梯度下降类优化算法概述*

Sebastian Ruder
Insight Centre for Data Analytics, NUI Galway
Aylien Ltd., Dublin
ruder.sebastian@gmail.com
翻译: 管枫 (初)，彭博 (复)，zhangdotcn(审)

Abstract

Gradient descent optimization algorithms, while increasingly popular, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by. This article aims to provide the reader with intuitions with regard to the behaviour of different algorithms that will allow her to put them to use. In the course of this overview, we look at different variants of gradient descent, summarize challenges, introduce the most common optimization algorithms, review architectures in a parallel and distributed setting, and investigate additional strategies for optimizing gradient descent.

梯度下降类算法在优化问题中非常流行，但是因为难以简单的对众多梯度下降类算法给出一个实用的优劣分析，所以在实际应用中这类算法常常被当作黑箱算法来使用。本文的目的是为读者提供不同算法效果的直观展示，并希望读者能够在实际问题中更合理的选择和使用梯度下降类算法。在这个概述中，我们考察梯度下降算法的不同类型，总结其面临的挑战，介绍几种常用的具体算法，简单介绍并行与分布式架构，并探索其他一些梯度下降类优化策略。

1 Introduction

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks. At the same time, every state-of-the-art Deep Learning library contains implementations of various algorithms to optimize gradient descent (e.g. lasagne's[2], caffe's[3], and keras'[4] documentation). These algorithms, however, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by.

梯度下降作为最流行的优化算法之一，目前也是神经网络问题中最常用的优化方法。同时每一个最先进的深度学习库里都包含各种算法来优化梯度下降法 (比如:lasagne[5], caffe[6], and keras[7] documentation). 然而难以合理的解释其优缺点，这些算法通常被当作黑箱算法来使用。

This article aims at providing the reader with intuitions with regard to the behaviour of different algorithms for optimizing gradient descent that will help her put them to use. In Section 2, we are first going to look at the different variants of gradient descent. We will then briefly summarize

*本文原载于 2016 年 1 月 19 号的博客http://sebastianruder.com/optimizing-gradient-descent/index.html

[2]http://lasagne.readthedocs.org/en/latest/modules/updates.html

[3]http://caffe.berkeleyvision.org/tutorial/solver.html

[4]http://keras.io/optimizers/

[5]http://lasagne.readthedocs.org/en/latest/modules/updates.html

[6]http://caffe.berkeleyvision.org/tutorial/solver.html

[7]http://keras.io/optimizers/

challenges during training in Section 3. Subsequently, in Section 4, we will introduce the most common optimization algorithms by showing their motivation to resolve these challenges and how this leads to the derivation of their update rules. Afterwards, in Section 5, we will take a short look at algorithms and architectures to optimize gradient descent in a parallel and distributed setting. Finally, we will consider additional strategies that are helpful for optimizing gradient descent in Section 6.

本文的目的是提供给读者一些用来优化梯度下降的不同算法的效果的直观展示，并希望能够帮助读者在实际问题中更合理的选择和使用梯度下降类算法。第2节，我们首先考察一下梯度下降算法的几种类型。然后在第3节中简单总结在使用中容易遇到的问题。接下来在第4节中，我们介绍最常用的几种算法的动机以及其数学实现，并在第5节中简单看一下在并行和分布式环境中，这些算法和框架是如何来优化梯度下降算法的。最后在第6节中，我们介绍其他一些可以用来优化梯度下降类算法的策略。

Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters. The learning rate η determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley.[8]

梯度下降法的基本思想是向着目标函数 $J(\theta)$ 梯度的反方向更新模型参数 $\theta \in \mathbb{R}^d$，并以此达到最小化目标函数的目的. 并通过定义学习率 η 来决定每一次参数更新时步伐的大小。如果把目标函数图像看作一个山丘，梯度下降法就是沿着山坡最陡峭的方向逐级下降直至山谷的过程。[9]

2 Gradient descent variants

There are three variants of gradient descent, which differ in how much data we use to compute the gradient of the objective function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

针对于在计算目标函数梯度时所使用数据量的不同, 梯度下降分为三种类型. 具体操作时，我们在参数更新的准确率和执行更新所消耗的时间之间进行权衡，来选用最合适的类型。

2.1 批梯度下降

Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters θ for the entire training dataset:

普通 (vanilla) 梯度下降, 又称为批梯度下降, 在整个数据集上计算损失函数关于参数 θ 的梯度:

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

As we need to calculate the gradients for the whole dataset to perform just *one* update, batch gradient descent can be very slow and is intractable for datasets that do not fit in memory. Batch gradient descent also does not allow us to update our model *online*, i.e. with new examples on-the-fly.

由于在批梯度下降中为了执行一次更新, 我们需要在整个数据集上计算梯度, 因此这将会是很慢的, 并且内存也是一个棘手的问题. 批梯度下降同样也不适用于在线更新我们的模型, 例如使用新的样例 on-the-fly.

In code, batch gradient descent looks something like this:

在代码中, 批梯度下降形如此:

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

[8] If you are unfamiliar with gradient descent, you can find a good introduction on optimizing neural networks at `http://cs231n.github.io/optimization-1/`.

[9] 对于梯度下降不熟悉的同学可以参考`http://cs231n.github.io/optimization-1/`.

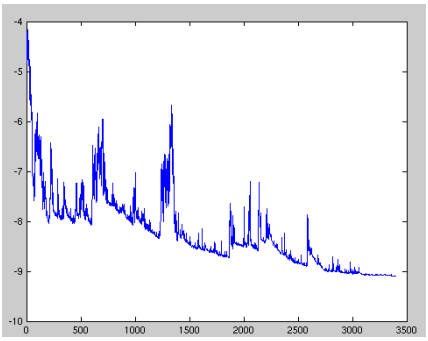


Figure 1: SGD fluctuation (Source: Wikipedia)

For a pre-defined number of epochs, we first compute the gradient vector `params_grad` of the loss function for the whole dataset w.r.t. our parameter vector `params`. Note that state-of-the-art deep learning libraries provide automatic differentiation that efficiently computes the gradient w.r.t. some parameters. If you derive the gradients yourself, then gradient checking is a good idea.[10]

对于预先确定的迭代次数`nb_epochs`，在每一次迭代中，我们首先在整个数据集上计算损失函数关于模型参数向量`params`的梯度`params_grad`. 最先进的深度学习代码库都会提供计算梯度的工具，但是如果使用者使用自己的梯度计算工具的话，最好使用梯度检查[11]确保不会出错。

We then update our parameters in the direction of the gradients with the learning rate determining how big of an update we perform. Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

然后我们向着梯度相反的方向更新参数，并利用学习率 (`learning rate`) 来决定更新步伐的大小。批梯度下降对于凸函数确保收敛于全局最小值，而对于非凸函数确保收敛于局部最小值。

2.2 随机梯度下降法

Stochastic gradient descent (SGD) in contrast performs a parameter update for *each* training example $x^{(i)}$ and label $y^{(i)}$:

与批梯度下降不同，随机梯度下降法 (SGD) 每一次参数更新时只使用一个样本 $x^{(i)}$ 以及其对应的标注 $y^{(i)}$:

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

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online.

在样本数量极大的时候，批梯度下降法在每一次参数更新前，要对大量相似的样本计算其损失函数的梯度，这造成很多冗余的计算。SGD 有效避免了这种冗余，所以他通常比较快，而且适用于样本的在线更新 (对于在线添加的样本，可以方便的直接带入迭代之中)

SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in Figure 1.

如图1所示，SGD 频繁更新参数，并使得目标函数值在剧烈震荡中逐步下降。

While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On

[10] Refer to `http://cs231n.github.io/neural-networks-3/` for some great tips on how to check gradients properly.

[11] 梯度检查请参考 `http://cs231n.github.io/neural-networks-3/`

the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively. Its code fragment simply adds a loop over the training examples and evaluates the gradient w.r.t. each example. Note that we shuffle the training data at every epoch as explained in Section 6.1.

相对比于批梯度下降法在参数空间中收敛于局部最小值的的表现来说，SGD 的高震荡性，一方面使迭代有可能跳出当前的局部最小值并找到更好的局部最小值，另一方面却因为过于活跃而使得收敛变得异常困难。但是，已知的结果显示，只要在迭代中逐渐缩小学习率，那么 SGD 也会展现出和批梯度下降法类似的收敛效果——分别在凸函数和非凸函数的情形下收敛于整体极小值和局部极小值。在代码上，仅仅是关于训练样本增加了一个子循环来计算关于每一个样本的梯度。请注意我们在每一个外层迭代开始时随机打乱了样本顺序，关于这种技巧我们在第6.1节进行详细介绍。

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

2.3 小批量梯度下降

Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples:

最后小批量梯度下降法集中了前述两种方法的优点，每一次参数更新时，都使用一个含有 n 个样本的小批量来计算损失函数的剃度。

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

This way, it a) reduces the variance of the parameter updates, which can lead to more stable convergence; and b) can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient. Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used. Note: In modifications of SGD in the rest of this post, we leave out the parameters $x^{(i:i+n)}; y^{(i:i+n)}$ for simplicity.

这种方法,a) 降低了 SGD 中参数更新的剧烈震荡，所以收敛效果更好, b) 可以很高效的利用目前最先进的深度学习代码库中那些已经得到了高度优化的矩阵运算工具。通常，根据不同的应用场景，小批量的大小取在 50 到 256 之间。目前在神经网络的训练中经常使用小批量梯度下降法，并习惯性的称之为随机梯度下降法 (SGD)。注：为了记号方便，在下文 SGD 改进版本的公式书写中，我们将省略 $x^{(i:i+n)}; y^{(i:i+n)}$ 。

In code, instead of iterating over examples, we now iterate over mini-batches of size 50:

在下面的代码中，与 SGD 每次使用一个样本相比而言，小批量梯度下降法每次使用一个大小为 50 的小批量。

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

3 问题与挑战

Vanilla mini-batch gradient descent, however, does not guarantee good convergence, but offers a few challenges that need to be addressed:

然而普通的小批量梯度下降法的收敛性仍然不太好，而且有如下的一些问题需要解决:

- Choosing a proper learning rate can be difficult. A learning rate that is too small leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.
- 很难选取一个合适的学习率。过小的学习率使得收敛极其缓慢，而过大的学习率会使损失函数在极小值附近剧烈震荡，甚至导致发散。
- Learning rate schedules [17] try to adjust the learning rate during training by e.g. annealing, i.e. reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics [4].
- 预定学习率衰减 [17] 的方法可以在训练过程中调整学习率，比如按照一开始定义好的计划逐渐削减学习率或者当相邻的两次迭代中目标函数的变化小于某一个阈值时减小学习率。但是这些计划和阈值都必须要在训练前就定义好，所以无法达到为训练样本量身定做的效果。
- Additionally, the same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.
- 另外，所有的参数更新都使用了同样的学习率。如果数据具有稀疏性而且不同的特征出现的频率有很大差异时，我们更希望对那些极少出现的特征使用较大的学习率，而不是对大家一视同仁。
- Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima. Dauphin et al. [5] argue that the difficulty arises in fact not from local minima but from saddle points, i.e. points where one dimension slopes up and another slopes down. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.
- 另一个重要的问题是在神经网络的训练中，经常会遇到极为非凸的目标函数，此时很难避免参数被困在一些次优的局部极小值附近。Dauphin et al. [5] 提出这类问题主要发生在鞍点附近。这些鞍点经常环绕着一些平原区域，而在这些区域中，损失函数的梯度很接近于零，所以 SGD 很难从中逃脱。

4 随机梯度下降优化算法

In the following, we will outline some algorithms that are widely used by the Deep Learning community to deal with the aforementioned challenges. We will not discuss algorithms that are infeasible to compute in practice for high-dimensional data sets, e.g. second-order methods such as Newton's method[12].

下面我们介绍一些优化梯度下降算法，他们经常在深度学习中被用来处理上节所提到的问题和挑战。在此我们不讨论那些不适用于高维数据的方法，比如类似于牛顿法[13]的二阶算法。

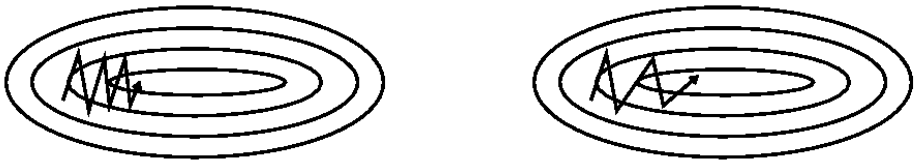
4.1 动量法

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another [19], which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum as in Figure 2a.

SGD 那种在一个方向上很陡峭而在其他方向上比较平坦，形如峡谷的函数上表现差强人意 [19]，而这种形状在局部极值附近却比较常见。在这种情况下，SGD 经常如下图所描述的那样，在峡谷的两边震荡而不能顺利的沿着峡谷方向直接前往最优点。

[12] https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization

[13] https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization



(a) SGD without momentum

(b) SGD with momentum

Figure 2: Source: Genevieve B. Orr

Momentum [16] is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Figure 2b. It does this by adding a fraction γ of the update vector of the past time step to the current update vector[14]

如图2b所示，动量法 [16] 可以通过消减这种无谓的震动来加速 SGD。动量法的基本原理是在当前的参数更新向量上添加之前一次参数更新向量的一部分[15]。

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

The momentum term γ is usually set to 0.9 or a similar value.

动量项 γ 通常被定在 0.9 左右。

Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity, if there is air resistance, i.e. $\gamma < 1$). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

动量法本质上就像是从山顶推一个球下山。球在向下滚动的过程中积累动量速度越来越快(如果存在空气阻力时，也就是 $\gamma < 1$ 的情况，最终会达到平衡速度)。我们的参数更新也是同样的情况: 动量项在那些梯度符号比较一致的方向上不断积累，而在那些梯度经常改变符号的方向上相互抵销，这帮助我们降低震荡从而加快迭代收敛速度。

4.2 Nesterov accelerated gradient

However, a ball that rolls down a hill, blindly following the slope, is highly unsatisfactory. We would like to have a smarter ball, a ball that has a notion of where it is going so that it knows to slow down before the hill slopes up again.

但是一个从山顶滚下越滚越快的球并不能满足我们的需求，我们希望这个球更加聪明一些，使得它在到达底部并再次上升之前就开始自动减速。

Nesterov accelerated gradient (NAG) [13] is a way to give our momentum term this kind of prescience. We know that we will use our momentum term $\gamma\, v_{t-1}$ to move the parameters θ. Computing $\theta - \gamma\, v_{t-1}$ thus gives us an approximation of the next position of the parameters (the gradient is missing for the full update), a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

Nesterov accelerated gradient (NAG) [13] 就可以赋予我们的动量项这样的先见之明。因为我们知道在第 t 次更新中将会用到动量项 $\gamma\, v_{t-1}$, 所以 $\theta - \gamma\, v_{t-1}$ 就给出了参数向量下一个位置的粗略估计。所以我们与其在 θ 处计算梯度，还不如在这个粗略估计的下一个位置计算梯度。

[14] Some implementations exchange the signs in the equations.

[15] 有些算法实现中会采取不同的符号.

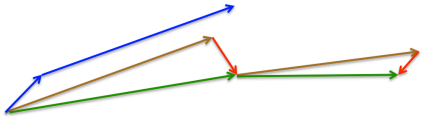


Figure 3: Nesterov update (Source: G. Hinton's lecture 6c)

$$
\begin{aligned}
v_t &= \gamma\, v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1}) \\
\theta &= \theta - v_t
\end{aligned}
\tag{5}
$$

Again, we set the momentum term γ to a value of around 0.9. While Momentum first computes the current gradient (small blue vector in Figure 3) and then takes a big jump in the direction of the updated accumulated gradient (big blue vector), NAG first makes a big jump in the direction of the previous accumulated gradient (brown vector), measures the gradient and then makes a correction (green vector). This anticipatory update prevents us from going too fast and results in increased responsiveness, which has significantly increased the performance of RNNs on a number of tasks [2].[16]

我们仍然把 γ 定为 0.9 左右。普通的动量法先计算当前的梯度 (图3中的短蓝向量)，然后沿着累积梯度方向做一个大幅度的参数更新 (图中的长蓝向量)。NAG 首先在前一次的累积梯度方向做一个大幅更新 (棕色向量)，然后再计算梯度，并进行修正 (绿色向量)。这种有先见之明的更新方式，可以避免我们更新的幅度过大。这种方法在若干个例子中都显著提升了 RNN 网络的表现 [2]。[17]

Now that we are able to adapt our updates to the slope of our error function and speed up SGD in turn, we would also like to adapt our updates to each individual parameter to perform larger or smaller updates depending on their importance.

现在我们可以让我们的参数更新自动适应误差函数的斜率，并确实加速了 SGD 的收敛。但是另一方面，我们也希望能够让我们的参数更新适应每一个参数，根据他们各自的重要性来决定每一个更新的幅度大小。

4.3 Adagrad

Adagrad [7] is an algorithm for gradient-based optimization that does just this: It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data. Dean et al. [6] have found that Adagrad greatly improved the robustness of SGD and used it for training large-scale neural nets at Google, which – among other things – learned to recognize cats in Youtube videos[18]. Moreover, Pennington et al. [15] used Adagrad to train GloVe word embeddings, as infrequent words require much larger updates than frequent ones.

Adagrad [7] 恰是如此：他是一种可以根据参数的重要性来调整学习率的算法，对于常常更新的参数使用较小的学习率，而对于很少得到更新的参数使用较大的学习率。所以这种方法很适合于稀疏性很强的数据集。Dean et al. [6] 发现 Adarad 可以大大提升 SGD 的鲁棒性，于是将其应用于 Google 的大规模神经网络训练中，并和其他的算法一起实现了在 YouTube 视频中对猫的自动识别[19]。另外 Pennington et al. [15] 使用 Adagrad 对 GloVe 词语嵌入进行训练，因为在这种情况下不常出现的词语需要比经常出现的词语更大的学习率。

Previously, we performed an update for all parameters θ at once as every parameter θ_i used the same learning rate η. As Adagrad uses a different learning rate for every parameter θ_i at every time step t, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we set $g_{t,i}$ to be the gradient of the objective function w.r.t. to the parameter θ_i at time step t:

[16] Refer to `http://cs231n.github.io/neural-networks-3/` for another explanation of the intuitions behind NAG, while Ilya Sutskever gives a more detailed overview in his PhD thesis [18].

[17] 在文章`http://cs231n.github.io/neural-networks-3/`中描述了 NAG 的另一种解释，Ilya Sutskever 在他的博士论文中也给出了详细的介绍 [18].

[18] `http://www.wired.com/2012/06/google-x-neural-network/`

[19] `http://www.wired.com/2012/06/google-x-neural-network/`

前面的方法中，我们每一次都用相同的学习率 η 对所有参数 θ_i 进行同步更新。而 Adagrad 在每一次更新中都对不同的参数采取不同的学习率，我们首先单独考察每一个参数的更新，然后再将其向量化。简单来说，我们用 $g_{t,i}$ 表示目标函数在第 t 步迭代时，关于参数 θ_i 的梯度。

$$g_{t,i} = \nabla_\theta J(\theta_i) \tag{6}$$

The SGD update for every parameter θ_i at each time step t then becomes:

SGD 对于参数 θ_i 在第 t 步的更新如下所示:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i} \tag{7}$$

In its update rule, Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i:

而 Adagrad 在每一次更新时，都针对每一个参数 θ_i 利用这个参数以往的方向导数值来动态调整其专属的学习率 η:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i} \tag{8}$$

$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t[20], while ϵ is a smoothing term that avoids division by zero (usually on the order of $1e-8$). Interestingly, without the square root operation, the algorithm performs much worse.

此处 $G_t \in \mathbb{R}^{d \times d}$ 是一个对角矩阵，其每一个对角元素 i, i 是在以往更新中到第 t 步为止，目标函数在 θ_i 方向上的导数的平方和。[21], 公式中的 ϵ 是一个平滑项，用来避免分母为零的情况发生，ϵ 通常取值在 $1e-8$ 量级上。有意思的是，如果去掉取平方根那一步，这个算法的表现就会立刻变差很多。

As G_t contains the sum of the squares of the past gradients w.r.t. to all parameters θ along its diagonal, we can now vectorize our implementation by performing an element-wise matrix-vector multiplication $\odot$ between G_t and g_t:

因为 G_t 包含了所有参数的以往方向导数的平方和，我们可以很容易的通过 G_t 和 g_t 之间元素级的矩阵——向量乘法 $\odot$ 来写出这个算法的向量形式:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t. \tag{9}$$

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that.

Adagrad 方法的一个主要的好处是它不需要手动调整学习率。大部分算法实现直接使用默认值 0.01 就好了。

Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. The following algorithms aim to resolve this flaw.

Adagrad 方法的主要弱点是它会在分母上不断累积梯度平方和: 因为每次加入的都是正数，所以分母随着训练步骤增加不断变大。这就使得学习率不断衰减，直到接近无限小（指趋近于 0），以至于参数更新无法继续前进。下面这个算法就是针对于此的一个改进。

[20] Duchi et al. [7] give this matrix as an alternative to the *full* matrix containing the outer products of all previous gradients, as the computation of the matrix square root is infeasible even for a moderate number of parameters d.

[21] Duchi et al. [7] 使用这个对角矩阵取代了完整的以往参数更新的协方差矩阵，因为对整个高维矩阵进行开方运算不划算。

4.4 Adadelta

Adadelta [21] is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size w.

Adadelta [21] 是针对 Adagrad 方法里面不断单调下降的学习率问题的一个改进版。Adadelta 方法采用指数衰减移动平均的方式来取代 Adagrad 里面将以往全部梯度值都平等纳入平方和的做法。

Instead of inefficiently storing w previous squared gradients, the sum of gradients is recursively defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at time step t then depends (as a fraction γ similarly to the Momentum term) only on the previous average and the current gradient:

如果用 $E[g^2]_t$ 表示在第 t 步时 g^2 的移动平均，那么我们采取如下迭代的方法对它进行更新，它只取决于前一项的平均和当前的梯度（其中 γ 类似于动量项）。

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

We set γ to a similar value as the momentum term, around 0.9. For clarity, we now rewrite our vanilla SGD update in terms of the parameter update vector $\Delta\theta_t$:

类似于动量法的做法，我们把 γ 定义为 0.9 左右。现在我们使用 $\Delta\theta_t$ 来简化 SGD 的更新算法的表达:

$$\begin{aligned} \Delta\theta_t &= -\eta \cdot g_{t,i} \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{11}$$

The parameter update vector of Adagrad that we derived previously thus takes the form:

那么，前面介绍的 Adagrad 算法就如下形式:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t \tag{12}$$

We now simply replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$:

只要在 Adagrad 算法里面将 G_t 换成指数衰减移动平均 $E[g^2]_t$，我们就得到了 Adadelta 算法的更新法则:

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \tag{13}$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

因为分母是梯度的一个均方根 (RMS)，我们可以将上面的算法简写为:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t \tag{14}$$

The authors note that the units in this update (as well as in SGD, Momentum, or Adagrad) do not match, i.e. the update should have the same hypothetical units as the parameter. To realize this, they first define another exponentially decaying average, this time not of squared gradients but of squared parameter updates:

Adadelta 的作者们注意到参数更新值 $\Delta\theta$ 的单位和参数 θ 本身的单位不匹配 (SGD，动量法，以及 Adagrad 也存在同样的问题)。为了解决这个问题，他们首先定义了一个新的指数指数衰减均值，这一次，他们使用的是参数更新值的平方，而不是梯度的平方。

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2 \tag{15}$$

The root mean squared error of parameter updates is thus:

进而得到参数更新值的均方根误差:

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon} \tag{16}$$

Since $RMS[\Delta\theta]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous time step. Replacing the learning rate η in the previous update rule with $RMS[\Delta\theta]_{t-1}$ finally yields the Adadelta update rule:

由于在 t 次更新前还不知道 $RMS[\Delta\theta]_t$ 的值，我们使用前一次的 $RMS[\Delta\theta]_{t-1}$ 对其进行估计。此时把学习率 η 替换成利用 $RMS[\Delta\theta]_{t-1}$ 定义的动态学习率就得到了如下的参数更新法:

$$\begin{aligned} \Delta\theta_t &= -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \\ \theta_{t+1} &= \theta_t + \Delta\theta_t \end{aligned} \tag{17}$$

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

因为学习率会得到自动更新，所以 Adadelta 方法的另一个特点 (优点) 是它甚至都不需要一个默认的学习率。

4.5 RMSprop

RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class[22].

RMSprop 是一个没有正式发表的方法。它来自 Geoff Hinton 课程[23]的第 6e 节的内容。

RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad's radically diminishing learning rates. RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

RMSprop 和 Adadelta 是为了解决 Adagrad 方法里面学习率衰减过快的问题而由不同的作者分别独立建立的两种方法。实际上 RMSprop 恰好和 Adadelta 方法的第一种形式完全相同 (RMSprop 不关注参数更新的单位问题)。

$$\begin{aligned} E[g^2]_t &= 0.9E[g^2]_{t-1} + 0.1g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned} \tag{18}$$

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

RMSprop 在学习率上除以了一个梯度的均方根。Hinton 建议 γ 取值为 0.9, 而默认学习率取为 0.001.

[22] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[23] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

4.6 Adam

Adaptive Moment Estimation (Adam) [9] is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum:

Adaptive Moment Estimation (Adam) [9] 是另一种针对每一个参数调整学习率的方法。这种方法不但像 Adadelta 和 RMSprop 一样记录梯度平方的指数衰减均值 v_t，它还像动量法一样记录梯度本身的指数衰减均值 m_t。

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \end{aligned} \tag{19}$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

m_t 和 v_t 分别是梯度的一阶 (均值) 和二阶 (方差) 矩估计，所以称为 Adaptive Moment Estimation(矩估计调整法)。因为 m_t 和 v_t 都初始化为 0 向量，所以 Adam 的作者发现，这两者在训练中都存在向零方向的偏差的问题，在训练初期这个问题尤为严重。

They counteract these biases by computing bias-corrected first and second moment estimates:

所以 Adam 的作者们使用下面的无偏化估计来取代上述的矩估计:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{20}$$

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

于是用类似于 Adadelta 和 RMSprop 的方法，我们得到了如下 Adam 的参数更新法则:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{21}$$

The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.

Adam 的作者们提出 β_1,β_2, 和 ϵ 分别取为 0.9,0.999,10^{-8}. 他们的数值结果显示 Adam 在实际应用中运行良好，而且比其它有动态调整功能的算法表现得更好。

4.7 算法可视化

The following two figures provide some intuitions towards the optimization behaviour of the presented optimization algorithms.[24]

下面的两个图中给出了前述几种算法的示意，希望读者可以从中得到一些关于这些算法的直观印象。[25]

In Figure 4a, we see the path they took on the contours of a loss surface. All started at the same point and took different paths to reach the minimum. Note that Adagrad, Adadelta, and RMSprop

[24] Also have a look at `http://cs231n.github.io/neural-networks-3/` for a description of the same images by Karpathy and another concise overview of the algorithms discussed.

[25] 感兴趣者可上 `http://cs231n.github.io/neural-networks-3/` 查看。

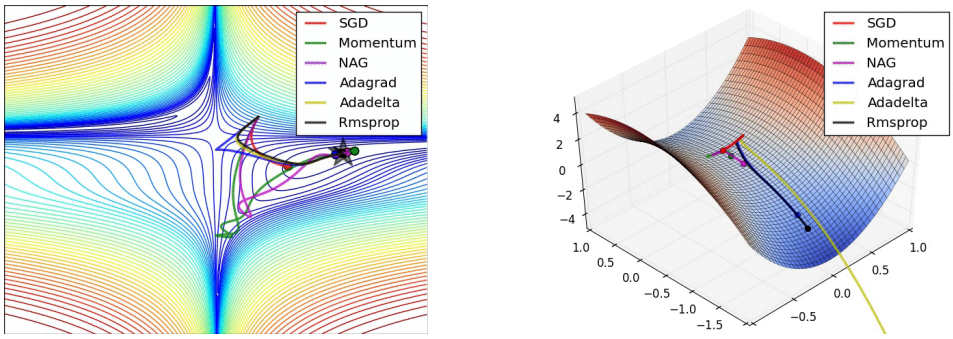
headed off immediately in the right direction and converged similarly fast, while Momentum and NAG were led off-track, evoking the image of a ball rolling down the hill. NAG, however, was able to correct its course sooner due to its increased responsiveness by looking ahead and headed to the minimum.

在图4a里，我们可以看到几种算法都从同一点出发，然后沿着不同的路径收敛到极小值。从中可以看出 Adagrad, Adadelta, and RMSprop 很快就找到了正确的方向，并且收敛速度都差不多快。但是模拟球滚下山坡的动量法与 NAG 一开始都遭到了误导，但是 NAG 的预见性帮助它更早的走回了正路。

Figure 4b shows the behaviour of the algorithms at a saddle point, i.e. a point where one dimension has a positive slope, while the other dimension has a negative slope, which pose a difficulty for SGD as we mentioned before. Notice here that SGD, Momentum, and NAG find it difficulty to break symmetry, although the latter two eventually manage to escape the saddle point, while Adagrad, RMSprop, and Adadelta quickly head down the negative slope, with Adadelta leading the charge.

Notice here that SGD, Momentum, and NAG find it difficulty to break symmetry, although the latter two eventually manage to escape the saddle point, while Adagrad, RMSprop, and Adadelta quickly head down the negative slope, with Adadelta leading the charge.

图4b展示了算法们在鞍点附近的表现, 如前所述，这种鞍点是 SGD 的一大心病. 我们可以看到，SGD, 动量法和 NAG 都遭遇了困难，其中动量法和 NAG 经过一番调整之后终于走出了误区。相比较而言 Adagrad, RMSprop, 与 Adadelta 法都很快就走出了鞍点区域，其中 Adadelta 走的最快。



(a) SGD optimization on loss surface contours

(b) SGD optimization on saddle point

Figure 4: Source and full animations: Alec Radford

As we can see, the adaptive learning-rate methods, i.e. Adagrad, Adadelta, RMSprop, and Adam are most suitable and provide the best convergence for these scenarios.

从这两张图中我们看到，动态调整学习率的几种方法 (Adagrad, Adadelta, RMSprop, 以及 Adam) 很适应这几种情况，而且表现良好，收敛迅速。

4.8 到底应该选取那个优化算法呢?

So, which optimizer should you use? If your input data is sparse, then you likely achieve the best results using one of the adaptive learning-rate methods. An additional benefit is that you will not need to tune the learning rate but will likely achieve the best results with the default value.

所以到底应该选择那个优化算法呢? 如果输入数据有稀疏性，那么可能这些动态调整学习率的算法会比较好。这些方法的另一个优点是，使用者不需要调整学习率，因为它们一般情况下在默认设定下就可以达到最佳效果。

In summary, RMSprop is an extension of Adagrad that deals with its radically diminishing learning rates. It is identical to Adadelta, except that Adadelta uses the RMS of parameter updates in the numinator update rule. Adam, finally, adds bias-correction and momentum to RMSprop. Insofar, RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances. Kingma et al. [9] show that its bias-correction helps Adam slightly outperform RMSprop towards the end of optimization as gradients become sparser. Insofar, Adam might be the best overall choice.

作为总结，RMSprop 是 Adagrad 的一个扩展算法，它主要解决的是 Adagrad 学习率单调衰减的问题。Adadelta 与 RMSprop 几乎一样，不同之处在于，Adadelta 在学习率的分子上使用了参数更新的均方根误差。而 Adam 则在 RMSprop 的基础之上加入了偏差修正，以及动量项。实际上 RMSprop, Adadelta, 和 Adam 是很类似的算法，他们也在类似的问题上表现差不多一样良好。Kingma et al [9] 指出 Adam 的偏差修正使得它在优化收敛的最后阶段，梯度值比较稀疏的时候表现略为优于 RMSprop. 所以目前为止，Adam 可能是这几个中最好的选择。

Interestingly, many recent papers use vanilla SGD without momentum and a simple learning rate annealing schedule. As has been shown, SGD usually achieves to find a minimum, but it might take significantly longer than with some of the optimizers, is much more reliant on a robust initialization and annealing schedule, and may get stuck in saddle points rather than local minima. Consequently, if you care about fast convergence and train a deep or complex neural network, you should choose one of the adaptive learning rate methods.

有意思的是，最近很多研究论文仅仅使用最原始不包含动量项的 SGD 以及简单的学习率衰减策略。如上面所述 SGD 一般可以找到极小值，但是可能比前述的优化算法慢很多，而且它还非常依赖于初始值与学习率衰减的选取，并且可能会卡在鞍点附近而找不到局部最小值。所以，如果研究者关心收敛速度，或者训练的是深度较大和比较复杂的神经网络时，就应当选取上述的动态调整学习率的模型。

5 并行与分布式 SGD

Given the ubiquity of large-scale data solutions and the availability of low-commodity clusters, distributing SGD to speed it up further is an obvious choice. SGD by itself is inherently sequential: Step-by-step, we progress further towards the minimum. Running it provides good convergence but can be slow particularly on large datasets. In contrast, running SGD asynchronously is faster, but suboptimal communication between workers can lead to poor convergence. Additionally, we can also parallelize SGD on one machine without the need for a large computing cluster. The following are algorithms and architectures that have been proposed to optimize parallelized and distributed SGD.

鉴于大数据分析的普遍性以及低廉的聚类方法的可用性，分布式 SGD 的加速方法会是一个更加明智的选择。SGD 本身就是一个按部就班的方法，一步一步逐渐到达最低值。SGD 的使用可以提供很好的收敛性，但是效率很低，尤其是在大数据集上。SGD 的异步使用速度回更快，但是 worker 之间的通信问题将会导致低的收敛性。除此之外，我们还可以在一台电脑上平行计算 SGD，以下就是曾被提出用来优化并行与分布式 SGD 计算的算法和框架。

5.1 Hogwild!

Niu et al. [14] introduce an update scheme called Hogwild! that allows performing SGD updates in parallel on CPUs. Processors are allowed to access shared memory without locking the parameters. This only works if the input data is sparse, as each update will only modify a fraction of all parameters. They show that in this case, the update scheme achieves almost an optimal rate of convergence, as it is unlikely that processors will overwrite useful information.

Niu et al. [14] 介绍了一种叫做 Hogwild! 的更新算法. 它使得 SGD 可以在 CPU 集群中并行进行。处理器可以在不锁定参数的情况下读取共享内存。因为这种更新只能改动一部分参数，所以这种方法只在输入数据比较稀疏的时候有效。他们的研究表明，这种情况下，Hogwild! 方法可以达到几乎最优的收敛速度，因为在更新过程中，处理器不会轻易改写有用的信息。

5.2 Downpour SGD

Downpour SGD is an asynchronous variant of SGD that was used by Dean et al. [6] in their DistBelief framework (the predecessor to TensorFlow) at Google. It runs multiple replicas of a model in parallel on subsets of the training data. These models send their updates to a parameter server, which is split across many machines. Each machine is responsible for storing and updating a fraction of the model's parameters. However, as replicas don't communicate with each other e.g. by sharing weights or updates, their parameters are continuously at risk of diverging, hindering convergence.

Downpour SGD 是由 Dean et al. [6] 在他们在 GOOGLE 的 DistBelief 架构 (TensorFlow 的前身) 下引进的 SGD 的一个并行变种。它把数据集分为若干子集合，并同时在每一个子集合

上训练模型。这些模型把他们的参数更新发送到一个分散在很多个机器上的参数服务器。每一个机器负责存储与更新一部分模型参数。但是因为每一个子集上的模型不与其它子集上的模型交换数据，以及参数更新，这种方法会一直面临模型参数发散的风险。

5.3 Delay-tolerant Algorithms for SGD

McMahan and Streeter [11] extend AdaGrad to the parallel setting by developing delay-tolerant algorithms that not only adapt to past gradients, but also to the update delays. This has been shown to work well in practice.

McMahan 与 Streeter [11] 通过建立 delay-tolerant 算法把 AdaGrad 扩展到并行架构。这种算法不但动态适应过去的参数更新值，而且也动态适应更新延迟。这个做法在实际操作中效果不错。

5.4 TensorFlow

TensorFlow[26] [1] is Google's recently open-sourced framework for the implementation and deployment of large-scale machine learning models. It is based on their experience with DistBelief and is already used internally to perform computations on a large range of mobile devices as well as on large-scale distributed systems. The distributed version, which was released in April 2016 recently in April 2016[27] relies on a computation graph that is split into a subgraph for every device, while communication takes place using Send/Receive node pairs.

TensorFlow[28] [1] 是 Google 最近开源的大规模机器学习研发平台。它的建立基于 Google 之前在 DistBelief 上的经验，并已经在 Google 内部的大规模移动设备以及大规模分布式系统中得到广泛应用。2016 年四月发布的分布式版本[29]，使用一个分布在各个设备中的计算图，使用发送和接受图节点的方式实现设备之间的数据沟通。

5.5 Elastic Averaging SGD

Zhang et al. [22] propose Elastic Averaging SGD (EASGD), which links the parameters of the workers of asynchronous SGD with an elastic force, i.e. a center variable stored by the parameter server. This allows the local variables to fluctuate further from the center variable, which in theory allows for more exploration of the parameter space. They show empirically that this increased capacity for exploration leads to improved performance by finding new local optima.

Zhang et al. [22] 提出 Elastic Averaging SGD (EASGD) 方法，这种方法用弹性的方式连接 SGD 的各个并行部分的参数，这种弹力通过一个存储在参数服务器中的参数来实现。这种方法允许局部参数获得更大的活动范围，这理论上增加了模型训练在参数空间上的探索能力。而作者们的数值试验结果显示，这种做法在寻找新的局部极值上确实表现良好。

6 其它优化 SGD 的方法

Finally, we introduce additional strategies that can be used alongside any of the previously mentioned algorithms to further improve the performance of SGD. For a great overview of some other common tricks, refer to [10].

最后我们来介绍其它一些与前述方法不同的作法来进一步优化 SGD。对于另一些在这里也没有涉及的技巧请参考 [10]。

6.1 Shuffling and Curriculum Learning

Generally, we want to avoid providing the training examples in a meaningful order to our model as this may bias the optimization algorithm. Consequently, it is often a good idea to shuffle the training data after every epoch.

[26] https://www.tensorflow.org/

[27] http://googleresearch.blogspot.ie/2016/04/announcing-tensorflow-08-now-with.html

[28] https://www.tensorflow.org/

[29] http://googleresearch.blogspot.ie/2016/04/announcing-tensorflow-08-now-with.html

一般来讲，我们不希望训练数据的特殊排序影响到优化算法的表现。所以，我们经常在每一次大循环后对训练数据进行洗牌重新排序。

On the other hand, for some cases where we aim to solve progressively harder problems, supplying the training examples in a meaningful order may actually lead to improved performance and better convergence. The method for establishing this meaningful order is called Curriculum Learning [3].

但是另一方面，在研究有些问题的时候，对训练数据进行适当排序会提升算法的表现和收敛的速度。这种对训练数据进行特殊排序的方法叫做 Curriculum Learning [3].

Zaremba and Sutskever [20] were only able to train LSTMs to evaluate simple programs using Curriculum Learning and show that a combined or mixed strategy is better than the naive one, which sorts examples by increasing difficulty.

Zaremba and Sutskever [20] 在训练 LSTMs 的时候发现必须使用 Curriculum Learning 才可以。而且他们指出，混合策略要优于把训练数据按照难易程度简单排序的单一的策略。

6.2 批规范化

To facilitate learning, we typically normalize the initial values of our parameters by initializing them with zero mean and unit variance. As training progresses and we update parameters to different extents, we lose this normalization, which slows down training and amplifies changes as the network becomes deeper.

为了改善训练效果，我们通常会在训练前对数据进行归一化处理 (零均值单位方差)。但是在训练过程中随着参数不断的更新，数据逐渐失去初始化时人为赋予的规则性，进而导致训练减速，这种现象在深度网络训练中尤为明显。

Batch normalization [8] reestablishes these normalizations for every mini-batch and changes are back-propagated through the operation as well. By making normalization part of the model architecture, we are able to use higher learning rates and pay less attention to the initialization parameters. Batch normalization additionally acts as a regularizer, reducing (and sometimes even eliminating) the need for Dropout.

批规范化 [8] 在每一次小批量计算之前重新进行规范化，并且确保改动进入模型训练的反向传播。批规范化可以通过在模型架构上的部分改动来实现，并令我们可以使用更大的学习率，而且不必太在意参数的初始化。批规范化还可以起到 regularizer 的作用，减少 (有时候甚至解除) 对 Dropout 的需求。

6.3 Early stopping

According to Geoff Hinton: "Early stopping (is) beautiful free lunch"[30]. You should thus always monitor error on a validation set during training and stop (with some patience) if your validation error does not improve enough.

按照 Geoff Hinton 的话说:"Early stopping (是) 免费午餐"[31]. 你总是可以通过在训练过程中监控模型在验证数据上的误差，并在每一步训练后误差的减小程度小于预设值时决定终止训练。

6.4 Gradient noise

Neelakantan et al. [12] add noise that follows a Gaussian distribution $N(0, \sigma_t^2)$ to each gradient update:

Neelakantan et al. [12] 在梯度更新中加入符合高斯分布的误差 $N(0, \sigma_t^2)$,

$$g_{t,i} = g_{t,i} + N(0, \sigma_t^2) \tag{22}$$

They anneal the variance according to the following schedule:

[30]NIPS 2015 Tutorial slides, slide 63, `http://www.iro.umontreal.ca/~bengioy/talks/DL-Tutorial-NIPS2015.pdf`

[31]NIPS 2015 Tutorial slides, slide 63, ă`http://www.iro.umontreal.ca/~bengioy/talks/DL-Tutorial-NIPS2015.pdf`

并通过下面的步骤逐渐衰减这个高斯分布的误差，

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma} \tag{23}$$

They show that adding this noise makes networks more robust to poor initialization and helps training particularly deep and complex networks. They suspect that the added noise gives the model more chances to escape and find new local minima, which are more frequent for deeper models.

方法的作者指出，加入这些噪音后，网络对于糟糕的初始化表现的更加稳定，并且尤其在深度和复杂的网络中表现良好。他们猜测，通过加入噪音，模型有更大的几率可以从深度模型里很常遇到的局部极小值中解脱出来。

7 结论

In this blog post, we have initially looked at the three variants of gradient descent, among which mini-batch gradient descent is the most popular. We have then investigated algorithms that are most commonly used for optimizing SGD: Momentum, Nesterov accelerated gradient, Adagrad, Adadelta, RMSprop, Adam, as well as different algorithms to optimize asynchronous SGD. Finally, we've considered other strategies to improve SGD such as shuffling and curriculum learning, batch normalization, and early stopping.

在本文中，我们首先考察了梯度法的三种类型，其中小批量随即梯度法是目前最常用的方法。然后我们研究了最常用的梯度优化方法:Momentum, Nesterov accelerated gradient, Adagrad, Adadelta, RMSprop, Adam, 以及梯度法的并行算法。最后，我们介绍了一下其它可以改善随机梯度法的几种做法包括：shuffling and curriculum learning, 批规范化, 以及 early stopping。

References

[1] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Man, Rajat Monga, Sherry Moore, Derek Murray, Jon Shlens, Benoit Steiner, Ilya Sutskever, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Oriol Vinyals, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow : Large-Scale Machine Learning on Heterogeneous Distributed Systems. 2015.

[2] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in Optimizing Recurrent Networks. 2012.

[3] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[4] C. Darken, J. Chang, and J. Moody. Learning rate schedules for faster stochastic gradient search. *Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop*, (September):1–11, 1992.

[5] 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. *arXiv*, pages 1–14, 2014.

[6] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large Scale Distributed Deep Networks. *NIPS 2012: Neural Information Processing Systems*, pages 1–11, 2012.

[7] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[8] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv preprint arXiv:1502.03167v3*, 2015.

[9] Diederik P. Kingma and Jimmy Lei Ba. Adam: a Method for Stochastic Optimization. *International Conference on Learning Representations*, pages 1–13, 2015.

[10] Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus Robert Müller. Efficient BackProp. *Neural Networks: Tricks of the Trade*, 1524:9–50, 1998.

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

[12] Arvind Neelakantan, Luke Vilnis, Quoc V. Le, Ilya Sutskever, Lukasz Kaiser, Karol Kurach, and James Martens. Adding Gradient Noise Improves Learning for Very Deep Networks. pages 1–11, 2015.

[13] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). *Doklady ANSSSR (translated as Soviet.Math.Docl.)*, 269:543–547.

[14] Feng Niu, Benjamin Recht, R Christopher, and Stephen J Wright. Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. pages 1–22, 2011.

[15] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global Vectors for Word Representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1532–1543, 2014.

[16] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks : the official journal of the International Neural Network Society*, 12(1):145–151, 1999.

[17] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[18] Ilya Sutskever. Training Recurrent neural Networks. *PhD thesis*, page 101, 2013.

[19] Richard S Sutton. Two problems with backpropagation and other steepest-descent learning procedures for networks, 1986.

[20] Wojciech Zaremba and Ilya Sutskever. Learning to Execute. pages 1–25, 2014.

[21] Matthew D. Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv preprint arXiv:1212.5701*, 2012.

[22] Sixin Zhang, Anna Choromanska, and Yann LeCun. Deep learning with Elastic Averaging SGD. *Neural Information Processing Systems Conference (NIPS 2015)*, pages 1–24, 2015.