The Gradient Descent Optimization Algorithms

Haobin Tan

Institute for Anthropomatics and Robotics,
Karlsruhe Institute of Technology
haobin.tan@student.kit.edu

Abstract

Gradient descent is one of the most important algorithms in machine learning and deep learning. In this paper, we first take a look at different variants of gradient descent. Addressing the problems that may occur during applying gradient descent, we introduce a number of practical optimization algorithms. Last but not least, we also give some empirical suggestions for the selection of gradient descent optimization algorithms.

1. Introduction

Most machine learning and deep learning algorithms involve optimization of some sort. It is the task of minimizing the model's loss function (or cost function) parameterized by the model's parameters. One of the most popular strategies to solve this minimization task is **gradient descent** algorithms.

This paper attempts to review different types of gradient descent and algorithms for optimizing gradient descent. In section 2, we initially introduce the gradient descent algorithm and its variants: Batch Gradient Descent (BGD), Stochastic Gradient Descent (SGD), and Mini-Batch Gradient Descent (MBGD). In section 3, two main challenges of gradient descent, choosing a proper learning rate and non-convex loss function, will be then summarized. In order to deal with these challenges, section 4 outlines some optimization algorithms: Momentum, Nesterov Momentum, Adaptive Gradient Algorithm (AdaGrad), Adadelta, RMSprop, and Adaptive Moment Estimation (Adam). They are able to improve the performance of gradient descent and therefore widely used in deep learning community.

2. Gradient Descent Variants

As it requires information of loss function's first derivative to update the model's parameters, gradient descent is a first-order optimization algorithm for finding the minimum of the loss function $J(\theta)$. It iteratively updates the parameters θ in the opposite direction of the gradient $\nabla_\theta J(\theta)$ with respect to the parameters $\theta \in \mathbb{R}^d$. The hyperparameter, learning rate η, determines the step size at each update while moving along the aforementioned direction [2].

There're three variants of gradient descent, which are defined on the basis of how much data we use to compute the gradient $\nabla_\theta J(\theta)$ at each update iteration. Depending upon the amount of data used per update, the required time and accuracy of the algorithms differ from each other.

2.1. Batch Gradient Descent

Batch Gradient Descent (BGD) computes the gradient of the cost function with respect to the parameters with the whole training dataset [1]:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) \tag{1}$$

Training using BGD can be presented as Algorithm 1. And the function `calculate_gradient`, which depends on the loss function, should be defined in advance.

Algorithm 1 Batch Gradient Descent

```
1:  $n \leftarrow$ number of epochs
2:  $\eta \leftarrow$ learning rate
3:  $J(\theta) \leftarrow$ loss function
4:  $\theta \leftarrow$ initialized parameters
5:  $M = \{(x^{(i)}, y^{(i)}) | i = 1, \ldots, m\} \leftarrow$ training dataset
6:  function BGD($J(\theta), M, \eta, \theta$)
7:      for $j \leftarrow 1$ to $n$ do
8:          $\nabla_\theta J \leftarrow$ calculate_gradient($J(\theta), M, \theta$)
9:          $\theta \leftarrow \theta - \eta \nabla_\theta J$
10:     end for
11:     return $\theta$
12: end function
```

BGD guarantees to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces [1]. Example in Figure 1 shows how BGD converges to the minimum of the basin the parameters are placed in.

Nevertheless, training with BGD can be very slow and is intractable for very large datasets that don't fit in memory, since we need to calculate the gradient based on the whole dataset to perform just one update. Moreover, BGD doesn't allow us to update our model online.

2.2. Stochastic Gradient Descent

Whereas BGD takes the entire training dataset into consideration before taking a single step, Stochastic Gradient Descent

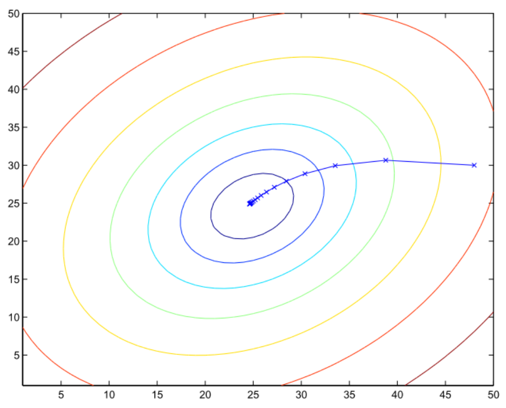
Figure 1: An example of applying BGD to minimize a quadratic loss function. The ellipses are the contours of the loss function. Successive values of θ that BGD went through are marked by ”$\times$”s (joined by straight lines). (Source: [3])

(SGD) can start making progress right away, and continues to make progress with each training example $x^{(i)}$ and label $y^{(i)}$ it looks at [1]:

$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta; x^{(i)}; y^{(i)}\right) \tag{2}$$

Pseudocode for SGD is provided in Algorithms 2. Note that it is necessary to shuffle the training dataset (line 8) in order to prevent providing the training examples in a meaningful order to the model since this may result in bias in the optimization algorithm.

Algorithm 2 Stochastic Gradient Descent

```
 1: $n \leftarrow$ number of epochs
 2: $\eta \leftarrow$ learning rate
 3: $J(\theta) \leftarrow$ loss function
 4: $\theta \leftarrow$ initialized parameters
 5: $M = \{(x^{(i)}, y^{(i)}) | i = 1, \dots, m\} \leftarrow$ training dataset
 6: function SGD($J(\theta), M, \eta, \theta$)
 7:     for $j \leftarrow 1$ to $n$ do
 8:         Shuffle $M$
 9:         for $k \leftarrow 1$ to $m$ do
10:             $\nabla_\theta J \leftarrow$ calculate_gradient($J(\theta), (x^{(k)}, y^{(k)}), \theta$)
11:             $\theta \leftarrow \theta - \eta \nabla_\theta J$
12:         end for
13:     end for
14:     return $\theta$
15: end function
```

As SGD evaluates the gradient $\nabla_\theta J(\theta)$ and updates parameters θ with only one single training example at a time, it is usually much faster than BGD and can also be used to learn online [1]. On the other hand, due to frequent updates with a high variance, the objective function may sometimes fluctuate heavily (Figure 2).

Furthermore, SGD’s fluctuation complicates convergence to the exact minimum, as SGD will keep overshooting

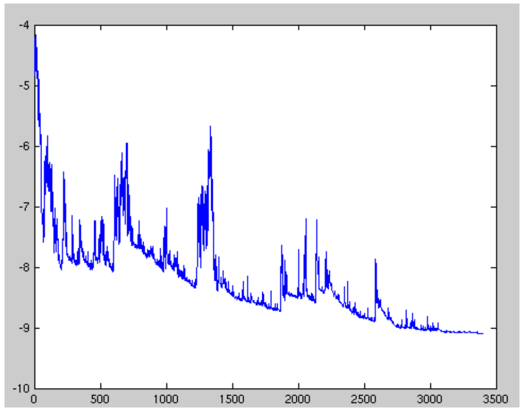
Figure 2: SGD fluctuation (Source: [1])

[1]. Due to its stochastic (i.e., random) nature, SGD is much less regular than BGD: the parameters θ will keep oscillating around the minimum of $J(\theta)$ [3] (Figure 3). Nonetheless, in practice, most of the values near the minimum can be reasonably considered as good approximations to the true minimum. Additionally, with the help of simulated annealing, which slowly decreases the learning rate, SGD almost converges to a local or the global minimum for non-convex and convex optimization respectively. For these reasons, when the training set is large, SGD is often preferred over BGD [3].

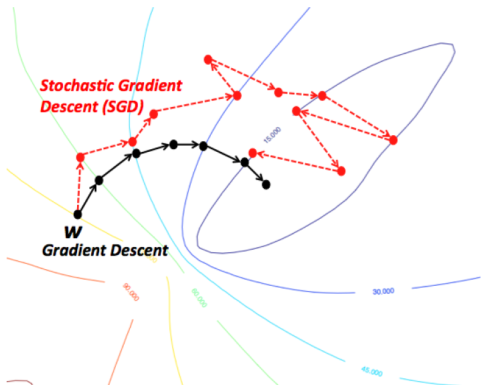

Figure 3: SGD vs. BGD (Source: https://wikidocs.net/3413)

2.3. Mini-batch Gradient Descent

At each step, instead of evaluating the gradients on the base of the full training set (as in BGD) or based on just one training example (as in SGD), Mini-Batch Gradient Descent (MBGD) performs an update for small random sets of training examples [1]:

$$\theta = \theta - \eta \cdot \nabla_\theta J\left(\theta; x^{(i:i+b)}; y^{(i:i+b)}\right) \tag{3}$$

The algorithm is described in pseudocode in Algorithm 3. Note that in line 10 the pre-defined function `get_batch` returns a set of mini-batches based on the mini-batch size. The size usually ranges between 50 and 256, but can vary depending on different applications.

Algorithm 3 Mini-Batch Gradient Descent

```
 1: $n \leftarrow$ number of epochs
 2: $\eta \leftarrow$ learning rate
 3: $J(\theta) \leftarrow$ loss function
 4: $\theta \leftarrow$ initialized parameters
 5: $M = \{(x^{(i)}, y^{(i)}) | i = 1, \dots, m\} \leftarrow$ training dataset
 6: $b \leftarrow$ batch size
 7: function MBGD($J(\theta), M, \eta, \theta$)
 8:     for $j \leftarrow 1$ to $n$ do
 9:         Shuffle $M$
10:         $\mathbb{B} = \{B_l | l = 1, \dots, \lfloor \frac{m}{b} \rfloor\} \leftarrow$ get_batch($M, b$)
11:         for $k \leftarrow 1$ to $l$ do
12:             $\nabla_\theta J \leftarrow$ calculate_gradient($J(\theta), B_k, \theta$)
13:             $\theta \leftarrow \theta - \eta \nabla_\theta J$
14:         end for
15:     end for
16:     return $\theta$
17: end function
```

The main advantage of MBGD is that it reduces the variance of the parameter updates, which can lead to more stable convergence [1]. Another benefit of MBGD over SGD is that we can achieve a performance boost from hardware optimization of matrix operations, especially when using GPUs [4].

2.4. Comparison and Trade-offs

Table 1 summarizes the difference and trade-offs between BGD, SGD, and MBGD. Furthermore, Figure 4 visualizes the comparison by showing the paths taken by three gradient descent algorithms in parameter space $\theta \in \mathbb{R}^2$ during training. All algorithms end up near the minimum, but the path of BGD actually stops right at the minimum, while SGD and MBGD continue to walk around. In addition, the oscillation of MBGD is obviously smaller than SGD.

Method	Accuracy	Update Speed	Memory Usage	Online Learning
BGD	very good	slow	high	no
SGD	good (with annealing)	high	low	yes
MBGD	good	medium	medium	yes

Table 1: Comparison and trade-offs between gradient descent variants

3. Challenges of Gradient Descents

During applying (vanilla) gradient descent, a few challenges, which may be detrimental to performance, need to be addressed.

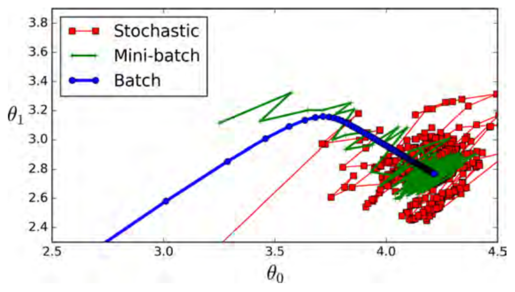

Figure 4: Comparison between gradient descent variants (Source: [4])

3.1. Learning Rate

A common challenge of gradient descent is to choose the proper learning rate η. As shown in Figure 5, a learning rate that is too small (like λ_0) can result in slow convergence and learning, while a learning rate that is too large can hinder convergence and lead to bouncing around the minimum (λ_2) or even to diverge (λ_3). Optimization algorithms aiming to handle this issue will be discussed in Section 4.3, 4.4, 4.5, and 4.6.

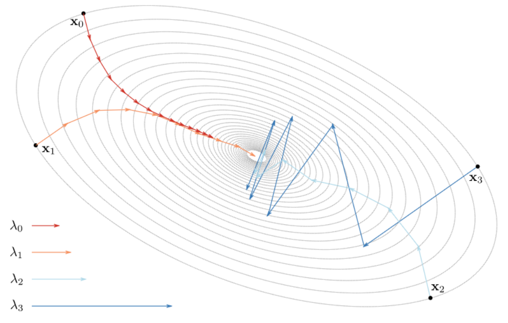

Figure 5: Effect of different learning rate on gradient descent (Source: https://blog.yani.io/sgd/)

3.2. Non-convex Loss function

Another typical challenge is highly non-convex loss functions, which are usually used in neural networks. Due to non-convexity, numerical suboptimal local minima can be a serious problem during training (Figure 6). Furthermore, difficulty also results from saddle points [7]. Such saddle points are commonly surrounded by a plateau of the same error. Since the gradient in all dimensions is close to zero, it is problematic for gradient descent to escapse [7]. Possible solutions for this challenge are described in Section 4.1 and 4.2.

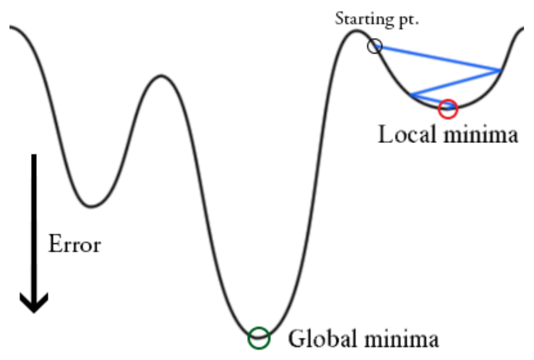

Figure 6: Problem caused by non-convex loss function. The loss function stucks in local minima and thus can not get to the global minima. (Source: Non-convex optimization)

4. Gradient Descent Optimization Algorithms

The update of parameters at each iteration consists of two parts: gradient and learning rate. We can consider the gradient as the update step direction and the learning rate as the update step size. Therefore, there're two perspectives to optimize gradient descent algorithm. Momentum and Nesterov Momentum aims to improve gradient descent from the perspective of the update step direction. Adaptive learning rate methods, including AdaGrad, Adadelta, RMSprop, and Adam, devote themselves to optimize the update step size.

4.1. Momentum

Motivating from a physical perspective of the optimization problem, momentum algorithms [5] is designed to accelerate learning and dampens oscillations (Figure 7). It takes the gradient from previous steps into account, accumulates an exponentially decaying moving average of past gradients and continues to move in their direction [6]:

$$\begin{aligned} v_{t+1} &= \mu v_t - \eta \nabla_\theta J(\theta_t) \\ \theta_{t+1} &= \theta_t + v_{t+1}, \end{aligned} \tag{4}$$

where

- $\mu \in [0, 1)$: momentum coefficient (hyperparameter), damps the velocity and reduces the kinetic energy of the system. From a physical perspective, it can be considered as the coefficient of friction. The typical value of μ used in practice is 0.9.
- v: velocity, the direction and speed at which the parameters move through parameter space. [6].

Figure 8 shows the momentum update at t-th step.

4.2. Nesterov Momentum

Motivated by Nesterov Accelerated Gradient (NAG) [8], a slight variant of the momentum algorithm (Section 4.1) was proposed. The update rules of Nesterov momentum are given by [5]:

$$\begin{aligned} v_{t+1} &= \mu v_t - \eta \nabla J_\theta \left(\theta_t + \mu v_t\right) \\ \theta_{t+1} &= \theta_t + v_{t+1} \end{aligned} \tag{5}$$

where μ and η play the similar role as in Equation 4.

The significant difference between Nesterov momentum and standard momentum is the position where the gradient is computed: while standard momentum directly evaluates the gradient from the current position θ_t (see Figure 8), Nesterov momentum computes the gradient from a "lookahead" position (red point in Figure 9), a point in the vicinity of where the parameters are soon going to be.

In the convex batch gradient case, Nesterov momentum brings the rate of convergence of the excess error from $O(1/k)$ (after k steps) to $O(1/k^2)$ as shown by [8]. Unfortunately, in the stochastic gradient case, Nesterov momentum

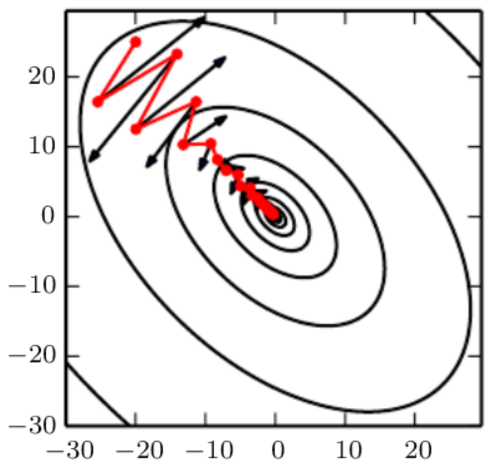

Figure 7: The ellipses are the contours of a quadratic loss function with a poorly conditioned Hessian matrix, whose shape looks like a long, narrow valley or canyon with steep sides. The black arrows indicate the step that (vanilla) gradient descent will take at that point. The red path is the path of steps applying momentum rule. While gradient steps spend a long time oscillating across the narrow axis of the canyon, momentum traverses the canyon lengthwise correctly and reaches the minimum more quickly. (Source: [6])

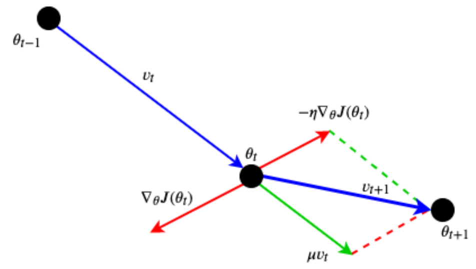

Figure 8: Momentum update at step t

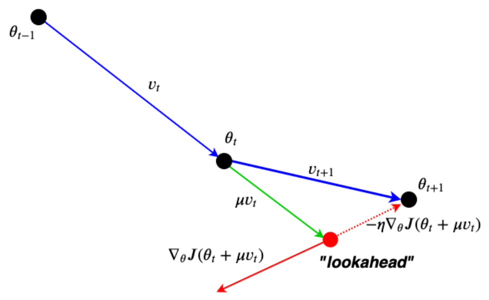
θ_{t-1}
v_t
θ_t
v_{t+1}
θ_{t+1}
μv_t
$-\eta\nabla_\theta J(\theta_t+\mu v_t)$
$\nabla_\theta J(\theta_t+\mu v_t)$
"lookahead"

Figure 9: Nesterov momentum update at step t

does not improve the rate of convergence [6].

4.3. AdaGrad

Algorithms in Section 4.1 and Section 4.2 focus on adapting the updates to the slope of the loss function and speeding up gradient descent. Another aspect of optimization is to make the learning rate to adapt to each parameter as well as its gradient.

In many application of machine learning and deep learning, the input instance are often of very high dimension. But only a few features are non-zero within some particular instance. These features are infrequent and the data is thus sparse. However, it is often the case that infrequently occurring features are much more informative and discriminative. I.e., infrequent features are usually more significant than the frequents. For example, for Natural Language Processing (NLP), although words like "the", "a" occur very frequently, they are not so important for dialog modeling.

Therefore, instead of using a fixed learning rate for all parameters, it can make more sense to adapt different learning rates to the parameters, in order to perform larger or smaller updates based on their importance.

One of the algorithms with adaptive learning rates is Adaptive Gradient Algorithm (AdaGrad) [10]. At each update, it individually adapts the learning rate of parameters by scaling them inversely proportional to the square root of the accumulated squared gradient [10]. In other words, parameters with large partial derivative of the loss have a correspondingly rapid decrease in their learning rate, while parameters with small partial derivatives have a relatively small decrease in their learning rate [6].

For the sake of brevity, we use g_t to denote the gradient of the loss function $J(\theta)$ with respect to the parameter θ at step t:

$$g_t = \begin{pmatrix} g_{t,1} \\ g_{t,2} \\ \vdots \\ g_{t,d} \end{pmatrix} = \begin{pmatrix} \nabla_\theta J(\theta_{t,1}) \\ \nabla_\theta J(\theta_{t,2}) \\ \vdots \\ \nabla_\theta J(\theta_{t,d}) \end{pmatrix} \in \mathbb{R}^{d\times 1} \tag{6}$$

The update rule of AdaGrad at time step t is [10]:

$$\begin{aligned} \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t \\ \Delta\theta_t &= -\frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t \end{aligned} \tag{7}$$

where

- $G_t \in \mathbb{R}^{d\times d}$: a diagonal matrix where each diagonal element $G_{t,ii}$ is the sum of squares of the gradient with respect to θ_i up to time step t [10], i.e.,

$$G_t = \begin{pmatrix} \sum_{\tau=1}^t g_{\tau,1}^2 & 0 & \cdots & 0 \\ 0 & \sum_{\tau=1}^t g_{\tau,2}^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sum_{\tau=1}^t g_{\tau,d}^2 \end{pmatrix} \tag{8}$$

- ϵ: a smoothing term that aims to prevent division by zero. Common value of ϵ is 10^{-7} [6].
- Division and square root are element-wise operation. For clarity, we expand Equation (7):

$$\begin{aligned} \Delta\theta_t &= \begin{pmatrix} \Delta\theta_{t,1} \\ \Delta\theta_{t,2} \\ \vdots \\ \Delta\theta_{t,d} \end{pmatrix} \\ &= -\begin{pmatrix} \frac{\eta}{\sqrt{\varepsilon + G_{t,11}}} & 0 & \cdots & 0 \\ 0 & \frac{\eta}{\sqrt{\varepsilon + G_{t,22}}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\eta}{\sqrt{\varepsilon + G_{t,dd}}} \end{pmatrix} \begin{pmatrix} g_{t,1} \\ g_{t,2} \\ \vdots \\ g_{t,d} \end{pmatrix} \\ &= -\begin{pmatrix} \frac{\eta}{\sqrt{\varepsilon + G_{t,11}}} g_{t,1} \\ \frac{\eta}{\sqrt{\varepsilon + G_{t,22}}} g_{t,2} \\ \vdots \\ \frac{\eta}{\sqrt{\varepsilon + G_{t,dd}}} g_{t,d} \end{pmatrix} \end{aligned} \tag{9}$$

The main benefit of AdaGrad is that the learning rate doesn't need to be manually tuned. Most implementation use a default value of 0.01 [1]. What's more, according to Equation (8) and (9), AdaGrad performs larger updates for infrequent and smaller updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data [1].

However, accumulation of the squared gradients from the beginning of training in the denominator (Equation (7)) results in the problem that the learning rate keeps shrinking and eventually become infinitesimally small [1]. At that point, the model is almost unable to learn additional knowledge.

4.4. Adadelta

As an extension of AdaGrad (Section 4.3), Adadelta [11] aims to improve upon the two main disadvantages:

a) continual shrinking of learning rate throughout training

b) the necessity of a manually selected global learning rate

4.4.1. *Accumulate over Window*

Regarding a), instead of accumulating all previous squared gradients, Adadelta restricts the window of past gradients that are accumulated to be some fixed size w. This ensures that learning keeps making progress, even after numbers of iteration [11].

As it is not efficient to store w past squared gradients, the accumulation of gradients is approximated with Exponentially Weighted Moving Average (EWMA) [11]. At step t, the running average $E[g^2]_t$ depends only on the last average and the current gradient [11]:

$$E\left[g^2\right]_t = \mu E\left[g^2\right]_{t-1} + (1-\mu)g_t^2 \tag{10}$$

where $\mu \in (0,1)$ is a degree of weighting decrease and plays a similar role as in the momentum method (see Equation (4)). We then replace G_t in Equation (7) with the decaying average $E[g^2]_t$:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t \tag{11}$$

where ϵ is a smoothing term that aims to prevent division by zero. The denominator is nothing but the (numerical) Root Mean Square (RMS) of the gradient [11]:

$$\mathbf{RMS}[g]_t = \sqrt{E[g^2]_t + \epsilon} \tag{12}$$

Therefore, Equation (11) can also be written as [11]:

$$\Delta\theta_t = -\frac{\eta}{\mathbf{RMS}[g]_t} \cdot g_t \tag{13}$$

4.4.2. *Correct Units with Hessian Approximation*

At each iteration, $\Delta\theta$ is applied to θ to perform an update. Therefore, the units of $\Delta\theta$ and θ should match. In other words, if the parameters θ have some hypothetical units, the changes to the parameters $\Delta\theta$ sholud be in the same units as well. However, first-order methods, such as SGD, Momentum, and AdaGrad, don't have correct units [11]:

$$\text{units of } \Delta\theta \propto \text{ units of } g \propto \frac{\partial J}{\partial \theta} \propto \frac{1}{\text{units of } \theta} \tag{14}$$

assuming the loss function J is unitless. In contrast, second-order methods like Newton's method, which make use of Hession information or an approximation to the Hessian, provide the correct units [11]:

$$\Delta\theta \propto \mathbf{H}^{-1}g \propto \frac{\frac{\partial J}{\partial \theta}}{\frac{\partial^2 J}{\partial \theta^2}} \propto \text{ units of } \theta \tag{15}$$

One of the most widely used second-order method is Newton's method, whose update rule at time step t is [16]:

$$\begin{aligned} \theta_{t+1} &= \theta_t - D^2 J(\theta_t)^{-1} \nabla_\theta J(\theta_t) \\ &= \theta_t - \mathbf{H}(J(\theta_t))^{-1} \cdot g_t \end{aligned} \tag{16}$$

where $D^2 J(\theta_t)$ denotes the second derivative of the loss function $J(\theta)$. Comparing Equation (16) with normal gradient descent update step (Equation (1)), $\mathbf{H}(J(\theta_t))^{-1}$ can then be considered as an automatically adaptive learning rate. Thus, there is no need to select a global learning rate manually.

Based on the approximation method proposed by Becker and Lecun [12]:

$$\Delta\theta_t = -\frac{1}{|\text{diag}(\mathbf{H}_t)| + \mu} g_t \tag{17}$$

where μ is a small constant to improve the conditioning of the Hessian for regions of small curvature [11], we rearrange Newton's method (assuming a diagonal Hessian) [11]:

$$\Delta\theta_t = \frac{\frac{\partial J}{\partial \theta_t}}{\frac{\partial^2 J}{\partial \theta_t^2}} \tag{18}$$

$$\Rightarrow \frac{1}{\frac{\partial^2 J}{\partial \theta_t^2}} = \frac{\Delta\theta_t}{\frac{\partial J}{\partial \theta_t}} \tag{19}$$

The denominater $\frac{\partial J}{\partial \theta_t}$ can be estimated with the RMS of the previous gradients $\mathbf{RMS}[g]_t$ (Equation (12)) [11]:

$$\frac{\partial J}{\partial \theta_t} \approx \mathbf{RMS}[g]_t \tag{20}$$

As the numerator, $\Delta\theta_t$, is unknown at the current step, we approximate it by computing the exponentially decaying RMS over a window of size w under the assumption that the curvature is locally smooth [11]:

$$E\left[\Delta\theta^2\right]_{t-1} = \mu E\left[\Delta\theta^2\right]_{t-2} + (1-\mu)\Delta\theta_{t-1}^2 \tag{21}$$

$$\mathbf{RMS}[\Delta\theta]_{t-1} = \sqrt{E\left[\Delta\theta^2\right]_{t-1} + \epsilon} \tag{22}$$

$$\Delta\theta_t \approx \mathbf{RMS}[\Delta\theta]_{t-1} \tag{23}$$

Combining Equation (16), (20), and (23) finally yields the update rule of Adadelta [11]:

$$\theta_{t+1} = \theta_t - \frac{\mathbf{RMS}[\Delta\theta]_{t-1}}{\mathbf{RMS}[g]_t} g_t \tag{24}$$

Notice that the term $\frac{\mathbf{RMS}[\Delta\theta]_{t-1}}{\mathbf{RMS}[g]_t}$ is an approximation to the diagonal Hessian using only RMS measure of g and $\Delta\theta$. This approximation is always positive [12], which thus ensures the update direction follows the negative gradient at each step. Furthermore, the numerator plays the role of acceleration term, which accumulates previous gradients over a window of time as in momentum [11]. In addition, the denominator is related to AdaGrad. Its squared gradient information per-dimension helps to even out the progress made in each dimension, but is computed over a window to ensure progress is made later in training [11]. The complete algorithm is presented in Algorithm 4.

4.5. RMSprop

RMSprop is an unpublished optimization algorithm designed for neural networks, first proposed by Geoff Hinton in Lec-

Algorithm 4 Adadelta update at time t

```
Require: Decay rate $\mu$, Constant $\epsilon$
Require: Initial parameter $\theta_0 \in \mathbb{R}^d$
1: Initialize accumulation variables $E[g^2]_0 = 0$, $E[\Delta x^2]_0 = 0$
2: for $t \leftarrow 1$ to $T$ do
3:     Compute gradient: $g_t$
4:     Accumulate Gradient: $E[g^2]_t = \mu E[g^2]_{t-1} + (1-\mu)g_t^2$
5:     Compute Update: $\Delta\theta_t = -\frac{\text{RMS}[\Delta\theta]_{t-1}}{\text{RMS}[g]_t} g_t$
6:     Accumulate Updates: $E[\Delta\theta^2]_t = \mu E[\Delta\theta^2]_{t-1} + (1-\mu)\Delta\theta_t^2$
7:     Apply Update: $\theta_{t+1} = \theta_t + \Delta\theta_t$
8: end for
```

ture 6e of his online Coursera course[1]. The update rule of RMSprop is essentially the same as Equation (13):

$$\begin{aligned} E\left[g^2\right]_t &= \mu E\left[g^2\right]_{t-1} + (1-\mu)g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t \\ &\overset{(12)}{=} \theta_t - \frac{\eta}{\mathbf{RMS}[g]_t} \cdot g_t \end{aligned} \tag{25}$$

Hinton suggests the hyperparameter μ have a value of 0.9 and the learning rate η should be 0.001.

Due to its good performance, especially for the non-convex case, RMSprop has been empirically shown to be an effective and useful optimization algorithm. And it is currently one of the go-to optimization methods being employed routinely by deep learning practitioners [6].

4.6. Adam

Another popular method with adaptive learning rates for each parameter is Adaptive Moment Estimation (Adam) [13]. It is essentially the combination of Momentum (Section 4.1) and RMSprop (Section 4.5).

At time step t, we update EWMA of the gradient (m_t) and the squared gradient (v_t). These moving averages are estimates of the first moment (the mean) m_t and the second raw moment (the uncentered variance) v_t [13]:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\, g_t \tag{26}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\, g_t^2 \tag{27}$$

Nevertheless, these moving averages are initialized as $\mathbf{0}$ (vector of 0's), which causes the problem that they are biased towards zero during the initial time steps, and especially when the decay rates are small [13]. Therefore, these biases have to be counteracted [1, 13]:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{28}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{29}$$

[1]`https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`

The update rule of Adam is similar to the form as in Adadelta and RMSprop [13]:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{30}$$

The default values for β_1, β_2, and ϵ are proposed to be 0.9, 0.999, and 10^{-8} [13]. The complete algorithm of Adam is described in Algorithm 5.

Algorithm 5 Adam. Default settings are $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$

```
Require: $\eta$: learning rate
Require: $\beta_1, \beta_2 \in [0,1)$: Exponential decay rates for the moment estimates
Require: $J(\theta)$: Loss function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
1: $m_0 \leftarrow \mathbf{0}$                          ▷ Initialize 1st moment vector
2: $v_0 \leftarrow \mathbf{0}$                          ▷ Initialize 2nd moment vector
3: $t \leftarrow 0$
4: while stopping criterion not met do
5:     $t \leftarrow t + 1$
6:     $g_t \leftarrow \nabla_\theta J(\theta)$
7:     $m_t = \beta_1 m_{t-1} + (1-\beta_1)\, g_t$
8:     $v_t = \beta_2 v_{t-1} + (1-\beta_2)\, g_t^2$
9:     $\hat{m}_t = \frac{m_t}{1-\beta_1^t}$            ▷ Bias correction for 1st moment
10:    $\hat{v}_t = \frac{v_t}{1-\beta_2^t}$            ▷ Bias correction for 2nd moment
11:    $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t$    ▷ Update parameters
12: end while
13: return $\theta_t$
```

It is empirically shown that Adam works well in practice and compares favorably to other stochastic optimization methods (Figure 10) [13]. In practice, Adam is currently recommended as the default algorithm to use. However, it is usually also worth trying SGD combining with Nesterov Momentum as an alternative [14].

4.7. Selection of Optimization Algorithms

After discussing a number of optimization algorithms that attempt to overcome the weakness of vanilla gradient descent and improve its performance, a natural question occurs: which algorithm should we choose?

Unfortunately, insofar there is no consensus on this point. Based on the analysis and comparison presented by Schaul [15], the family of algorithms with adaptive learning rate (AdaGrad, Adadelta, RMSprop, Adam) have fairly robust performance. RMSprop, Adadelta, and Adam are very similar to each other and perform well in similar situations. It is shown that Adam slightly outperforms RMSprop thanks to its bias correction. However, there's no absolute best algorithm.

Currently, the most popular optimization algorithms actively in use are SGD, SGD with momentum, RMSprop, RMSprop with momentum, Adadelta, and Adam [6]. For

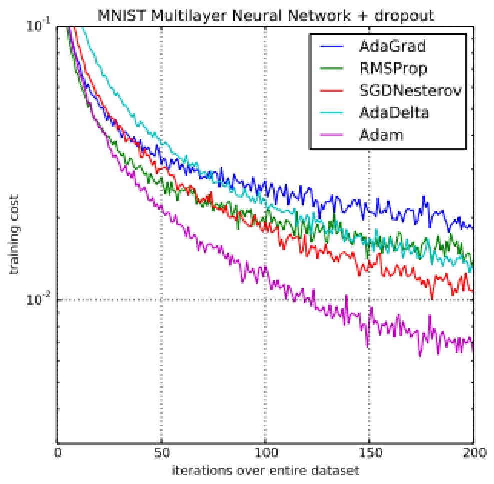

Figure 10: Comparison between gradient descent optimization algorithms (Source: [13])

ease of hyperparameter tuning, the choice of optimization algorithm seems to be heavily reliant on the practitioner's familiarity with the algorithm [6]. A practical suggestion is that the best result can be likely achieved using one of the adaptive learning rate methods, if the input data is sparse [1]. Moreover, if we care about fast convergence and train a deep or complex neural network (such as CNN, RNN), we should prefer the member of the adaptive learning rate methods [1].

5. Conclusions

In this paper, we have initially introduced three variants of gradient descent, among which Mini-Batch Gradient Descent is usually the algorithm of choice. Two common issues of gradient descent, improper learning rate and non-convex loss function, have also been addressed. Aiming to overcome these weaknesses, we have investigated numbers of popular optimization algorithms: Momentum, Nesterov momentum, AdaGrad, Adadelta, RMSprop, and Adam. Based on practical experience, it turned out that adaptive learning rate methods (AdaGrad, Adadelta, RMSprop, and Adam) provide a fairly robust performance. Therefore, they are preferred for training a deep or complex neural network, especially when the input data is sparse.

6. References

[1] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[2] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[3] Andrew Ng. Cs229 lecture notes. *CS229 Lecture notes*, 1(1):1–3, 2000.

[4] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. " O'Reilly Media, Inc.", 2017.

[5] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[7] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941, 2014.

[8] Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.

[9] Yurii Nesterov. *Lectures on convex optimization*, volume 137. Springer, 2018.

[10] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[11] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[12] Sue Becker, Yann Le Cun, et al. Improving the convergence of back-propagation learning with second order methods. In *Proceedings of the 1988 connectionist models summer school*, pages 29–37, 1988.

[13] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[14] Andrej Karpathy et al. Cs231n convolutional neural networks for visual recognition. *Neural networks*, 1, 2016.

[15] Tom Schaul, Ioannis Antonoglou, and David Silver. Unit tests for stochastic optimization. *arXiv preprint arXiv:1312.6055*, 2013.

[16] Mordecai Avriel. *Nonlinear programming: analysis and methods*. Courier Corporation, 2003.