1.梯度下降法

1.1 什么是梯度下降

梯度下降法是一个一阶最优化算法，通常也称为最速下降法。要使用梯度下降法找到一个函数的局部极小值，必须向函数上当前点对应梯度的反方向的规定步长距离点进行迭代搜索。如果相反地向梯度正方向迭代搜索，则会接近函数的局部极大值，这个过程称为梯度上升法。

如果J(θ)是一个多元函数，在θ_0点附近对J(θ)做线性逼近

$$J(\theta_0+\Delta_\theta)=J(\theta_0)+\Delta_\theta^T\cdot\nabla J(\theta_0)+o(|\Delta_\theta|)$$

这个线性逼近不能告诉我们极值点在什么地方，他只能告诉我们极值点在什么方向.所以我们只能选取一个比较“小”的学习率η来沿着这个方向走下去，并得到梯度下降法的序列：

$$\theta_n=\theta_{n-1}-\eta_{n-1}\nabla J(\theta_{n-1})$$

1.2 遇到的困难

困难：

1. **梯度的计算**

   在机器学习和统计参数估计问题中目标函数经常是求和函数的形式

   $$J_X(\theta)=\sum_i J_{x_i}(\theta)$$

   其中每一个函数$J_{x_i}(\theta)$都对应于一个样本x_i。

   - **当样本量极大时，梯度的计算就变得非常耗时耗力。**

2. **学习率的选择**

   学习率选择过小会导致算法收敛太慢，学习率选择过大容易导致算法不收敛。

   - **如何选择学习率需要具体问题具体分析**

1.3 牛顿下降法

假设 $f(x)$ 具有二阶连续偏导数，若第 k 次迭代值为 $x^{(k)}$，则可将 $f(x)$ 在 $x^{(k)}$ 附近进行二阶泰勒展开：

$$f(x)=f(x^{(k)})+g_k^{\mathrm{T}}(x-x^{(k)})+\frac{1}{2}(x-x^{(k)})^{\mathrm{T}}H(x^{(k)})(x-x^{(k)}) \tag{B.2}$$

这里，$g_k=g(x^{(k)})=\nabla f(x^{(k)})$ 是 $f(x)$ 的梯度向量在点 $x^{(k)}$ 的值，$H(x^{(k)})$ 是 $f(x)$ 的海赛矩阵（Hesse matrix）

$$H(x)=\left[\frac{\partial^2 f}{\partial x_i\partial x_j}\right]_{n\times n} \tag{B.3}$$

在点 $x^{(k)}$ 的值．函数 $f(x)$ 有极值的必要条件是在极值点处一阶导数为 0，即梯度向量为 0．特别是当 $H(x^{(k)})$ 是正定矩阵时，函数 $f(x)$ 的极值为极小值.

牛顿法利用极小点的必要条件

$$\nabla f(x)=0 \tag{B.4}$$

用抛物线进行逼近，这样逼近的好处：1.知道逼近的方向；2.知道逼近的步长（抛物线有极值）

$$f(\theta_0+\Delta_\theta)=f(\theta_0)+\Delta_\theta^T\cdot\bigtriangledown J(\theta_0)+\frac{1}{2}\Delta_\theta^T HJ(\theta_0)\Delta_\theta+o(|\Delta_\theta|)^2,$$

于是关于Δ_θ的梯度为：

$\bigtriangledown J(\theta_0)+HJ(\theta_0)\Delta_\theta+o|\Delta_\theta|$，零点的近似值为：

$$\Delta_\theta=HJ(\theta_0)^{-1}\cdot\bigtriangledown J(\theta_0)$$

a-1几何意义：

对函数进行二阶逼近（更精确），并直接估计函数的极小值点

1.4 为什么不用牛顿法

1. 牛顿法要求计算目标函数的二阶导数（Hessian matrix），在高维特征情形下这个矩阵非常巨大，计算和存储都成问题
2. 在使用小批量情形下，牛顿法对于二阶导数的估计噪音太大
3. 在目标函数非凸时，牛顿法更容易受到鞍点甚至最大值点的吸引

2. 随机梯度下降

随机梯度下降法主要为了解决第一个问题：梯度计算

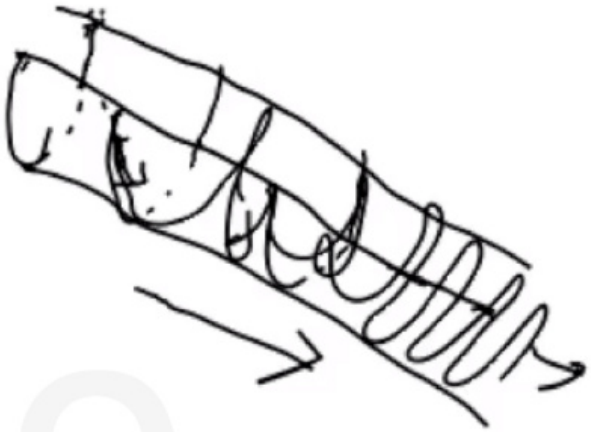
由于梯度下降法的引入，通常将梯度下降法分为三种类型：

1. **批梯度下降法(GD)**
   原始的梯度下降法
2. **随机梯度下降法(SGD)**
   每次梯度计算只使用一个样本
   - 避免在类似样本上计算梯度造成的冗余计算
   - 增加了跳出当前的局部最小值的潜力
   - 在逐渐缩小学习率的情况下，有与批梯度下降法类似的收敛速度
3. **小批量随机梯度下降法(Mini Batch SGD)**
   每次梯度计算使用一个小批量样本
   - 梯度计算比单样本更加稳定
   - 可以很好的利用现成的高度优化的矩阵运算工具

注意：神经网络训练的文献中经常把 Mini Batch SGD 称为 SGD

随机梯度下降法的问题在于第二个：学习率的选取

- 局部梯度的反方向不一定是函数整体下降的方向，对图像比较崎岖的函数，尤其是隧道性曲面，做的是局部估计，梯度下降表现不佳。如下图，会做来回的震荡估计，但总体方向向下：



- 预定学习率衰减法的问题，学习率衰减法很难根据当前数据进行自适应
- 对不同参数采取不同的学习率的问题。在数据有一定稀疏性时，希望对不同特征采取不同的学习率

- 神经网络训练中梯度下降法容易被困在鞍点附近的问题。比起局部极小值，鞍点更加可怕（关于鞍点及鞍点为什么可怕：https://blog.csdn.net/baidu_27643275/article/details/79250537）

3.SGD的几种优化算法

3.1 动量法（适用于隧道型曲面）

1. 定义

每次更新都会吸收一部分上次更新的余势，这样主体方向的更新就得到了更大的保留，从而效果被不断扩大

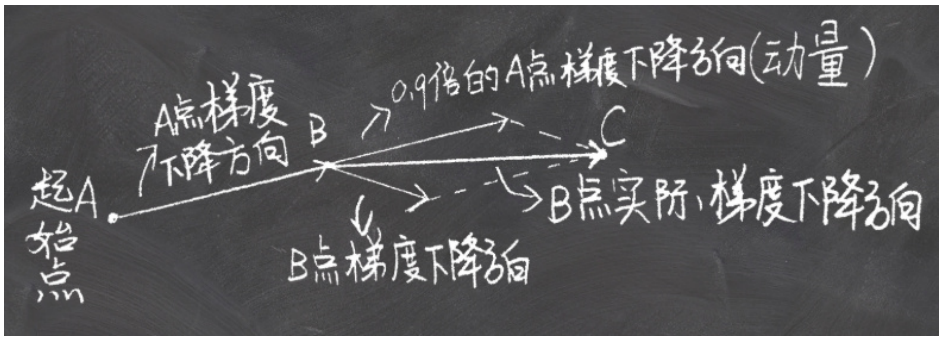
$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta_t = \theta_{t-1} - v_t$$

2. 物理理解

就像是推一个很重的铁球下山，因为铁球保持了下山主体方向的动量，所以隧道上沿两侧震荡的次数就会越来越少，如下图。

(a) SGD without momentum　　(b) SGD with momentum

3. 动量法实际路线



4. 存在问题

从山顶推下的雪球会越滚越快，以至于到了山底停不下来，我们希望在到达底部就自己刹车。

3.2 Nesterov accelerated gradient(NAG)

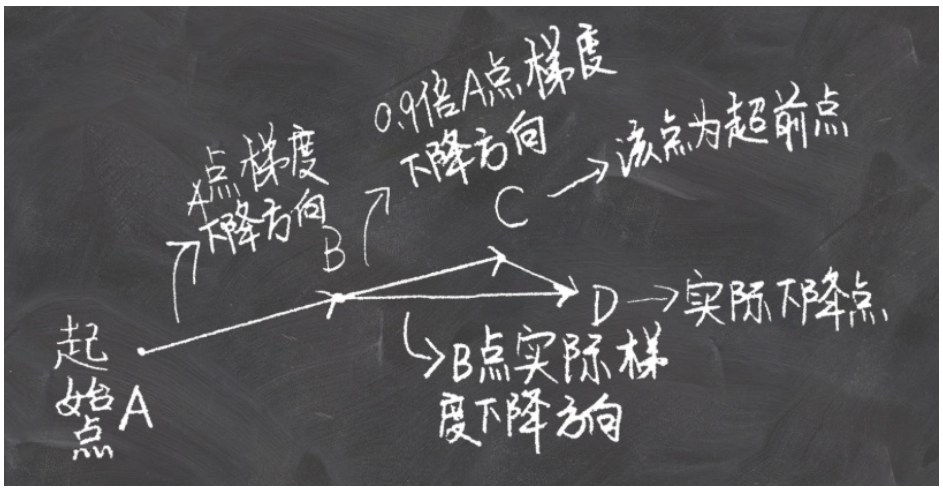
1. 定义

动量法的改进算法，利用主体下降方向提供先见之明，预判自己的下一步位置，并到预判位置计算梯度。

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

2. 实际路线

达到刹车的目的



3.4 Adagrad(自动调整学习率，适用于稀疏数据)

1.定义

梯度下降法在每一步对每一个参数使用相同的学习率，这种一刀切的做法不能有效的利用每一个数据集自身的特点。

Adagrad是一种自动调整学习率的方法

- 随着模型的训练，学习率自动衰减
- 对于更新频繁的参数，采取较小的学习率
- 对于更新不频繁的参数，选取较大的学习率

为了实现对于更新频繁的参数使用较小的学习率，Adagrad对每个参数历史上的更新进行叠加，并以此来做下一次更新的惩罚系数

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

梯度历史矩阵： G_t 对角矩阵，其中$G_{t,ii} = \sum_k g_{k,i}^2$

参数更新：

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

2.存在的问题

随着训练的进行，学习率快速单调衰减，可能使得训练过程提前结束；

Adagrad及一般的梯度下降法的另一个问题在于，梯度与参数的单位不匹配

3.5 Adadelta(Adagrad的改进算法）/RMSprop

1.定义

Adagrad的一个问题在于随着训练的进行，学习率快速单调衰减，Adadelta则使用梯度平方的移动平均来取代全部历史平方和。

定义移动平均： $E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$

于是就得到参数更新法则：

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[g^2]_{t,ii} + \epsilon}} \cdot g_{t,i}$$

- Adagrad 以及一般的梯度下降法的另一个问题在于，梯度与参数的单位不匹配。

  Adadelta 使用参数更新的移动平均来取代学习率η. 于是参数更新法则：

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\sqrt{\mathrm{E}[\Delta\theta]_{t-1}}}{\sqrt{E[g^2]_{t,ii} + \epsilon}} \cdot g_{t,i}$$

注意：Adadelta 的第一个版本也叫做 RMSprop，是Geoff Hinton独立于Adadelta提出来的。

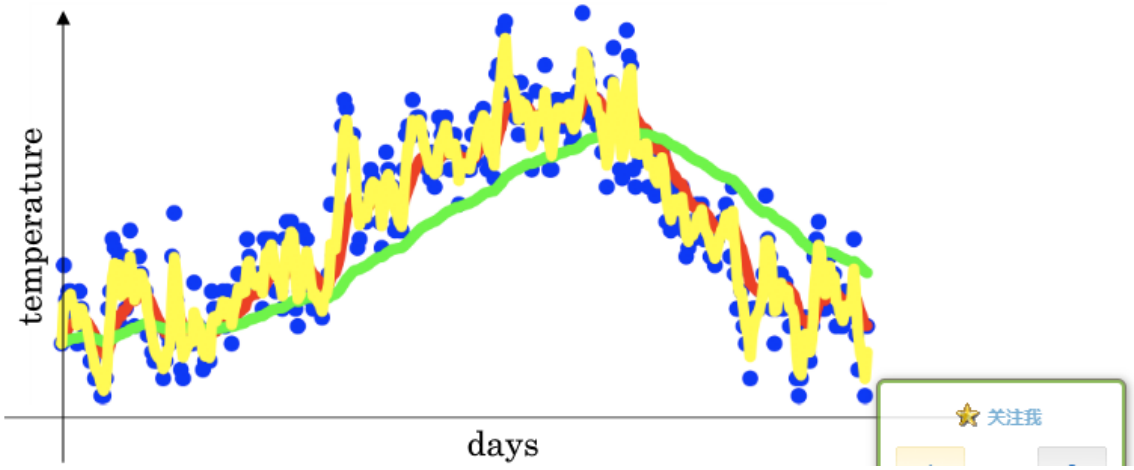
什么是指数加权平均

指数加权平均

在深度学习优化算法中，例如Momentum、RMSprop、Adam，都提到了一个概念，指数加权平均，看了Andrew Ng的深度学习课程后，总结一下什么是指数加权平均。

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

式中v_t可近似代表1/(1-β)个θ的平均值。



当$\beta = 0.9$，则近似代表近10个数据的平均值，证明如下：

$v_{100} = \beta v_{99} + (1-\beta)\theta_{100} = 0.9v_{99} + 0.1\theta_{100}$

$v_{99} = 0.9v_{98} + 0.1\theta_{99}$

$v_{98} = 0.9v_{97} + 0.1\theta_{98}$

$\cdots$

展开v_{100}

$v_{100} = 0.9v_{99} + 0.1\theta_{100} = 0.9(0.9v_{98} + 0.1\theta_{99}) + 0.1\theta_{100}$

$= 0.9(0.9(0.9v_{97} + 0.1\theta_{98}) + 0.1\theta_{99}) + 0.1\theta_{100}$

$= 0.1\theta_{100} + 0.1\times 0.9^1\theta_{99} + 0.1\times 0.9^2\theta_{98} + 0.1\times 0.9^3 v_{97}$

再依次展开$v_{97}\cdots v_0$

$\because \beta^{(\frac{1}{1-\beta})} = 0.9^{10} \approx \frac{1}{e} \approx 0.35$

此时系数已经近似为$\frac{1}{3}$以后会越来越小

$\therefore$ 指数加权平均可以近似为$\frac{1}{1-\beta}$个数据的移动平均

3.6 Adam（结合了动量法与Adadelta)

除了像 Adadelta 和 RMSprop 一样存储了过去梯度的平方 vt 的指数衰减平均值，也像 momentum 一样保持了过去梯度 mt 的指数衰减平均值：

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t.$$

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

如果 mt 和 vt 被初始化为 0 向量，那它们就会向 0 偏置，所以做了**偏差校正**，通过计算偏差校正后的 mt 和 vt 来抵消这些偏差：

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

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

梯度更新规则：

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

超参数设定值：

建议 β1 = 0.9，β2 = 0.999，ϵ = 10e−8

偏差修正

由以上证明可以看出，每个最新数据值，依赖于以前的数据结果。

一般令第一个数值为0，即v0=0；但此时初期的几个计算结果就会与真实的平均值有较大偏差，具体如下：

$v_0 = 0$

$v_1 = 0.98v_0 + 0.02\theta_1 = 0.02\theta_1$

$v_2 = 0.98v_1 + 0.02\theta_2 = 0.98(0.02\theta_1) + 0.02\theta_2$

$= 0.0196\theta_1 + 0.02\theta_2$

数值偏小，与计算值有较大偏差

进行偏差修正：

$$\frac{v_t}{1-\beta^t}$$

随着t的增大β^t越来越接近0

该偏差修正对数据前期影响大，后期影响逐渐减小

有了指数加权平均、偏差修正的基础，就可以研究一下深度学习中优化算法的实现原理了。

如果把Adadelta里面的梯度平方和看成是梯度的二阶矩，那么梯度自身的求和就是一阶矩。Adam算法在Adadelta的二阶矩基础之上引入了一阶矩。

而一阶矩，其实就类似于动量法里面的动量。

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

于是参数更新法则为：

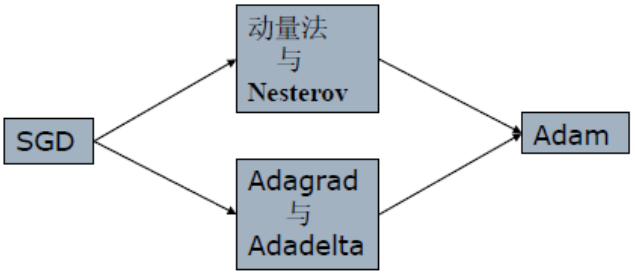
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} m_t$$

注意：实际操作中 v_t与m_t采取了更好的无偏估计，来避免前几次更新时候数据不足的问题。

究竟如何选择算法呢？

- 动量法与Nesterov的改进方法着重解决目标函数图像崎岖的问题
- Adagrad与Adadelta主要解决学习率更新的问题
- Adam集中了前述两种做法的主要优点

目前为止 Adam **可能**是几种算法中综合表现最好的

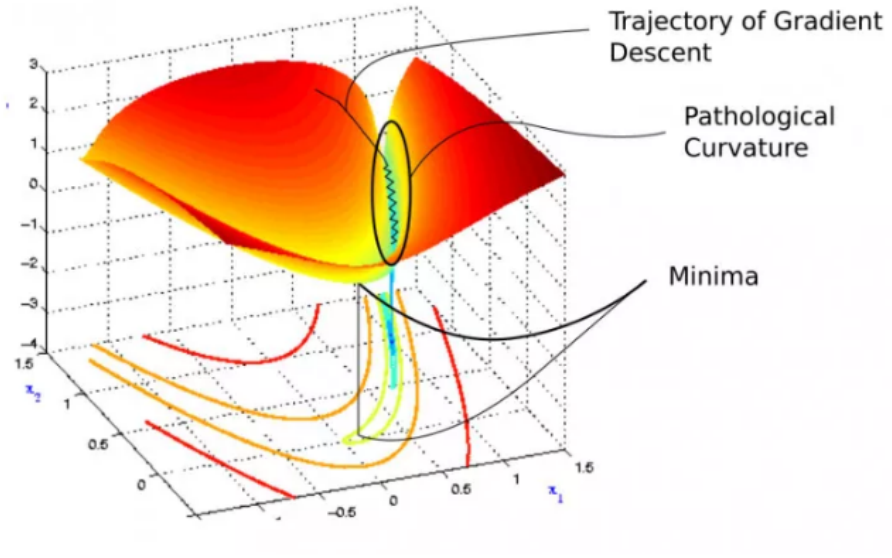


4. 从病态曲率看几种优化算法

虽然局部极小值和鞍点会阻碍我们的训练，但病态曲率会减慢训练的速度，以至于从事机器学习的人可能会认为搜索已经收敛到一个次优的极小值。让我们深入了解什么是病态曲率。

4.1 病态曲率

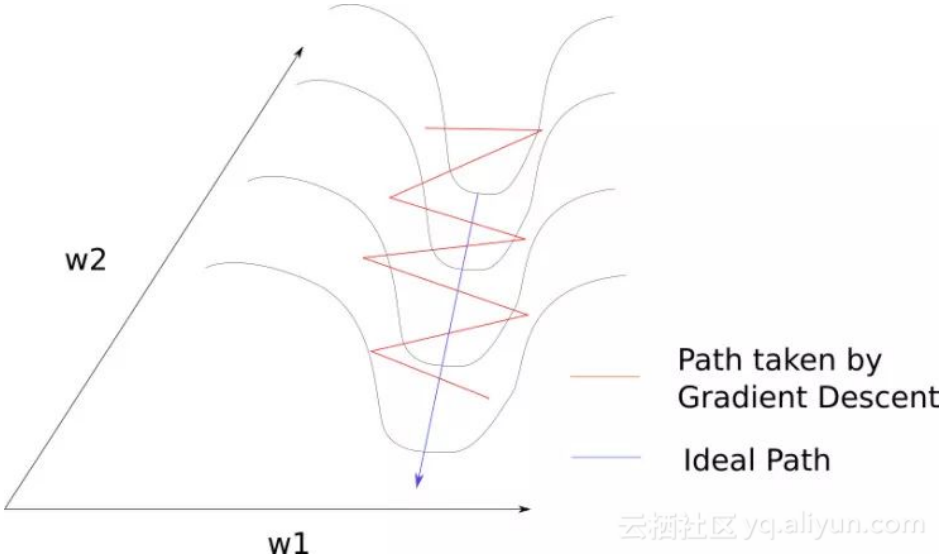
考虑以下损失曲线图



病态曲率

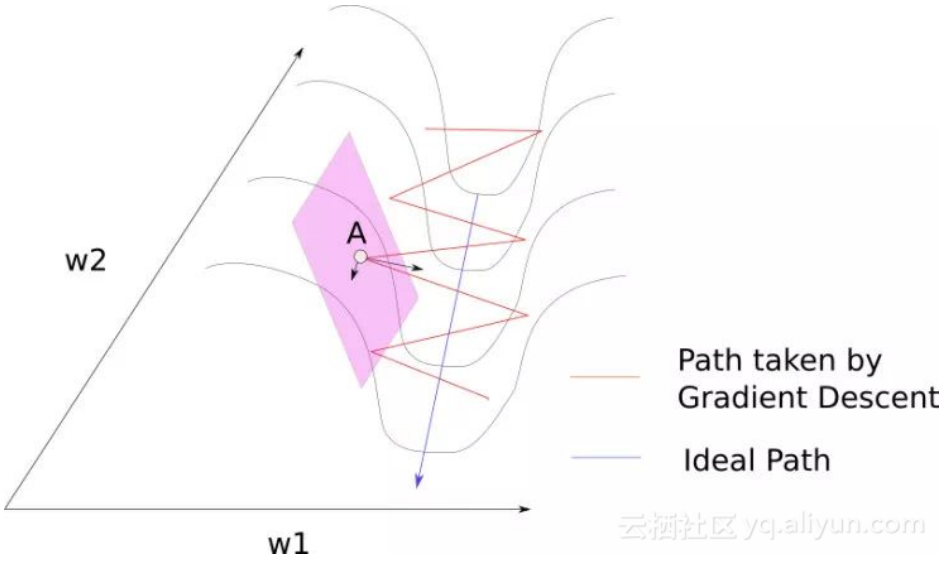
我们在进入一个以蓝色为标志的像沟一样的区域之前是随机的，这些颜色实际上代表了在特定点上的损失函数的值，红色代表最高的值，蓝色代表最低的值。

我们想要下降到最低点，因此需要穿过峡谷，这个区域就是所谓的病态曲率。



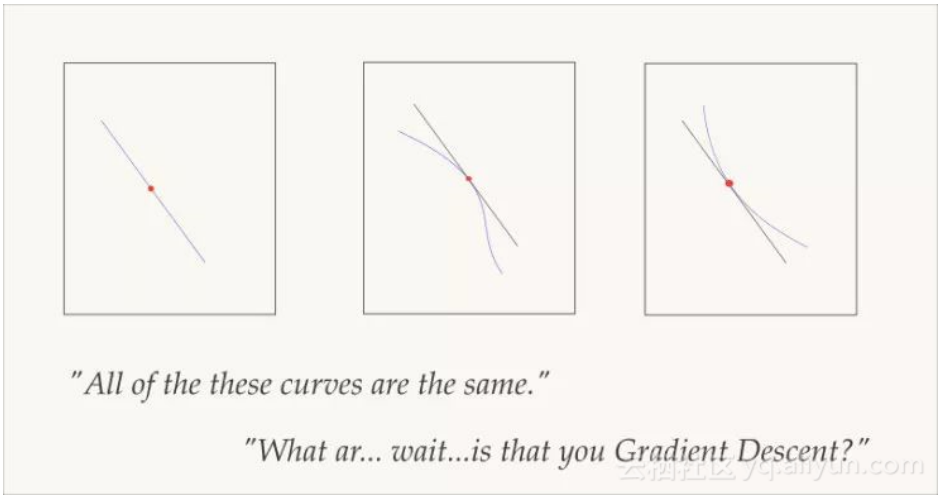
梯度下降沿着峡谷的山脊反弹，向最小的方向移动的速度非常慢。这是因为山脊的曲线在 W1 方向上弯曲的更陡。

考虑山脊表面的 A 点。我们看到，梯度在这点可以分解为两个分量，一个沿着 W1 方向，另外一个沿着 W2 方向。如果 f 显著下降的唯一方向是低曲率的，那么优化可能会变得太慢而不切实际，甚至看起来完全停止，造成局部最小值的假象。



正常情况下，我们使用一个较慢的学习率来解决这种山脊间反弹的问题，但是如果考虑梯度下降进入病态曲率的区域以及到最小值的绝对距离，使用较慢的学习率可能花费更多的时间才能到达极小值点。事实上，有研究论文报道过使用足够小的学习率来阻值山脊间的反弹可能导致参与者以为损失根本没有改善，从而放弃训练。

解决以上问题，可以使用二阶导数。因为梯度下降只考虑一阶导数，所以不知道损失函数的曲率。



就好像上图中红色的点，三个曲线在这一点上的梯度是相同的。如何解决？使用二阶导数，或者考虑梯度变化的速率。

4.2 使用牛顿法

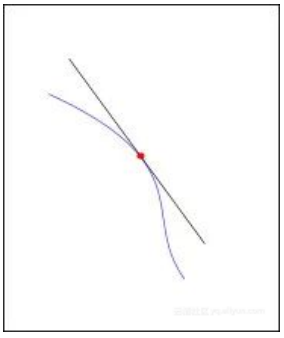
牛顿法通过计算Hessian矩阵来实现，Hessian矩阵是损失函数的二阶导数组成的权值组合，

$$H(e) = \begin{bmatrix} \frac{\partial^2 e}{\partial w_1^2} & \frac{\partial^2 e}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_1 \partial w_n} \\ \frac{\partial^2 e}{\partial w_2 \partial w_1} & \frac{\partial^2 e}{\partial w_2^2} & \cdots & \frac{\partial^2 e}{\partial w_2 \partial w_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2 e}{\partial w_n \partial w_1} & \frac{\partial^2 e}{\partial w_n \partial w_2} & \cdots & \frac{\partial^2 e}{\partial w_n^2} \end{bmatrix}$$

Hessian 矩阵给出了一个点的损失曲面曲率的估计。一个损失的表面可以有一个正曲率，这意味着当我们移动时，表面会迅速变得不那么陡峭。如果我们有一个负曲率，这意味着当我们移动时，曲面变得越来越陡。

$$\epsilon^* = \frac{g^\top g}{g^\top H g}$$

注意，如果这一步是负的，那就意味着我们可以使用任意的步骤。换句话说，我们可以切换回原来的算法。这对应于下面的情况，梯度变得越来越陡。



然而，如果梯度变得不那么陡峭，我们可能会走向一个处于病态曲率底部的区域。在这里，牛顿法给了我们一个修正的学习步骤，正如你所看到的，它与曲率成反比，或者曲面变得越来越小。

如果表面变得不那么陡峭，那么学习步骤就会减少。

为什么很少使用牛顿法？Hessian矩阵要求的值得数量是神经网络中权值数量的平方。对于现代的网络来说，通常都含有数十亿个参数，使用高阶的优化方法很难计算。

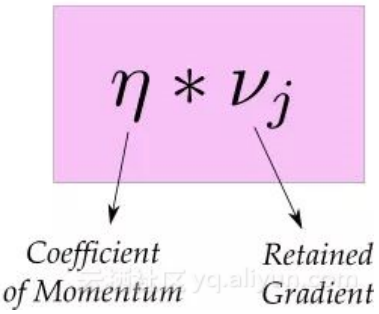
4.3 Momentum

Repeat Until Convergence {

$$\nu_j \leftarrow \eta * \nu_j - \alpha * \nabla_w \sum_1^m L_m(w)$$

$$\omega_j \leftarrow \nu_j + \omega_j$$

}



我们设 v 的初始为 0，动量系数为 0.9，那么迭代过程如下：

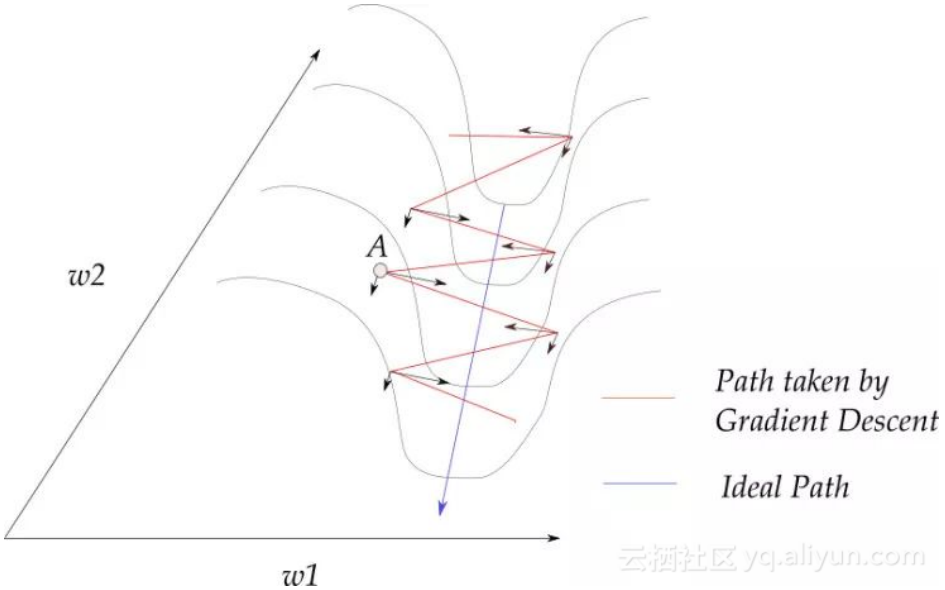
$$\nu_1 = -G_1$$

$$\nu_2 = -0.9 * G_1 - G_2$$

$$\nu_3 = -0.9 * (0.9 * G_1 - G_2) - G3 = -0.81 * (G_1) - (0.9) * G_2 - G_3$$

我们可以看到之前的梯度会一直存在后面的迭代过程中，只是越靠前的梯度其权重越小。（说的数学一点，我们取的是这些梯度步长的指数平均）

这对我们的例子有什么帮助呢？观察下图，注意到大部分的梯度更新呈锯齿状。我们也注意到，每一步的梯度更新方向可以被进一步分解为 w1 和 w2 分量。如果我们单独的将这些向量求和，沿 w1 方向的的分量将抵消，沿 w2 方向的分量将得到加强。



对于权值更新来说，将沿着 w2 方向进行，因为 w1 方向已抵消。这就可以帮助我们快速朝着极小值方向更新。所以，动量也被认为是一种抑制迭代过程中锯齿下降问题的技术。

4.4 RMSProp

RMSProp 算法也旨在抑制梯度的锯齿下降，但与动量相比， RMSProp 不需要手动配置学习率超参数，由算法自动完成。 更重要的是，RMSProp 可以为每个参数选择不同的学习率。

在 RMSprop 算法中，每次迭代都根据下面的公式完成。 它是对每个参数单独迭代。

$$For\ each\ Parameter\ w^j$$

$$(j\ subscript\ dropped\ for\ clarity)$$

$$\nu_t = \rho\nu_{t-1} + (1-\rho) * g_t^2$$

$$\Delta\omega_t = -\frac{\eta}{\sqrt{\nu_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

$\eta : Initial\ Learning\ rate$
$\nu_t : Exponential\ Average\ of\ squares\ of\ gradients$
$g_t : Gradient\ at\ time\ t\ along\ \omega^j$

在第一个方程中，我们计算一个梯度平方的指数平均值。由于我们需要针对每个梯度分量分别执行平方，所以此处的梯度向量gt对应的是正在更新的参数各个方向的投影分量。

注意我们表示病态曲率的图，梯度沿 w1 方向的分量比沿 w2 方向的分量大的多。我们以平方的方式将 w1 和 w2 叠加，w1 不会发生抵消，w2 在指数平均后会更小。

第二个方程定义了步长，我们沿梯度负方向移动，但是步长受到指数平均值的影响。我们设置了一个初始学习率eta，用它除指数平均值。在我们的例子中，因为 w1 平均后比 w2 大很多，所以 w1 的迭代步长就比 w2 要小很多。因此这将避免我们在山脊之间跳跃而朝着正确的方向移动。

第三个方程是更新操作，超参数 p 通常选为 0.9，但是你可能需要调整它。方程 2 中的 epsilon 是为了防止被 0 除，通常取 1e-10。

4.5 Adam

尽管 Momentum 加速了我们对极小值方向的搜索，但 RMSProp 阻碍了我们在振荡方向上的搜索。

$$For\ each\ Parameter\ w^j$$

$$(j\ subscript\ dropped\ for\ clarity)$$

$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$

$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$

$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$

$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

$\eta : Initial\ Learning\ rate$

$g_t : Gradient\ at\ time\ t\ along\ \omega^j$

$\nu_t : Exponential\ Average\ of\ gradients\ along\ \omega_j$

$s_t : Exponential\ Average\ of\ squares\ of\ gradients\ along\ \omega_j$

$\beta_1, \beta_2 : Hyperparameters$

我们计算了每个梯度分量的指数平均和梯度平方指数平均（方程 1、方程 2）。为了确定迭代步长我们在方程 3 中用梯度的指数平均乘学习率（如 Momentum 的情况）并除以根号下的平方指数平均（如RMSProp的情况），然后方程 4 执行更新步骤.超参数 beta1 一般取 0.9 左右，beta_2 取 0.99。Epsilon 一般取1e-10。

二阶动量的出现（Adagrad开始），是自适应学习率优化算法时代的开始。

5. 从通用框架看优化算法

5.1 几种算法演变

优化算法

优化算法通用框架：

首先定义：待优化参数 w，目标函数 $f(w)$，初始学习率 α。而后开始进行迭代优化，在每个 epoch：

(1) 计算目标函数关于当前参数的梯度：

$$g_t = \nabla f(w_t)$$

(2) 根据历史梯度计算一阶动量和二阶动量：

$$m_t = \phi(g_1, g_2, \cdots, g_t)$$

$$V_t = \psi(g_1, g_2, \cdots, g_t)$$

(3) 计算当前时刻的下降梯度

$$\eta_t = \alpha \cdot \frac{m_t}{\sqrt{V_t}}$$

(4) 根据下降梯度进行更新

$$w_{t+1} = w_t - \eta_t$$

各个算法主要差别是 (1) (2)

1. SGD.

SGD 没有动量的概念：

$$m_t = g_t, \quad V_t = I^2$$

$$\therefore \eta_t = \alpha \cdot g_t$$

SGD 最大的缺点是下降速度慢，而且可能会在沟壑的两边持续震荡，停留在一个局部最优点。

2. SGD with Momentum (为了抑制 SGD 的震荡，SGDM 认为梯度下降过程可以加入惯性)

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) g_t \quad \text{一阶动量}$$

t 时刻的下降方向，不仅由当前点的梯度方向决定，而且由此前累积的下降方向决定。β_1 经验值为 0.9

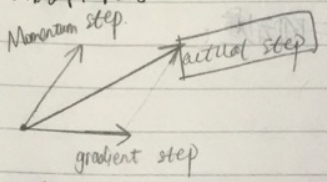
3. SGD with Nesterov Acceleration:

SGD 还有一个问题是困在局部最优的沟壑里面震荡

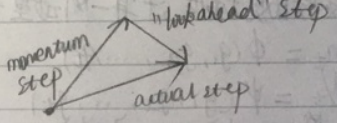
因此，我们可选择向前多看一步，

$g_t = \nabla f(w_t - \lambda \cdot m_{t-1}/\sqrt{V_{t-1}})$.

不计算当前位置的梯度方向，而是计算如果按照累积动量走了一步，那个时候的下降方向：



Momentum.



NAG.

4. AdaGrad. (自适应学习率)

怎样度量历史更新幅度？二阶动量.

——该维度上，迄今为止所有梯度值的平方和.

$V_t = \sum_{\tau=1}^{t} g_\tau^2$

对于经常更新参数，我们已经积累了大量关于它的知识，不希望被单个样本影响太大，希望学习速率慢一些；对于偶尔更新参数，我们了解的信息太少，希望学习速率大一些.

5 AdaDelta / RMSprop

由于 AdaGrad 单调递减的学习率变化过于激进，改变二阶动量计算方法：不累积全部历史梯度，只关注过去一段时间窗口的下降梯度.

指数移动平均值：

$V_t = \beta_2 \cdot V_{t-1} + (1-\beta_2) g_t^2$

6 Adam (结合一阶动量和二阶动量)

$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$

$V_t = \beta_2 \cdot V_{t-1} + (1-\beta_2) g_t^2$.

Nadam

最后是Nadam。我们说Adam是集大成者，但它居然遗漏了Nesterov，这还能忍？必须给它加上——只需要按照NAG的步骤1来计算梯度：

$$g_t = \nabla f(w_t - \alpha \cdot m_{t-1}/\sqrt{V_t})$$

这就是Nesterov + Adam = Nadam了。

说到这里，大概可以理解为什么说 Adam / Nadam 目前最主流、最好用的算法了。无脑用Adam/Nadam，收敛速度嗖嗖滴，效果也是杠杠滴。

那为什么Adam还老招人黑，被学术界一顿鄙夷？难道只是为了发paper灌水吗？

5.2 Adam的问题

1. 可能不收敛

这篇是正在深度学习领域顶级会议之一 ICLR 2018 匿名审稿中的一篇论文《On the Convergence of Adam and Beyond》，探讨了Adam算法的收敛性，通过反例证明了Adam在某些情况下可能会不收敛。

回忆一下上文提到的各大优化算法的学习率：

$$\eta_t = \alpha/\sqrt{V_t}$$

其中，SGD没有用到二阶动量，因此学习率是恒定的（实际使用过程中会采用学习率衰减策略，因此学习率递减）。AdaGrad的二阶动量不断累积，单调递增，因此学习率是单调递减的。因此，这两类算法会使得学习率不断递减，最终收敛到0，模型也得以收敛。

但AdaDelta和Adam则不然。二阶动量是固定时间窗口内的累积，随着时间窗口的变化，遇到的数据可能发生巨变，使得
V_t
可能会时大时小，不是单调变化。这就可能在训练后期引起学习率的震荡，导致模型无法收敛。

这篇文章也给出了一个修正的方法。由于Adam中的学习率主要是由二阶动量控制的，为了保证算法的收敛，可以对二阶动量的变化进行控制，避免上下波动。

$$V_t = max(\beta_2 * V_{t-1} + (1-\beta_2)g_t^2, V_{t-1})$$

通过这样修改，就保证了

$$||V_t|| \geq ||V_{t-1}||$$

从而使得学习率单调递减。

2. 可能错过全局最优解

《Improving Generalization Performance by Switching from Adam to SGD》，进行了实验验证。他们CIFAR-10数据集上进行测试，Adam的收敛速度比SGD要快，但最终收敛的结果并没有SGD好。他们进一步实验发现，主要是后期Adam的学习率太低，影响了有效的收敛。他们试着对Adam的学习率的下界进行控制，发现效果好了很多。

于是他们提出了一个用来改进Adam的方法：前期用Adam，享受Adam快速收敛的优势；后期切换到SGD，慢慢寻找最优解。这一方法以前也被研究者们用到，不过主要是根据经验来选择切换的时机和切换后的学习率。这篇文章把这一切换过程傻瓜化，给出了切换SGD的时机选择方法，以及学习率的计算方法，效果看起来也不错。

6 不同算法的选择与使用策略

三.如何选择优化算法

如果数据是稀疏的，就用自适用方法，即 Adagrad, Adadelta, RMSprop, Adam。

RMSprop, Adadelta, Adam 在很多情况下的效果是相似的。

Adam 就是在 RMSprop 的基础上加了 bias-correction 和 momentum，

随着梯度变的稀疏，Adam 比 RMSprop 效果会好。

整体来讲，**Adam 是最好的选择。**

很多论文里都会用 SGD，没有 momentum 等。**SGD 虽然能达到极小值，但是比其它算法用的时间长，而且可能会被困在鞍点。**

如果需要更快的收敛，或者是训练更深更复杂的神经网络，需要用一种自适应的算法。

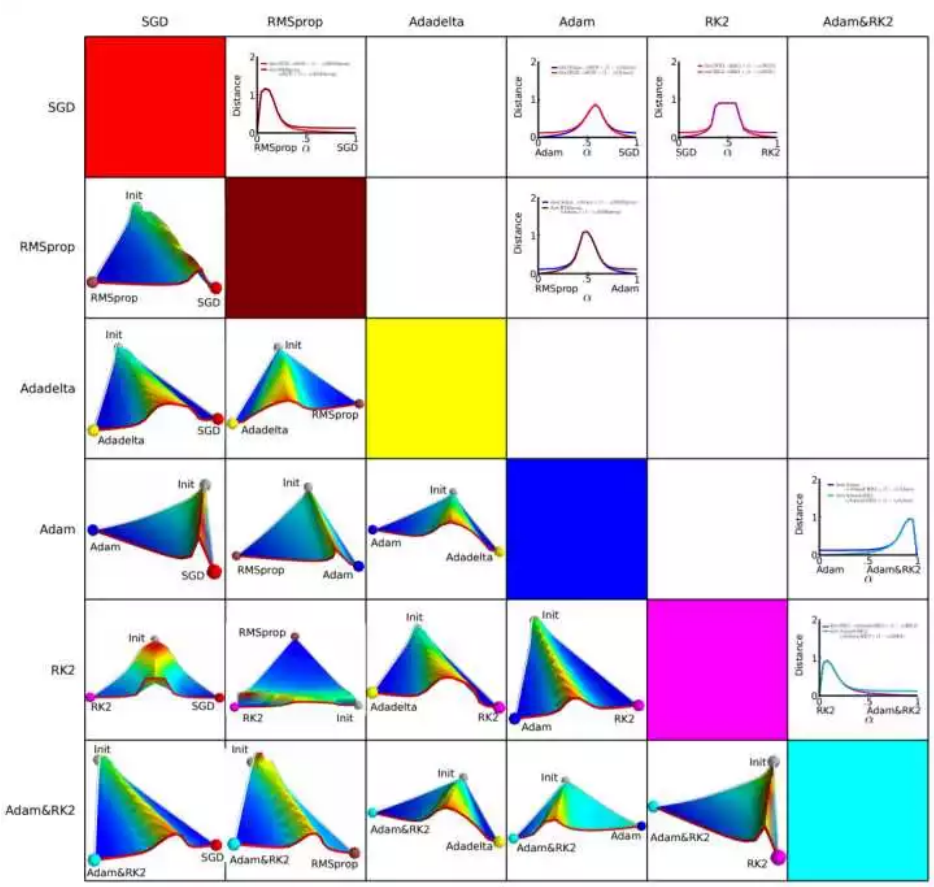
6.1 不同算法的核心差异：下降方向

从第一篇的框架中我们看到，不同优化算法最核心的区别，就是第三步所执行的下降方向：

$$\eta_t = (\alpha/\sqrt{V_t}) \cdot m_t$$

这个式子中，前半部分是实际的学习率（也即下降步长），后半部分是实际的下降方向。SGD算法的下降方向就是该位置的梯度方向的反方向，带一阶动量的SGD的下降方向则是该位置的一阶动量方向。自适应学习率类优化算法为每个参数设定了不同的学习率，在不同维度上设定不同步长，因此其下降方向是缩放过（scaled）的一阶动量方向。

由于下降方向的不同，可能导致不同算法到达完全不同的局部最优点。《An empirical analysis of the optimization of deep network loss surfaces》 这篇论文中做了一个有趣的实验，他们把目标函数值和相应的参数形成的超平面映射到一个三维空间，这样我们可以直观地看到各个算法是如何寻找超平面上的最低点的。



上图是论文的实验结果，横纵坐标表示降维后的特征空间，区域颜色则表示目标函数值的变化，红色是高原，蓝色是洼地。他们做的是配对儿实验，让两个算法从同一个初始化位置开始出发，然后对比优化的结果。可以看到，几乎任何两个算法都走到了不同的洼地，他们中间往往隔了一个很高的高原。这就说明，不同算法在高原的时候，选择了不同的下降方向。

6.2 Adam+SGD组合策略

主流的观点认为：Adam等自适应学习率算法对于稀疏数据具有优势，且收敛速度很快；但精调参数的SGD（+Momentum）往往能够取得更好的最终结果。

那么我们就会想到，可不可以把这两者结合起来，先用Adam快速下降，再用SGD调优，一举两得？思路简单，但里面有两个技术问题：

1. 什么时候切换优化算法？——如果切换太晚，Adam可能已经跑到自己的盆地里去了，SGD再怎么好也跑不出来了。
2. 切换算法以后用什么样的学习率？——Adam用的是自适应学习率，依赖的是二阶动量的累积，SGD接着训练的话，用什么样的学习率？

上一篇中提到的论文 Improving Generalization Performance by Switching from Adam to SGD 提出了解决这两个问题的思路。

首先来看第二个问题，切换之后的学习率。

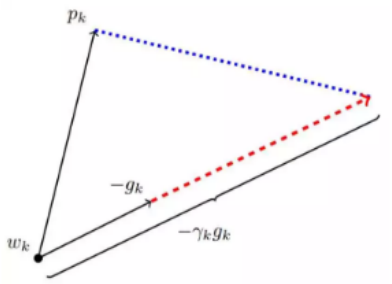
Adam的下降方向是

$$\eta_t^{Adam} = (\alpha/\sqrt{V_t}) \cdot m_t$$

而SGD的下降方向是

$$\eta_t^{SGD} = \alpha^{SGD} \cdot g_t$$

SGD下降方向必定可以分解为Adam下降方向及其正交方向上的两个方向之和，那么其在Adam下降方向上的投影就意味着SGD在Adam算法决定的下降方向上前进的距离，而在Adam下降方向的正交方向上的投影是 SGD 在自己选择的修正方向上前进的距离。



(图片来自原文，这里p为Adam下降方向，g为梯度方向，r为SGD的学习率。)

如果SGD要走完Adam未走完的路，那就首先要接过Adam的大旗——沿着 η_t^{Adam} 方向走一步，而后在沿着其正交方向走相应的一步。

这样我们就知道该如何确定SGD的步长（学习率）了——SGD在Adam下降方向上的正交投影，应该正好等于Adam的下降方向（含步长）。也即：

$$proj_{\eta_t^{SGD}} = \eta_t^{Adam}$$

解这个方程，我们就可以得到接续进行SGD的学习率：

$$\alpha_t^{SGD} = ((\eta_t^{Adam})^T \eta_t^{Adam})/((\eta_t^{Adam})^T g_t)$$

为了减少噪声影响，我们可以使用移动平均值来修正对学习率的估计：

$$\lambda_t^{SGD} = \beta_2 \cdot \lambda_{t-1}^{SGD} + (1-\beta_2) \cdot \alpha_t^{SGD}$$
$$\tilde{\lambda}_t^{SGD} = \lambda_t^{SGD}/(1-\beta_2^t)$$

这里直接复用了Adam的 beta 参数。

然后来看第一个问题，何时进行算法的切换。

作者提出的方法很简单，那就是当 SGD的相应学习率的移动平均值基本不变的时候，即：

$$|\tilde{\lambda}_t^{SGD} - \alpha_t^{SGD}| < \epsilon$$

6.3 优化算法常用tricks

1.首先，各大算法孰优孰劣并无定论。

如果是刚入门，优先考虑 SGD+Nesterov Momentum或者Adam.（Standford 231n ： The two recommended updates to use are either SGD+Nesterov Momentum or Adam）

主流的观点认为：Adam等自适应学习率算法对于稀疏数据具有优势，且收敛速度很快；但精调参数的SGD（+Momentum）往往能够取得更好的最终结果。

2. 选择你熟悉的算法——这样你可以更加熟练地利用你的经验进行调参。

3.充分了解你的数据——如果模型是非常稀疏的，那么优先考虑自适应学习率的算法。

4. 根据你的需求来选择——在模型设计实验过程中，要快速验证新模型的效果，可以先用Adam进行快速实验优化；在模型上线或者结果发布前，可以用精调的SGD进行模型的极致优化。

5. 先用小数据集进行实验。有论文研究指出，随机梯度下降算法的收敛速度和数据集的大小的关系不大。（The mathematics of stochastic gradient descent are amazingly independent of the training set size. In particular, the asymptotic SGD convergence rates are independent from the sample size. [2]）因此可以先用一个具有代表性的小数据集进行实验，测试一下最好的优化算法，并通过参数搜索来寻找最优的训练参数。

6. **考虑不同算法的组合**。先用Adam进行快速下降，而后再换到SGD进行充分的调优。切换策略可以参考本文介绍的方法。

7. **数据集一定要充分的打散（shuffle）**。这样在使用自适应学习率算法的时候，可以避免某些特征集中出现，而导致的有时学习过度、有时学习不足，使得下降方向出现偏差的问题。

8. **训练过程中持续监控训练数据和验证数据上的目标函数值以及精度或者AUC等指标的变化情况**。对训练数据的监控是要保证模型进行了充分的训练——下降方向正确，且学习率足够高；对验证数据的监控是为了避免出现过拟合。

9. **制定一个合适的学习率衰减策略**。可以使用定期衰减策略，比如每过多少个epoch就衰减一次；或者利用精度或者AUC等性能指标来监控，当测试集上的指标不变或者下跌时，就降低学习率。