

# 重慶大學

最优化



梯度下降法实验报告

姓名：贾旺旺

学号：20204251

# 梯度下降法及其改进方法分析

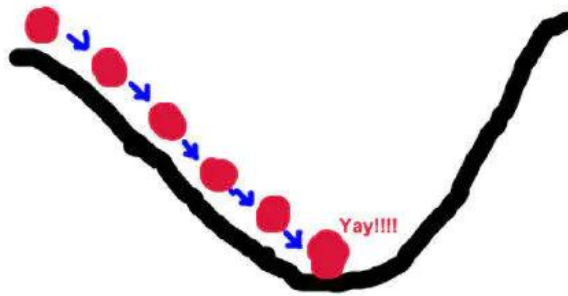
## 摘要

在本章，以及我们首先研究了不同数据量对梯度下降法的影响。紧接着，我们通过实例探讨了梯度下降法的优缺点，并我们通过研究梯度下降法的超因素冲量和学习率衰减率对梯度下降法的改进得出改进后算法的优缺点。

最后我们总结了一些关于梯度下降法的看法。

## 一、引言

我们假设这样一个场景：一个人需要从某处开始下山，尽快到达山底。在下山之前他需要确认下山的方向和下山的距离两件事。这是因为下山的路有很多，他必须利用一些信息，找到从该处开始最陡峭的方向下山，这样可以保证他尽快到达山底。此外，这座山最陡峭的方向并不是一成不变的，每当走过一段规定的距离，他必须停下来，重新利用现有的信息找到新的最陡峭的方向。通过反复进行该过程，最终抵达山底。



下山过程简图

这一过程形象的描述了梯度下降法求解无约束最优化问题的过程。下面我们将例子中的关键信息与梯度下降法中的关键信息对应起来：山代表了需要优化的函数表达式；山的最低点就是该函数的最优值，也就是我们的目标；每次下山的距离代表后面要解释的学习率；寻找方向利用的信息即为样本数据；最陡峭的下山方向则与函数表达式梯度的方向有关，之所以要寻找最陡峭的方向，是为了满足最快到达山底的限制条件；细心的读者可能已经发现上面还有一处加粗的词组：某处——代表了我们给优化函数设置的初始值，算法后面正是利用这个初始值进行不断的迭代求出最优解。

在选择每次行动的距离时，如果所选择的距离过大，则有可能偏离最陡峭的方向，甚至已经到达了最低点却没有停下来，从而跨过最低点而不自知，一直无法到达山底；如果距离过小，则需要频繁寻找最陡峭的方向，会非常耗时。要知道，每次寻找最陡峭的方向是非常复杂的！同样的，梯度下降法也会面临这个问题，因此需要我们找到最佳的学习率，在不偏离方向的同时耗时最短。

在下一个板块，我们将从梯度和学习率两个板块来展开梯度下降法介绍。

## 二、相关概念

### 2.1 学习率 $\alpha$

学习率也被称为迭代的步长，优化函数的梯度一般是不断变化的（梯度的方向随梯度的变化而变化），因此需要一个适当的学习率约束着每次下降的距离不会太多也不会太少。读者可参考上面讲过的下山例子中下山距离的讲解，此处不再赘述。

### 2.2 梯度

在一元函数中，梯度其实就是微分，既函数的变化率，而在多元函数中，梯度变为了向量，同样表示函数变化的方向，从几何意义来讲，梯度的方向表示的是函数增加最快的方向，这正是我们下山要找的“最陡峭的方向”的**反方向**！因此后面要讲到的迭代公式中，梯度前面的符号为“-”，代表梯度方向的反方向。在多元函数中，梯度向量的模（一般指二模）表示函数变化率，同样的，模数值越大，变化率越快。

## 2.3 成本函数

设计完神经网络结构之后，接下来就要设计神经网络的学习算法。假设向量为神经网络的输入向量，向量为神经网络的输出向量，向量为实际输出。我们算法的目的是使实际输出尽可能等于期望输出。为了量化算法的好坏，我们定义一个损耗函数(cost function)：

$$C(\omega, b) = \frac{1}{2n} \sum_{i=1}^n \|Y_i - A_i\|^2 \dots\dots\dots(0)$$

上式中， $\omega, b$  分别为网络中权值和偏置的集合，即  $\omega = \omega_1, \omega_2, \dots$ ，而  $b = b_1, b_2, \dots$  是输入训练样本的总数。 $Y_i$  表示在第  $i$  个样本输入下的期望输出， $A_i$  表示在第  $i$  个样本下的实际输出。 $\|\bullet\|$  为范数运算。

## 三、梯度下降算法原理

根据计算梯度时所用的数据量不同，分为三种基本方法，它们分别是批量梯度下降法 (Batch Gradient Descent, BGD)、小批量梯度下降法 (Mini-batch Gradient Descent, MBGD) 以及随机梯度下降法 (Stochastic Gradient Descent, SGD)。接下来，我们将分别介绍这三种算法。

### 3.1 批量梯度下降法 (BGD)

我们把成本函数 想象成一个山谷表面，其梯度的负方向就是下降速度最快的方向，假设我们有一个小球，那么小球只要沿着这个方向就可以最快到达谷底，即最快取得最小值(有可能停在局部极小值处，我们要避免这种情况)。梯度的定义为：

$$\nabla C = (\frac{\partial C}{\partial \omega_1}, \frac{\partial C}{\partial \omega_2}, \dots, \frac{\partial C}{\partial b_1}, \frac{\partial C}{\partial b_2}, \dots) \dots\dots\dots(1)$$

假设球在  $\omega_i$  方向上移动的距离为  $\Delta \omega_i$ 。由全微分可得，求得在曲面上移动的距离为

$$\Delta C = (\frac{\partial C}{\partial \omega_1} \Delta \omega_1, \frac{\partial C}{\partial \omega_2} \Delta \omega_2, \dots, \frac{\partial C}{\partial b_1} \Delta b_1 + \dots) \dots\dots\dots(2)$$

接下来要找一种方式，保证  $\Delta C$  是负的，只有这样，球才是往下走的。由公式（1）和（2）我们可以得出：

$$\Delta C = \nabla C \cdot \Delta V \dots\dots\dots (3)$$

其中  $\Delta V = (\Delta \omega_1, \dots, \Delta b_1, \dots)^T$ ，从公式（3）我们可以看出如何选择才能保证为负数，假设我们的选择如下：

$$\Delta V = -\eta (\nabla C)^T \dots\dots\dots (4)$$

其中  $\eta$  是学习率，此时公式（3）变为：

$$\Delta C = -\eta \nabla C \cdot (\nabla C)^T = -\eta \|\nabla C\|^2 \dots\dots\dots (5)$$

因为  $\|\nabla C\|^2 \geq 0$ ，所以  $\Delta C \leq 0$ 。

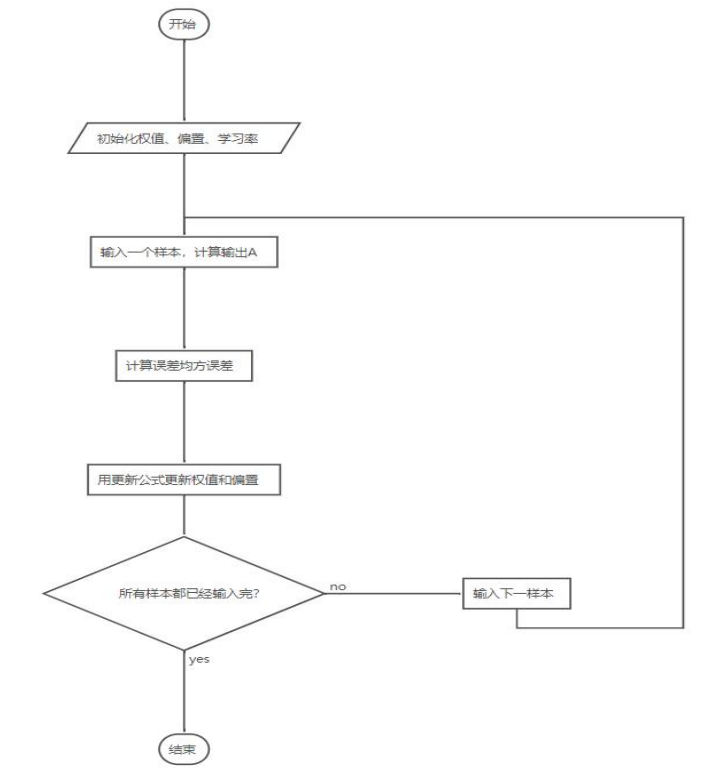
由公式（4）我们可以得到权值和偏置的更新公式  $V' = V - \eta (\nabla C)^T$ ，具体来说就是：

$$\omega'_k = \omega_k - \eta \frac{\partial C}{\partial \omega_k} \dots\dots\dots (6)$$

$$b'_k = b_k - \eta \frac{\partial C}{\partial b_k} \dots\dots\dots (6)$$

上述算法称为**批量梯度下降法**（Batch Gradient Descent, BGD），由公式（0）（4）可以看出，为了计算出需要计算所有样本的误差。另言之，为了得到一次更新值需要把所有样本都输入计算一遍。

其流程图为：



### 3.2 随机梯度下降法（SGD）

当样本很多时，梯度下降算法每更新一次都要输入所有样本，因此非常耗时。因此就有了**随机梯度下降算法（SGD）**，它的基本思想是：随机选择一个样本作为输入然后更新一次权值和偏置。

此时，公式（0）变为

$$C(\omega, b) = \frac{\|Y_i - A_i\|^2}{2} \dots \dots \dots (8)$$

更新公式为：

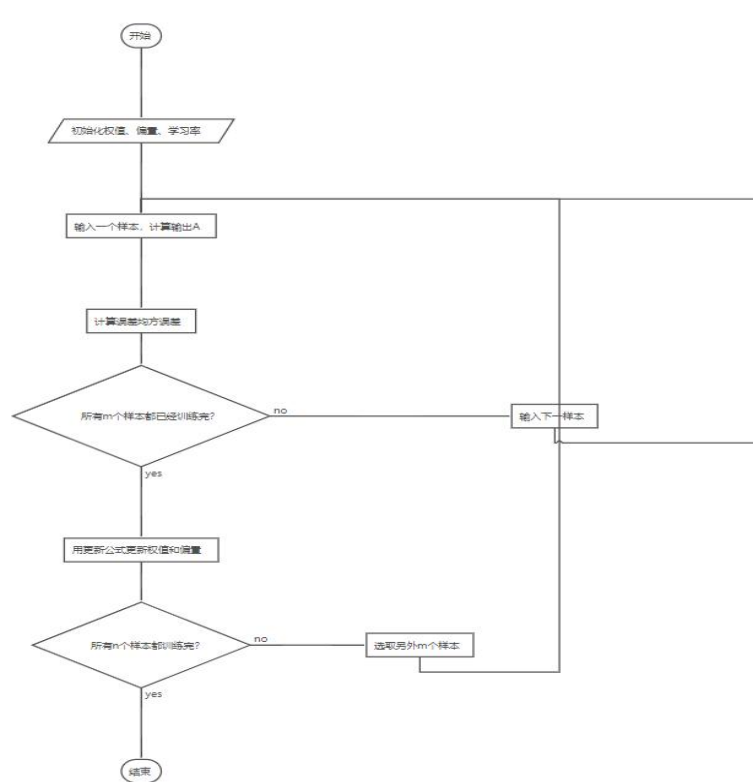
$$\omega'_k = \omega_k - \eta \frac{\partial C}{\partial \omega_k} \dots \dots \dots (9)$$

$$b'_k = b_k - \eta \frac{\partial C}{\partial b_k} \dots \dots \dots (10)$$

当样本很多时，梯度下降算法每更新一次都要输入所有样本，因此非常耗时。因此就有了**随机梯度下降算法（SGD）**，

实际上，随机梯度下降算法计算的并不是真正的梯度，真正的梯度应该像批量梯度算法那样需要所有的样本才能计算。随机梯度只是用一个样本进行对真正梯度的一次预估，可能存在偏差、即随机梯度下降算法那并不能保证每次前进的方向都是正确的，所以会带来波动。

其流程图为：



### 3.3 小批量梯度下降法（MGD）

随机梯度下降算法虽然更新速度变快，但每一次更新的方向不一定正确，所以有可能“绕远路”。因此又提出了小批量梯度算法（MGD）。其核心思想是：每次选择若干样本估计梯度，比如每学习 100 个样本更新一次梯度和权值。

首先，从所有样本中随机挑选出  $m$  个样本。我们把这  $m$  个样本编号为  $X_1, X_2, \dots, X_i, \dots, X_m$ ，并称他们为小批量（mini-batch）。此时，均方误差变为：

$$C(w, b) = \frac{1}{2m} \sum_{i=1}^m \|Y_i - A_i\|^2 = \frac{1}{m} \sum_{i=1}^m C_{X_i} \dots\dots\dots (11)$$

其中，由梯度  $C_{X_i} = \frac{\|Y_i - A_i\|^2}{2}$  可得下式：

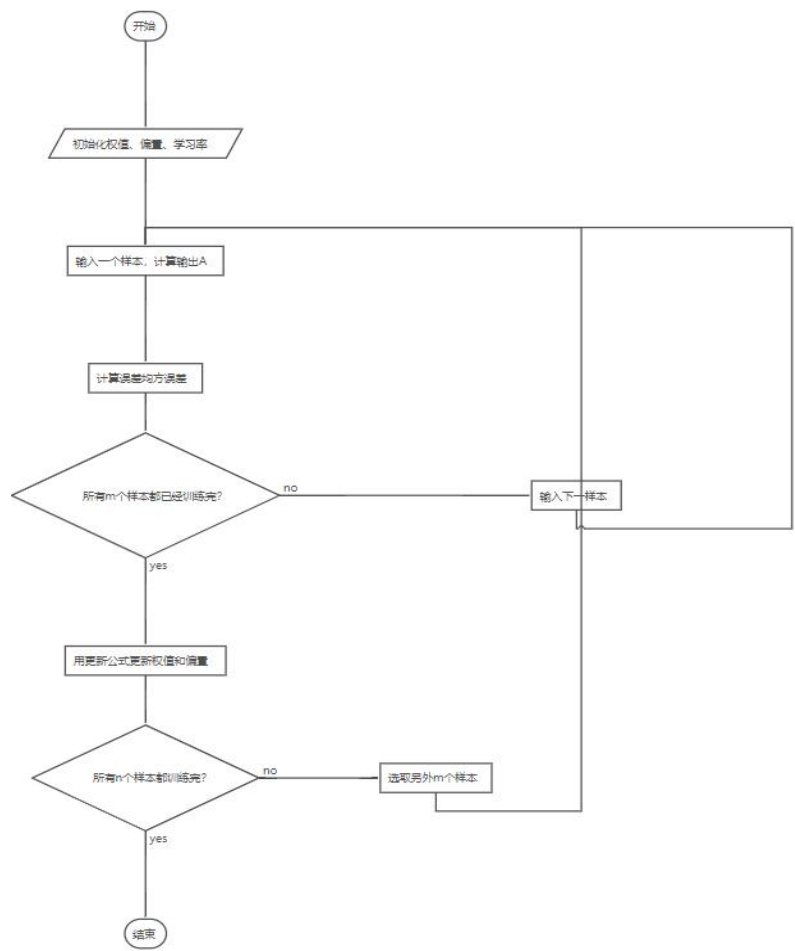
$$\nabla C = \frac{1}{m} \sum_{i=1}^m \nabla C_{X_i} \dots\dots\dots (12)$$

因此，公式（12）就用  $m$  个样本数据估计出整体的梯度，当  $m$  越大估计越准确，此时更新公式为：

$$\omega'_k = \omega_k - \eta \frac{\partial C}{\partial \omega_k} = \omega_k - \frac{\eta}{m} \sum_{i=1}^m \frac{\partial C_{X_i}}{\partial \omega_k} \dots\dots\dots (10)$$

$$b'_k = b_k - \eta \frac{\partial C}{\partial b_k} = b_k - \frac{\eta}{m} \sum_{i=1}^m \frac{\partial C_{X_i}}{\partial b_k} \dots\dots\dots (11)$$

其流程图为：



3.4 三种方法优缺点对比

	BGD	SGD	MGD
优点	非凸函数可保证 收敛至全局最优 解	计算速度快	计算速度快，收敛 稳定
缺点	计算速度缓慢，不 允许新样本中途 进入	计算结果不易收 敛，可能会陷入局 部最优解中	——



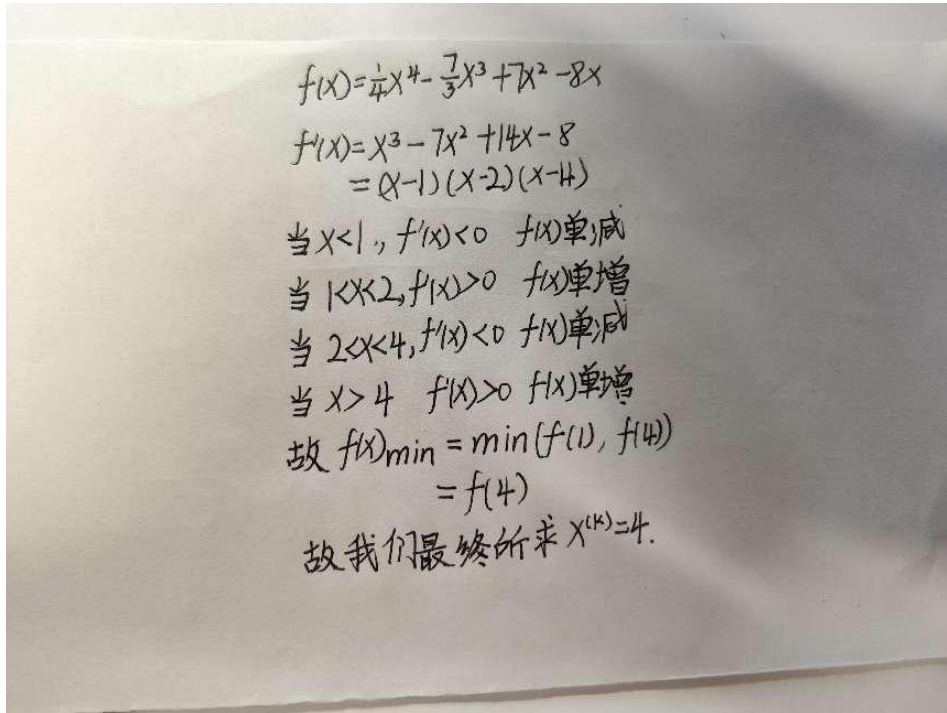
#### 四、梯度下降法的优缺点的研究

在这里，我们以求一个二次函数

$$f(x) = \frac{1}{4}x^4 - \frac{7}{3}x^3 + 7x^2 - 8x$$

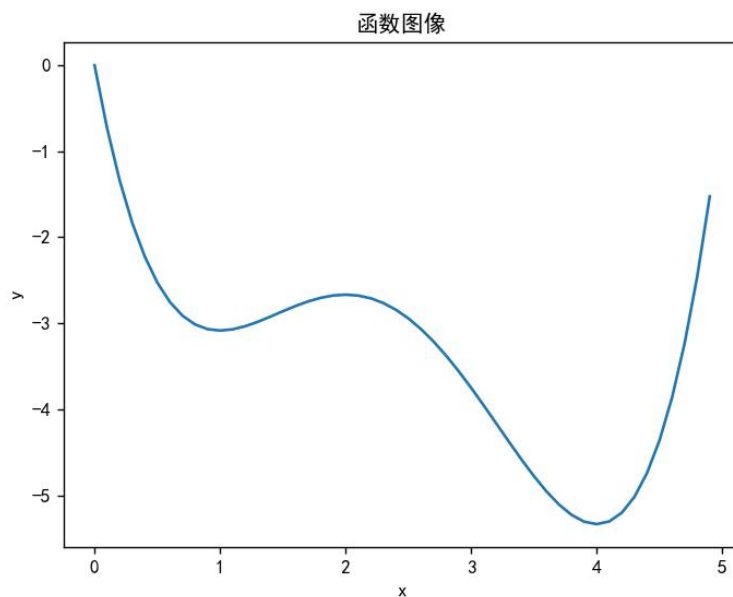
的最小值为例来展开研究。

为什么选这个函数呢？如下：


$$\begin{aligned} f(x) &= \frac{1}{4}x^4 - \frac{7}{3}x^3 + 7x^2 - 8x \\ f'(x) &= x^3 - 7x^2 + 14x - 8 \\ &= (x-1)(x-2)(x-4) \end{aligned}$$

当  $x < 1$ ,  $f'(x) < 0$   $f(x)$  单减  
当  $1 < x < 2$ ,  $f'(x) > 0$   $f(x)$  单增  
当  $2 < x < 4$ ,  $f'(x) < 0$   $f(x)$  单减  
当  $x > 4$ ,  $f'(x) > 0$   $f(x)$  单增  
故  $f(x)_{\min} = \min(f(1), f(4))$   
 $= f(4)$   
故我们最终所求  $x^{(k)} = 4$ .

下面是它的函数图像：



即我们的优化模型为：

$$\min_{x \in R} f(x)$$

## 4.1 梯度下降法

### 4.1.1 相关概念

梯度下降法又叫最速下降法。它由于只考虑当前下降最快而不是全局下降最快，在求解非线性无约束问题时，最重要的是得到每一步迭代的方向  $d^{(k)}$  和每一步下降的长度  $\lambda^{(k)}$ 。考虑到函数  $f(x)$  在点  $x^{(k)}$  处沿着方向  $d$  的方向导数，其意义是  $f(x)$  在点  $x^{(k)}$  处沿  $d$  的变化率。当  $f$  连续可微时，方向导数为负，说明函数值沿着该方向下降；方向导数越小（负值），表明下降的越快，因此确定搜索方向  $d^{(k)}$  的一个想法就是以  $f(x)$  在点  $x^{(k)}$  方向导数最小  $d^{(k)}$  的方向作为搜索方向。

(1) 搜索方向  $d^{(k)}$  的确定

设方向  $d$  为单位向量， $\|d\| = 1$ ，从点  $x^{(k)}$  按方向  $d$ ，步长  $\lambda$  进行搜索得到下一点  $x^{(k+1)} = x^{(k)} + \lambda_k d^{(k)}$ ，对该式进行泰勒展开得到：

$$f(x^{(k)} + \lambda_k d^{(k)}) = f(x^{(k)}) + \lambda_k \nabla f(x^{(k)})^T d^{(k)} + o(\lambda)$$

可以得到  $x^{(k)}$  处的变化率：

$$\lim_{t \rightarrow 0} \frac{f(x^{(k)} + \lambda_k d^{(k)}) - f(x^{(k)})}{\lambda_k} = \lim_{t \rightarrow 0} \frac{\lambda_k \nabla f(x^{(k)})^T d^{(k)} + o(\lambda)}{\lambda_k} = \nabla f(x^{(k)})^T d^{(k)}$$

容易看出来在  $x^{(k)}$  下降最快就是要在  $x^{(k)}$  出的变化率最大，所以就是要使  $\nabla f(x^{(k)})^T d^{(k)}$  最小（ $\nabla f(x^{(k)})^T d^{(k)} < 0$ ），而对于

$$\nabla f(x^{(k)})^T d^{(k)} = \|\nabla f(x^{(k)})\| \cdot \|d^{(k)}\| \cdot \cos\theta, \text{ 要使其最小就是当 } \cos\theta = -1 \text{ 时,}$$

$$d^{(k)} = -\frac{\nabla f(x^{(k)})}{\|\nabla f(x^{(k)})\|}, \text{ 即可以确定最速下降方向为 } -\nabla f(x^{(k)}), \text{ 这也是最速下降法名字的由来。}$$

(2) 步长  $\lambda^{(k)}$  的确定

最速下降法采用的搜索步长通常采取的策略是精确步长搜索法，即： $\lambda_k = \operatorname{argmin} f(x^{(k)} + \lambda_k d^{(k)})$ ，通过求该式子的最小值点来求取步长，一般有：

$\frac{df(x^{(k)} + \lambda d^{(k)})}{d\lambda} = d^{(k)} \nabla f(x^{(k)}) = 0$ ，该式表明  $d^{(k)}$  和  $d^{(k+1)}$  是正交的。在这里我没有用该方法，而是用一维搜索方法（黄金分割法 <0.618法>）来近似找到最小值点，通过自己编程实现一维搜索更好的理解这个过程，最终的结果与精确搜索几乎一致。

(3) 算法过程

求解问题:  $\min_{x \in \mathbb{R}^2} f(x)$

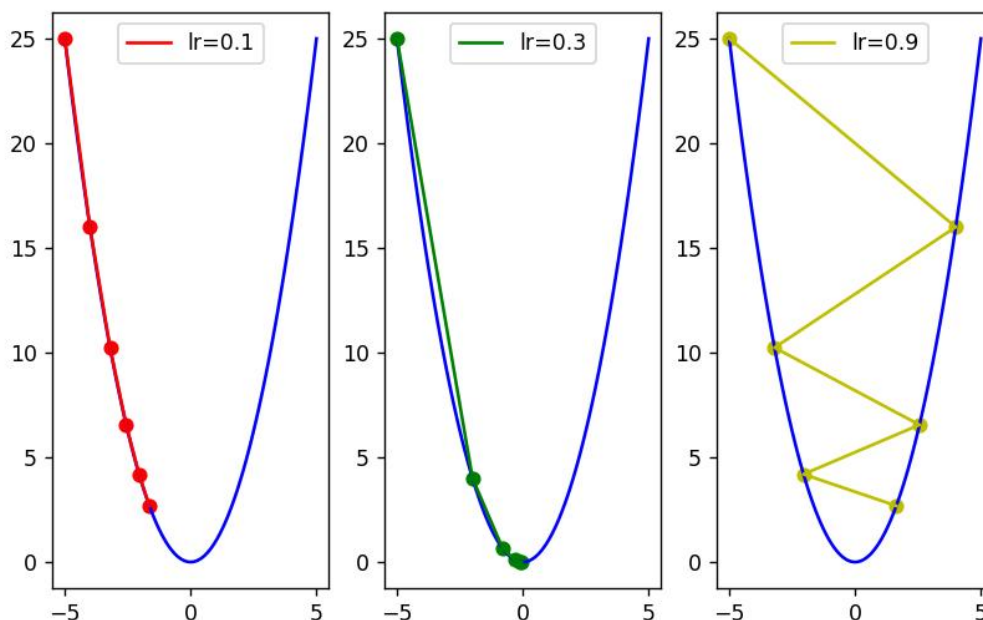
最速下降法的具体步骤为:

1. 选定初始点  $x^{(k)}$ ,  $k = 1$  给定精度要求  $\varepsilon > 0$ 。
2. 计算  $\nabla f(x^{(k)})$ , 若  $\|\nabla f(x^{(k)})\| < \varepsilon$ , 则停止, 否则令  $d^{(k)} = -\nabla f(x^{(k)})$ ;
3. 在  $x^{(k)}$  处沿方向  $d^{(k)}$  作线搜索得  $x^{(k+1)} = x^{(k)} + \lambda_k d^{(k)}$ ,  $k = k + 1$ , 返回2.

#### 4.1.2 最速下降法与学习率的关系 (代码: 附录 1)

这里我们用一个简单的二次函数  $y=x^2$  来查看。

最终的运行结果如下:



从下图输出结果可以看出两点, 在迭代周期不变的情况下:

学习率较小时, 收敛到正确结果的速度较慢。

学习率较大时, 容易在搜索过程中发生震荡。

这是最速的下降的一个缺点:

#### 4.1.3 最速下降法存在的问题的探讨 (代码: 附录 2)

这里, 我们以求  $f(x) = \frac{1}{4}x^4 - \frac{7}{3}x^3 + 7x^2 - 8x$  的最小值为例来探讨, 我们选取

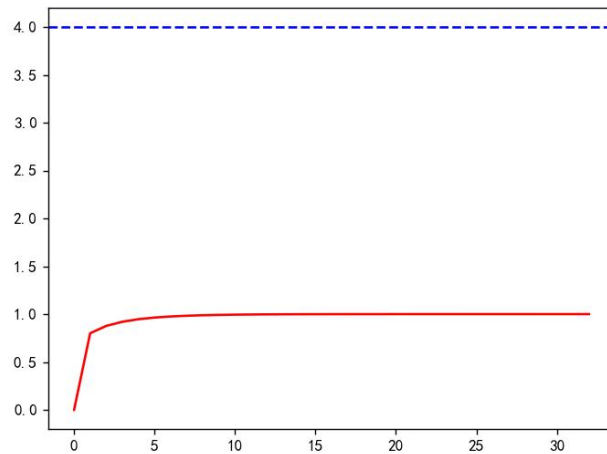
的迭代周期为 100, 误差 err 为 0.00001。并不断改变学习率 lr 和初始值 x 来研究。具体如下:

①  $x=0$ ,  $lr=0.1$

最终迭代结果如下:

最终迭代结果为1.000  
与最终值得绝对误差为3.000

迭代中间过程如下：

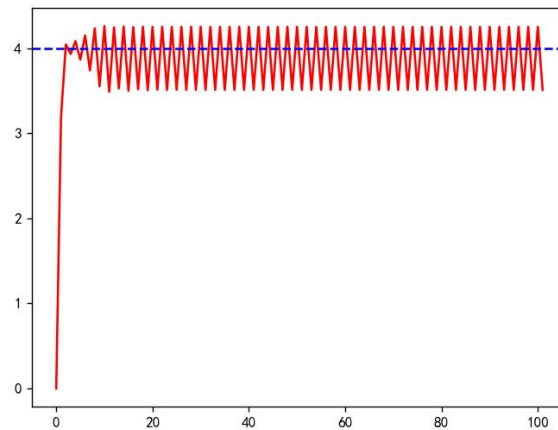


②  $x=0$ ,  $lr=0.4$

最终迭代结果如下：

```
import numpy as np
迭代次数达到，但仍然没有找到
最终迭代结果为3.511
与最终值得绝对误差为0.489
□
```

迭代中间过程如下：

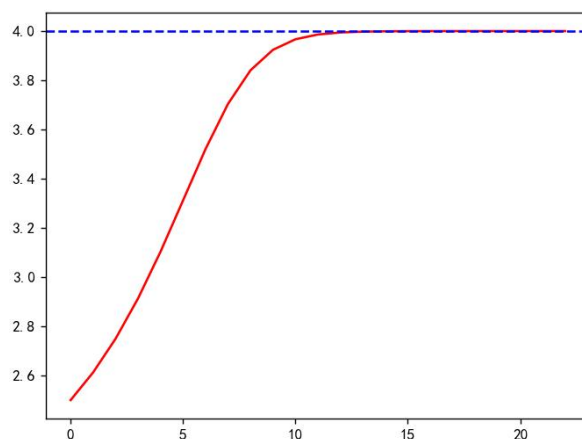


③  $x=2.5$ ,  $lr=0.1$

最终迭代结果如下：

```
import numpy as np
最终迭代结果为4.000
与最终值得绝对误差为0.000
□
```

迭代中间过程如下：

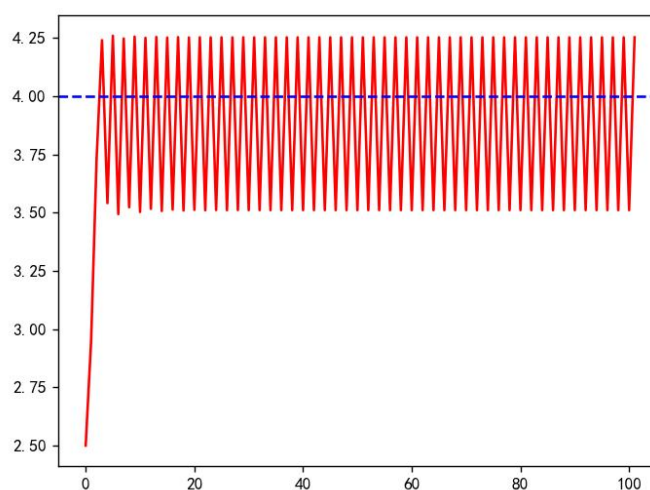


④  $x=2.5$ ,  $lr=0.4$

最终迭代结果如下：

```
import Axes3D.py
迭代次数达到，但仍然没有找到
最终迭代结果为4.253
与最终值得绝对误差为0.253
□
```

迭代中间过程如下：



经由上述四个实验，我总结了最速下降法可以存在的如下几个问题：

1. 最速下降法无法跳出局部最优，而找到全局最优解。这个我们可以通过上述实验 1 和实验 3 的迭代曲线图可以得出。

2. 最速下降法可以会产生“之”字型的下降，会发生震荡（当前下降方向与前次下降方向垂直）。显然，第二和第四个实验就出现了这样的问题，以至于迭代了 100 次仍然没有找到最优解。

#### 4.1.4 最速下降法优缺点

最速下降法	
优点	1. 时间复杂度低，在每一个迭代中，只需要计算梯度 2. 空间复杂度低，因为梯度向量为一个 $n \times 1$ 的向量
缺点	1. 学习率较小时，收敛到正确结果的速度较慢。学习率较大时，容易在搜索过程中发生震荡。 2. 最速下降法无法跳出局部最优，而找到全局最优解。 3. 最速下降法可以会产生“之”字型的下降，会发生震荡（当前下降方向与前次下降方向垂直）。 4. 靠近极小值时收敛速度减慢。

## 五、梯度下降算法改进研究

在第四阶段，我们通过两个函数研究了梯度下降法，探讨了它的优缺点。然而，梯度下降算法存在的这些问题我们有没有办法改进呢？事实上是可以的。但梯度下降算法有很多改进算法，比如一阶方法主要有：**Momentum 冲量法**、**Nesterov accelerated gradient (NAG)**、**Adagrad (自适应梯度算法)**、**RMSprop 算法**，二阶方法有：**牛顿法**、**拟牛顿法**等。因为有些改进方法仅仅是某些情况下才有较好的效果，而有的方法较为复杂，增加了计算量。因而它们大部分应用并不广泛。

接下来我们就其中几种较为简单而应用广泛的方法来进行研究。

### 5.1 Momentum 冲量法

#### (1) 原理

Momentum 冲量法是一种非常简单的改进方法，并已经成功应用了数十年。其核心思想是：在梯度方向一致的地方加速，在梯度方向不断改变的地方减速。其更新公式为：

$$\begin{aligned}\Delta V_t &= \rho \Delta V_{t-1} - \eta [\nabla C(X_{t-1})] \\ X_t &= X_{t-1} + \Delta V_t\end{aligned}$$

其中  $\rho$  是一个取值  $0 \sim 1$  的常数，称为动量因子，常取 0.9。它的大小决定这动量项作用的强弱，当  $\rho = 0$  时没有影响，当  $\rho = 1$  时影响最强，平滑效果明显。这种方法对于像一些又长有窄的山谷一样的复杂的成本函数面时更为有效。尽管沿着山谷方向的梯度要比横跨山谷方向的梯度要小的多，但因为沿着山



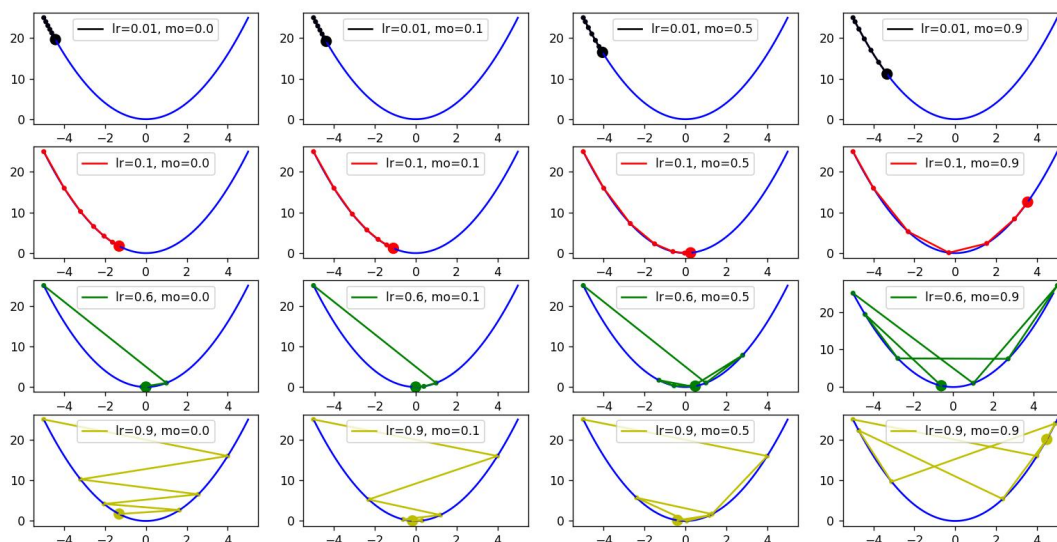
谷方向的梯度方向是一致的，因此动量项能加快其学习速度。而横跨山谷的方向梯度的方向不断变化，因此动量项能减小其更新量，减慢学习速度，这有效减小了在横跨山谷方向上的来回震荡。

## (2) 探究冲量大小和学习率对梯度下降法的影响。（附录 3 代码）

这里我们依旧选取  $y=x^2$  来探究学习率

在这里我们设置学习率为  $lr=[0.01, 0.1, 0.6, 0.9]$ ，冲量依次为  $momentum=[0.0, 0.1, 0.5, 0.9]$ 。

运行结果如下：



在上述运行结果中，我们可以看到每一行的图的学习率相同，每一列的冲量 momentum 相同，最左列为不使用 momentum 的收敛情况。

简单分析结果，我们可以得到如下结论：

1. 从第一行可看出：在学习率较小的时候，适当的 momentum 能够起到一个加速收敛速度的作用。
2. 从第四行可看出：在学习率较大的时候，适当的 momentum 能够起到一个减小收敛时震荡幅度的作用。

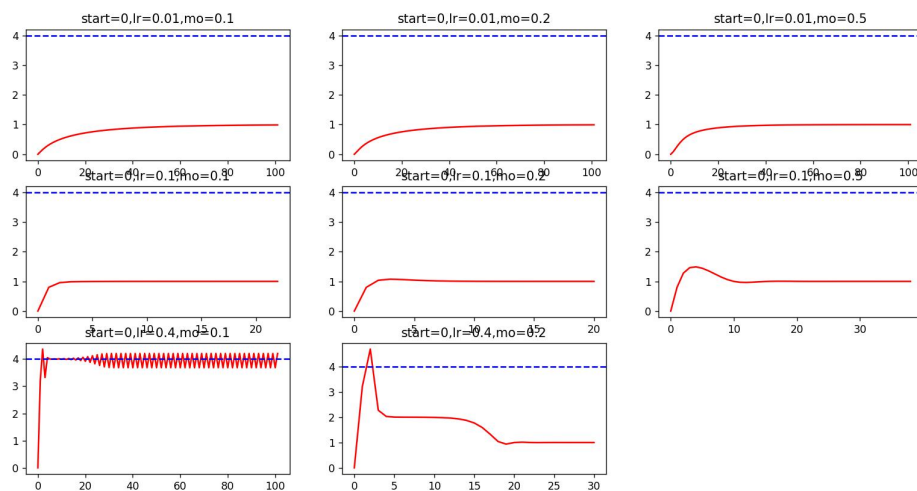
因此，我们得出了 momentum 能够较好地解决最速下降法学习率较小时，收敛到正确结果的速度较慢。学习率较大时，容易在搜索过程中发生震荡的问题。

## (3) 冲量法优点和缺点探究（附录 4 代码）

这里，我们以求  $f(x) = \frac{1}{4}x^4 - \frac{7}{3}x^3 + 7x^2 - 8x$  的最小值为例来探讨，我们选取

的迭代周期为 100，误差 err 为 0.00001。并不断改变学习率 lr、初始值 start 和冲量 momentum 来研究。

运行结果具体如下：

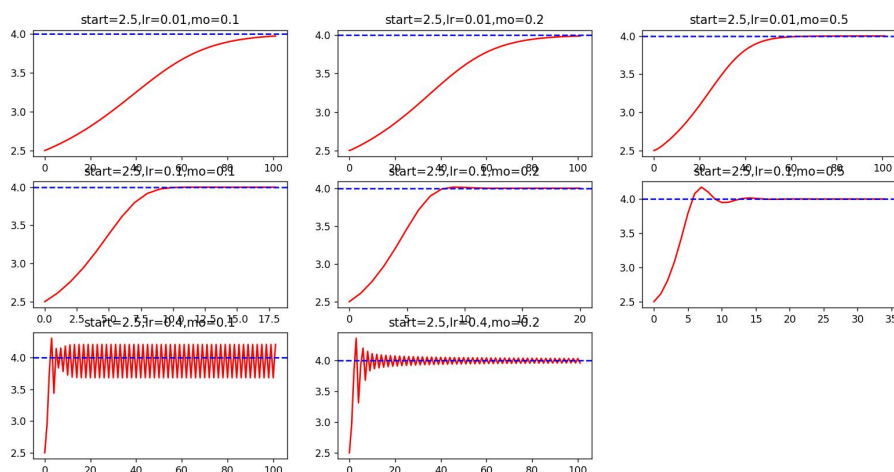


start=0

从左往右、从上到下对应结果如下：

迭代次数达到，但仍然没有找到  
最终迭代结果为0.987  
与最终值得绝对误差为3.013  
迭代次数达到，但仍然没有找到  
最终迭代结果为0.992  
与最终值得绝对误差为3.008  
迭代次数达到，但仍然没有找到  
最终迭代结果为1.000  
与最终值得绝对误差为3.000  
最终迭代结果为1.000  
与最终值得绝对误差为3.000  
最终迭代结果为1.000  
与最终值得绝对误差为3.000  
最终迭代结果为1.000  
与最终值得绝对误差为3.000  
迭代次数达到，但仍然没有找到  
最终迭代结果为4.204  
与最终值得绝对误差为0.204  
最终迭代结果为1.000  
与最终值得绝对误差为3.000





start=2.5

从左往右、从上到下对应结果如下：

迭代次数达到，但仍然没有找到  
最终迭代结果为3.974  
与最终值得绝对误差为0.026  
迭代次数达到，但仍然没有找到  
最终迭代结果为3.989  
与最终值得绝对误差为0.011  
迭代次数达到，但仍然没有找到  
最终迭代结果为4.000  
与最终值得绝对误差为0.000  
最终迭代结果为4.000  
与最终值得绝对误差为0.000  
最终迭代结果为4.000  
与最终值得绝对误差为0.000  
最终迭代结果为4.000  
与最终值得绝对误差为0.000  
迭代次数达到，但仍然没有找到  
最终迭代结果为4.204  
与最终值得绝对误差为0.204  
迭代次数达到，但仍然没有找到  
最终迭代结果为3.963  
与最终值得绝对误差为0.037

分析上述各个试验，我们可以得出如下推断：

1. 在 start=0 这一张图中，我们知道相比于最速下降法，在选取恰当的冲量和学习率，冲量法是有一定的概率跳出局部最优，如第三行第一张图。
2. 冲量法的迭代速率明显快于最速下降法。
3. 在 start=0 这一张图中，在学习率较小的时候，适当的 momentum 能够起到一个加速收敛速度的作用。在学习率较大的时候，适当的 momentum 能够起到一个减小收敛时震荡幅度的作用。
4. 但是同样也存在着这样的问题，当 momentum 较大时，原本能够正确收敛

的时候却因为刹不住车跑过头了。如  $\text{start}=2.5$  中的最后两幅图。

(4) 冲量法优缺点总结以及相对于最速下降法的提升

优点	<ul style="list-style-type: none"><li>1. 收敛速度快</li><li>2. 在学习率较小的时候，适当的 momentum 能够起到一个加速收敛速度的作用。在学习率较大的时候，适当的 momentum 能够起到一个减小收敛时震荡幅度的作用。</li><li>3. 适当的学习率和冲量的选取可能有一定几率跳出局部最优</li></ul>
缺点	<ul style="list-style-type: none"><li>1. 当 momentum 较大时，原本能够正确收敛的时候却因为刹不住车跑过头了</li><li>2. 计算量大增，尤其是在迭代次数非常大的时候。</li></ul>
相对于最速下降法的提升	<ul style="list-style-type: none"><li>1. 收敛速度快</li><li>2. 解决了最速下降法学习率较小时，收敛到正确结果的速度较慢。学习率较大时，容易在搜索过程中发生震荡的问题。</li><li>3. 解决了最速下降法靠近极小值时收敛速度减慢。</li></ul>

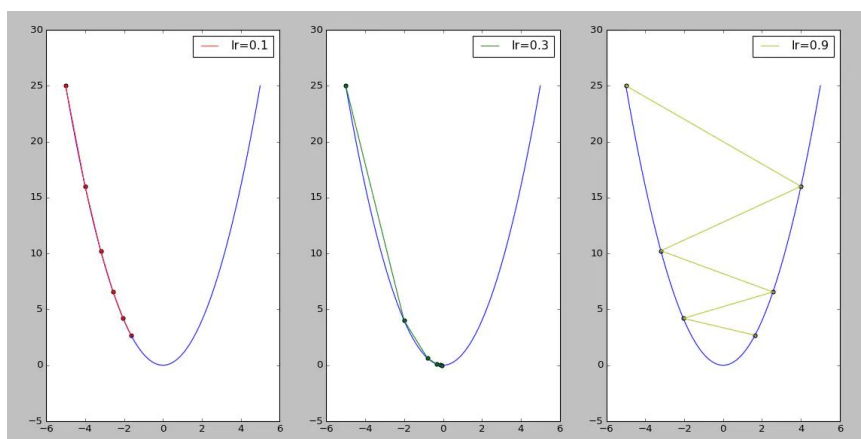
总之，相对于最速下降法，冲量法有了一个非常大的提升，但是我们不得不正视它存在的一个较大的问题，如何选取冲量？如何选取学习率。故接下来，我将引入学习率衰减率因子这一个奇妙的创造！

5.2 学习率衰减率因子（decay）

(1) 原理

我们知道，在梯度下降法里如何调整搜索的步长（也叫学习率，Learning Rate）、如何加快收敛速度以及如何防止搜索时发生震荡却是一门值得深究的学问。

首先我们回顾一下上面梯度下降法中探究的不同学习率下梯度下降法的收敛过程。



从上图可看出，学习率较大时，容易在搜索过程中发生震荡，而发生震荡的根本原因无非就是搜索的步长迈的太大了。

我们回顾一下问题本身，在使用梯度下降法求解目标函数  $f(x) = x^2$  的极小值时，更新公式为  $x += v$ ，其中每次  $x$  的更新量  $v$  为  $v = -dx * lr$ ， $dx$  为目标函数  $f(x)$  对  $x$  的一阶导数。可以想到，如果能够让  $lr$  随着迭代周期不断衰减变小，那么搜索时迈的步长就能不断减少以减缓震荡。学习率衰减因子由此诞生：

$$lr\_i = lr\_start * 1.0 / (1.0 + decay * i)$$

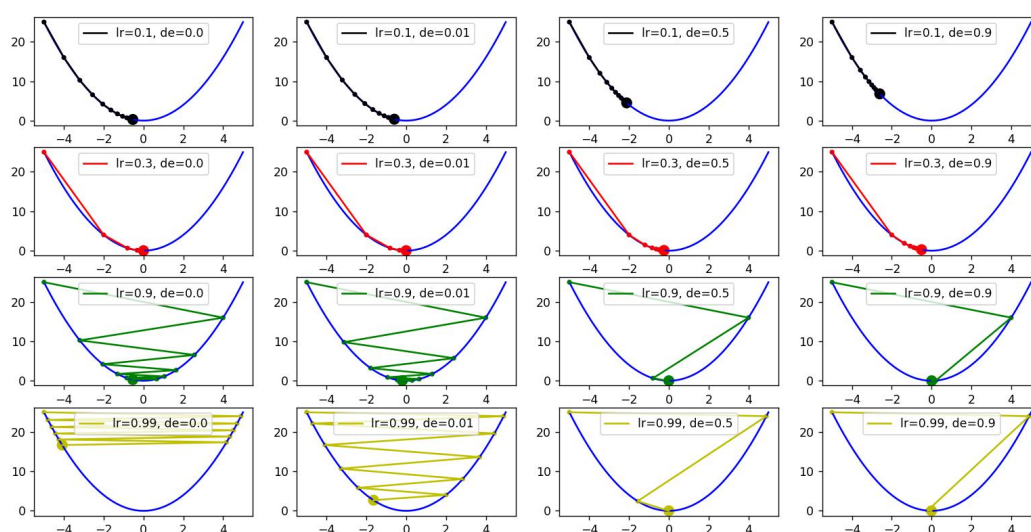
上面的公式即为学习率衰减公式，其中  $lr\_i$  为第  $i$  次迭代时的学习率， $lr\_start$  为原始学习率， $decay$  为一个介于  $[0.0, 1.0]$  的小数。

从上述公式，我们可以看出：

$decay$  越小，学习率衰减地越慢，当  $decay = 0$  时，学习率保持不变。

$decay$  越大，学习率衰减地越快，当  $decay = 1$  时，学习率衰减最快。

## (2) 测试 (附录 5)



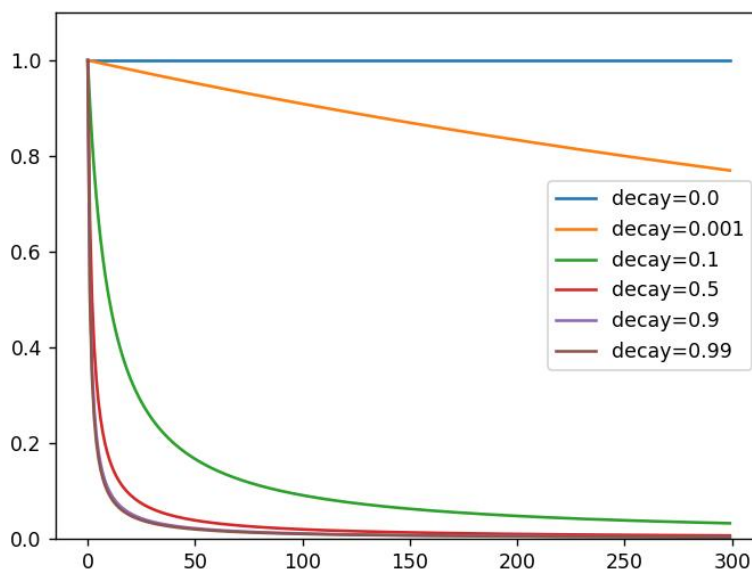
如上，我们会发现如下两点：

1. 在所有行中均可以看出， $decay$  越大，学习率衰减地越快。

2. 在第三行与第四行可看到，decay 确实能够对震荡起到减缓的作用。

### (3) 探究不同学习率衰减率因子对学习率的影响（附录 7）

在这里我们设置初始学习率为 1，研究在迭代 300 次后学习率的变化。



如上，可以推测出如果 decay 设置得太大，则可能会收敛到一个不是极值的地方。

故我们可以知道在梯度下降法中，参数的调整是一个非常重要的工作。一个适当的参数组合能够让我们的求解更加理想！

### 5.3 涅斯捷罗夫梯度加速法 (NAG) (了解部分)

我们知道，动量法更新其实包括两部分：一部分是上一时刻的更新值，这一部分是上一次就已经算出来了，是已知的；另一部分是基于当前位置计算出来的梯度。涅斯捷罗夫梯度加速法 (NAG, Nesterov accelerated gradient) 则指出：既然已经知道本次更新一定会走的量，我们就可以先走上，然后再根据那里的梯度前进一下，岂不美哉？于是就有了以下更新公式：

原始形式中，Nesterov Accelerated Gradient (NAG) 算法相对于 Momentum 的改进在于，以“向前看”看到的梯度而不是当前位置梯度去更新。经过变换之后的等效形式中，NAG 算法相对于 Momentum 多了一个本次梯度相对上次梯度的变化量，这个变化量本质上是对目标函数二阶导的近似。由于利用了二阶导的信息，NAG 算法才会比 Momentum 具有更快的收敛速度。同时 Nesterov Accelerated Gradient (NAG) 相比于 Momentum 能够更好地降低震荡。

## 六、总结与归纳思考

实际上，梯度下降法是一个用于局部最优值的方法。虽然我们可以通过一定

的方法来增加其突破局部最优解的可能性，但是这已经在一定程度上脱离了梯度下降法的本质。

那么我们该如何用梯度下降法来求解全局最优解呢？我的看法是不断对参数的调整和取舍，我们可以通过梯度下降法来不断求取不同的极小值来进行不断地取舍，最后终会达到或无限逼近于我们的全局最优解。当然，这建立在我们基数的尽可能多的层次上。

同时，在实验中，我发现，学习率、冲量、学习率衰减因子这三个参数对梯度下降法收敛速度和震荡程度的影响非常大。这启发，亦是在告诉我们，梯度下降法最难的部分是参数的调整，如何准确的选取恰当的参数，将会是我们在梯度下降法这个领域内无限追求的未来！而我相信，这一切的实现就会在现在，亦或是不久的将来！

## 七、附录

### （一）

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3.
4. # 目标函数:y=x^2
5. def func(x):
6.     return np.square(x)
7.
8.
9. # 目标函数一阶导数:dy/dx=2*x
10. def dfunc(x):
11.     return 2 * x
12.
13. # Gradient Descent
14. def GD(x_start, df, epochs, lr):
15.     """
16.     梯度下降法。给定起始点与目标函数的一阶导函数，求在 epochs 次迭代中 x 的更新值
17.     :param x_start: x 的起始点
18.     :param df: 目标函数的一阶导函数
19.     :param epochs: 迭代周期
20.     :param lr: 学习率
21.     :return: x 在每次迭代后的位置（包括起始点），长度为 epochs+1
22.     """
23.     xs = np.zeros(epochs+1)
24.     x = x_start
25.     xs[0] = x
26.     for i in range(epochs):
27.         dx = df(x)
28.         # v 表示 x 要改变的幅度
29.         v = - dx * lr
```

```

30.         x += v
31.         xs[i+1] = x
32.     return xs
33.
34. def demo1_GD_lr():
35.     # 函数图像
36.     line_x = np.linspace(-5, 5, 100)
37.     line_y = func(line_x)
38.     plt.figure('Gradient Descent: Learning Rate')
39.
40.     x_start = -5
41.     epochs = 5
42.
43.     lr = [0.1, 0.3, 0.9]
44.
45.     color = ['r', 'g', 'y']
46.     size = np.ones(epochs+1) * 10
47.     size[-1] = 70
48.     for i in range(len(lr)):
49.         x = GD(x_start, dfunc, epochs, lr=lr[i])
50.         plt.subplot(1, 3, i+1)
51.         plt.plot(line_x, line_y, c='b')
52.         plt.plot(x, func(x), c=color[i], label='lr={}'.format(lr[i]))
53.         plt.scatter(x, func(x), c=color[i])
54.         plt.legend()
55.     plt.show()
56. demo1_GD_lr()

```

## (二)

```

1.     #导入相关依赖包
2.     import math
3.     import numpy as np
4.     from matplotlib import pyplot as plt
5.     plt.rcParams['font.sans-serif'] = ['SimHei'] #用来正常显示中文标签
6.     plt.rcParams['axes.unicode_minus'] = False #用来正常显示负号
7.
8.
9.     #函数定义
10.    def f(x):
11.        return 1/4*pow(x,4)-7/3*pow(x,3)+7*pow(x,2)-8*x
12.    #函数求导
13.    def f1(x):
14.        return pow(x,3)-7*pow(x,2)+14*x-8
15.

```

```

16. #函数作图
17. def drawing():
18.     x = np.arange(0, 5, 0.01)
19.     plt.xlabel('x')
20.     plt.ylabel('y')
21.     plt.title("函数图像")
22.     plt.plot(x, f(x))
23. #最速下降法
24. #x0 表示初始化
25. #err 表示误差
26. #learn 表示学习率
27. #用一个list 把每次找的x 和对应的序号记录
28. xList=[]
29. countList=[]
30. def MaxSpeed(x0,err,learn):
31.     count=0
32.     xList.append(x0)
33.     countList.append(count)
34.     while(count<=100):
35.         x1=x0-learn*f1(x0)
36.         xList.append(x1)
37.         countList.append(count+1)
38.         if(abs(f1(x0))<=err):
39.             print("最终迭代结果为%.3f"%(x1))
40.             print("与最终值得绝对误差为%.3f"%(abs(x1-4)))
41.             return x1
42.         else:
43.             x0=x1
44.             count=count+1
45.     print("迭代次数达到，但仍然没有找到")
46.     print("最终迭代结果为%.3f"%(x1))
47.     print("与最终值得绝对误差为%.3f"%(abs(x1-4)))
48. #设置不同参数即可
49. MaxSpeed(2.5,0.00001,0.4)
50. plt.plot(countList,xList,color='r')
51. plt.axhline(y=4, ls='--', c='blue') # 添加水平线
52. x = np.arange(0, 5, 0.01)
53. plt.show()

```

### (三)

```

1. import numpy as np
2. import matplotlib.pyplot as plt
3.
4.

```

```

5.     # 目标函数:  $y=x^2$ 
6.     def func(x):
7.         return np.square(x)
8.
9.
10.    # 目标函数一阶导数:  $dy/dx=2*x$ 
11.    def dfunc(x):
12.        return 2 * x
13.
14.    def M(x_start, df, epochs, lr, momentum):
15.        """
16.        带有冲量的梯度下降法。
17.        :param x_start: x 的起始点
18.        :param df: 目标函数的一阶导函数
19.        :param epochs: 迭代周期
20.        :param lr: 学习率
21.        :param momentum: 冲量
22.        :return: x 在每次迭代后的位置（包括起始点），长度为 epochs+1
23.        """
24.        xs = np.zeros(epochs+1)
25.        x = x_start
26.        xs[0] = x
27.        v = 0
28.        for i in range(epochs):
29.            dx = df(x)
30.            # v 表示 x 要改变的幅度
31.            # 冲量法原理
32.            v = - df(x-v) * lr + momentum * v
33.            x += v
34.            xs[i+1] = x
35.        return xs
36.
37.    def demo2_GD_momentum():
38.        line_x = np.linspace(-5, 5, 100)
39.        line_y = func(line_x)
40.        plt.figure('Gradient Descent: Learning Rate, Momentum')
41.
42.        x_start = -5
43.        epochs = 6
44.
45.        lr = [0.01, 0.1, 0.6, 0.9]
46.        momentum = [0.0, 0.1, 0.5, 0.9]
47.
48.        color = ['k', 'r', 'g', 'y']

```



```

49.
50.     row = len(lr)
51.     col = len(momentum)
52.     size = np.ones(epochs+1) * 10
53.     size[-1] = 70
54.     for i in range(row):
55.         for j in range(col):
56.             x = M(x_start, dfunc, epochs, lr=lr[i], momentum=momentum[j])
57.             plt.subplot(row, col, i * col + j + 1)
58.             plt.plot(line_x, line_y, c='b')
59.             plt.plot(x, func(x), c=color[i], label='lr={}, mo={}'.format(lr[i], momentum[j]))
60.             plt.scatter(x, func(x), c=color[i], s=size)
61.             plt.legend(loc=0)
62.     plt.show()
63. demo2_GD_momentum()

```

#### (四)

```

1.     import numpy as np
2.     import matplotlib.pyplot as plt
3.
4.
5.     #函数定义
6.     def f(x):
7.         return 1/4*pow(x,4)-7/3*pow(x,3)+7*pow(x,2)-8*x
8.     #函数求导
9.     def f1(x):
10.        return pow(x,3)-7*pow(x,2)+14*x-8
11.    #冲量法实现
12.    xList=[]
13.    countList=[]
14.    def GD_momentum(x0,err,learn,momentum):
15.        count=0
16.        xList.append(x0)
17.        countList.append(count)
18.        v=0
19.        while(count<=100):
20.            #v 表示要改变的幅度
21.            v=v*momentum-f1(x0)*learn
22.            x0=x0+v
23.            xList.append(x0)
24.            countList.append(count+1)
25.            if(abs(f1(x0))<=err):
26.                print("最终迭代结果为%.3f"%(x0))
27.                print("与最终值得绝对误差为%.3f"%(abs(x0-4)))

```

```

28.         return x0
29.         count=count+1
30.     print("迭代次数达到，但仍然没有找到")
31.     print("最终迭代结果为%.3f"%(x0))
32.     print("与最终值得绝对误差为%.3f"%(abs(x0-4)))
33. def draw(start,lr,momentum):
34.     GD_momentum(start,0.00001,lr,momentum)
35.     plt.plot(countList,xList,color='r')
36.     plt.title('start={},lr={},mo={}'.format(start,lr,momentum))
37.     plt.axhline(y=4, ls='--', c='blue') # 添加水平线
38. def intMain(start):
39.     lrs=[0.01,0.1,0.4]
40.     momentums=[0.1,0.2,0.5]
41.     row = len(lrs)
42.     col = len(momentums)
43.     for i in range(row):
44.         for j in range(col):
45.             if(i==row-1 and j==col-1):
46.                 break
47.             plt.subplot(row, col, i * col + j + 1)
48.             draw(start,lrs[i],momentums[j])
49.
50.             xList.clear()
51.             countList.clear()
52.     plt.show()
53. intMain(2.5)

```

## (五)

```

1. import numpy as np
2. import matplotlib.pyplot as plt
3. # 目标函数:y=x^2
4. def func(x):
5.     return np.square(x)
6. # 目标函数一阶导数:dy/dx=2*x
7. def dfunc(x):
8.     return 2 * x
9. def GD_decay(x_start, df, epochs, lr, decay):
10.     """
11.     带有学习率衰减因子的梯度下降法。
12.     :param x_start: x 的起始点
13.     :param df: 目标函数的一阶导函数
14.     :param epochs: 迭代周期
15.     :param lr: 学习率
16.     :param decay: 学习率衰减因子

```

```

17.         :return: x 在每次迭代后的位置（包括起始点），长度为 epochs+1
18.         """
19.         xs = np.zeros(epochs+1)
20.         x = x_start
21.         xs[0] = x
22.         v = 0
23.         for i in range(epochs):
24.             dx = df(x)
25.             # 学习率衰减
26.             lr_i = lr * 1.0 / (1.0 + decay * i)
27.             # v 表示 x 要改变的幅度
28.             v = - dx * lr_i
29.             x += v
30.             xs[i+1] = x
31.         return xs
32.     def decay():
33.         line_x = np.linspace(-5, 5, 100)
34.         line_y = func(line_x)
35.         plt.figure('Gradient Descent: Decay')
36.
37.         x_start = -5
38.         epochs = 10
39.
40.         lr = [0.1, 0.3, 0.9, 0.99]
41.         decay = [0.0, 0.01, 0.5, 0.9]
42.
43.         color = ['k', 'r', 'g', 'y']
44.
45.         row = len(lr)
46.         col = len(decay)
47.         size = np.ones(epochs + 1) * 10
48.         size[-1] = 70
49.         for i in range(row):
50.             for j in range(col):
51.                 x = GD_decay(x_start, dfunc, epochs, lr=lr[i], decay=decay[j])
52.                 plt.subplot(row, col, i * col + j + 1)
53.                 plt.plot(line_x, line_y, c='b')
54.                 plt.plot(x, func(x), c=color[i], label='lr={}, de={}'.format(lr[i], decay[j]))
55.                 plt.scatter(x, func(x), c=color[i], s=size)
56.                 plt.legend(loc=0)
57.         plt.show()
58.     decay()

```

(六)

```

1. import numpy as np
2. import matplotlib.pyplot as plt
3. def test():
4.     lr = 1.0
5.     iterations = np.arange(300)
6.
7.     decay = [0.0, 0.001, 0.1, 0.5, 0.9, 0.99]
8.     for i in range(len(decay)):
9.         decay_lr = lr * (1.0 / (1.0 + decay[i] * iterations))
10.        plt.plot(iterations, decay_lr, label='decay={}'.format(decay[i]))
11.
12.        plt.ylim([0, 1.1])
13.        plt.legend(loc='best')
14.        plt.show()
15. test()

```

## (七)

```

1. import numpy as np
2. import matplotlib.pyplot as plt
3.
4.
5. # 目标函数:  $y=x^2$ 
6. def func(x):
7.     return np.square(x)
8.
9.
10. # 目标函数一阶导数:  $dy/dx=2*x$ 
11. def dfunc(x):
12.     return 2 * x
13.
14. def NAG(x_start, df, epochs, lr, momentum):
15.     """
16.     带有冲量的梯度下降法。
17.     :param x_start: x 的起始点
18.     :param df: 目标函数的一阶导函数
19.     :param epochs: 迭代周期
20.     :param lr: 学习率
21.     :param momentum: 冲量
22.     :return: x 在每次迭代后的位置（包括起始点），长度为 epochs+1
23.     """
24.     xs = np.zeros(epochs+1)
25.     x = x_start
26.     xs[0] = x
27.     v = 0

```

```

28.     for i in range(epochs):
29.         dx = df(x)
30.         # v 表示 x 要改变的幅度
31.         # 冲量法原理
32.         v = - df(x-momentum*v) * lr + momentum * v
33.         x += v
34.         xs[i+1] = x
35.     return xs
36.
37. def NAG_show():
38.     line_x = np.linspace(-5, 5, 100)
39.     line_y = func(line_x)
40.     plt.figure('Gradient Descent: Learning Rate, Momentum')
41.
42.     x_start = -5
43.     epochs = 6
44.
45.     lr = [0.01, 0.1, 0.6, 0.9]
46.     momentum = [0.0, 0.1, 0.5, 0.9]
47.
48.     color = ['k', 'r', 'g', 'y']
49.
50.     row = len(lr)
51.     col = len(momentum)
52.     size = np.ones(epochs+1) * 10
53.     size[-1] = 70
54.     for i in range(row):
55.         for j in range(col):
56.             x = NAG(x_start, dfunc, epochs, lr=lr[i], momentum=momentum[j])
57.             plt.subplot(row, col, i * col + j + 1)
58.             plt.plot(line_x, line_y, c='b')
59.             plt.plot(x, func(x), c=color[i], label='lr={}, mo={}'.format(lr[i], momentum[j]))
60.             plt.scatter(x, func(x), c=color[i], s=size)
61.             plt.legend(loc=0)
62.     plt.show()
63.     NAG_show()

```

## 八、参考文献

1. 简书 机器学习板块: [机器学习 - 文集 - 简书 \(jianshu.com\)](https://jianshu.com)
2. Csdn 梯度学习介绍(14 条消息) 梯度下降算法及其改进方法详解 LVLV 苗的博客-CSDN 博客 [梯度下降法改进](#)

