最全的机器学习中的优化算法介绍

在机器学习中，有很多的问题并没有解析形式的解，或者有解析形式的解但是计算量很大（譬如，超定问题的最小二乘解），对于此类问题，通常我们会选择采用一种迭代的优化方式进行求解。

这些常用的优化算法包括：梯度下降法（Gradient Descent），共轭梯度法（Conjugate Gradient），Momentum算法及其变体，牛顿法和拟牛顿法（包括L-BFGS），AdaGrad，Adadelta，RMSprop，Adam及其变体，Nadam。

梯度下降法（Gradient Descent）

想象你在一个山峰上，在不考虑其他因素的情况下，你要如何行走才能最快的下到山脚？当然是选择最陡峭的地方，这也是梯度下降法的核心思想：它通过每次在当前梯度方向（最陡峭的方向）向前“迈”一步，来逐渐逼近函数的最小值。

在第 n 次迭代中，参数 $\theta_n = \theta_{n-1} + \Delta\theta$

我们将损失函数在 θ_{n-1} 处进行一阶泰勒展开：

$$L(\theta_n) = L(\theta_{n-1} + \Delta\theta) \approx L(\theta_{n-1}) + L^{'}(\theta_{n-1})\Delta\theta$$

为了使 $L(\theta_n) < L(\theta_{n-1})$，可取 $\Delta\theta = -\alpha L^{'}(\theta_{n-1})$，即得到我们的梯度下降的迭代公式：

$$\theta_n := \theta_{n-1} - \alpha L^{'}(\theta_{n-1})$$

梯度下降法根据每次求解损失函数 L 带入的样本数，可以分为：**全量梯度下降**（计算所有样本的损失），**批量梯度下降**（每次计算一个batch样本的损失）**和随机梯度下降**（每次随机选取一个样本计算损失）。

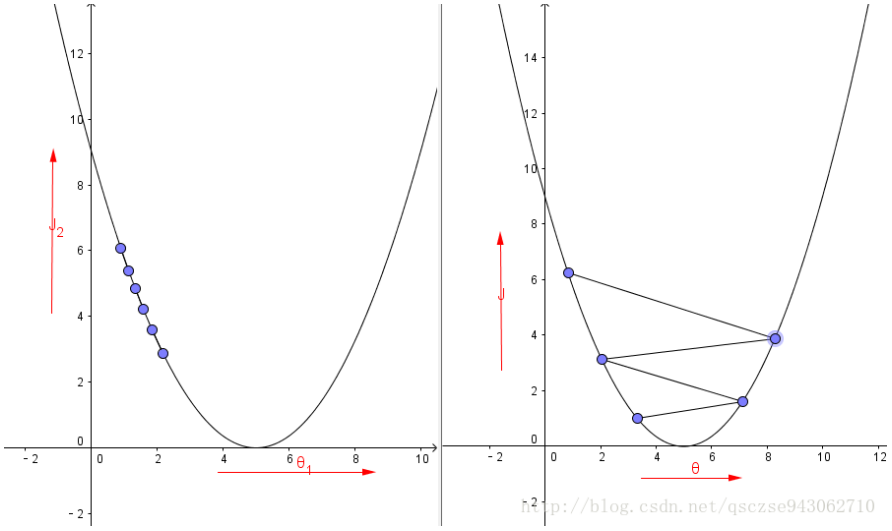
PS：现在所说的SGD（随机梯度下降）多指Mini-batch-Gradient-Descent（批量梯度下降），后文用 g_n 来代替 $L^{'}(\theta_n)$

SGD的优缺点

优点：操作简单，计算量小，在损失函数是凸函数的情况下能够保证收敛到一个较好的全局最优解。

缺点：

- α 是个定值（在最原始的版本），它的选取直接决定了解的好坏，过小会导致收敛太慢，过大会导致震荡而无法收敛到最优解。



- 对于非凸问题，只能收敛到局部最优，并且没有任何摆脱局部最优的能力（一旦梯度为0就不会再有任何变化）。
  PS：对于非凸的优化问题，我们可以将其转化为对偶问题，对偶函数一定是凹函数，但是这样求出来的解并不等价于原函数的解，只是原函数的一个确下界

Momentum

SGD中，每次的步长一致，并且方向都是当前梯度的方向，这会收敛的不稳定性：无论在什么位置，总是以相同的“步子”向前迈。

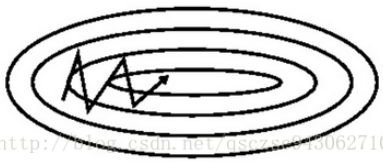
Momentum的思想就是模拟物体运动的惯性：当我们跑步时转弯，我们最终的前进方向是由我们之前的方向和转弯的方向共同决定的。Momentum在每次更新时，保留一部分上次的更新方向：

$$\Delta\theta_n = \rho\Delta\theta_{n-1} + g_{n-1}$$

$$\theta_n := \theta_{n-1} - \alpha\Delta\theta_n$$

这里
ρ
值决定了保留多少上次更新方向的信息，值为0~1，初始时可以取0.5，随着迭代逐渐增大；
α
为学习率，同SGD。

优点：一定程度上缓解了SGD收敛不稳定的问题，并且有一定的摆脱局部最优的能力（当前梯度为0时，仍可能按照上次迭代的方向冲出局部最优点），直观上理解，它可以让每次迭代的“掉头方向不是那个大“。左图为SGD，右图为Momentum。

图片来源于《An overview of gradient descent optimization algorithms》

缺点：这里又多了另外一个超参数
ρ
需要我们设置，它的选取同样会影响到结果。

Nesterov Momentum

Nesterov Momentum又叫做Nesterov Accelerated Gradient（NAG），是基于Momentum的加速算法。

通过上述，我们知道，在每次更新的时候，都在
$\rho\Delta\theta_{n-1} + L^{'}(\theta_n)$
走
α
这么远，那么我们为什么不直接走到这个位置，然后从这个位置的梯度再走一次呢？为此，引出NAG的迭代公式：

$$\Delta\theta_n = \rho\Delta\theta_{n-1} + g(\theta_{n-1} - \alpha\Delta\theta_{n-1})$$

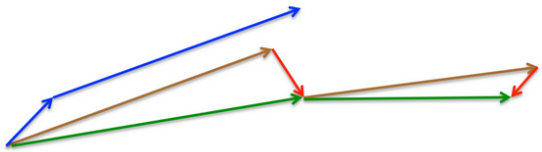
$$\theta_n := \theta_{n-1} - \alpha\Delta\theta_n$$

我们可以这样理解，每次走之前，我们先用一个棍子往前探一探，这根棍子探到的位置就是
$L(\theta_{n-1} - \alpha\Delta\theta_{n-1})$
，然后我们求解此处的梯度：如果梯度大，我们迈一大步，反之，迈一小步。如果我们将上式改写一下：

$$\Delta\theta_n = \rho\Delta\theta_{n-1} + g_{n-1} + \rho(g_{n-1} - g_{n-2}))$$

$$\theta_n := \theta_{n-1} - \alpha\Delta\theta_n$$

如果这次的梯度比上次大，那么我们有理由认为梯度还会继续变大！于是，当前就迈一大步，因为使用了二阶导数的信息（二阶导数>0即一阶导单调递增，也即
$g^{'}_{n-1} \geq g^{'}_{n-2}$
，因此可以加快收敛。



图片来自Hinton在Coursera上DL课程的slides

蓝色的线代表原始的Momentum更新方向，在NAG中，我们先求解得到了这个方向，也即棕色的线，然后求解此处的梯度（红色的线），从而

得到最终的前进方向。

共轭梯度法（Conjugate Gradient）

同样的，CG也在选取前进方向上，对SGD做了改动。它对性能有很大的提升，但是不适用高维数据，求解共轭的计算量过大。网上有很多讲CG的，但是个人感觉都是从某一篇文献里面摘出来的几个图，这里推荐一个专门讲解CG的painless conjugate gradient，讲的很细致。

不同于上述算法对前进方向进行选择和调整，后面这些算法主要研究沿着梯度方向走多远的问题，也即如何选择合适的学习率 α 。

Adagrad

即adaptive gradient，自适应梯度法。它通过记录每次迭代过程中的前进方向和距离，从而使得针对不同问题，有一套自适应调整学习率的方法：

$$\Delta\theta_n = \frac{1}{\sqrt{\sum_{i=1}^{n-1} g_i + \epsilon}} g_{n-1}$$

$$\theta_n := \theta_{n-1} - \alpha\Delta\theta_n$$

可以看到，随着迭代的增加，我们的学习率是在逐渐变小的，这在“直观上”是正确的：当我们越接近最优解时，函数的“坡度”会越平缓，我们也必须走的更慢来保证不会穿过最优解。这个变小的幅度只跟当前问题的函数梯度有关，ϵ 是为了防止0除，一般取1e-7。

优点：解决了SGD中学习率不能自适应调整的问题

缺点：

- 学习率单调递减，在迭代后期可能导致学习率变得特别小而导致收敛及其缓慢。
- 同样的，我们还需要手动设置初始 α

Adagrad-like

在《No More Pesky Learning Rates》一文中，提到另外一种利用了二阶导信息的类adagrad算法。它是由Schaul于2012年提出的，使用了如下形式的更新公式：

$$\Delta\theta_n = \frac{1}{|diag(H_n)|} * \frac{(\sum_{i=n-t}^{n-1} g_i)^2}{\sum_{i=n-t}^{n-1} g_i^2} g_{n-1}$$

$$\theta_n := \theta_{n-1} - \Delta\theta_n$$

H_n 是二阶梯度的Hession矩阵，这里只使用了前t个梯度来缩放学习率。它是由LecCun提出来的一种逼近Hession矩阵的更新方式的变体，原始版本为：

$$\Delta\theta_n = \frac{1}{|diag(H_n)| + \epsilon} g_{n-1}$$

$$\theta_n := \theta_{n-1} - \Delta\theta_n$$

优点：缓解了Adagrad中学习率单调递减的问题

缺点：Hession矩阵的计算必须采用较好的近似解，其次t也成为了新的超参数需要手动设置，即我们需要保留参数前多少个梯度值用来缩放学习率。

Adadelta

Adadelta在《ADADELTA: An Adaptive Learning Rate Method 》一文中提出，它解决了Adagrad所面临的问题。定义：

$$RMS[g]_n = \sqrt{E[g^2]_n + \epsilon}$$

$$E[g^2]_n = \rho E[g^2]_{n-1} + (1-\rho) g_n^2$$

则更新的迭代公式为：

$$\theta_n := \theta_{n-1} - \frac{RMS[\Delta\theta]_{n-1}}{RMS[g]_n} g_n$$

这里 ρ 为小于1的正数，随着迭代次数的增加，同一个 $E[g^2]_i$ 会因为累乘一个小于1的数而逐渐减小，即使用了一种自适应的方式，让距离当前越远的梯度的缩减学习率的比重越小。分子是为了单位的统一性，其实上述的算法中，左右的单位是不一致的，为了构造一致的单位，我们可以模拟牛顿法（一阶导\二阶导），它的单位是一致的，而分子就是最终推导出的结果，具体参考上面那篇文章。这样，也解决了Adagrad初始学习率需要人为设定的问题。

优点：完全自适应全局学习率，加速效果好
缺点：后期容易在小范围内产生震荡

RMSprop

其实它就是Adadelta，这里的RMS就是Adadelta中定义的RMS，也有人说它是一个特例，$\rho = 0.5$ 的Adadelta，且分子 α，即仍然依赖于全局学习率。

Adam

Adam是Momentum和Adaprop的结合体，我们先看它的更新公式：

$$E[g^2]_n = \rho E[g^2]_{n-1} + (1-\rho) g_n^2$$

$$E[g]_n = \phi E[g]_{n-1} + (1-\phi) g_n^2$$

$$\bar{E[g^2]}_n = \frac{E[g^2]_n}{1-\rho^n}$$

$$\bar{E[g]}_n = \frac{E[g]_n}{1-\phi^n}$$

$$\theta_n := \theta_{n-1} - \alpha \frac{\bar{E[g]}_n}{\sqrt{\bar{E[g^2]}_n + \epsilon}} g_n$$

它利用误差函数的一阶矩估计和二阶矩估计来约束全局学习率。
优点：结合Momentum和Adaprop，稳定性好，同时相比于Adagrad，不用存储全局所有的梯度，适合处理大规模数据
一说，adam是世界上最好的优化算法，不知道用啥时，用它就对了。
详见《Adam: A Method for Stochastic Optimization》

Adamax

它是Adam的一个变体，简化了二阶矩估计的取值：

$$E[g^2]_n = \max(|g_n|, E[g^2]_{n-1} * \rho)$$

$$\theta_n := \theta_{n-1} - \alpha \frac{\bar{E[g]}_n}{E[g^2]_n + \epsilon} g_n$$

Nadam和NadaMax

Nadam是带有NAG的adam：

$$\bar{g}_n = \frac{g_n}{1 - \prod_{i=1}^{n} \phi_i}$$

$$E[g]_n = \phi E[g]_{n-1} + (1-\phi) g_n^2$$

$$\bar{E[g]}_n = \frac{E[g]_n}{1 - \prod_{i=1}^{n+1} \phi_i}$$

$$E[g^2]_n = \rho E[g^2]_{n-1} + (1-\rho) g_n^2$$

$$\bar{E[g^2]}_n = \frac{E[g]_n}{1-\rho^n}$$

$$\bar{E[g]}_n = (1-\phi_n)\bar{g}_n + \phi_{n_1} \bar{E[g]}_n$$

$$\theta_n := \theta_{n-1} - \alpha \frac{\bar{E[g]}_n}{\sqrt{E[g^2]_n} + \epsilon}$$

每次迭代的
ϕ
都是不同的，如果参考Adamax的方式对二阶矩估计做出修改，我们可以得到NadaMax，
详见：《Incorporating Nesterov Momentum intoAdam》

牛顿法

牛顿法不仅使用了一阶导信息，同时还利用了二阶导来更新参数，其形式化的公式如下：

$$\theta_n := \theta_{n-1} - \alpha \frac{L_{n-1}^{'}}{L_{n-1}^{''}}$$

回顾之前的
$\theta_n = \theta_{n-1} + \Delta\theta$
，我们将损失函数在
θ_{n-1}
处进行二阶泰勒展开：

$$L(\theta_n) = L(\theta_{n-1} + \Delta\theta) \approx L(\theta_{n-1}) + L^{'}(\theta_{n-1})\Delta\theta + \frac{L^{''}(\theta_{n-1})\Delta\theta^2}{2}$$

要使
$L(\theta_n) < L(\theta_{n-1})$
，我们需要极小化
$L^{'}(\theta_{n-1})\Delta\theta + \frac{L^{''}(\theta_{n-1})\Delta\theta^2}{2}$
，对其求导，令导数为零，可以得到：

$$\Delta\theta = -\frac{L_{n-1}^{'}}{L_{n-1}^{''}}$$

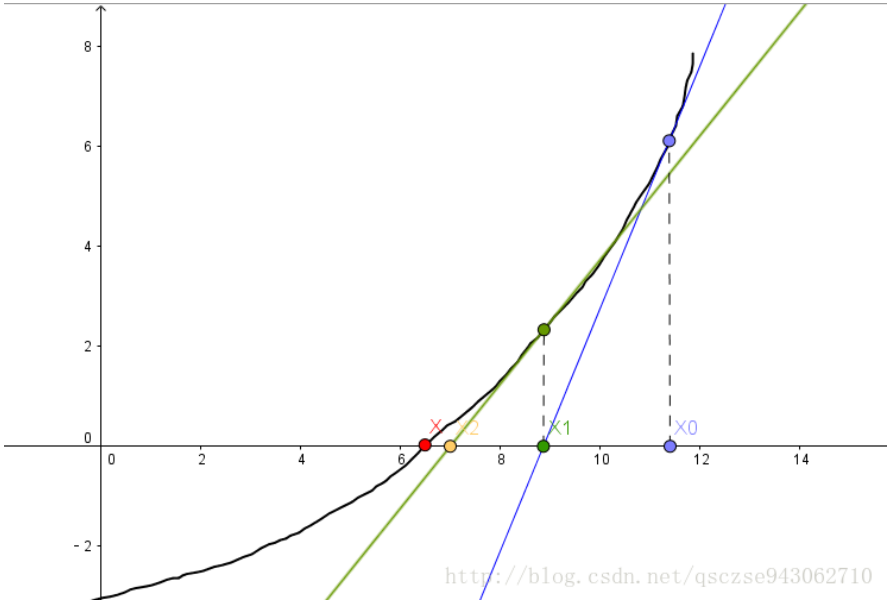
也即牛顿法的迭代公式，拓展到高维数据，二阶导变为Hession矩阵，上式变为：

$$\Delta\theta = -H^{-1} L_{n-1}^{'}$$

直观上，我们可以这样理解：我们要求一个函数的极值，假设只有一个全局最优值，我们需要求得其导数为0的地方，我们把下图想成是损失函数的导数的图像
$f(x)$
，那么：

$$k = \tan\theta = f^{'}(x_0) = \frac{f(x_0)}{x_0 - x_1} \rightarrow x_1 = x_0 - \frac{f(x_0)}{f^{'}(x_0)}$$

我们一直这样做切线，最终
x_n
将逼近与
$f^{'}(x)$
的0点，对于原函数而言，即
$\Delta\theta = -\frac{L_{n-1}^{'}}{L_{n-1}^{''}}$
。



牛顿法具有二阶收敛性，每一轮迭代会让误差的数量级呈平方衰减。

即在某一迭代中误差的数量级为0.01，则下一次迭代误差为0.0001，再下一次为0.00000001。收敛速度快，但是大规模数据时，Hession矩阵的计算与存储将是性能的瓶颈所在。

为此提出了一些算法，用来近似逼近这个Hession矩阵，最著名的有L-BFGS，优于BFGS，可适用于并行计算从而大大提高效率，详见：Large-scale L-BFGS using MapReduce

有人会问，既然有这么多方法，为什么很多论文里面还是用的SGD?需要注意的是，其他的方法在计算性能和收敛方面确实优秀很多，有的甚至不用认为干涉，它会自适应的调整参数，**但是**，在良好的调参情况下，SGD收敛到的最优解一般是最好的。