梯度

【参考】

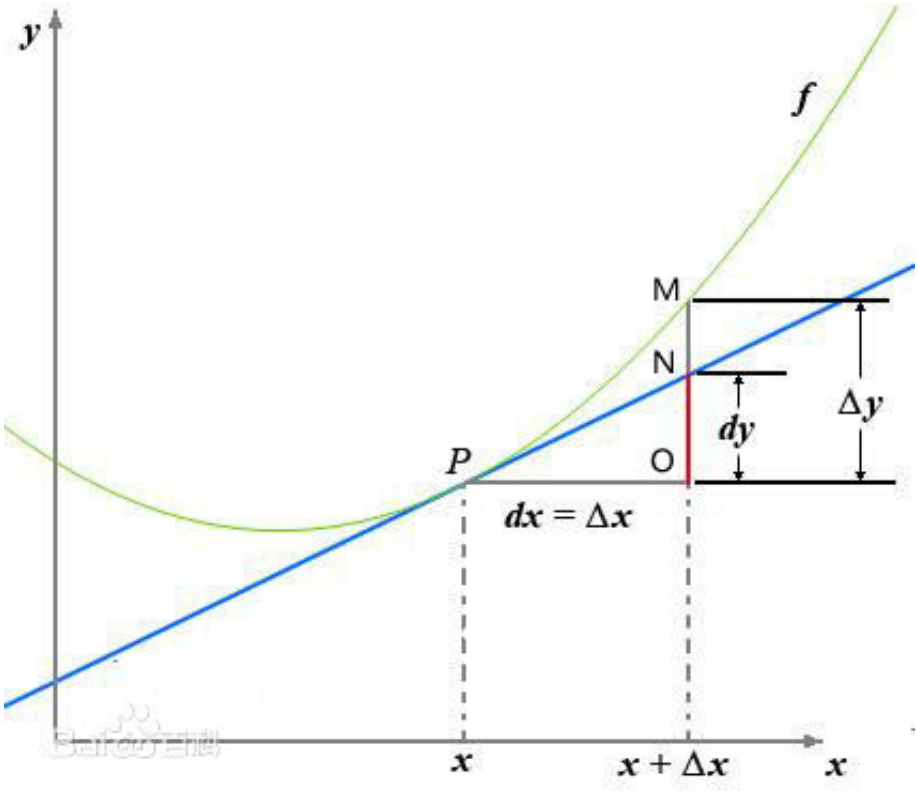
- csdn - ML重要概念：梯度（Gradient）与梯度下降法（Gradient Descent）
- cnblogs - 梯度下降（Gradient Descent）小结

在机器学习中，大部分问题都是优化问题，而绝大部分的优化问题都可以通过梯度下降（Gradient Descent）算法来解决。要明白什么是梯度下降算法，就需要明白什么是梯度，要不然就无法理解什么是梯度下降。

提到梯度，就必须从**导数（derivative）**、**偏导数（partial derivative）**和**方向导数（directional derivative）**讲起，弄清楚这些概念，才能够正确理解为什么在优化问题中使用梯度下降法来优化**目标函数**，并熟练掌握梯度下降法（Gradient Descent）。

导数

从下图可以看到导数的定义：



导数的定义如下：

$$f'(x_0) = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}$$

它反映了函数 $y = f(x)$ 在某一点处沿x轴正方向的**变化率**，即 函数 $f(x)$ 在x轴上某一点处沿着x轴正方向的**变化率/变化趋势**。直观地看，也就是在x轴上某一点处，如果 $f'(x) > 0$，说明 $f(x)$ 的函数值在x点沿x轴正方向是趋于增加的；如果$f'(x) < 0$，说明f(x)的函数值在x点沿x轴正方向是趋于减少的。

偏导数

偏导数是对于多变量函数而言的，定义如下：

$$\begin{aligned}\frac{\partial}{\partial x_j} f(x_0, x_1, \ldots, x_n) &= \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} \\ &= \lim_{\Delta x \to 0} \frac{f(x_0, \ldots, x_j + \Delta x, \ldots, x_n) - f(x_0, \ldots, x_j, \ldots, x_n)}{\Delta x}\end{aligned}$$

可以看到，**导数与偏导数本质是一致的，都是当自变量的变化量趋于0时，函数值的变化量与自变量变化量比值的极限。直观地说，偏导数也就是函数在某一点上沿坐标轴正方向的的变化率。**

区别在于：

- 导数，指的是一元函数中，函数$y = f(x)$在某一点处沿x轴正方向的变化率；
- 偏导数，指的是多元函数中，函数$y = f(x_1, x_2, \ldots, x_n)$在某一点处沿某一坐标轴$(x_1, x_2, \ldots, x_n)$正方向的变化率。

方向导数

偏导数的定义如下：

$$\begin{aligned}\frac{\partial}{\partial l} f(x_0, x_1, \ldots, x_n) &= \lim_{\rho \to 0} \frac{\Delta y}{\Delta x} \\ &= \lim_{\rho x \to 0} \frac{f(x_0 + \Delta x_0, \ldots, x_j + \Delta x_j, \ldots, x_n + \Delta x_n) - f(x_0, \ldots, x_j, \ldots, x_n)}{\rho} \\ \rho &= \sqrt{(\Delta x_0)^2 + \ldots + (\Delta x_j)^2 + \ldots + (\Delta x_n)^2}\end{aligned}$$

在前面导数和偏导数的定义中，均是沿坐标轴正方向讨论函数的变化率。那么当我们讨论函数沿任意方向的变化率时，也就引出了方向导数的定义，即：**某一点在某一趋近方向上的导数值。**

通俗的解释是：
我们不仅要知道函数在坐标轴正方向上的变化率（即偏导数），而且还要设法求得函数在其他特定方向上的变化率。而**方向导数就是函数在其他特定方向上的变化率。**

导数与梯度

梯度的定义如下：

$$grad f(x_0, x_1, \ldots, x_n) = \nabla f(x_0, x_1, \ldots, x_n) = \left(\frac{\partial f}{\partial x_0}, \ldots, \frac{\partial f}{\partial x_j}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

其实就是：**对多元函数的参数求∂偏导数，把求得的各个参数的偏导数以向量的形式写出来，就是梯度。**

比如函数$f(x,y)$, 分别对x,y求偏导数，求得的梯度向量就是$(\partial f/\partial x, \partial f/\partial y)^T$,简称$grad f(x,y)$或者$\nabla f(x,y)$。对于在点$(x_0,y_0)$的具体梯度向量就是$(\partial f/\partial x_0, \partial f/\partial y_0)^T$ 。或者$\nabla f(x_0,y_0)$，如果是3个参数的向量梯度，就是$(\partial f/\partial x, \partial f/\partial y, \partial f/\partial z)^T$，以此类推。

梯度的提出只为回答一个问题： **函数在变量空间的某一点处，沿着哪一个方向有最大的变化率**?
梯度定义如下：**函数在某一点的梯度是这样一个向量，它的方向与取得最大*方向导数*的方向一致，而它的模为方向导数的最大值。**

如：对于函数$f(x,y)$, 在点(x_0,y_0)，沿着梯度向量的方向就是$(\partial f/\partial x_0, \partial f/\partial y_0)^T$ 的方向是f(x,y)增加最快的地方。或者说，沿着梯度向量的方向，更加容易找到函数的最大值。反过来说，沿着梯度向量相反的方向，也就是 $-(\partial f/\partial x_0, \partial f/\partial y_0)^T$ 的方向，梯度减少最快，也就是更加容易找到函数的最小值。

梯度下降算法

概念：负梯度、

既然在变量空间的某一点处，函数沿梯度方向具有最大的变化率，那么在优化目标函数的时候，自然是沿着**负梯度**方向去减小函数值，以此达到我们的优化目标。

如何沿着负梯度方向减小函数值呢？既然梯度是偏导数的集合，如下：

$$grad f(x_0, x_1, \ldots, x_n) = \nabla f(x_0, x_1, \ldots, x_n) = \left(\frac{\partial f}{\partial x_0}, .., \frac{\partial f}{\partial x_j}, \ldots, \frac{\partial f}{\partial x_n} \right)$$

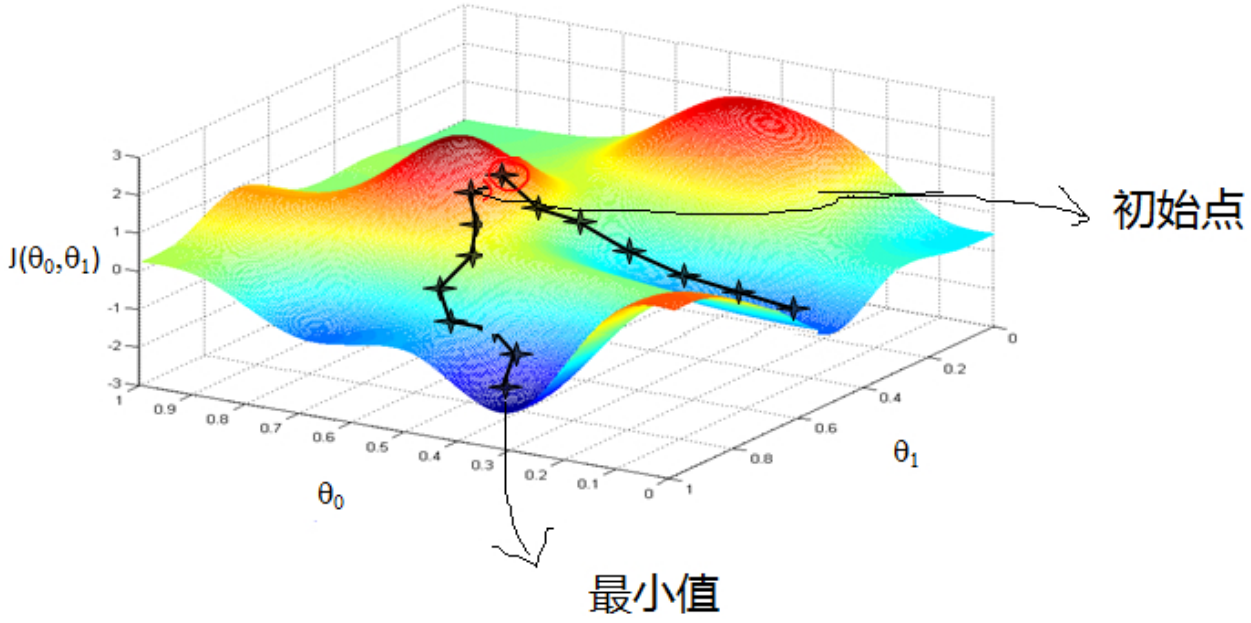
同时梯度和偏导数都是向量，那么参考向量运算法则，我们在每个变量轴上减小对应变量值即可，梯度下降法可以描述如下：

$$\begin{aligned} & Repeat\{ \\ & x_0 := x_0 - \alpha \frac{\partial f}{\partial x_0} \\ & \quad \vdots \\ & x_j := x_j - \alpha \frac{\partial f}{\partial x_j} \\ & \quad \vdots \\ & x_n := x_n - \alpha \frac{\partial f}{\partial x_n} \\ & \} \end{aligned} \tag{1}$$

直观解释

比如我们在一座大山上的某处位置，由于我们不知道怎么下山，于是决定走一步算一步，也就是在每走到一个位置的时候，求解当前位置的梯度，沿着梯度的负方向，也就是当前最陡峭的位置向下走一步，然后继续求解当前位置梯度，向这一步所在位置沿着最陡峭最易下山的位置走一步。这样一步步的走下去，一直走到觉得我们已经到了山脚。当然这样走下去，有可能我们不能走到山脚，而是到了某一个局部的山峰低处。

从上面的解释可以看出，梯度下降不一定能够找到全局的最优解，有可能是一个局部最优解。当然，如果损失函数是凸函数，梯度下降法得到的解就一定是全局最优解。



相关概念

1.步长（Learning rate）：即**学习速率**，式子（1）中的 α，步长决定了在梯度下降迭代的过程中，每一步沿梯度负方向前进的长度。用上面下山的例子，步长就是在当前这一步所在位置沿着最陡峭最易下山的位置走的那一步的长度。

2.特征（feature）：指的是样本中输入部分，比如2个单特征的样本 $(x^{(0)},y^{(0)})$，$(x^{(1)},y^{(1)})$,则第一个样本特征为$x^{(0)}$，第一个样本输出为 $y^{(0)}$。

3.假设函数（hypothesis function）：在监督学习中，为了拟合输入样本，而使用的假设函数，记为$h_\theta(x)$。比如对于单个特征的m个样本$(x^{(i)},y^{(i)})\ i=1,2,...m$,可以采用拟合函数如下：$h_\theta(x)=\theta_0+\theta_1 x$。

4.损失函数（loss function）：为了评估模型拟合的好坏，通常用损失函数来度量拟合的程度。损失函数极小化，意味着拟合程度最好，对应的模型参数即为最优参数。在线性回归中，损失函数通常为样本输出和假设函数的差取平方。比如对于m个样本（$(x^{(i)},y^{(i)})\ i=1,2,...m$，采用线性回归，损失函数为：

$$\mathcal{J}(\theta_0,\theta_1)=\sum_{i=1}^{m}(h_\theta(x_i)-y_i)^2$$

其中x_i表示第i个样本特征，y_i表示第i个样本对应的输出（标签），$h_\theta(x_i)$为假设函数。

算法详情

梯度下降法的算法可以有代数法和矩阵法（也称向量法）两种表示，代数法更加容易理解，但矩阵法更加的简洁，且由于使用了矩阵，实现逻辑更加的一目了然。

代数法实现

1）先决条件： 确认优化模型的假设函数和损失函数。
比如对于线性回归，假设函数表示为 $h_\theta(x_1,x_2,\ldots,x_n)=\theta_0+\theta_1x_1+\ldots+\theta_nx_n$，其中 $\theta_i(i=0,1,2...n)$为模型参数，$x_i(i=0,1,2...n)$为每个样本的 n 个特征值。这个表示可以简化，我们增加一个特征$x_0=1$，这样:

$$h_\theta(x_0,x_1,x_2,\ldots,x_n)=\sum_{i=0}^{n}\theta_ix_i。$$

同样是线性回归，对应于上面的假设函数，损失函数为：

$$\mathcal{J}(\theta_0,\theta_1\ldots,\theta_n)=\frac{1}{2m}\sum_{j=0}^{m}\left(h_\theta(x_0^{(j)},x_1^{(j)},\ldots x_n^{(j)})-y_j\right)^2 \tag{A1}$$

m 为训练集中样本的个数。

2.算法相关参数初始化：主要是初始化$\theta_0,\theta_1,\ldots,\theta_n$，算法终止距离 ε 以及步长 α。通常将所有的 θ 初始化为0，将步长初始化为 1，在调优的时候再优化。

3.算法过程：

a）确定当前位置的损失函数的梯度，对于 θ_i,其梯度表达式如下：

$$\frac{\partial}{\partial\theta_i}\mathcal{J}(\theta_0,\theta_1\ldots,\theta_n)$$

b）用步长乘以损失函数的梯度，得到当前位置下降的距离，即$\alpha\frac{\partial}{\partial\theta_i}\mathcal{J}(\theta_0,\theta_1\ldots,\theta_n)$对应于前面登山例子中的某一步。

c）确定是否所有的θ_i梯度下降的距离都小于 ε，如果小于 ε 则算法终止，当前所有的 $\theta_i(i=0,1,\ldots,n)$即为最终结果。否则进入步骤 d.

d）更新所有的θ，对于θ_i，其更新表达式如下。更新完毕后继续转入步骤 a:

$$\theta_i:=\theta_i-\alpha\frac{\partial}{\partial\theta_i}\mathcal{J}(\theta_0,\theta_1\ldots,\theta_n)$$

代数法例子

假设我们的样本是 $(x_1^{(0)},x_2^{(0)},\ldots x_n^{(0)},y_0),(x_1^{(1)},x_2^{(1)},\ldots x_n^{(1)},y_1),\ldots(x_1^{(m)},x_2^{(m)},\ldots x_n^{(m)},y_m)$,损失函数如前面先决条件所述：

$$\mathcal{J}(\theta_0,\theta_1\ldots,\theta_n)=\frac{1}{2m}\sum_{j=0}^{m}\left(h_\theta(x_0^{(j)},x_1^{(j)},\ldots x_n^{(j)})-y_j\right)^2 \tag{2}$$

假设函数为：

$$h_\theta(x_0, x_1, x_2, \ldots, x_n) = \theta_0 x_0 + \theta_1 x_1 + \ldots + \theta_n x_n \tag{3}$$

则在算法过程步骤1中对于θ_i 的偏导数通过式子（2）和（3）可以推导出：

$$\frac{\partial}{\partial \theta_i} \mathcal{J}(\theta_0, \theta_1 \ldots, \theta_n) = \frac{1}{m} \sum_{j=0}^{m} \left(h_\theta(x_0^{(j)}, x_1^{(j)}, \ldots x_n^{(j)}) - y_j \right) x_i^{(j)} \tag{4}$$

由于样本中没有x_0上式中令所有的x_0^j为1。那么算法中 d 步的跟新公式如下：

$$\theta_i := \theta_i - \alpha \frac{1}{m} \sum_{j=0}^{m} \left(h_\theta(x_0^{(j)}, x_1^{(j)}, \ldots x_n^{(j)}) - y_j \right) x_i^{(j)} \tag{5}$$

从这个例子以及（4）式可以看出**当前点的梯度方向是由所有的样本决定的**，加$\frac{1}{m}$ 是为了好理解。由于步长也为常数，他们的乘机也为常数，所以这里$\alpha \frac{1}{m}$可以用一个常数表示。

矩阵方法的实现

此处会用到矩阵相关的知识，尤其是矩阵求导的知识。

1. 先决条件： 需要确认优化模型的假设函数和损失函数。对于线性回归，假设函数 $h_\theta(x_1, x_2, \ldots, x_n) = \theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n$的矩阵表达方式为：

$$h_\theta(\mathbf{X}) = \mathbf{X}\theta \tag{B1}$$

其中， 假设函数$h_\theta(\mathbf{X})$为 mx1 的向量，θ为 (n+1)×1 的向量，里面有 n+1 $(\theta_0, .., \theta_n)$个代数法的模型参数。X 为 m×(n+1) 维的矩阵。m 代表样本的个数，n+1$(x_0, \ldots, x_n)$代表样本的特征数。

损失函数的表达式为：

$$\mathcal{J}(\theta) = \frac{1}{2}(\mathbf{X}\theta - \mathbf{Y})^{\mathsf{T}}(\mathbf{X}\theta - \mathbf{Y}) \tag{B2}$$

其中 Y 是样本的输出向量，维度为 mx1 .

2. 算法相关参数初始化: θ向量可以初始化为默认值，或者调优后的值。算法终止距离ε，步长α。
3. 算法过程：

a）确定当前位置的损失函数的梯度，对于 θ,其梯度表达式如下：

$$\frac{\partial}{\partial \theta} \mathcal{J}(\theta) \tag{B3}$$

b）用步长乘以损失函数的梯度，得到当前位置下降的距离，即$\alpha \frac{\partial}{\partial \theta} \mathcal{J}(\theta)$对应于前面登山例子中的某一步。

c）确定是否所有的θ_i梯度下降的距离都小于 ε，如果小于 ε 则算法终止，当前所有的 $\theta_i (i = 0, 1, \ldots, n)$即为最终结果。否则进入步骤 d.

d）更新所有的θ，对于θ_i，其更新表达式如下。更新完毕后继续转入步骤 a:

$$\theta := \theta - \alpha \frac{\partial}{\partial \theta} \mathcal{J}(\theta) \tag{B4}$$

矩阵方法例子

还是用线性回归的例子来描述具体的算法过程。矩阵常见的求导规则：

- 加减法：$d(X \pm Y) = dX \pm dY$
- 乘法：$d(XY) = (dX)Y + XdY$
- 转置：$d(X^T) = (dX)^T$
- 逆：$dX^{-1} = -X^{-1}dXX^{-1}$

损失函数对于θ向量的偏导数，即对式（B2）求导，计算如下：

$$\begin{aligned} \frac{\partial}{\partial \theta} \mathcal{J}(\theta) &= \frac{\partial}{\partial \theta} \frac{1}{2} (\mathbf{X}\theta - \mathbf{Y})^{\mathsf{T}} (\mathbf{X}\theta - \mathbf{Y}) \\ &= \mathbf{X}^T (\mathbf{X}\theta - \mathbf{Y}) \end{aligned} \tag{B5}$$

那么 θ 向量的更新表达式如下：

$$\theta := \theta - \alpha \mathbf{X}^T (\mathbf{X}\theta - \mathbf{Y})$$

调优

1. 算法的步长选择。在前面的算法描述中，我提到取步长为1，但是实际上取值取决于数据样本，可以多取一些值，从大到小，分别运行算法，看看迭代效果，如果损失函数在变小，说明取值有效，否则要增大步长。前面说了。步长太大，会导致迭代过快，甚至有可能错过最优解。步长太小，迭代速度太慢，很长时间算法都不能结束。所以算法的步长需要多次运行后才能得到一个较为优的值。
2. 算法参数的初始值选择。 初始值不同，获得的最小值也有可能不同，因此梯度下降求得的只是局部最小值；当然如果损失函数是凸函数则一定是最优解。由于有局部最优解的风险，需要多次用不同初始值运行算法，关键损失函数的最小值，选择损失函数最小化的初值。
3. 归一化。由于样本不同特征的取值范围不一样，可能导致迭代很慢，为了减少特征取值的影响，可以对特征数据归一化，也就是对于每个特征x，求出它的均值 $\bar{x}$ 和标准差std(x)，然后转化为：

$$\frac{x - \bar{x}}{std(x)}$$

这样特征的均值为0，方差为1，迭代次数可以大大加快。

其他梯度下降算法

批量梯度下降（Batch Gradient Descent）

批量梯度下降法，是梯度下降法最常用的形式，具体做法也就是在更新参数时使用**所有的样本**来进行更新，这即前面描述的线性回归的梯度下降算法，也就是上面的梯度下降算法就是批量梯度下降法。提应的更新公式为：

$$\theta_i := \theta_i - \alpha \frac{1}{m} \sum_{j=0}^{m} \left(h_\theta(x_0^{(j)}, x_1^{(j)}, \dots x_n^{(j)}) - y_j \right) x_i^{(j)} \tag{6}$$

由于我们有m个样本，这里求梯度的时候就用了所有m个样本的梯度数据。

随机梯度下降法（Stochastic Gradient Descent）

随机梯度下降法，与批量梯度下降法原理类似，区别在与求梯度时没有用所有的 m 个样本的数据，而是仅仅选取一个样本 j 来求梯度。因此每次得到一个样本，就执行一次参数更新，又被成为**在线学习**。对应的更新公式是：

$$\theta_i := \theta_i - \alpha \left(h_\theta(x_0^{(j)}, x_1^{(j)}, \dots x_n^{(j)}) - y_j \right) x_i^{(j)} \tag{7}$$

随机梯度下降法，和的批量梯度下降法是两个极端，一个采用所有数据来梯度下降，一个用一个样本来梯度下降。自然各自的优缺点都非常突出。

- 对于训练速度来说，随机梯度下降法由于每次仅仅采用一个样本来迭代，训练速度很快，而批量梯度下降法在样本量很大的时候，训练速度不能让人满意，而且还会面临内存不足的问题。
- 对于准确度来说，随机梯度下降法用于仅仅用一个样本决定梯度方向，导致解很有可能不是最优。
- 对于收敛速度来说，由于随机梯度下降法一次迭代一个样本，导致迭代方向变化很大，不能很快的收敛到局部最优解。

那么，有没有一个中庸的办法能够结合两种方法的优点呢？有！这就是小批量梯度下降法。

小批量梯度下降法（Mini-batch Gradient Descent）

小批量梯度下降法是批量梯度下降法和随机梯度下降法的折衷，也就是对于m个样本，我们采用x个样子来迭代，1<x<m。一般可以取x=10，当然根据样本的数据，可以调整这个 x 的值。对应的更新公式是：

$$\theta_i := \theta_i - \alpha \sum_{j=t}^{t+x-1} \left(h_\theta(x_0^{(j)}, x_1^{(j)}, \dots x_n^{(j)}) - y_j \right) x_i^{(j)} \tag{8}$$

与其他优化算法比

在机器学习中的无约束优化算法，除了梯度下降以外，还有前面提到的最小二乘法，此外还有牛顿法和拟牛顿法。

梯度下降法和最小二乘法相比，梯度下降法需要**选择步长**，而最小二乘法不需要。梯度下降法是**迭代求解**，最小二乘法是计算解析解。如果样本量不算很大，且存在解析解，最小二乘法比起梯度下降法要有优势，计算速度很快。但是如果样本量很大，用最小二乘法由于需要求一个超级大的逆矩阵，这时就很难或者很慢才能求解解析解了，使用迭代的梯度下降法比较有优势。

梯度下降法和牛顿法/拟牛顿法相比，两者都是迭代求解，不过梯度下降法是**梯度求解**，而牛顿法/拟牛顿法是用**二阶海森矩阵的逆矩阵或伪逆矩阵**求解。相对而言，使用牛顿法/拟牛顿法收敛更快。但是每次迭代的时间比梯度下降法长。

梯度下降算法改进

【参考】

- csdn - 一文看懂常用的梯度下降算法
- 知乎- 一文看懂常用的梯度下降算法与上文类似

对于神经网络模型，借助于BP算法可以高效地计算梯度，从而实施梯度下降算法。但梯度下降算法一个老大难的问题是：不能保证全局收敛。如果这个问题解决了，深度学习的世界会和谐很多。梯度下降算法针对凸优化问题原则上是可以收敛到全局最优的，因为此时只有唯一的局部最优点。而实际上深度学习模型是一个复杂的非线性结构，一般属于非凸问题，这意味着存在很多局部最优点（鞍点），采用梯度下降算法可能会陷入局部最优，这应该是最头疼的问题。这点和进化算法如遗传算法很类似，都无法保证收敛到全局最优。因此，我们注定在这个问题上成为“高级调参师”。可以看到，梯度下降算法中一个重要的参数是学习速率，适当的学习速率很重要：学习速率过小时收敛速度慢，而过大时导致训练震荡，而且可能会发散。理想的梯度下降算法要满足两点：**收敛速度要快；能全局收敛**。为了这个理想，出现了很多经典梯度下降算法的变种，如带动量的梯度下降算法、NAG、AdaGrad、RMSprop、Adam。

Momentum optimization

冲量（Momentum）梯度下降算法是BorisPolyak在1964年提出的，其基于这样一个物理事实：将一个小球从山顶滚下，其初始速率很慢，但在加速度作用下速率很快增加，并最终由于阻力的存在达到一个稳定速率。对于冲量梯度下降算法，其更新方程如下：

$$\begin{aligned}\mathbf{m} &\leftarrow \gamma \cdot \mathbf{m} + \eta \cdot \nabla_\theta \mathcal{J}(\theta) \\ \theta &\leftarrow \theta - \mathbf{m}\end{aligned} \tag{C1}$$

可以看到，参数更新时不仅考虑当前梯度值，而且加上了一个积累项（冲量），但多了一个超参 γ，一般初始值取接近1的值如0.9。相比原始梯度下降算法，冲量梯度下降算法有助于加速收敛。当梯度与冲量方向一致时，冲量项会增加，而相反时，冲量项减少，因此冲量梯度下降算法可以减少训练的震荡过程。

有时会还可以写成：

$$\begin{aligned}\mathbf{m} &\leftarrow \beta \cdot \mathbf{m} + (1-\beta) \cdot \nabla J(\theta) \\ \theta &\leftarrow \theta - \eta \cdot \mathbf{m}\end{aligned}$$

此时我们就可以清楚地看到，所谓的冲量项其实只是梯度的指数加权移动平均值。这个实现和之前的实现没有本质区别，只是学习速率进行了放缩一下而已。

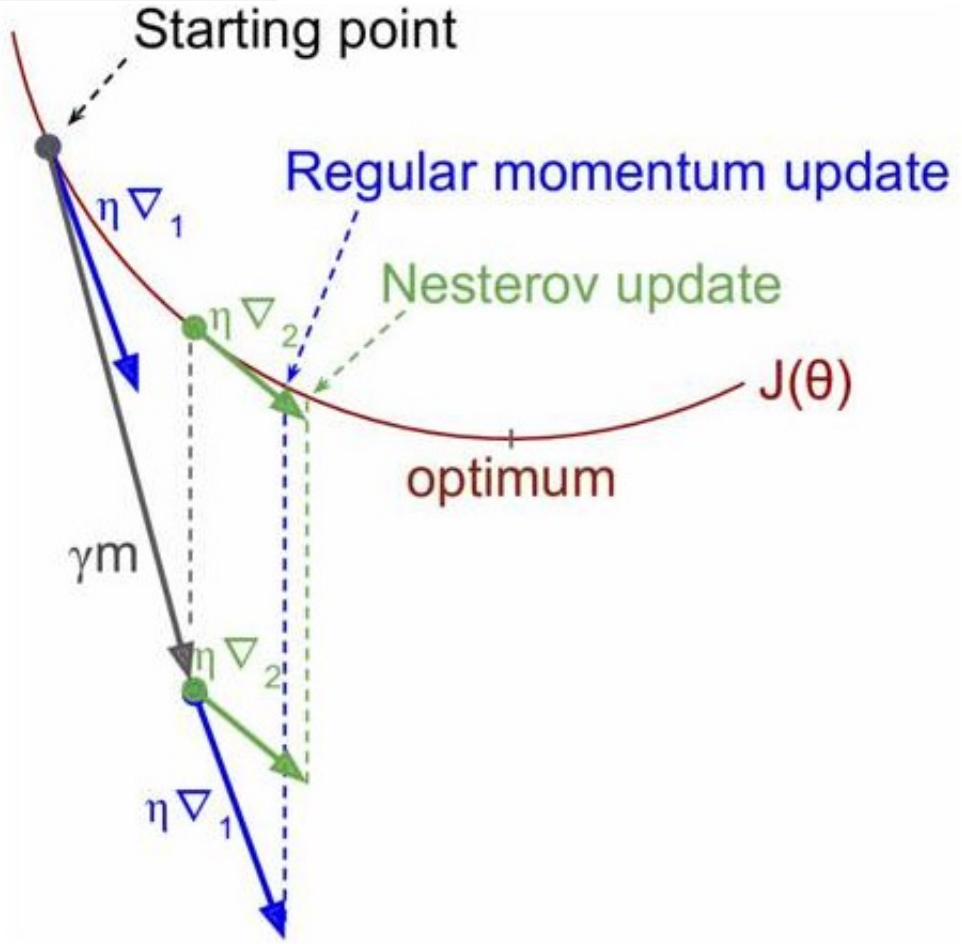
TensorFlow中提供了这一优化器：`tf.train.MomentumOptimizer(learning_rate=learning_rate,momentum=0.9)`。

NAG

NAG 算法全称 Nesterov Accelerated Gradient，是YuriiNesterov在1983年提出的对冲量梯度下降算法的改进版本，其速度更快。其变化之处在于计算**超前梯度**更新冲量项，具体公式如下：

$$\begin{aligned}\mathbf{m} &\leftarrow \gamma \cdot \mathbf{m} + \eta \cdot \nabla_\theta \mathcal{J}(\theta - \gamma \cdot \mathbf{m}) \\ \theta &\leftarrow \theta - \mathbf{m}\end{aligned} \tag{C2}$$

既然参数要沿着 $\gamma \cdot \mathbf{m}$ 更新，不妨计算未来位置 $\theta - \gamma \cdot \mathbf{m}$ 的梯度，然后合并两项作为最终的更新项，其具体效果如下图所示，可以看到一定的加速效果。在TensorFlow中，NAG优化器为：`tf.train.MomentumOptimizer(learning_rate=learning_rate,momentum=0.9, use_nesterov=True)`

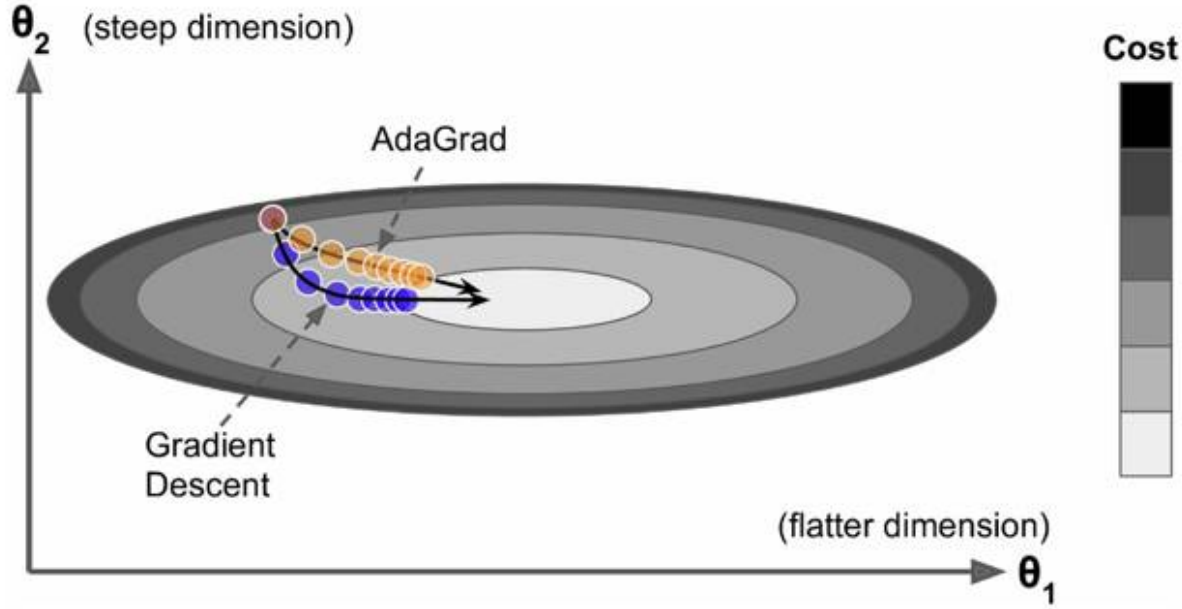


AdaGrad

AdaGrad是Duchi在2011年提出的一种**学习速率自适应**的梯度下降算法。在训练迭代过程，其学习速率是**逐渐衰减**的，经常更新的参数其学习速率衰减更快，这是一种**自适应算法**。其更新过程如下：

$$\begin{aligned}\mathbf{s} &\leftarrow \mathbf{s} + \nabla_\theta \mathcal{J}(\theta) \odot \nabla_\theta \mathcal{J}(\theta) \\ \theta &\leftarrow \theta - \frac{\eta}{\sqrt{\mathbf{s} + \varepsilon}} \odot \nabla_\theta \mathcal{J}(\theta)\end{aligned} \tag{C3}$$

其中 s 是梯度平方的积累量，在进行参数更新时，学习速率要除以这个积累量的平方根，其中加上一个很小值是为了防止除0的出现。由于是该项逐渐增加的，那么学习速率是衰减的。考虑下图所示的情况，目标函数在两个方向的坡度不一样，如果是原始的梯度下降算法，在接近坡底时收敛速度比较慢。而当采用AdaGrad，这种情况可以被改观。由于比较陡的方向梯度比较大，其学习速率将衰减得更快，这有利于参数沿着更接近坡底的方向移动，从而加速收敛。



前面说到AdaGrad其学习速率实际上是不断衰减的，这会导致一个很大的问题，就是训练后期学习速率很小，导致训练过早停止，因此在实际中AdaGrad一般不会被采用，下面的算法将改进这一致命缺陷。不过TensorFlow也提供了这一优化器：`tf.train.AdagradOptimizer`。

RMSprop

RMSprop 是 Hinton 在他的课程上讲到的，其算是对Adagrad算法的改进，主要是解决学习速率过快衰减的问题。其实思路很简单，类似 Momentum 思想，引入一个超参数，在积累梯度平方项进行衰减：

$$\begin{aligned} \mathbf{s} &\leftarrow \gamma \cdot \mathbf{s} + (1-\gamma)\nabla_\theta \mathcal{J}(\theta) \odot \nabla_\theta \mathcal{J}(\theta) \\ \theta &\leftarrow \theta - \frac{\eta}{\sqrt{\mathbf{s}+\varepsilon}} \odot \nabla_\theta \mathcal{J}(\theta) \end{aligned} \tag{C4}$$

此时可以看到 s 是梯度平方的指数加权移动平均值，其中 γ 一般取值 **0.9**，其实这样就是一个指数衰减的均值项，减少了出现的爆炸情况，因此有助于避免学习速率很快下降的问题。同时Hinton也建议学习速率设置为 **0.001**。RMSprop是属于一种比较好的优化算法了，在TensorFlow中当然有其身影：`tf.train.RMSPropOptimizer(learning_rate=learning_rate, momentum=0.9, decay=0.9, epsilon=1e-10)`。

同时期还有一个 Adadelta 算法，其也是 Adagrad 算法的改进，而且改进思路和RMSprop很像，但是其背后是基于一次梯度近似代替二次梯度的思想。

Adam

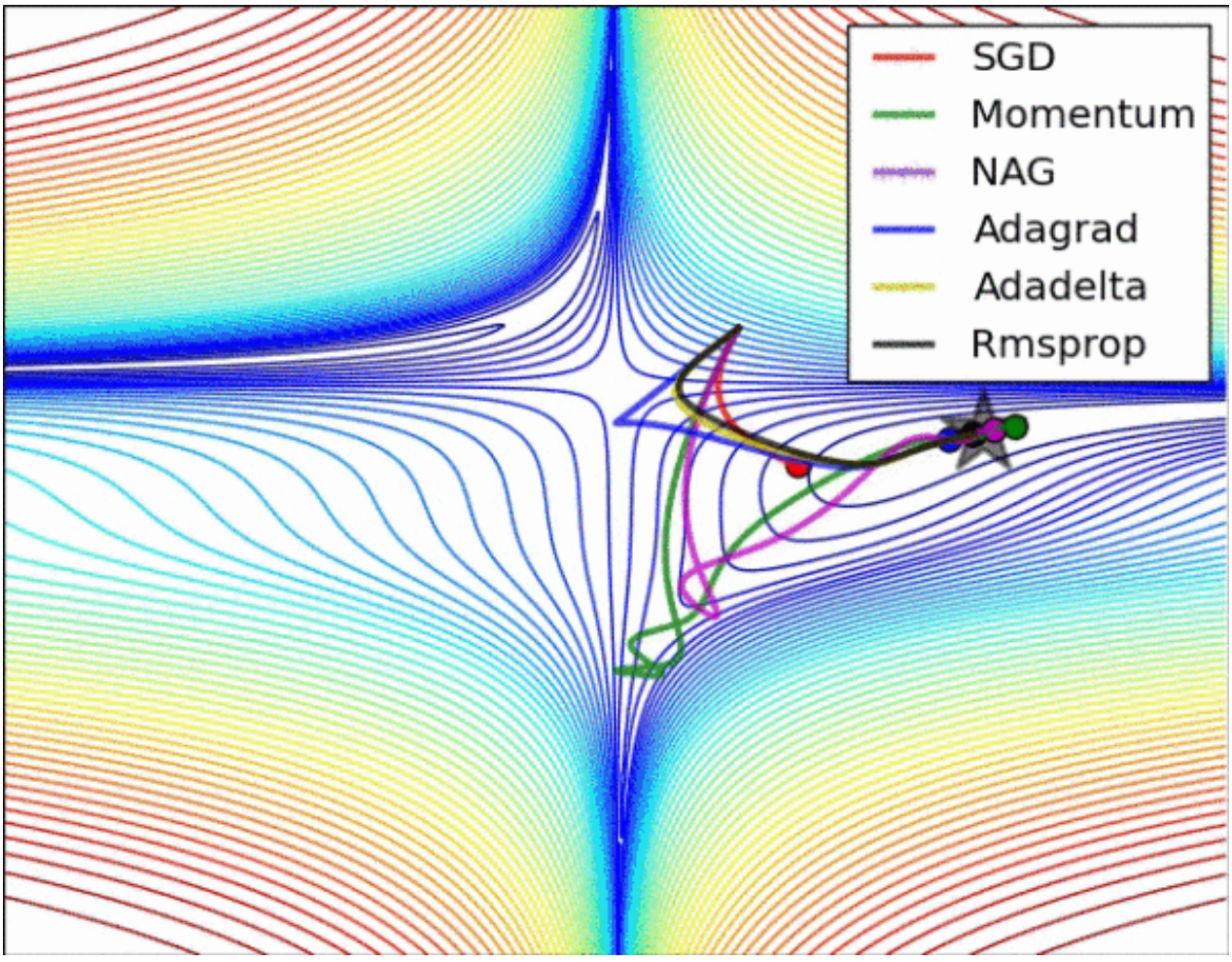
Adam 全称 Adaptive moment estimation，是Kingma等在2015年提出的一种新的优化算法，其结合了 Momentum 和 RMSprop 算法的思想。相比 Momentum 算法，其学习速率是自适应的，而相比 RMSprop，其增加了冲量项。所以，Adam是两者的结合体：

$$
\begin{aligned}
\mathbf{m} &\leftarrow \beta_1 \cdot \mathbf{m} + (1-\beta_1) \cdot \nabla_\theta \mathcal{J}(\theta) \\
\mathbf{s} &\leftarrow \beta_2 \cdot \mathbf{s} + (1-\beta_2) \nabla_\theta \mathcal{J}(\theta) \odot \nabla_\theta \mathcal{J}(\theta) \\
\mathbf{m} &\leftarrow \frac{\mathbf{m}}{1-\beta_1} \\
\mathbf{s} &\leftarrow \frac{\mathbf{s}}{1-\beta_2} \\
\theta &\leftarrow \theta - \frac{\eta}{\sqrt{\mathbf{s}+\varepsilon}} \odot \mathbf{m}
\end{aligned}
\tag{C45}
$$

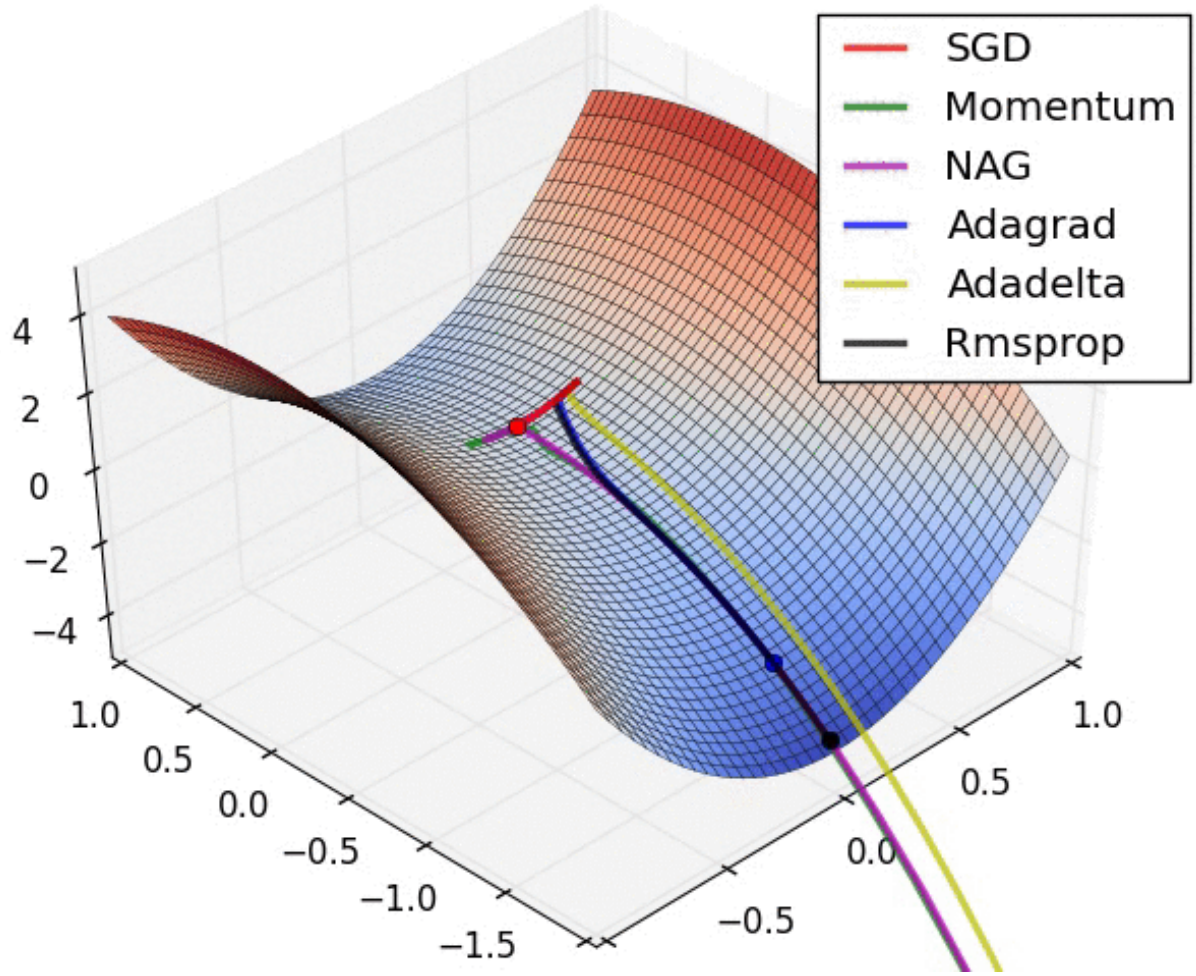
可以看到前两项和 Momentum 和 RMSprop 是非常一致的，由于和的初始值一般设置为0，在训练初期其可能较小，第三和第四项主要是为了放大它们，最后一项是参数更新。其中超参数的建议值是 $\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 1e-8$。Adm是性能非常好的算法，在TensorFlow其实现如下：`tf.train.AdamOptimizer(learning_rate=0.001,beta1=0.9, beta2=0.999, epsilon=1e-08)`。

算法的比较

在非鞍点的情况：

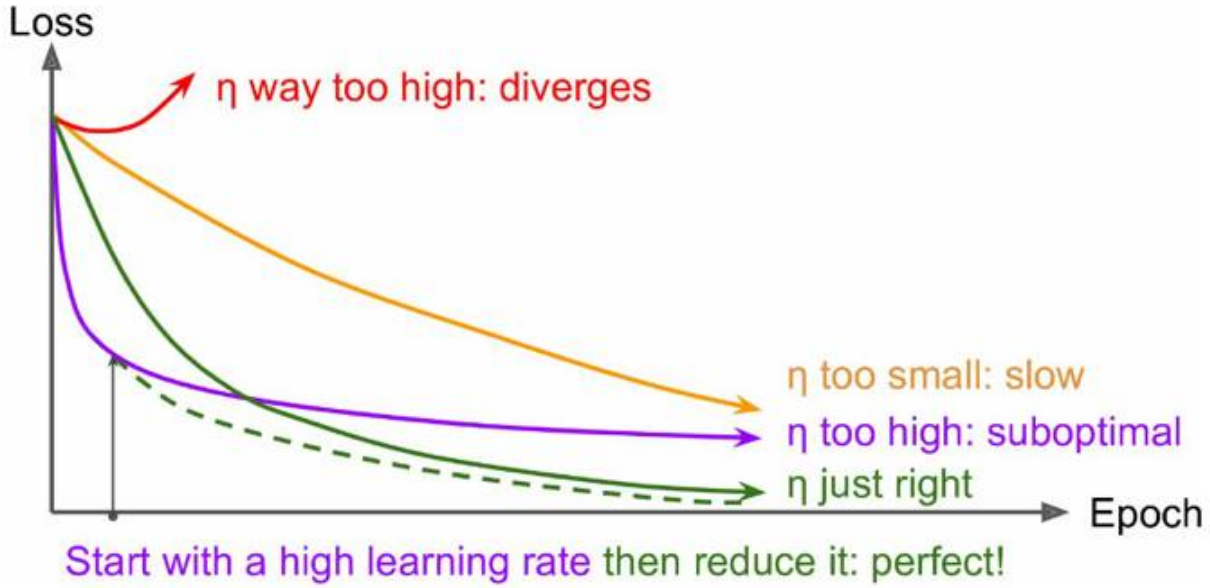


在鞍点的情况：



学习速率问题

对于梯度下降算法，这应该是一个最重要的超参数。如果学习速率设置得非常大，那么训练可能不会收敛，就直接发散了；如果设置的比较小，虽然可以收敛，但是训练时间可能无法接受；如果设置的稍微高一些，训练速度会很快，但是当接近最优点会发生震荡，甚至无法稳定。不同学习速率的选择影响可能非常大，如下图所示。

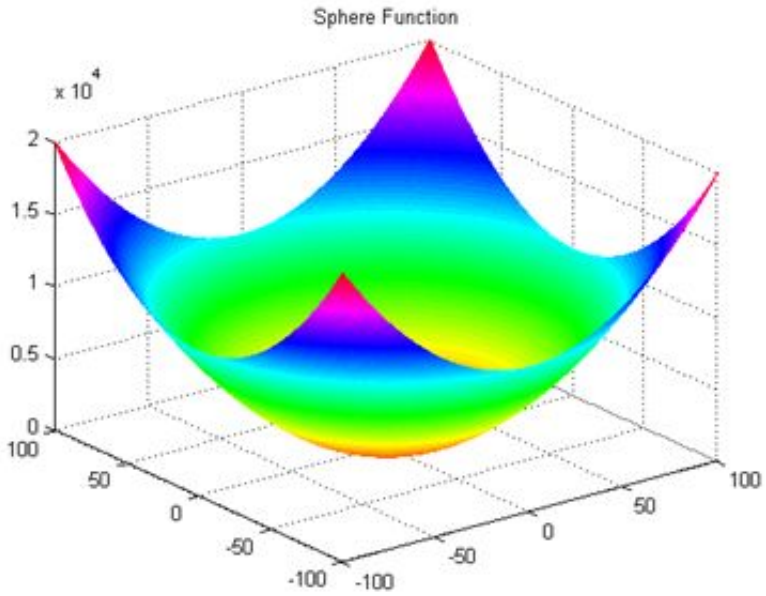


理想的学习速率是：刚开始设置较大，有很快的收敛速度，然后慢慢衰减，保证稳定到达最优点。所以，前面的很多算法都是学习速率自适应的。除此之外，还可以手动实现这样一个自适应过程，如实现学习速率指数式衰减：

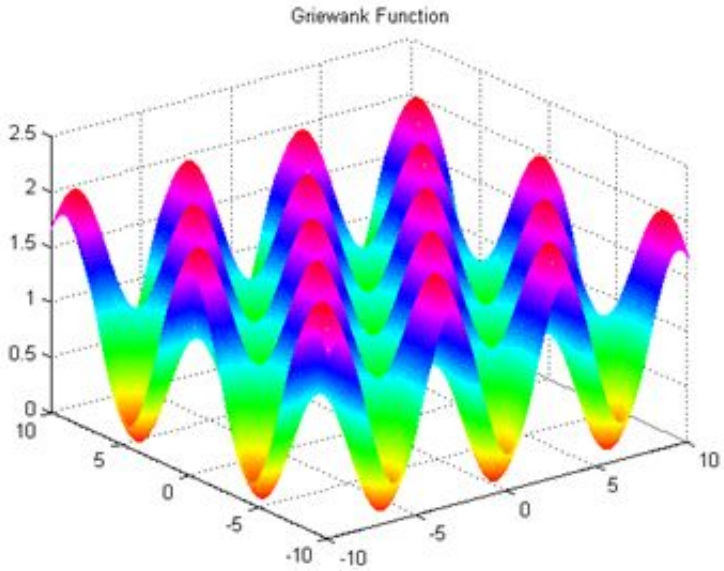
$$\eta(t) = \eta_0 \cdot 10^{-\frac{t}{r}}$$

局部最优与鞍点

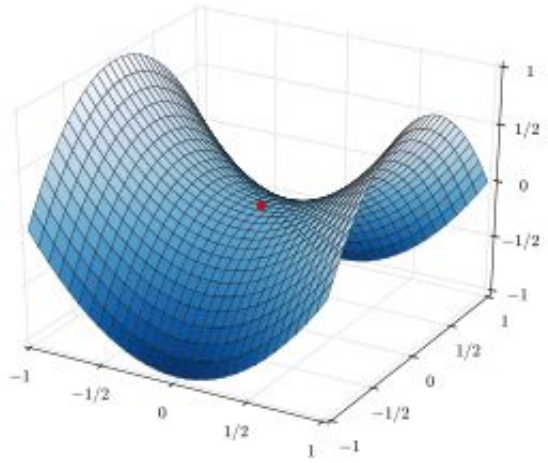
前面说到，我们希望优化算法能够收敛速度快，并且想找到全局最优。对于凸函数来说，其仅有一个极值点，就是全局最优点，此时采用梯度下降算法是可以收敛到最优点的，因为沿着下坡的道路走就可以了。如下图：



但是其实现在的深度学习模型是一个庞大的非线性结构，这样其一般是非凸函数，就如下图所示那样，存在很多局部最优点（local optimum），一旦梯度下降算法跳进局部陷阱，可以想象其很难走出来，此时梯度下降算法变得不再那么可靠，因为我们想要的是全局最优。很难找到全局最优，这可能是目前优化算法共同面对的问题。



不过到底深度学习的损失函数是不是存在很多局部最优点呢？前面所有的分析都是基于低维空间，我们很容易观察到局部最优点。但是深度学习的参数一般庞大，其损失函数已经成为了超高维空间。但是Bengio等最新的研究表明，对于高维空间，非凸函数最大的存在不是局部最优点，而是鞍点（saddle point），鞍点也是梯度为0的点，但是它不像局部最优点或者全局最优点。如下图：



对于局部最优或者全局最优点，其周围的所有方向要朝向上（最小）或者朝向上（最大），但是考虑到参数庞大，很有可能是一部分方向朝下，一部分方向朝上，这就成为了鞍点。意思就是说在高维度空间，不大可能像低维度空间那样出现很多局部最优。而且鞍点也不大可能会成为梯度下降算法的葬身之地。那么真正影响梯度下降算法会是什么呢？可能是**平稳区（plateaus）**，如果出现大面积梯度很小或者近似为0的区域，那么梯度下降算法就找不到方向，想象你自己站在一望无际的平原，估计你也方向感全无了。当前上面所有的谈论仅是停留在经验，对于高维空间，无论如何很难直观想象，这还是一个历史难题吧。