<1> 前言
<2> 梯度下降法变体
    <2.1> 批量梯度下降法（Batch GradientDescent）
    <2.2> 随机梯度下降法（Stochastic GradientDescent）
    <2.3> 小批量梯度下降法（Mini-Batch GradientDescent）
<3> 面临的挑战
<4> 梯度下降优化算法
    <4.1> 动量法
    <4.2> Nesterov加速梯度法
    <4.3> Adagrad 法
    <4.4> Adadelta 法
    <4.5> RMSprop
    <4.6> 适应性动量估计法（Adam）
    <4.7> 算法可视化
    <4.8> 如何选择优化器？
<5> 对SGD进行平行计算或分布式计算
    <5.1> Hogwild!
    <5.2> Downpour SGD
    <5.3> 容忍延迟的 SGD 算法
    <5.4> TensorFlow
    <5.5> 弹性平均梯度下降法（Elastic Averaging SGD）
<6> 优化SGD的其他手段
    <6.1> 重排法（Shuffling）和递进学习（CurriculumLearning）
    <6.2> 批量标准化（Batch Normalization）
    <6.3> 早停（Early Stopping）
    <6.4> 梯度噪声（Gradient Noise）
<7> 结论
<8> 鸣谢
<9> 该文的打印版及引文

<1> 前言

梯度下降法，是当今最流行的最优化（optimization）算法，亦是至今最常用的最优化神经网络的方法。与此同时，最新的深度学习程序库都包含了各种优化梯度下降的算法（可以参见如 lasagne、caffe 及Kera 等程序库的说明文档）。但他们的算法则不被公开，都作为黑箱优化器被使用，这也就是为什么它们的优势和劣势往往难以被实际地解释。

该文章旨在让你对不同的优化梯度下降法的算法有一个直观认识，以帮助你使用这些算法。我们首先会考察梯度下降法的各种变体，然后会简要地总结在训练（神经网络或是机器学习算法）的过程中可能遇到的挑战。接着，我们将会讨论一些最常见的优化算法，研究他们的解决这些挑战的动机及他们推导出更新规律（update rules）的过程。我们还会简要探讨一下，在平行计算或是分布式处理情况下优化梯度下降法的算法和架构。最后，我们会考虑一下其他有助于优化梯度下降法的策略。

梯度下降法的核心，是最小化目标函数 $J(\theta)$ ，其中 θ 是模型的参数，$\theta \in R^d$。它的方法是，在每次迭代中，对每个变量，按照目标函数在该变量梯度的相反方向，更新对应的参数值。其中，学习率η决定了函数到达（局部）最小值的迭代次数。换句话说，我们在目标函数的超平面上，沿着斜率下降的方向前进，直到我们遇到了超平面构成的“谷底”。如果你不熟悉梯度下降法的话，你可以在这里找到一个很好的关于优化神经网络的介绍。

<2> 梯度下降法变体

本文讨论了三种梯度下降法的变体——他们的不同之处在于，一次性使用多少数据来计算目标函数的梯度。对于不同的数据量，我们需要在参数更新准确性和参数更新花费时间两方面做出权衡。

<2.1> 批量梯度下降法（Batch GradientDescent）

Vanilla梯度下降法（译者注：Vanilla是早期机器学习算法相关的名词，也是如今一个机器学习 python程序库的名字，在该处指的是后者，参见：https://github.com/vinhkhuc/VanillaML），也就是所谓的批量梯度下降法，在整个数据集上（求出目标函数J(θ)并）对每个参数θ 求目标函数 J(θ) 的偏导数：

--formula--

在该方法中，每次更新我们都需要在整个数据集上求出所有的偏导数。因此批量梯度下降法的速度会比较慢，甚至对于较大的、内存无法容纳的数据集，该方法都无法被使用。同时，梯度下降法不能以“在线”的形式更新我们的模型，也就是不能再运行中加入新的样本进行运算。

批量梯度下降法的实现代码，如下所示：

--code--

对于给定的迭代次数，我们首先基于输入的罚函数 loss_function 对输入的参数向量 params 计算梯度向量 params_grad。注意，最新的深度学习程序库中，提供了自动求导的功能，能够高效、快速地求给定函数对于特定系数的导数。如果你希望自己写代码求出梯度值，那么“梯度检查”会是一个不错的注意。（你可以参考这里，了解关于如何检查梯度的相关建议。）

然后，我们对参数减去梯度值乘学习率的值，也就是在反梯度方向，更新我们参数。当目标函数 J(θ) 是一凸函数时，则批量梯度下降法必然会在全局最小值处收敛；否则，目标函数则可能会局部极小值处收敛。

<2.2> 随机梯度下降法（Stochastic GradientDescent）

相比批量梯度下降法，随机梯度下降法（下简称为SGD）的每次更新，是对数据集中的一个样本（x，y）求出罚函数，然后对其求相应的偏导数：
--formula--

因为批量梯度下降法在每次更新前，会对相似的样本求算梯度值，因而它在较大的数据集上的计算会有些冗余（redundant）。而SGD通过每次更新仅对一个样本求梯度，去除了这种冗余的情况。因而，它的运行速度被大大加快，同时也能够“在线”学习。

SGD更新值的方差很大，在频繁的更新之下，它的目标函数有着如下图所示的剧烈波动。
--image--SGD函数波动，来源：Wikipedia

相比批量梯度下降法的收敛会使目标函数落入一个局部极小值，SGD收敛过程中的波动，会帮助目标函数跳入另一个可能的更小的极小值。另一方面，这最终会让收敛到特定最小值的过程复杂化，因为该方法可能持续的波动而不停止。但是，当我们慢慢降低学习率的时候，SGD表现出了与批量梯度下降法相似的收敛过程，也就是说，对非凸函数和凸函数，必然会分别收敛到它们的极小值和最小值。

相比批量梯度下降法的代码，在如下的代码中，我们仅仅加入了一个循环，用以遍历所有的训练样本并求出相应的梯度值。注意，如这里所说，在每次迭代中，我们会打乱训练数据集。

<2.3> 小批量梯度下降法（Mini-Batch GradientDescent）

小批量梯度下降法集合了上述两种方法的优势，在每次更新中，对n个样本构成的一批数据，计算罚函数J(θ)，并对相应的参数求导：
--formula--

这种方法，(a) 降低了更新参数的方差（variance），使得收敛过程更为稳定；(b) 能够利用最新的深度学习程序库中高度优化的矩阵运算器，能够高效地求出每小批数据的梯度。通常一小批数据含有的样本数量在50至256之间，但对于不同的用途也会有所变化。小批量梯度下降法，通常是我们训练神经网络的首选算法。同时，有时候我们也会使用随机梯度下降法，来称呼小批量梯度下降法（译者注：也就是说，在下文中就不再随机梯度下降法和小批量梯度下降法，统一用SGD代指）。注意：在下文对于随机梯度法优化的介绍中，为方便起见，我们会省略式子中的参数x(i:i+n),y(i:i+n)。

如下的代码所示，我们不再对每个样本进行循环，而是对每批带有50个样本的小批数据进行循环：
--code--

<3> 面临的挑战

由于Vanilla小批量梯度下降法并不能保证良好地收敛，这给我们留下了如下待解决的挑战：

- 选择适当的学习率是一个难题。太小的学习率会导致较慢的收敛速度，而太大的学习率则会阻碍收敛，并会引起罚函数在最小值处震荡，甚至有可能导致结果发散；
- 我们可以设置一个关于学习率的列表，通过如退火的方法，在学习过程中调整学习率——按照一个预先定义的列表、或是当每次迭代中目标函数的变化小于一定阈值时来降低学习率。但这些列表或阈值，需要根据数据集地特性，被提前定义。
- 此外，我们对所有的参数都采用了相同的学习率。但如果我们的数据比较稀疏，同时特征有着不同的出现频率，那么我们不希望以相同的学习率来更新这些变量，我们希望对较少出现的特征有更大的学习率。
- 在对神经网络最优化非凸的罚函数时，另一个通常面临的挑战，是如何避免目标函数被困在无数的局部最小值中，以导致的未完全优化的情况。Dauphin 及其他人[19]认为，这个困难并不来自于局部最小值，而是来自于“鞍点”，也就是在一个方向上斜率是正的、在一个方向上斜率是负的点。这些鞍点通常由一些函数值相同的面环绕，它们在各个方向的梯度值都为0，所以SGD很难从这些鞍点中脱开。

<4> 梯度下降优化算法

在如下的讨论中，我们将会列举一些应对上述问题的算法，它们被广泛应用于深度学习社区。同时，我们不会讨论那些不能应用于高维数据集的方法，例如牛顿法等针对二阶问题的方法。

<4.1> 动量法

SGD很难在陡谷——一种在一个方向的弯曲程度远大于其他方向的表面弯曲情况——中找到正确更新方向。而这种陡谷，经常在局部极值中出现。在这种情况下，如图2所示，SGD在陡谷的周围震荡，向局部极值处缓慢地前进。
--image2 image3--

动量法[2]，如图3所示，则帮助SGD在相关方向加速前进，并减少它的震荡。他通过修改公式中，在原有项前增加一个折损系数γ，来实现这样的功能：
--formula--

注意：在其他的一些算法实现中，公式中的符号也许有所不同。动量项γ 往往被设置为0.9或为其他差不多的值。

从本质上说，动量法，就仿佛我们从高坡上推下一个球，小球在向下滚动的过程中积累了动量，在途中他变得越来越快（直到它达到了峰值速度，如果有空气阻力的话，γ<1）。在我们的算法中，相同的事情发生在我们的参数更新上：动量项在梯度指向方向相同的方向逐渐增大，对梯度指向改变的方向逐渐减小。由此，我们得到了更快的收敛速度以及减弱的震荡。

<4.2> Nesterov加速梯度法

但当一个小球从山谷上滚下的时候，盲目的沿着斜率方向前行，其效果并不令人满意。我们需要有一个更“聪明”的小球，它能够知道它再往哪里前行，并在知道斜率再度上升的时候减速。

Nesterov加速梯度法（NAG）是一种能给予梯度项上述“预测”功能的方法。我们知道，我们使用动量项γvt-1 来“移动”参数项θ。通过计算θ-γvt-1，我们能够得到一个下次参数位置的近似值——也就是能告诉我们参数大致会变为多少。那么，通过基于未来参数的近似值而非当前的参数值计算相得应罚函数 J(θ-γvt-1) 并求偏导数，我们能让优化器高效地“前进”并收敛：
--formula--

在该情况下，我们依然设定动量系数γ 在0.9左右。如下图4所示，动量法首先计算当前的梯度值（小蓝色向量），然后在更新的积累向量（大蓝色向量）方向前进一大步。但 NAG 法则首先（试探性地）在之前积累的梯度方向（棕色向量）前进一大步，再根据当前地情况修正，以得到最终的前进方向（绿色向量）。这种基于预测的更新方法，使我们避免过快地前进，并提高了算法地响应能力（responsiveness），大大改进了 RNN 在一些任务上的表现[8]。
--image4: Nesterov Update法，来源：G.Hinton's lecture 6c--

参考这里，以查看Ilya Sutskever在它博士论文中，对NAG机理的更为详尽的解释[9]。

因为我们现在能根据我们罚函数的梯度值来调整我们的更新，并能相应地加速SGD，我们也希望能够对罚函数中的每个参数调整我们的更新值，基于它们的重要性以进行或大或小的更新。

<4.3> Adagrad 法

Adagrad[3]是一个基于梯度的优化算法，它的主要功能是：他对不同的参数调整学习率，具体而言，对低频出现的参数进行大的更新，对高频出现的参数进行晓得更新。因此，他很适合于处理稀疏数据。Dean等人[14]发现，Adagrad法大大提升了SGD的鲁棒性，并在谷歌使用它训练大规模的神经网络，其诸多功能包括识别Youtube视频中的猫。此外，Pennington等人[5]使用它训练GloVe单词向量映射（Word Embedding），在其中不频繁出现的词语需要比频繁出现的更大的更新值。

在这之前，我们对于所有的参数使用相同的学习率进行更新。但Adagrad 则不然，对不同的训练迭代次数t，adagrad 对每个参数都有一个不同的学习率。我们首先考察adagrad每个参数的的更新过程，然后我们再使之向量化。为简洁起见，我们记在迭代次数t下，对参数θi求目标函数梯度的结果为gt,i：
--formula--

那么普通SGD的更新规则为：
--formula--

而adagrad将学习率η进行了修正，对迭代次数t，基于每个参数之前计算的梯度值，将每个参数的学习率η按如下方式修正：

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

其中 $G_t \in \mathbb{R}^{d \times d}$ 是一个对角阵，其中对角线上的元素 $(G_t)_{i,i}$ 是从一开始到 t 时刻目标函数对于参数 θ_i 梯度的平方和。ϵ 是一个平滑项，以避免分母为0的情况，它的数量级通常在 $1e-8$。有趣的是，如果不开方的话，这个算法的表现会变得很糟。

因为 G_t 在其对角线上，含有过去目标函数对于参数 θ_i 梯度的平方和，我们可以利用一个元素对元素的向量乘法$\odot$，将我们的表达式向量化：

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

Adagrad主要优势之一，是它不需要对每个学习率手工地调节。而大多数算法，只是简单地使用一个相同地默认值如0.1，来避免这样地情况。

Adagrad地主要劣质，是他在分母上的 G_t 项中积累了平方梯度和。因为每次加入的项总是一个正值，所以累积的和将会随着训练过程而增大。因而，这会导致学习率不断缩小，并最终变为一个无限小值——此时，这个算法已经不能从数据中学到额外的信息。而下面的算法，则旨在解决这个问题。

<4.4> Adadelta 法

Adadelta 法[6]是Adagrad 法的一个延伸，它旨在解决它学习率不断单调下降的问题。相比计算之前所有梯度值的平方和，Adadelta 法仅计算在一个大小为 w 的时间区间内梯度值的累积和。

但该方法并不会存储之前 w 个梯度的平方值，而是将梯度值累积值按如下的方式递归地定义：它被定义为关于过去梯度值的衰减均值（decade average），当前时间 t 的梯度均值 $E[g^2]_t$ 是基于过去梯度均值 $E[g^2]_{t-1}$ 和当前梯度值平方 g_t^2 的加权平均，其中 γ 是类似上述动量项的权值。

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

与动量项的设定类似，我们设定 γ 为以0.9左右的值。为明确起见，我们将我们的 SGD 更新规则写为关于参数更新向量 $\Delta\theta_t$ 的形式：

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

由此，我们刚刚在Adagrad法中推导的的参数更新规则的向量表示，变为如下形式：

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

我们现在将其中的对角矩阵 G_t 用上述定义的基于过去梯度平方和的衰减均值 $E[g^2]_t$ 替换：

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

因为分母表达式的形式与梯度值的方均根（root mean squared,RMS）形式类似，因而我们使用相应的简写来替换：

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

作者还注意到，在该更新中（在SGD、动量法或者Adagrad也类似）的单位并不一致，也就是说，更新值的量纲与参数值的假设量纲并不一致。为改进这个问题，他们定义了另外一种指数衰减的衰减均值，他是基于参数更新的平方而非梯度的平方来定义的：

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

因此，对该问题的方均根为：

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

因为$\mathrm{RMS}[\Delta\theta]_t$ 值未知，所以我们使用 $t-1$ 时刻的方均根来近似。将前述规则中的学习率 η 替换为 $\mathrm{RMS}[\Delta\theta]_{t-1}$ ，我们最终得到了Adadelta 法的更新规则：

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

借助 Adadelta 法，我们甚至不需要预设一个默认学习率，因为它已经从我们的更新规则中被删除了。

<4.5> RMSprop

RMSprop 是由Geoff Hinton 在他 Coursera 课程中提出的一种适应性学习率方法，至今仍未被公开发表。

RMSprop 法和 Adadelta 法几乎同时被发展出来。他们 解决 Adagrad 激进的学习率缩减问题。实际上， RMSprop 和我们推导出的 Adadelta 法第一个更规则相同：

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

RMSprop 也将学习率除以了一个指数衰减的衰减均值。Hinton 建议设定 γ 为0.9，对 η 而言，0.001是一个较好的默认值。

<4.6> 适应性动量估计法（Adam）

适应性动量估计法（Adam）[15]是另一种能对不同参数计算适应性学习率的方法。除了存储类似 Adadelta 法或 RMSprop 中指数衰减的过去梯度平方均值 v_t 外，Adam 法也存储像动量法中的指数衰减的过去梯度值均值 m_t：

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

m_t 和 v_t 分别是梯度的一阶矩（均值）和二阶矩（表示不确定度的方差），这也就是该方法名字的来源。因为当 m_t 和 v_t 一开始被初始化为0向量时，Adam的作者观察到，该方法会有趋向0的偏差，尤其是在最初的几步或是在衰减率很小（即 β_1 和 β_2 接近1）的情况下。

他们使用偏差纠正系数，来修正一阶矩和二阶矩的偏差：

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

他们使用这些来更新参数，更新规则很我们在Adadelta 和 RMSprop 法中看到的一样，服从Adam的更新规则：

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

作者认为参数的默认值应设为：$\beta_1 = 0.9,\ \beta_2 = 0.999,\ \epsilon = 10^-8$。他们的经验表明，Adam 在实践中表现很好，和其他适应性学习算法相比也比较不错。

<4.7> 算法可视化

如下的两个动画（图像版权：Alec Radford）给了我们关于特定优化算法在优化过程中行为的直观感受。你可以参见这里，以获取 Karpathy 对相同图像的一些描述，及另关于一些相关算法的细致讨论。

在图5中，我们可以看到，在罚函数的等高线图中，优化器的位置随时间的变化情况。注意到，Adagrad、 Adadelta 及 RMSprop 法几乎立刻就找到了正确前进方向并以相似的速度很快收敛。而动量法和 NAG 法，则找错了方向，如图所示，让小球沿着梯度下降的方向前进。但 NAG 法能够很快改正它的方向向最小指出前进， 因为他能够往前看并对前面的情况做出响应。

图6展现了各算法在鞍点附近的表现。如上面所说，这对对于SGD 法、动量法及 NAG 法制造了一个难题。他们很难打破”对称性“带来的壁垒，尽管最后两者设法逃脱了鞍点。而 Adagrad 法、RMSprop 法及 Adadelta 法都能快速的沿着负斜率的方向前进。
--image5 SGD优化器在罚函数等高线图中的表现--
--image6 SGD优化器在鞍点处的表现--

如我们所见，适应性学习率方法，也就是 Adagrad 法、Adadelta 法 、RMSprop 法及 Adam 法最适合处理上述情况，并有最好的收敛效果。

<4.8> 如何选择优化器？

那么，我们该如何选择优化器呢？如果你的输入数据较为稀疏（sparse），那么使用适应性学习率类型的算法会有助于你得到好的结果。此外，使用该方法的另一好处是，你在不调参、直接使用默认值的情况下，就能得到最好的结果。

总的来说，RMSprop 法是一种基于Adagrad 法的拓展，他从根本上解决学习率骤缩的问题。Adadelta 法于 RMSprop 法大致相同，除了前者使用了。而Adam法，则基于RMSprop法添加了偏差修正项和动量项。在我们地讨论范围中，RMSprop、Adadelta及Adam法都是非常相似地算法，在相似地情况下都能做的很好。Kingma及其他人[15]展示了他们的偏差修正项帮助Adam法，在最优化过程快要结束、梯度变得越发稀疏的时候，表现略微优于RMSprop法。总的来说，Adam也许是总体来说最好的选择。

有趣的是，很多最新的论文，都直接使用了（不带动量项的）Vanilla SGD法，配合一个简单的学习率（退火）列表。如论文所示，这些 SGD 最终都能帮助他们找到一个最小值，但会花费远多于上述方法的时间。并且这些方法非常依赖于鲁棒的初始化值及退火列表。因此，如果你非常在你的模型能快速收敛，或是你需要训练一个深度或复杂模型，你可能需要选择上述的适应性模型。

<5> 对SGD进行平行计算或分布式计算

现如今，大规模数据集随处可见、小型计算机集群也易于获得。因而，使用分布式方法进一步加速 SGD 是一个惯常的选择。

SGD 它本事是序列化的：通过一步一步的迭代，我们最终求到了最小值。运行它能够得到不错的收敛结果，但是特别是对于大规模的数据集，它的运行速度很慢。相比而言，异步 SGD 的运行速度相对较快，但在不同的工作机之间的关于非完全优化的沟通可能会导致较差的收敛结果。此外，我们能够对 SGD 进行平行运算而不需要一个计算机集群。下文讨论了相关的算法或架构，它们或关于平行计算或者对其进行了分布式优化。

<5.1> Hogwild!

Niu 等人提出了一种叫做 Hogwild! 的更新规则，它允许在平行 GPU 上进行 SGD 更新。处理器。这仅能在输入数据集是稀疏的时起效，在每次更新过程中仅会修正一部分的参数值。他们展示了，在这种情况下，这个更新规则达到了最优化的收敛速度，因为处理器不太会覆盖有用的信息。

<5.2> Downpour SGD

Downpour SGD 是一个异步的 SGD 法变体，它被 Dean 等人[4] 用在了谷歌的 DistBelief 架构中（它是TensorFlow 的前身）。他对训练集地子集同步地运行模型的多个副本。这些模型将它们的更新值发送到参数服务器，服务器被分为了许多台主机。每一台主机都负责存储和上载模型的一部分参数。但是，副本之间却没有相互的通信——例如，共享权重值或者更新值——其参数面临着发散的风险，会阻止收敛。

<5.3> 容忍延迟的 SGD 算法

McMahan 和 Streeter [12] 改良了 AdaGrad 法使之能够用于平行运算的场景。通过实现延迟容忍的算法，它不仅能能够适应于过去的梯度，还能够适应于更新的延迟。在实践中，它的表现很好。

<5.4> TensorFlow

TensorFlow[13]是谷歌最近开源的一个实现和部署大规模机器学习模型的架构。它基于他们之前对于使用 DistBelief 的经验，并已在内部被部署在一系列的移动设备及大规模的分布式系统上进行计算。为了分布式执行，一个计算图被分为了许多子图给不同的设备，设备之间的通信使用了发送和接受节点对。~~但是，目前 TensorFlow 的开源版本并不支持分布式功能（参见这里）~~。2016年4月13日更新：一个分布式TensorFlow 的版本已经被发布。

<5.5> 弹性平均梯度下降法（Elastic Averaging SGD）

张等人[14]提出了弹性平均梯度下降法（EASGD），他使不同工作机之间不同的SGD以一个”弹性力“连接，也就是一个储存于参数服务器的中心变量。这允许局部变量比中心变量更大地波动，理论上允许了对参数空间更多的探索。他们的经验表明，提高的探索能力有助于在寻找新的局部极值中提升（优化器的）表现。

<6> 优化SGD的其他手段

最后，我们将讨论一些其他手段，他们可以与前述的方法搭配使用，并能进一步提升SGD的效果。你可以参考[22]，以了解一些其他常用策略。

<6.1> 重排法（Shuffling）和递进学习（CurriculumLearning）

总体而言，我们希望避免训练样本以某种特定顺序传入到我们的学习模型中，因为这会向我们的算法引入偏差。因此，在每次迭代后，对训练数据集中的样本进行重排（shuffling），会是一个不错的注意。

另一方面，在某些情况下，我们会需要解决难度逐步提升的问题。那么，按照一定的顺序遍历训练样本，会有助于改进学习效果及加快收敛速度。这种构建特定遍历顺序的方法，叫做递进学习（Curriculum Learning）[16]。*这个词目前没有标准翻译，我根据表意和意义翻译成这个。

Zaremba 和 Sutskever [17] 仅使用了递进学习法训练LSTMs来学习简单的项目，但结果表明，递进学习法使用的混合策略的表现好于朴素策略——后者不断地重排数据，反而增加了学习过程的难度。

<6.2> 批量标准化（Batch Normalization）

我们通常设置我们参数初值的均值和方差分别为0和单位值，以帮助模型进行学习。随着学习过程的进行，每个参数被不同程度地更新，相应地，参数的正则化特征也随之失去了。因此，随着训练网络的越来越深，训练的速度会越来越慢，变化值也会被放大。

批量标准化[18]对每小批数据都重新进行标准化，并也会在操作中逆传播（back-propgate）变化量。在模型中加入批量标准化后，我们能使用更高的学习率且不要那么在意初始化参数。此外，批量正则化还可以看作是一种正则化手段，能够减少（甚至去除）留出法的使用。

<6.3> 早停（Early Stopping）

诚如Geoff Hinton所言：“Early stopping (is) beautiful free lunch（早停是美妙的免费午餐，又简单效果又好）”（NIPS 2015 Tutorial Sildes, Slide 63）。在训练过程中，你应该时刻关注模型在验证集上的误差情况，并且在改误差没有明显改进的时候停止训练。

<6.4> 梯度噪声（Gradient Noise）

Neelakentan等人[21]在每次梯度的更新中，向其中加入一个服从合高斯分布N(0,σ^2)的噪声值：

--formula--

并按照如下的方式修正方差：

--formula--

他们指出，这种方式能够提升神经网络在不良初始化前提下的鲁棒性，并能帮助训练特别是深层、复杂的神经网络。他们发现，加入噪声项之后，模型更有可能发现并跳出在深度网络中频繁出现的局部最小值。

<7> 结论

在本文中，我们首先分析了梯度下降法的三个变体，在其中小批量梯度下降法最受欢迎。接着，我们研究了常用的优化SGD的算法，包括：动量法、Nesterov accelerated gradient法、Adagrad法、Adadelta法、RMSprop法、Adam法及其他优化异步SGD的算法。最终，我们讨论了另外一些改进SGD的策略，包括样本重排（shuffling）、递进学习（curriculum learning）、批量标准化（Batch Normalization）及早停（early stopping）等。

我希望本文能增进读者关于这些优化算法的认识，能对这些算法的行为与动机有一个了解。也许我遗漏了一些常用的优化SGD的算法，或是你有一些自己使用SGD训练的技巧。如果有的话，请在下方留言区留言让我知道。

<8> 鸣谢

感谢 Denny Britz 及 Ceasr Salgado 阅读本文的草稿并提出了修改建议。

<9> 该文的打印版及引文

如果你之后需要引用该博客中的内容，该博客在 arXiv 也有一个打印版。

如果你觉得该文章对你有用，你可以按照如下格式引用本文在 arXiv 上的版本：*Sebastian Ruder (2016). An overview of gradientdescent optimisation algorithms. arXiv preprint arXiv:1609.04747.*