梯度下降优化算法概述

> 原文作者简介：Sebastian Ruder 是我非常喜欢的一个博客作者，是 NLP 方向的博士生，目前供职于一家做 NLP 相关服务的爱尔兰公司 AYLIEN，博客主要是写机器学习、NLP和深度学习相关的文章。

> - 本文原文是 An overview of gradient descent optimization algorithms，同时作者也在 arXiv 上发了一篇同样内容的 论文。
> - 本文结合了两者来翻译，但是阅读原文我个人建议读博客中的，感觉体验更好点。
> - 文中括号中或者引用块中的 *斜体字* 为对应的英文原文或者我自己注释的话（会标明 `译者注` ），否则为原文中本来就有的话。
> - 为方便阅读，我在引用序号后加了所引用论文的题目，用 *斜体字* 表示，例如 Learning rate schedules [11，*A stochastic approximation method*] 。
> - 水平有限，如有错误欢迎指出。翻译尽量遵循原文意思，但不意味着逐字逐句。

Abstract

梯度下降算法虽然最近越来越流行，但是始终是作为一个「黑箱」在使用，因为对他们的优点和缺点的实际解释（practical explainations）很难实现。这篇文章致力于给读者提供这些算法工作原理的一个直观理解。在这篇概述中，我们将研究梯度下降的不同变体，总结挑战，介绍最常见的优化算法，介绍并行和分布式设置的架构，并且也研究了其他梯度下降优化策略。

Introduction

梯度下降是最流行的优化算法之一，也是目前优化神经网络最常用的算法。同时，每一个最先进的深度学习库都包含了梯度下降算法的各种变体的实现（例如 lasagne，caffe，keras）。然而始终是作为一个「黑箱」在使用，因为对他们的优点和缺点的实际解释很难实现。这篇文章致力于给读者提供这些算法工作原理的一个直观理解。我们首先介绍梯度下降的不同变体，然后简单总结下在训练中的挑战。接着，我们通过展示他们解决这些挑战的动机以及如何推导更新规则来介绍最常用的优化算法。我们也会简要介绍下在并行和分布式架构中的梯度下降。最后，我们会研究有助于梯度下降的其他策略。

梯度下降是一种最小化目标函数 $J(\theta)$ 的方法，其中 $\theta \in \mathbb{R}^d$ 是模型参数，而最小化目标函数是通过在其关于 θ 的 梯度 $\nabla_\theta J(\theta)$ 的相反方向来更新 θ 来实现的。而学习率（learning rate）则决定了在到达（局部）最小值的过程中每一步走多长。换句话说，我们沿着目标函数的下坡方向来达到一个山谷。如果你对梯度下降不熟悉，你可以在 这里 找到一个很好的关于优化神经网络的介绍。

Gradient descent variants

依据计算目标函数梯度使用的数据量的不同，有三种梯度下降的变体。根据数据量的大小，我们在参数更新的准确性和执行更新所需时间之间做了一个权衡。

Batch gradient descent

标准的梯度下降，即批量梯度下降（batch gradient descent）（ *译者注：以下简称 BGD* ），在整个训练集上计算损失函数关于参数 θ 的梯度。

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$$

由于为了一次参数更新我们需要在整个训练集上计算梯度，导致 BGD 可能会非常慢，而且在训练集太大而不能全部载入内存的时候会很棘手。BGD 也不允许我们在线更新模型参数，即实时增加新的训练样本。

下面是 BGD 的代码片段：

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

其中 `nb_epochs` 是我们预先定义好的迭代次数（epochs），我们首先在整个训练集上计算损失函数关于模型参数 `params` 的梯度向量 `params_grad`。其实目前最新的深度学习库都已经提供了关于一些参数的高效自动求导。如果你要自己求导求梯度，那你最好使用梯度检查（gradient checking），在 这里 查看关于如何进行合适的梯度检查的提示。

然后我们在梯度的反方向更新模型参数，而学习率决定了每次更新的步长大小。BGD 对于凸误差曲面（convex error surface）保证收敛到全局最优点，而对于非凸曲面（non-convex surface）则是局部最优点。

Stochastic gradient descent

随机梯度下降（ *译者注：以下简称 SGD* ）则是每次使用一个训练样本 $x^{(i)}$ 和标签 $y^{(i)}$ 进行一次参数更新。

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

BGD 对于大数据集来说执行了很多冗余的计算，因为在每一次参数更新前都要计算很多相似样本的梯度。SGD 通过一次执行一次更新解决了这种冗余。因此通常 SGD 的速度会非常快而且可以被用于在线学习。SGD 以高方差的特点进行连续参数更新，导致目标函数严重震荡，如图 1 所示。

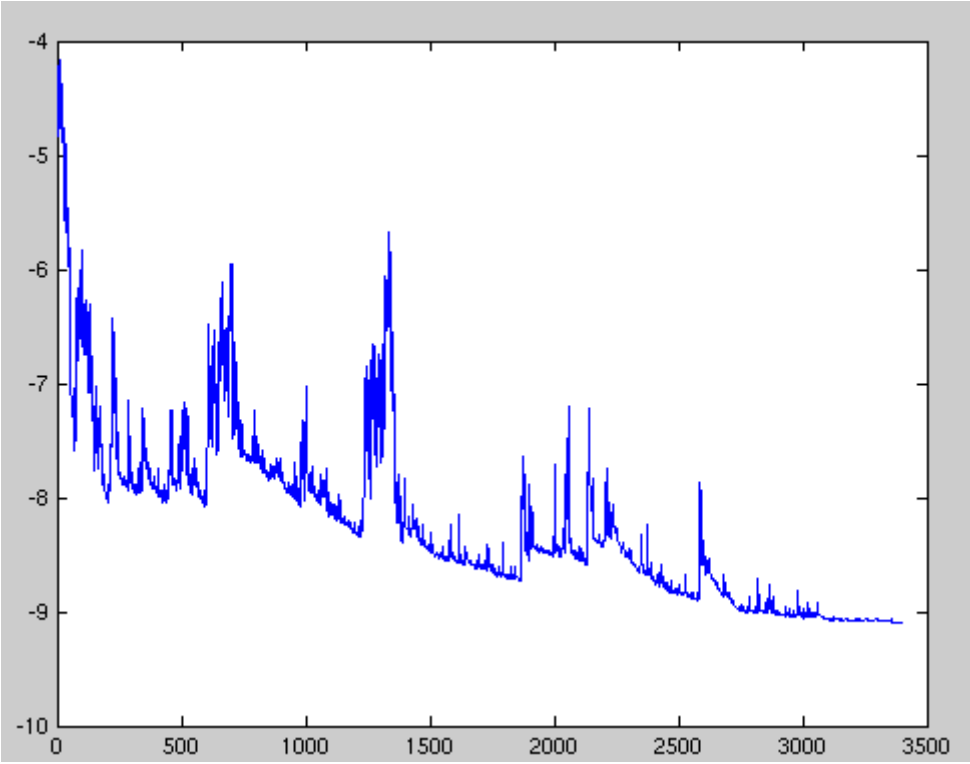


图 1：SGD 震荡，来自 Wikipedia

BGD 能够收敛到（局部）最优点，然而 SGD 的震荡特点导致其可以跳到新的潜在的可能更好的局部最优点。已经有研究显示当我们慢慢的降低学习率时，SGD 拥有和 BGD 一样的收敛性能，对于非凸和凸曲面几乎同样能够达到局部或者全局最优点。

代码片段如下，只是加了个循环和在每一个训练样本上计算梯度。注意依据 这里 的解释，我们在每次迭代的时候都打乱训练集。

```
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
```

Mini-batch gradient descent

Mini-batch gradient descent（ *译者注：以下简称 MBGD* ）则是在上面两种方法中采取了一个折中的办法：每次从训练集中取出 n 个样本作为一个 mini-batch，以此来进行一次参数更新。

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

这样做有两个好处：

- 减小参数更新的方差，这样可以有更稳定的收敛。
- 利用现在最先进的深度学习库对矩阵运算进行了高度优化的特点，这样可以使得计算 mini-batch 的梯度更高效。

通常来说 mini-batch 的大小为 50 到 256 之间，但是也会因为任务的差异而不同。MBGD 是训练神经网络时的常用方法，而且通常即使实际上使用的是 MBGD，也会使用 SGD 这个词来代替。注意：在本文接下来修改 SGD 时，为了简单起见我们会省略参数 $x^{(i:i+n)}; y^{(i:i+n)}$。

代码片段如下，我们每次使用 mini-batch 为 50 的样本集来进行迭代：

```
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
```

Challenges

标准的 MBGD 并不保证好的收敛，也提出了一下需要被解决的挑战：

- **选择一个好的学习率是非常困难的**。太小的学习率导致收敛非常缓慢，而太大的学习率则会阻碍收敛，导致损失函数在最优点附近震荡甚至发散。
- Learning rate schedules [11，*A stochastic approximation method*] 试图在训练期间调整学习率即退火（annealing），根据先前定义好的一个规则来减小学习率，或者两次迭代之间目标函数的改变低于一个阈值的时候。然而这些规则和阈值也是需要在训练前定义好的，所以**也不能做到自适应数据的特点** [10，*Learning rate schedules for faster stochastic gradient search*]。
- 另外，**相同的学习率被应用到所有参数更新中**。如果我们的数据比较稀疏，特征有非常多不同的频率，那么此时我们可能并不想要以相同的程度更新他们，反而是对更少出现的特征给予更大的更新。
- 对于神经网络来说，另一个最小化高度非凸误差函数的关键挑战是**避免陷入他们大量的次局部最优点（suboptimal）**。Dauphin 等人 [19，*Identifying and attacking the saddle point problem in high-dimensional non-convex optimization*] 指出事实上困难来自于鞍点而不是局部最优点，即损失函数在该点的一个维度上是上坡（slopes up）（ *译者注：斜率为正* ），而在另一个维度上是下坡（slopes down）（ *译者注：斜率为负* ）。这些鞍点通常被一个具有相同误差的平面所包围，这使得对于 SGD 来说非常难于逃脱，因为在各个维度上梯度都趋近于 0 。

Gradient descent optimization algorithms

接下来，我们将会概述一些在深度学习社区常用的算法，这些算法解决了我们前面提到的挑战。我们不会讨论实际上在高维数据集上不可行的算法，例如二阶方法中的 牛顿法。

Momentum

SGD 在遇到沟壑（ravines）会比较困难，即在一个维度上比另一个维度更陡峭的曲面 [1，*Two problems with backpropagation and other steepest-descent learning procedures for networks*] ，这些

曲面通常包围着局部最优点。在这些场景中，SGD 震荡且缓慢的沿着沟壑的下坡方向朝着局部最优点前进，如图 2 所示。

图 2：不带动量的 SGD

动量（Momentum）[2，*On the momentum term in gradient descent learning algorithms*] 是一种在相关方向加速 SGD 的方法，并且能够减少震荡，如图 3 所示。

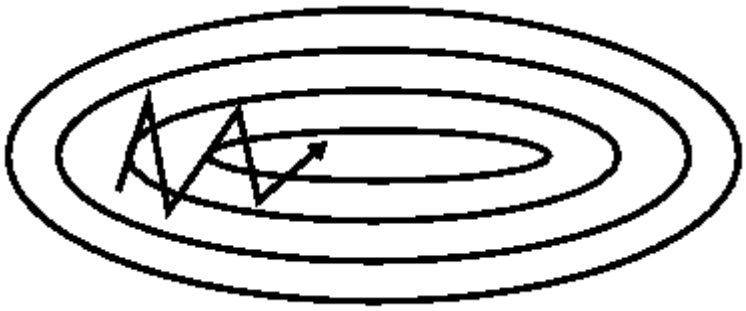


图 3：带动量的 SGD

它在当前的更新向量中加入了先前一步的状态：

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

注意：一些实现可能改变了公式中的符号。动量项 γ 通常设置为 0.9 或者相似的值。

本质上来说，当我们使用动量时，类似于我们把球推下山的过程。在球下山的过程中，球累积动量使其速度越来越快（直到达到其最终速度，如果有空气阻力的话，即 $\gamma < 1$）。相同的事情也发生在我们的参数更新中：对于梯度指向方向相同的维度动量项增大，对于梯度改变方向的维度动量项减小。最终，我们获得了更快的收敛并减少了震荡。

Nesterov accelerated gradient

然而，一个球盲目的沿着斜坡下山，这不是我们希望看到的。我们希望有一个聪明的球，他知道将要去哪并可以在斜坡变成上坡前减速。

Nesterov accelerated gradient（ *译者注：以下简称 NAG* ）[7，*A method for unconstrained convex minimization problem with the rate of convergence o(1/k2)*] 就是这样一种给予我们的动量项预知能力的方法。我们知道我们使用动量项 γv_{t-1} 来更新 θ。因此计算 $\theta - \gamma v_{t-1}$ 给了我们一个关于的参数下一个位置的估计（这里省略了梯度项），这是一个简单粗暴的想法。我们现在可以通过计算 θ 下一个位置而不是当前位置的梯度来实现「向前看」。

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta - \gamma v_{t-1})$$
$$\theta = \theta - v_t$$

我们仍然设置 γ 为 0.9。动量法首先计算当前梯度（图 4 中的小蓝色向量）,然后在更新累积梯度（updated accumulated gradient）方向上大幅度的跳跃（图 4 中的大蓝色向量）。与此不同的是，NAG 首先在先前的累积梯度（previous accumulated gradient）方向上进行大幅度的跳跃（图 4 中的棕色向量），评估这个梯度并做一下修正（图 4 中的红色向量），这就构成一次完整的 NAG 更新（图 4 中的绿色向量）。这种预期更新防止我们进行的太快，也带来了更高的相应速度，这在一些任务中非常有效的提升了 RNN 的性能 [8，*Advances in Optimizing Recurrent Networks*] 。

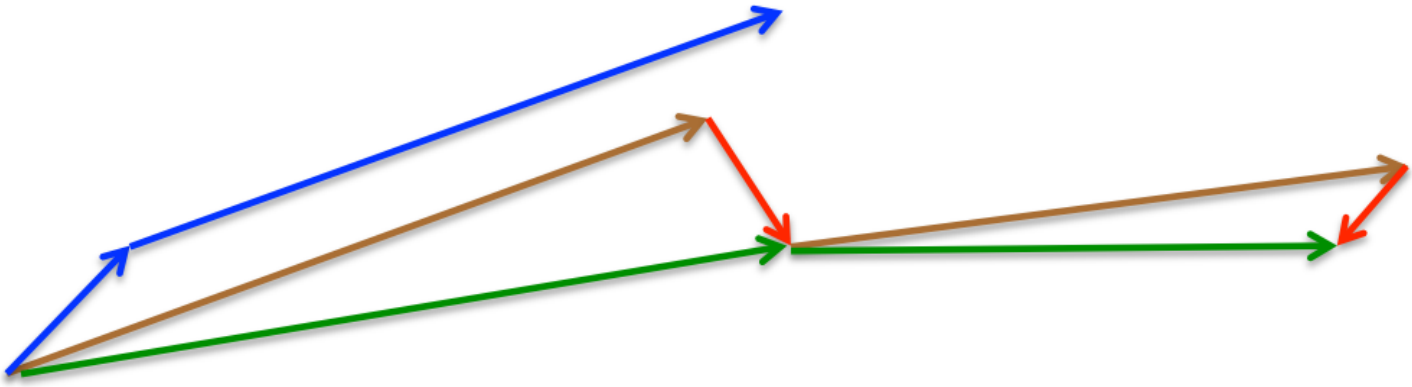


图 4：Nesterov 更新，来自 G. Hinton's lecture 6c

可以在 这里 查看对 NAG 的另一种直观解释，此外 Ilya Sutskever 在他的博士论文中也给出了详细解释 [9，*Training Recurrent neural Networks*] 。

现在我们已经能够依据误差函数的斜率来调整更新，并加快 SGD 的速度，此外我们也想根据每个参数的重要性来决定进行更大还是更小的更新。

Adagrad

Adagrad [3，*Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*] 就是这样一种解决这个问题的基于梯度的优化算法：根据参数来调整学习率，对于不常见的参数给予更大的更新，而对于常见的给予更小的更新。因此，Adagrad 非常适用于稀疏数据。Dean 等人 [4，*Large Scale Distributed Deep Networks*] 发现 Adagrad 能够大幅提高 SGD 的鲁棒性，并在 Google 用其训练大规模神经网络，这其中就包括 在 YouTube 中学习识别猫。除此之外，Pennington 等人 [5，*Glove: Global Vectors for Word Representation*] 使用 Adagrad 来训练 GloVe 词嵌入，因为罕见的词汇需要比常见词更大的更新。

前面我们在对所有参数 θ 更新时每个参数 θ_i 使用相同的学习率 η。Adagrad 在每个时间点 t 对每个参数 θ_i 使用的学习率都不同，我们首先展现每个参数的更新，然后再向量化。简单起见，我们使用 $g_{t,i}$ 来表示目标函数关于参数 θ_i 在时间点 t 时的梯度：

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

SGD 在每个时间点 t 对每个参数 θ_i 的更新变为：

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

在这个更新规则里，Adagrad 在每个时间点 t 对每个参数 θ_i 都会基于过去的梯度修改学习率 η。

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

其中 $G_t \in \mathbb{R}^{d \times d}$ 是一个对角矩阵，对角元素 $G_t[i,i]$ 是参数 θ_i 从开始到时间点 t 为止的梯度平方和 [25]，ϵ 是一个平滑项，用于防止分母为 0 ，通常为 10^{-8} 左右。有趣的是，如果去掉开方操作，算法性能会大幅下降。

由于 G_t 的对角元素是关于所有参数的过去的梯度的平方和，我们可以将上面的实现向量化，即使用点乘 $\odot$ ：

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

Adagrad 最大的一个优点是我们可以不用手动的调整学习率。大多数实现使用一个默认值 0.01 。

Adagrad 主要的缺点是分母中累积的平方和梯度：由于每一个新添加的项都是正的，导致累积和在训练期间不断增大。这反过来导致学习率不断减小，最终变成无限小，这时算法已经不能再继续学习新东西了。下面的这个算法就解决了这个问题。

Adadelta

Adadelta [6，*ADADELTA: An Adaptive Learning Rate Method*] 是 Adagrad 的扩展，旨在帮助缓解后者学习率单调下降的问题。与 Adagrad 累积过去所有梯度的平方和不同，Adadelta 限制在过去某个窗口大小为 w 的大小内的梯度。

存储先前 w 个梯度的平方效率不高，Adadelta 的梯度平方和被递归的定义为过去所有梯度平方的衰减平均值（decaying average）。在 t 时刻的平均值 $E[g^2]_t$ 仅仅取决于先前的平均值和当前的梯度：

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

其中 γ 类似于动量项，我们同样设置为 0.9 左右。为清楚起见，我们根据参数更新向量 $\Delta\theta_t$ 来重写一般的 SGD 更新公式：

$$\Delta\theta_t = -\eta \cdot g_{t,i}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

先前我们推导过的 Adagrad 的参数更新向量是：

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

我们现在用过去梯度平方和的衰减平均 $E[g^2]_t$ 来代替对角矩阵 G_t ：

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

由于分母只是一个梯度的均方误差（Root Mean Squared，RMS），我们可以用缩写来代替：

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

作者注意到这个更新中的单位（units）不匹配（SGD，动量和 Adagrad 也是），即更新向量应该具有和参数一样的单位。为解决这个问题，他们首先定义了另一个指数衰减平均,但是这次不是关于梯度的平方，而是关于参数更新的平方：

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

那么参数更新的均方误差是：

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

由于 $RMS[\Delta\theta]_t$ 未知，我们可以使用上一步的来估计。用带有 $RMS[\Delta\theta]_{t-1}$ 的参数更新规则代替学习率 η 就得到了 Adadelta 最终的更新规则：

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

使用 Adadelta 时我们甚至不需要指定一个默认的学习率，因为它已经不在更新规则中了。

RMSprop

RMSprop 是一种未发布的自适应学习率的方法，由 Geoff Hinton 在 Lecture 6e of his Coursera Class 中提出。

RMSprop 和 Adadelta 在同一时间被独立地发明出来，都是为了解决 Adagrad 的学习率递减问题。事实上 RMSprop 与我们上面讨论过的 Adadelta 的第一个更新向量一模一样：

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

RMSprop 也是将学习率除以平方梯度的指数衰减平均值。Hinton 建议将 γ 设为 0.9 ，默认学习率 η 设为 0.001 。

Adam

Adaptive Moment Estimation (Adam) [15，*Adam: a Method for Stochastic Optimization*] 是另一种为每个参数计算自适应学习率的方法。除了像 Adadelta 和 RMSprop 一样存储历史平方梯度 v_t 的指数衰减平均值外，Adam 也存储历史梯度 m_t 的指数衰减平均值，类似于动量：

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

其中 m_t 和 v_t 分别是梯度在第一时刻（平均值，the mean）和第二时刻（未中心化的方差，the uncentered variance）的估计值，也就是这个方法的名称。由于 m_t 和 v_t 用零向量初始化，Adam 的作者发现他们趋向于 0 ，特别是最开始的时候（*the initial time steps*）和衰减率很小的时候（即 β_1 和 β_2 接近于 1）。

他们通过计算偏差纠正的（bias-corrected）的第一和第二时刻的估计值来抵消这个问题：

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

然后他们使用这些公式来更新参数，就像 Adadelta 和 RMSprop 一样：

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

作者建议 β_1 的默认值为 0.9 ， β_2 的默认值为 0.999 ，ϵ 的默认值为 10^{-8} 。他们证明 Adam 在实践中非常有效，而且对比其他自适应学习率算法也有优势。

AdaMax

Adam 的更新规则中的 v_t 成比例的缩放了梯度，正比于历史梯度的 ℓ_2 范数（通过 v_{t-1} 项）和当前梯度 $|g_t|^2$（*译者注：此段话我非常不确定是这么翻译的，贴上原文：The v_t factor in the Adam update rule scales the gradient inversely proportionally to the ℓ_2 norm of the past gradients (via the vt−1vt−1 term) and current gradient $|g_t|^2$*）：

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

我们可以将此推广到 ℓ_p 范数。注意 Kingma 和 Ba 也将 β_2 参数化为 β_2^p：

$$v_t = \beta_2^p v_{t-1} + (1 - \beta_2^p) |g_t|^p$$

当 p 非常大的时候通常会导致数值上的不稳定（*numerically unstable*），这也是实际中通常使用 ℓ_1 和 ℓ_2 的原因。然而，ℓ_∞ 通常也会比较稳定。因此，作者提出了 AdaMax（Kingma and Ba, 2015），显示了结合了 ℓ_∞ 的 v_t 也能够收敛到下面的更稳定的值。为了避免与 Adam 混淆，我们使用 u_t 来表示无限范数约束的 v_t（infinity norm-constrained）：

$$
\begin{aligned}
u_t &= \beta_2^{\infty} v_{t-1} + (1-\beta_2^{\infty})|g_t|^{\infty} \\
&= \max(\beta_2 \cdot v_{t-1}, |g_t|)
\end{aligned}
$$

我们现在可以将此加进 Adam 的更新规则里，用 u_t 代替 $\sqrt{\hat{v}_t} + \epsilon$，得到 AdaMax 的更新规则：

$$
\theta_{t+1} = \theta_t - \frac{\eta}{u_t}\hat{m}_t
$$

注意到 u_t 依赖于 **max** 操作，这不像 Adam 中的 m_t 和 v_t 那样容易趋向于 0（*bias towards zero*），这也是我们不需要为 u_t 计算偏差纠正的原因。建议的默认值是 $\eta = 0.002$，$\beta_1 = 0.9$ 和 $\beta_2 = 0.999$。

Nadam

正如前面说的那样，Adam 可以看成 RMSprop 和动量的组合：RMSprop 贡献了历史平方梯度的指数衰减的平均值 v_t ，而动量则负责历史梯度的指数衰减平均值 m_t 。我们也看到 NAG 要优于普通的动量。

Nadam（Nesterov-accelerated Adaptive Moment Estimation）[24，*Incorporating Nesterov Momentum into Adam*] 结合了 Adam 和 NAG 。为了将 NAG 融进 Adam 中，我们需要更改其中的动量项 m_t 。

首先，让我们使用当前的符号回顾一下动量更新规则：

$$
\begin{aligned}
g_t &= \nabla_{\theta_t} J(\theta_t) \\
m_t &= \gamma m_{t-1} + \eta g_t \\
\theta_{t+1} &= \theta_t - m_t
\end{aligned}
$$

其中 J 是我们的目标函数，γ 是动量衰减项，η 是我们的步长。将 m_t 代入上面的第三个式子展开得到：

$$
\theta_{t+1} = \theta_t - (\gamma m_{t-1} + \eta g_t)
$$

这再次证明了动量更新涉及两个方向：先前动量向量的方向和当前梯度的方向。

在计算梯度前通过动量更新参数，这使得 NAG 允许我们在梯度方向上执行一个更准确的步骤。因此我们只需更改 g_t 来将 NAG 融进去：

$$
\begin{aligned}
g_t &= \nabla_{\theta_t} J(\theta_t - \gamma m_{t-1}) \\
m_t &= \gamma m_{t-1} + \eta g_t \\
\theta_{t+1} &= \theta_t - m_t
\end{aligned}
$$

Dozat 提出按以下方式来修改 NAG ：与应用动量步骤两次不同的是 —— 一次用来更新梯度 g_t 和一次用来更新参数 θ_{t+1} ，我们现在直接对当前参数应用一个「向前看的」（*look-ahead*）动量向量：

$$
\begin{aligned}
g_t &= \nabla_{\theta_t} J(\theta_t) \\
m_t &= \gamma m_{t-1} + \eta g_t \\
\theta_{t+1} &= \theta_t - (\gamma m_t + \eta g_t)
\end{aligned}
$$

注意我们现在不再使用如上面展开的动量更新规则中的先前动量向量 m_{t-1} ，而是使用当前动量向量 m_t 来「向前看」。为了将 Nesterov 动量加进 Adam ，相似地我们可以用当前动量向量代替先前动量向量。首选，我们先回顾一下 Adam 的更新规则（注意我们不用修改 $\hat{v}_t$ ）：

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t
\end{aligned}
$$

将 $\hat{m}_t$ 和 m_t 定义代入展开得到：

$$
\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\frac{\beta_1 m_{t-1}}{1-\beta_1^t} + \frac{(1-\beta_1) g_t}{1-\beta_1^t} \right)
$$

注意到 $\frac{m_{t-1}}{1-\beta_1^t}$ 是先前动量向量 m_{t-1} 的偏差修正，因此我们可以用 m_{t-1} 来代替之：

$$
\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\beta_1 \hat{m}_{t-1} + \frac{(1-\beta_1) g_t}{1-\beta_1^t} \right)
$$

这个式子与我们上面展开得到的动量更新规则非常相似。我们可以加进 Nesterov 动量，就像我们之前一样简单地用当前动量向量的偏差修正估计 $\hat{m}_t$ 来代替先前动量向量的偏差修正估计 $\hat{m}_{t-1}$，得到 Nadam 的更新规则：

$$
\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \left(\beta_1 \hat{m}_t + \frac{(1-\beta_1) g_t}{1-\beta_1^t} \right)
$$

Visualization of algorithms

下面的两个动画（来自 Alec Radford）提供了对当前优化算法行为的直观解释。也可以在 这里 查看 Karpathy 对这些动画的描述和对我们讨论过的算法的解释。

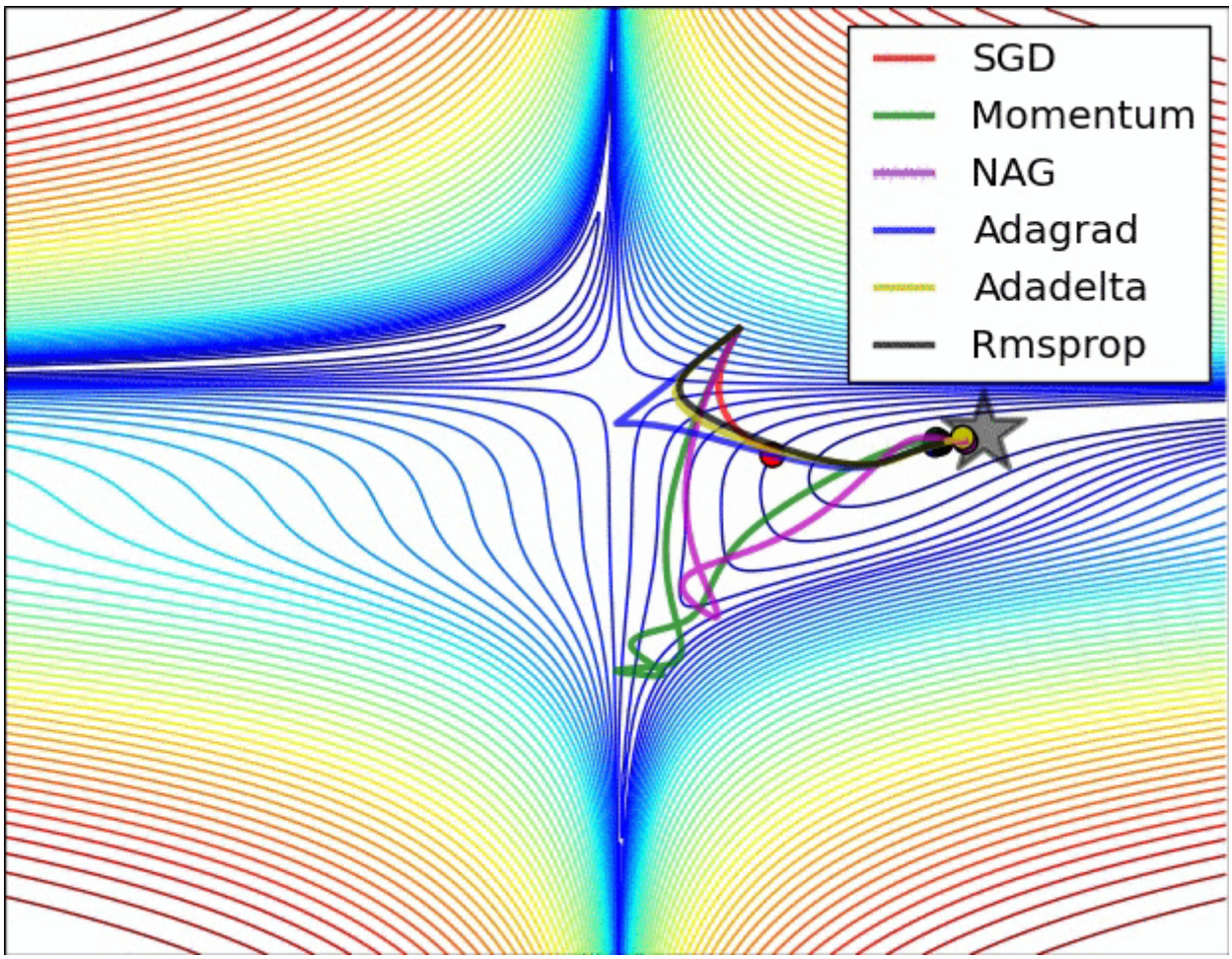
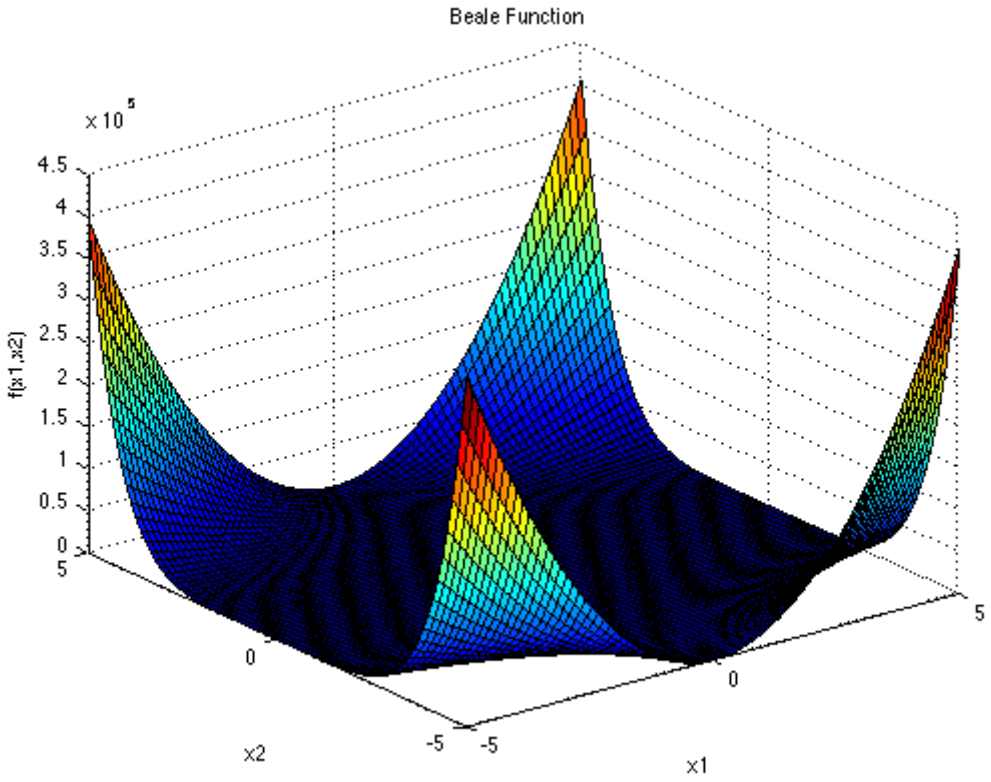


图 5：在损失曲面等值线上的 SGD 优化

在图 5 中我们可以看到他们在损失曲面的等值线上（the Beale function）随时间的变化趋势。注意到 Adadelta 、Adagrad 和 RMSprop 几乎立即开始在正确的方向下降并收敛速度几乎一样快，然而动量和 NAG 则「脱轨」了。不过由于 NAG 的「向前看」能力使其很快的纠正方向并朝着最小点前进。

> *译者注：方便起见我把 Beale 函数的图像和解析式放在这里。*



$$f(\mathbf{x}) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 + x_2^3)^2$$

通常来说该函数的定义域为 $x_i \in [-4.5, 4.5]$ ，其中 $i = 1, 2$ 。全局最小点为 $f(\mathbf{x}^*) = 0$ ，当 $\mathbf{x}^* = (3, 0.5)$ 时 。

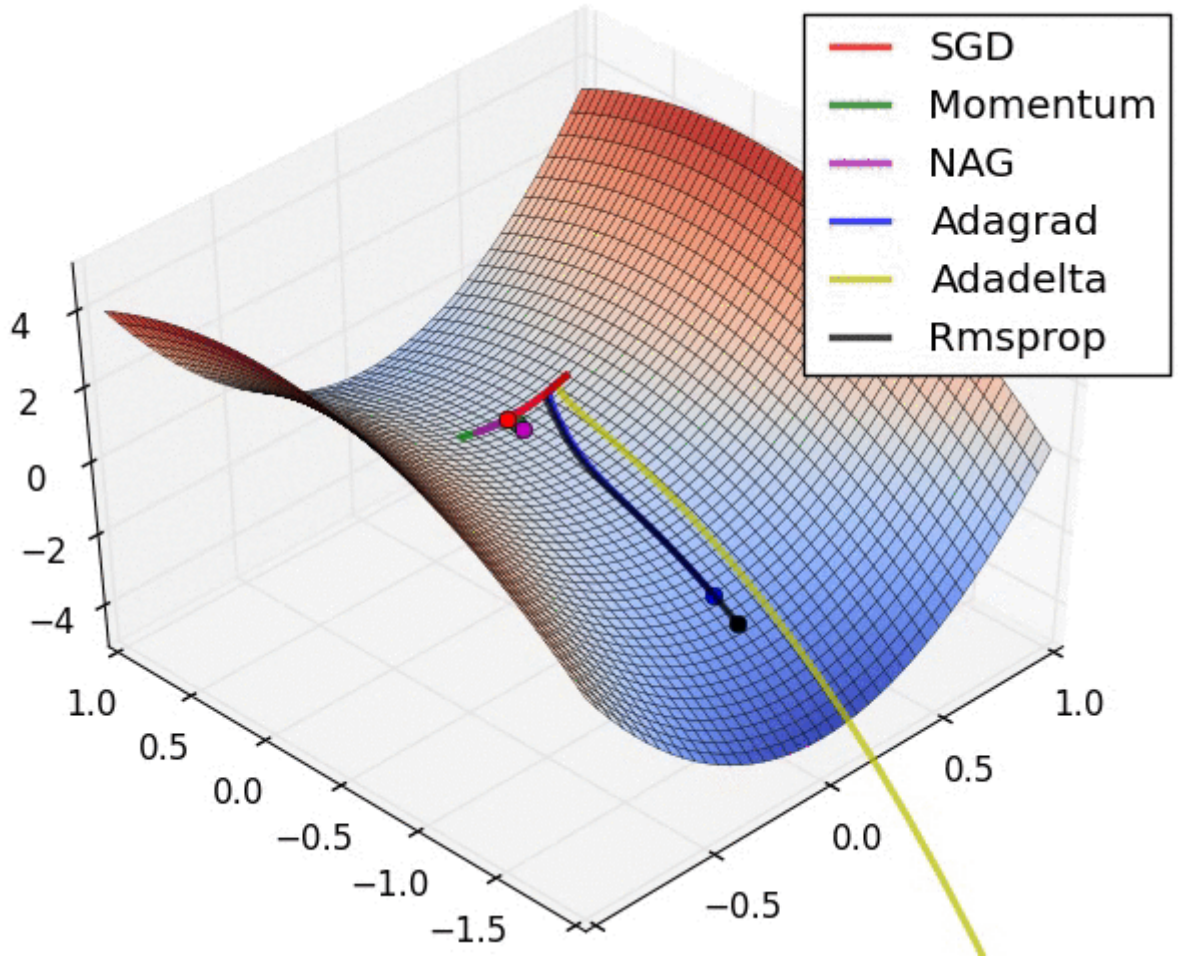


图 6：在鞍点处的 SGD 优化

图 6 显示了在鞍点处的算法行为，即该点在一个方向斜率为正，其他方向斜率为负，正如我们之前提到的这对于 SGD 是一个难点。注意到 SGD 、动量和 NAG 很难打破对称性，尽管后两者最终逃离的鞍点，但是 Adagrad 、RMSprop 和 Adadelta 很快朝着斜率为负的方向前进了。

可以看到自适应学习率的方法，例如 Adagrad 、Adadelta 、RMSprop 和 Adam 在这些场景中是最合适的并且提供了最好的收敛。

Which optimizer to use?

那么，你应该使用哪种优化算法呢？如果你的数据比较稀疏，那么使用自适应学习率的算法很可能会让你得到最好的结果。另外一个好处是你不用去调节学习率，使用默认的设置可能就会让你达到最好的效果。

总的来说，RMSprop 是 Adagrad 的一种扩展，用来解决后者学习率逐渐递减的问题。它和 Adadelta 非常像，除了 Adadelta 在更新规则的分子上使用参数更新的 RMS （*译者注：均方误差*）。Adam 最终在 RMSprop 的基础上加了偏差修正和动量。在这方面，RMSprop 、Adadelta 和 Adam 非常相似，在相似的环境下也表现地一样好。Kingma 等人 [15，*Adam: a Method for Stochastic Optimization*] 表示随着梯度越来越稀疏，Adam 的偏差修正使其略微优于 RMSprop 。在这个方面总体上来说 Adam 可能是最好的选择。

有趣的是，许多最近的论文仅仅使用普通的不带动量 SGD 和一个简单的学习率退火机制（*annealing schedule*）。正如我们前面所讨论的，SGD 通常会找到最优点，但是相比其他一些优化算法可能花费的时间比较长，更依赖于一个好的初始化和退火机制，而且也可能陷入鞍点而不是局部最优点。所以，**如果你比较关心收敛速度并且在训练一个深度或者复杂的神经网络，你应该选择一个自适应学习率算法。**

Parallelizing and distributing SGD

鉴于大数据解决方案的普及以及低价集群的可用性，使用分布式 SGD 来进一步加速训练是一个很明显的选择。SGD 本质上是按顺序执行的：我们一步一步朝着最优点前进。SGD 提供了较好的收敛但是在大数据集上可能会速度较慢。相反，异步执行 SGD 比较快，但是 worker 之间不理想的通信可能会造成比较差的收敛。另外，我们也可以不需要大型计算集群，在一台机器上并行执行 SGD 。下面是目前提出的用于优化并行和分布式 SGD 的算法和架构。

Hogwild!

Niu 等人 [23，*Hogwild! : A Lock-Free Approach to Parallelizing Stochastic Gradient Descent*] 引入了一个更新机制称为 Hogwild!，可以在 CPU 上并行执行 SGD 。处理器可以在不锁参数的情况下访问共享内存。由于没次更新只修改一部分参数，所以这种方法只适合于输入数据比较稀疏的情况。他们表明在这种情况下该方法可以达到几乎最优的收敛速度，因为处理器是不太可能覆盖有用信息的。

Downpour SGD

Downpour SGD 是一种 SGD 的异步变体，由 Dean 等人 [4，*Large Scale Distributed Deep Networks*] 在谷歌的 DistBelief 框架（TensorFlow 的前身）中使用。它在训练数据的子集上并行的运行一个模型的多个副本。这些模型将他们的更新发送到一个参数服务器，他们分布在多个机器上。每个机器只负责存储和更新全部模型参数的一部分。然而由于这些机器并不需要相互通信，例如共享权重或者更新，导致他们的参数一直有发散的风险，阻碍收敛。

Delay-tolerant Algorithms for SGD

McMahan 和 Streeter [12，*Delay-Tolerant Algorithms for Asynchronous Distributed Online Learning*] 通过开发一个延迟容忍（delay-tolerant）算法来扩展 Adagrad 使其并行化，该算法不仅自适应历史梯度，而且也更新延迟（delays）。在实际中也被证明该方法很有效。

TensorFlow

TensorFlow [13，*TensorFlow : Large-Scale Machine Learning on Heterogeneous Distributed System*] 是 Google 最近开源的用于实现和部署大规模机器学习模型的框架。TensorFlow 基于他们使用 DistBelief 的经验，并且已经在内部使用，用于在大范围的移动设备和大规模分布式系统上执行计算。分布式执行中，一个计算图针对每一个设备被拆分成多个子图，使用发送/接收节点对进行通信。

Elastic Averaging SGD

Zhang 等人 [14，*Deep learning with Elastic Averaging SGD*] 提出了 Elastic Averaging SGD（EASGD），将异步 SGD 中每个 worker 参数用一个「弹力」（*elastic force*）连接起来，即一个由参数服务器保存的中心变量。这允许本地变量在中心变量附近进一步震荡，这理论上可以在更大的参数空间中进行探索。他们以经验证明这种增加的探索能力可以寻找新的局部最优点从而提升性能。

Additional strategies for optimizing SGD

最后，我们介绍一些可以和前面讨论的算法一起使用的策略，用以进一步提升 SGD 的性能。对于一些其他常用的技巧，可以参见 [22，*Efficient BackProp. Neural Networks: Tricks of the Trade*] 。

Shuffling and Curriculum Learning

通常，我们想要避免给模型提供的训练数据是有特定顺序的，因为这会使模型带有偏见。因此，每次迭代完之后打乱训练数据是一个很好地想法。

但是另一方面，有些情况下我们想要逐步解决更难的问题，我们将训练数据以一种有意义的顺序提供给模型，这可能会提升性能和得到更好的收敛。构建这种有意义的顺序的方法称为课程学习（*Curriculum Learning*）[16，*Curriculum learning*] 。

Zaremba 和 Sutskever [17，*Learning to Execute*] 只能训练 LSTMs 来评估使用课程学习的简单程序，而且表明组合或者混合的方法要比单一方法更有效，通过增加难度来排序样本。

Batch normalization

为方便学习，我们一般会正规化（*normalize*）参数的初始值，以 0 均值和单位方差来初始化。随着训练的进行和我们将参数更新到不同的程度，我们损失了这种正规化，导致训练速度变慢并且随着网络越来越深，这种影响被渐渐放大。

Batch normalization [18，*Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift*] 重新为每一个 mini-batch 建立了这种正规化并且变化也会随着这个操作反向传播。通过在模型中加入正规化，我们可以使用更高的学习率而且不用太关心参数的初始化。Batch normalization 此外也扮演者正则化的角色，可以减少（甚至有些时候消除）Dropout 的需要.

Early stopping

根据 Geoff Hinton: "*Early stopping (is) beautiful free lunch*"（NIPS 2015 Tutorial slides，slide 63），你应该在训练时时刻监视着验证集误差，并且在你的验证集误差不再足够地降低时停止训练。

Gradient noise

Neelakantan 等人 [21，*Adding Gradient Noise Improves Learning for Very Deep Network*] 在每次梯度更新时加上一个服从高斯分布 $N(0, \sigma_t^2)$ 的噪声：

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

他们依照下面的公式更新（*anneal*）方差：

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

他们表明添加这个噪声使得网络对较差的初始化更具有鲁棒性并且尤其对训练深度和复杂的网络很有帮助。他们怀疑添加的噪声使得模型有更多机会逃离和找到新的局部最优点，这在深度模型中很常见。

Conclusion

本文中，我们首先看了梯度下降的 3 中变体，其中 mini-batch 梯度下降最流行。我们然后研究了几种最常使用的用于优化 SGD 的算法：动量，Nesterov accelerated gradient，Adagrad，Adadelta，RMSprop，Adam 以及为优化异步 SGD 的不同算法。最后，我们考虑了用于提升 SGD 性能的其他策略，例如 shuffling 与 curriculum learning，batch normalization 以及 early stopping。我希望这篇文章可以给你提供一个关于不同优化算法的行为和动机的直观理解。

References

1. Sutton, R. S. (1986). Two problems with backpropagation and other steepest-descent learning procedures for networks. Proc. 8th Annual Conf. Cognitive Science Society.
2. Qian, N. (1999). On the momentum term in gradient descent learning algorithms. Neural Networks : The Official Journal of the International Neural Network Society, 12(1), 145–151. http://doi.org/10.1016/S0893-6080(98)00116-6
3. Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12, 2121–2159. Retrieved from http://jmlr.org/papers/v12/duchi11a.html
4. Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V, … Ng, A. Y. (2012). Large Scale Distributed Deep Networks. NIPS 2012: Neural Information Processing Systems, 1–11. http://doi.org/10.1109/ICDAR.2011.95
5. Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global Vectors for Word Representation. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, 1532–1543. http://doi.org/10.3115/v1/D14-1162
6. Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. Retrieved from http://arxiv.org/abs/1212.5701
7. Nesterov, Y. (1983). A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), vol. 269, pp. 543–

547.
8. Bengio, Y., Boulanger-Lewandowski, N., & Pascanu, R. (2012). Advances in Optimizing Recurrent Networks. Retrieved from http://arxiv.org/abs/1212.0901
9. Sutskever, I. (2013). Training Recurrent neural Networks. PhD Thesis.
10. Darken, C., Chang, J., & Moody, J. (1992). Learning rate schedules for faster stochastic gradient search. Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop, (September), 1–11. http://doi.org/10.1109/NNSP.1992.253713
11. H. Robinds and S. Monro, “A stochastic approximation method,” Annals of Mathematical Statistics, vol. 22, pp. 400–407, 1951.
12. Mcmahan, H. B., & Streeter, M. (2014). Delay-Tolerant Algorithms for Asynchronous Distributed Online Learning. Advances in Neural Information Processing Systems (Proceedings of NIPS), 1–9. Retrieved from http://papers.nips.cc/paper/5242-delay-tolerant-algorithms-for-asynchronous-distributed-online-learning.pdf
13. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., … Zheng, X. (2015). TensorFlow : Large-Scale Machine Learning on Heterogeneous Distributed Systems.
14. Zhang, S., Choromanska, A., & LeCun, Y. (2015). Deep learning with Elastic Averaging SGD. Neural Information Processing Systems Conference (NIPS 2015), 1–24. Retrieved from http://arxiv.org/abs/1412.6651
15. Kingma, D. P., & Ba, J. L. (2015). Adam: a Method for Stochastic Optimization. International Conference on Learning Representations, 1–13.
16. Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum learning. Proceedings of the 26th Annual International Conference on Machine Learning, 41–48. http://doi.org/10.1145/1553374.1553380
17. Zaremba, W., & Sutskever, I. (2014). Learning to Execute, 1–25. Retrieved from http://arxiv.org/abs/1410.4615
18. Ioffe, S., & Szegedy, C. (2015). Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv Preprint arXiv:1502.03167v3.
19. Dauphin, Y., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., & Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. arXiv, 1–14. Retrieved from http://arxiv.org/abs/1406.2572
20. Sutskever, I., & Martens, J. (2013). On the importance of initialization and momentum in deep learning. http://doi.org/10.1109/ICASSP.2013.6639346
21. Neelakantan, A., Vilnis, L., Le, Q. V., Sutskever, I., Kaiser, L., Kurach, K., & Martens, J. (2015). Adding Gradient Noise Improves Learning for Very Deep Networks, 1–11. Retrieved from http://arxiv.org/abs/1511.06807
22. LeCun, Y., Bottou, L., Orr, G. B., & Müller, K. R. (1998). Efficient BackProp. Neural Networks: Tricks of the Trade, 1524, 9–50. http://doi.org/10.1007/3-540-49430-8_2
23. Niu, F., Recht, B., Christopher, R., & Wright, S. J. (2011). Hogwild! : A Lock-Free Approach to Parallelizing Stochastic Gradient Descent, 1–22.
24. Dozat, T. (2016). Incorporating Nesterov Momentum into Adam. ICLR Workshop, (1), 2013–2016.

25. Duchi et al. [3] 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 dd.