pp. 223-311, 2018.

[2] R. S. Sutton, A. G. Barto, *Reinforcement learning: An introduction* 2018.

[3] L. Sutskever, J. Martens, G. Dahl, and G. Hinton, *On the importance of initialization and momentum in deep learning*, in 30th International Converence on Machine Learning (ICML), 2013.

[4] Y. Nesterov, *A method of solving a convex programming problem with convergence rate* $\mathcal{O}\left(\frac{1}{k^2}\right)$, Soviet Math. Dokl., 27 (1983)

[5] S. Ruder, *An overview of gradient descent optimization algorithms*, arXiv:1609.04747v2 [cs.LG], (2017).

[6] A. Karpathy, *Convolutional Neural Networks for Visual Recognition*, Course notes, Accessed 2019 June 9.

[7] G. Hinton, *Lecture 6a: Overview of mini-batch gradient descent*, Coursera lecture, "Neural networks for machine learning," Accessed 2019 June 9.

[8] D. P. Kingma, J. L. Ba, *Adam: A method for stochastic optimization.* International Conference on Learning Representations, 1-13. (2015).

[9] I. Goodfellow, Y. Bengio, A. Courville, *Optimization for Training Deep Models*, from *Deep Learning*, MIT Press, (2016).

[10] F. Niu, B. Recht, R. Christopher, S. J. Wright, *Hogwild! : A Lock-Free Approach to Parallelizing Stochastic Gradient Descent*, 1–22. (2011).

[11] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, A. Y. Ng, *Large Scale Distributed Deep Networks.* NIPS 2012: Neural Information Processing Systems, 1–11. (2012).

[12] H. B. Mcmahan, M. Streeter, *Delay-Tolerant Algorithms for Asynchronous Distributed Online Learning.* Advances in Neural Information Processing Systems (Proceedings of NIPS), 1–9. (2014).

[13] T. J. O'Shea, T. Roy, N. West, *Approximating the void: Learning stochastic channel models from observation with variational generative adversarial networks.* arXiv:1805.06350v2, (2018).

[14] W. Zaremba, I. Sutskever, *Learning to Execute,* arXiv:1410.4615v3 [cs.NE] (2014).

[15] S. Ioffe, C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal co-variate shift.* arXiv:1502.03167v3 [cs.LG] (2015).