1、最优化概念

1.1、最大值最小值

最优化问题就是求 $f(x)$ 的极大值或者极小值

$$f(x) \quad x \in R$$

我们往往求极小值，在这里 x 是优化变量，就是自变量，$f(x)$ 称为目标函数，可能还带有约束条件，有些优化问题既有等式约束，又有不等式约束

$$\max f(x) \rightarrow \min(-f(x))$$

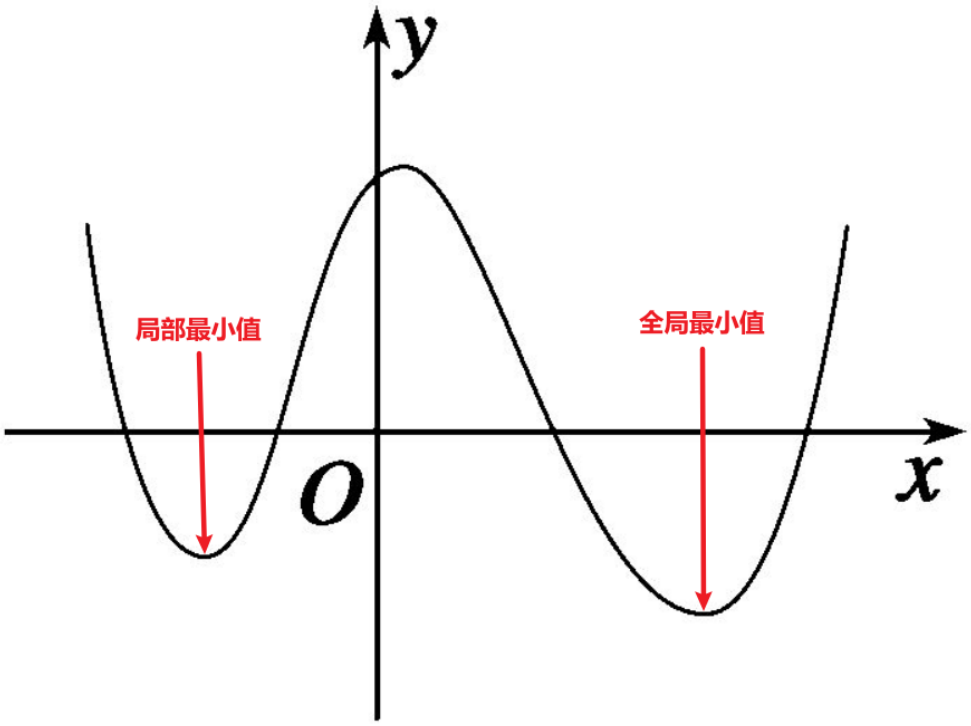
$$g_i(x) = 0, \quad i = 1, 2, \cdots, n$$

$$h_i(x) \leq 0, \quad i = 1, 2, \cdots, n$$

满足这种约束条件，并且还在 $f(x)$ 定义域之内的那些 x 构成的集合叫可行域 D 。

1.2、局部极小值

函数 $f(x)$ 在 x_0 的邻域 $[x_0 - \Delta x, x_0 + \Delta x]$ 内存在，并且 $f(x) \geq f(x_0)$，那么 $f(x_0)$ 为局部最小值点。



1.3、全局最小值

其实初中我们就学求极值的问题了，比如二次函数抛物线的顶点就是极大值或者极小值，高中我们也会有求极值的题目，算一个非常复杂的函数的极值，借用初等数学一些的技巧来完成的。而真正的飞跃是发生在大学里面，微积分这门课为求解极值提供了一个统一的方法，就是找它导数等于 0 的点，如果是多元函数的话，我们说它的梯度等于 0，我们去解方程组就可以了，在机器学习里面我们遇到的几乎所有的函数都是可导的。光滑即可导

$$f'(x) = 0$$

局部最小值， 通过大量实践发现在高维度的优化问题中，局部最小值和全局最小值通常没有太大的区别，甚至在有些情况下局部最小值有更好的归纳能力（泛化能力）。

2、求导与迭代求解

2.1、求导遇到的问题

前面咱们说了求极值就是导数或梯度等于 0，找到疑似的极值，就是驻点

$$\left.\begin{array}{l} f'(x)=0 \\ \nabla f(x)=0 \end{array}\right\} \text{驻点}$$

有了微积分里面这样的手段，只要函数可导，我们求解不就行了？

因为很多时候去求方程等于 0 或方程组等于 0 的根并不是那么容易，比如：

$$f(x,y)=x^3-5x^2+e^{xy}-3y^4+12y^2+1024sin(xy)$$

分别对 x,y 求偏导数：

$$\begin{cases} \dfrac{\partial f(x,y)}{\partial x}=3x^2-10x+ye^{xy}+1024ycos(xy) \\ \dfrac{\partial f(x,y)}{\partial y}=xe^{xy}-12y^3+24y+1024xcos(xy) \end{cases}$$

令上面的导数为 0 依然无法求解。对于 5 次或以上的代数方程它都没有求根公式了，所以这个思路看上去很美，但是实际上是行不通的，我们得想其它的办法。

2.2、近似迭代求解

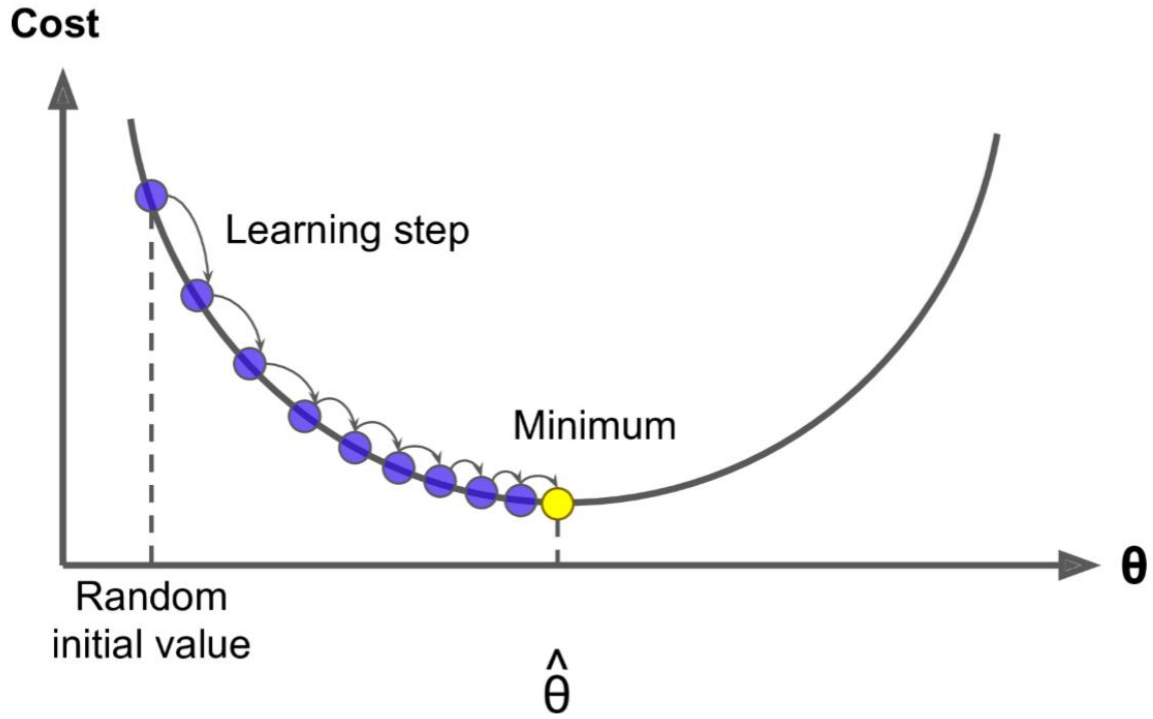
近似求解，我们先给个初始的值 x_0，看看它是不是极值点，当然得满足等式或者不等式约束，它如果不是，我们想办法再前进一步，只要我们得 x_n 它越来越接近我们的极值点，那这个问题不就解决了嘛！

$$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_n$$

因此，构造一个序列，满足这样一个极限，当 k 趋近于 $+\infty$ 的时候，$f(x_k)$ 这点的梯度值极限趋近于 0 的话，那我们就找到了函数的驻点

$$\lim_{k\rightarrow+\infty}\nabla f(x_k)=0$$

由此我们要解决的核心问题就变成了，怎么从当前一个点移动到下一个点上面去，也就是怎么从 x_k 到 x_{k+1}，迭代法是我们计算数学中经常采用的一种方法，它不单单可以求极值，还可以用来求方程组的解，包括线性方程，非线性方程，一个矩阵的特征值等等。



在最优化中，这种迭代思想又叫梯度下降！

3、梯度下降

3.1、公式推导

数值优化算法，是求近似解，而不是理论上面的精确解

$$x_k = x_{k-1} - \eta \nabla f(x_{k-1})$$

下面我们来推导这个公式，需要用到多元函数泰勒展开公式（函数 $f(x)$ 在 x_0 的某个邻域内），最后加上一个高阶无穷小

$$f(x) = f(x_0) + [\nabla f(x_0)]^T \times (x - x_0) + o(x - x_0)$$

如果我们忽略高阶无穷小，也就是说 x 它是属于 x_0 的 δ 邻域里面的话，那么等号就会变成约等于，那么我们把 $f(x_0)$ 放到等号左边：

$$f(x) - f(x_0) \approx [\nabla f(x_0)]^T \times (x - x_0)$$

$$A^T B = ||A|| \times ||B|| \times cos\theta$$

这时要想下降的更快，就要让等式右边得到最小值，即当 $cos\theta = -1$，下降的幅度最大！

单位向量代表的是方向，长度为 1 的向量，即模为 1 的向量称为单位向量。

向量乘以标量不会改变它的方向，只会改变它的幅值。这就是缩放向量的方法。

所以一个向量可以表示为一个标量乘以一个单位向量。

为了使得下降幅度最大，那么向量 $(x - x_0)$【这里的向量用 θ 表示】 的方向和梯度的方向相反：

$$v = -\frac{\nabla f(\theta_{n-1})}{||\nabla f(\theta_{n-1})||}$$

$$\theta_n = \theta_{n-1} - \eta \frac{\nabla f(\theta_{n-1})}{||\nabla f(\theta_{n-1})||}$$

分母是个标量，可以并入到 η 中，简化为：

$$\theta_n = \theta_{n-1} - \eta \nabla f(\theta_{n-1})$$

需要注意的是，x 和 x_0 离的充分近，也就是在它的 δ 邻域里面，才能忽略掉泰勒展开里面的一次以上的项，否则就不能忽略它了，它就不是高阶无穷小了，约等于的条件就不成立了，所以 η 步长不能够太大，由此我们就得到了梯度下降法的公式。

$$x = x_0$$

$$for(\cdots):$$

$$x_k = x_{k-1} - \eta \nabla f(x_{n-1})$$

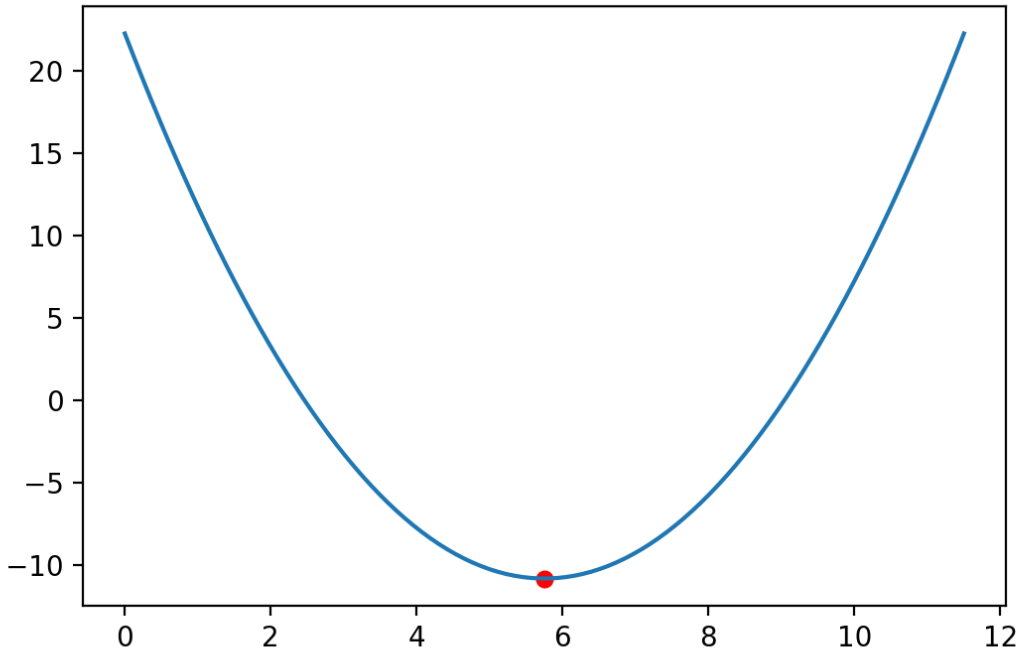
实现细节问题：

- 初始值的设定，随机或者设定满足约束条件
- 步长的设定，一个比较小的，也可以动态的调整
- 循环终止的判定，达到max_iteration（最大迭代次数）；上一次减下一次的更新，差异基本不变时

3.2、代码演示

1、创建模拟数据

```
import numpy as np
import matplotlib.pyplot as plt
# 构建方程
f = lambda x : (x - 3.5)**2 - 4.5*x + 10
# 导函数
g = lambda x :2 * (x - 3.5) - 4.5
# 创建模拟数据并可视化
x = np.linspace(0,11.5,100)
y = f(x)
plt.plot(x,y)
plt.scatter(5.75,f(5.75),color = 'red',s = 30)
```



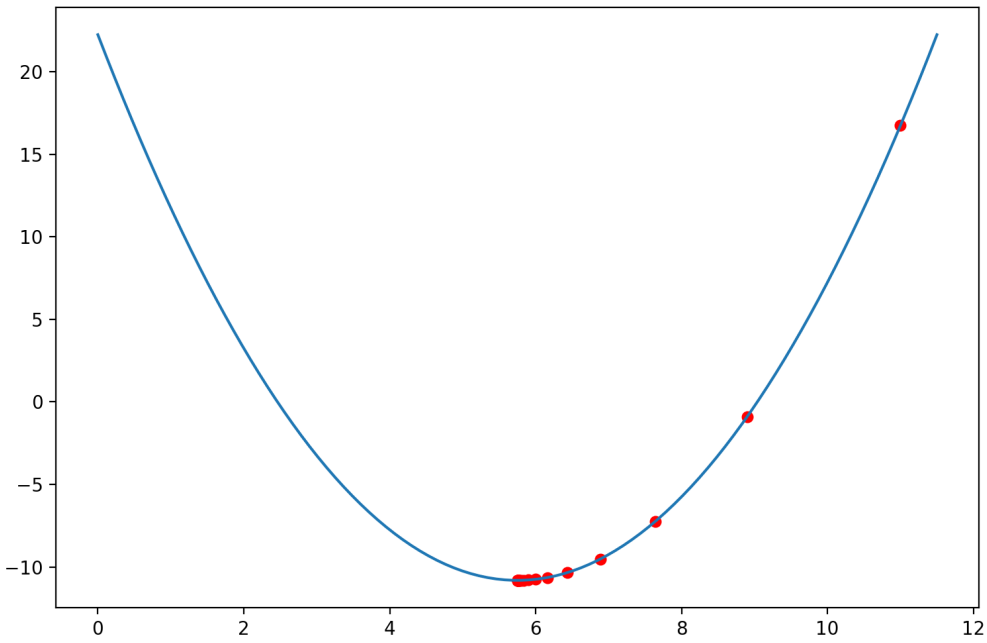
2、迭代法求解（梯度下降）

```
eta = 0.2 # 学习率
# 随机（瞎蒙）,初始值
x = np.random.randint(0,12,size = 1)[0]
# 多次while 循环，每次梯度下降，更新，记录一下上一次的值
# 比较精确度
# 一开始故意设置差异，目的为了有区分，不能一上来就停止
last_x = x + 0.1
# 精确度，人为设定
precision = 0.0001
print('-----------------随机x是：',x)
# 每次梯度下降，求解出来的x值，一开始随机给的
x_ = [x] # Python中列表
count = 0
while True:
    if np.abs(x - last_x) < precision: # 更新时，变化甚微，满足精确度，终止
        break
    # 更新，梯度下降
    # x是当前数值，赋值给上一个值
    last_x = x
    count += 1
    x = x - eta * g(x) # 梯度下降公式
    x_.append(x)
    print('+++++++++++++++更新之后的x是：%0.5f'%(x))
print('+++++++++++++++梯度下降次数：',count)

# x1是NumPy数组
x1 = np.linspace(0,11.5,100)
y1 = f(x1)
plt.figure(figsize=(9,6))
plt.plot(x1,y1)

# 散点图
x_ = np.array(x_)
plt.scatter(x_, f(x_),color = 'red',s = 30)
```

```
plt.savefig('./4-迭代法求解.png',dpi = 200)
# 执行结果
'''
-----------------随机x是： 11
+++++++++++++++更新之后的x是：8.90000
+++++++++++++++更新之后的x是：7.64000
+++++++++++++++更新之后的x是：6.88400
+++++++++++++++更新之后的x是：6.43040
+++++++++++++++更新之后的x是：6.15824
+++++++++++++++更新之后的x是：5.99494
+++++++++++++++更新之后的x是：5.89697
+++++++++++++++更新之后的x是：5.83818
+++++++++++++++更新之后的x是：5.80291
+++++++++++++++更新之后的x是：5.78174
+++++++++++++++更新之后的x是：5.76905
+++++++++++++++更新之后的x是：5.76143
+++++++++++++++更新之后的x是：5.75686
+++++++++++++++更新之后的x是：5.75411
+++++++++++++++更新之后的x是：5.75247
+++++++++++++++更新之后的x是：5.75148
+++++++++++++++更新之后的x是：5.75089
+++++++++++++++更新之后的x是：5.75053
+++++++++++++++更新之后的x是：5.75032
+++++++++++++++更新之后的x是：5.75019
+++++++++++++++更新之后的x是：5.75012
+++++++++++++++梯度下降次数： 21
'''
```



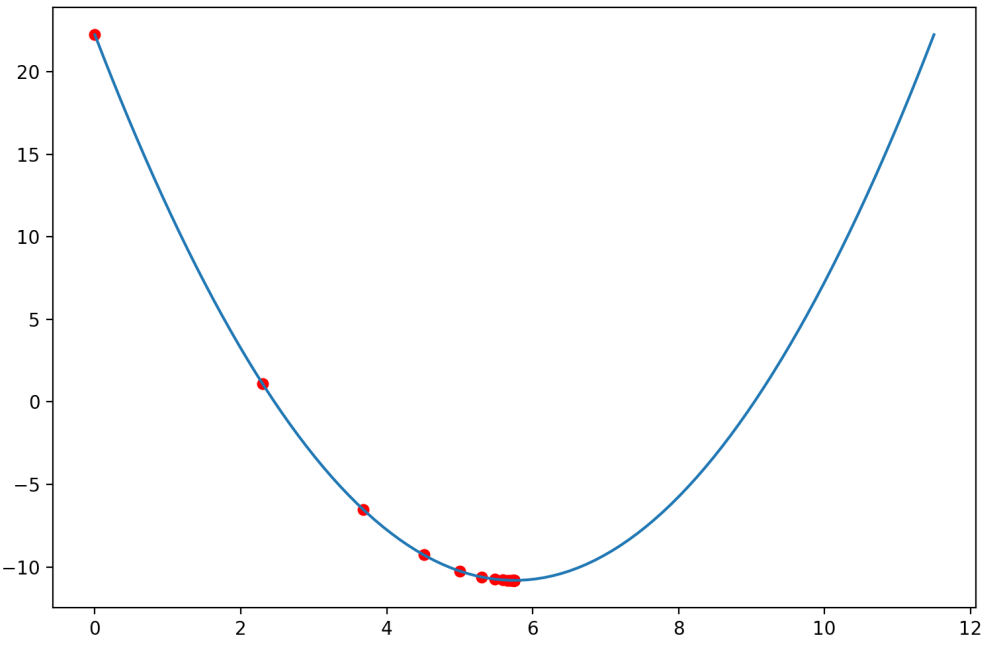
3、迭代法求解（退出条件最大迭代次数）

```
eta = 0.2 # 学习率
# 随机（瞎蒙）,初始值
x = np.random.randint(0,12,size = 1)[0]
# 多次while 循环，每次梯度下降，更新，记录一下上一次的值
# 一开始故意设置差异，目的为了有区分，不能一上来就停止
last_x = x + 0.1
```

```
# 最大迭代次数，人为设定
count = 30
print('-----------------随机x是：',x)
# 每次梯度下降，求解出来的x值，一开始随机给的
x_ = [x] # Python中列表
count = 0
for i in range(count):
    # 更新，梯度下降
    # x是当前数值，赋值给上一个值
    last_x = x
    count += 1
    x = x - eta * g(x) # 梯度下降公式
    x_.append(x)
    print('+++++++++++++++更新之后的x是：%0.5f'%(x))

# x1是NumPy数组
x1 = np.linspace(0,11.5,100)
y1 = f(x1)
plt.figure(figsize=(9,6))
plt.plot(x1,y1)

# 散点图
x_ = np.array(x_)
plt.scatter(x_, f(x_),color = 'red',s = 30)
plt.savefig('./5-迭代法求解最大迭代次数.png',dpi = 200)
```



4、牛顿法

4.1、牛顿法原理

牛顿法的原理是使用函数 $f(x)$ 的泰勒级数的前面几项来寻找方程 $f(x)=0$ 的根。

将函数 $f(x)$ 在 x_0 处展开成泰勒级数：

$$f(x)=f(x_0)+f'(x_0)(x-x_0)+\frac{1}{2}f''(x_0)(x-x_0)^2+\cdots$$

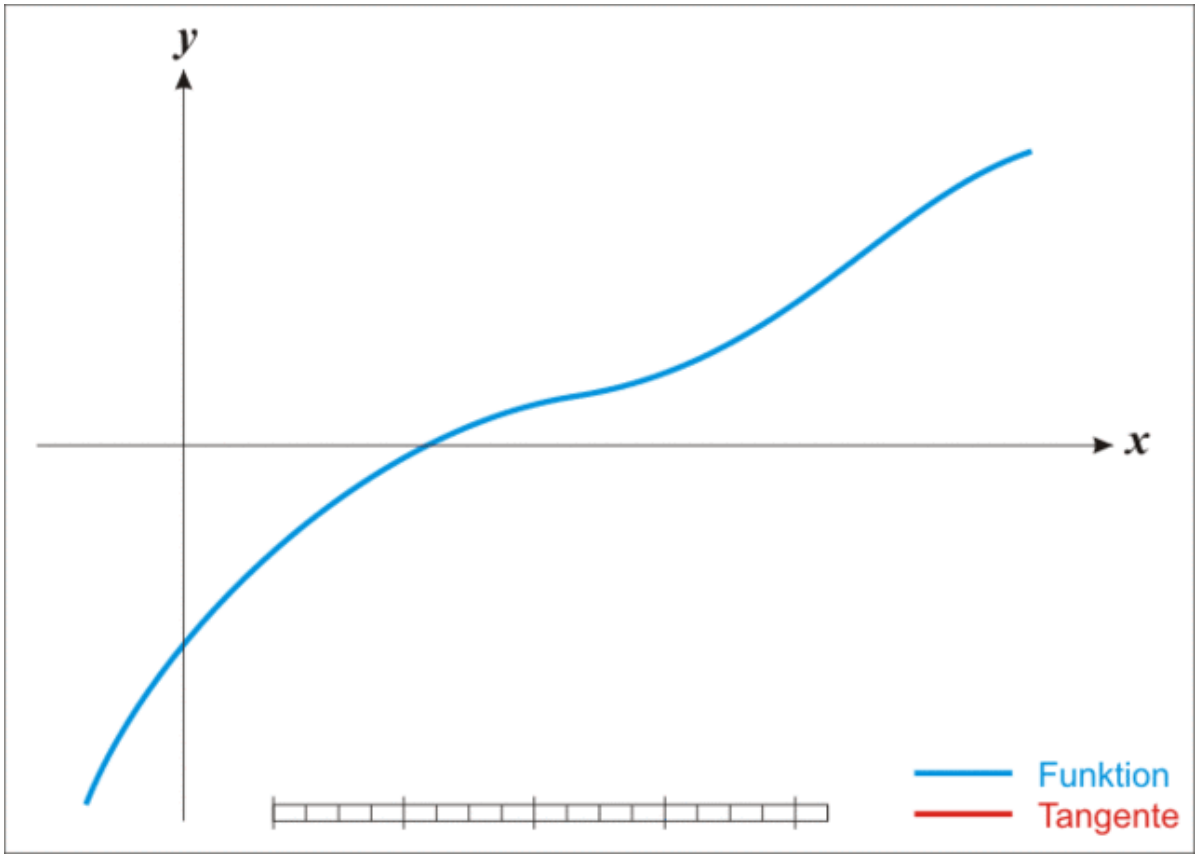
取线性部分，作为 $f(x)$ 的近似，则 $f(x_0)+f'(x_0)(x-x_0)=0$ 的解来近似 $f(x)=0$ 的解，其解为：

$$x_1=x_0-\frac{f(x_0)}{f'(x_0)}$$

由于 $f(x)$ 的近似只是一阶展开，因此 x_1 并非 $f(x)=0$ 的解，只能说 $f(x_1)$ 比 $f(x_0)$ 更接近 0 。于是，考虑迭代求解：

$$x_n=x_{n-1}-\frac{f(x_{n-1})}{f'(x_{n-1})}$$

迭代过程如下：



4.2、牛顿法代码演示

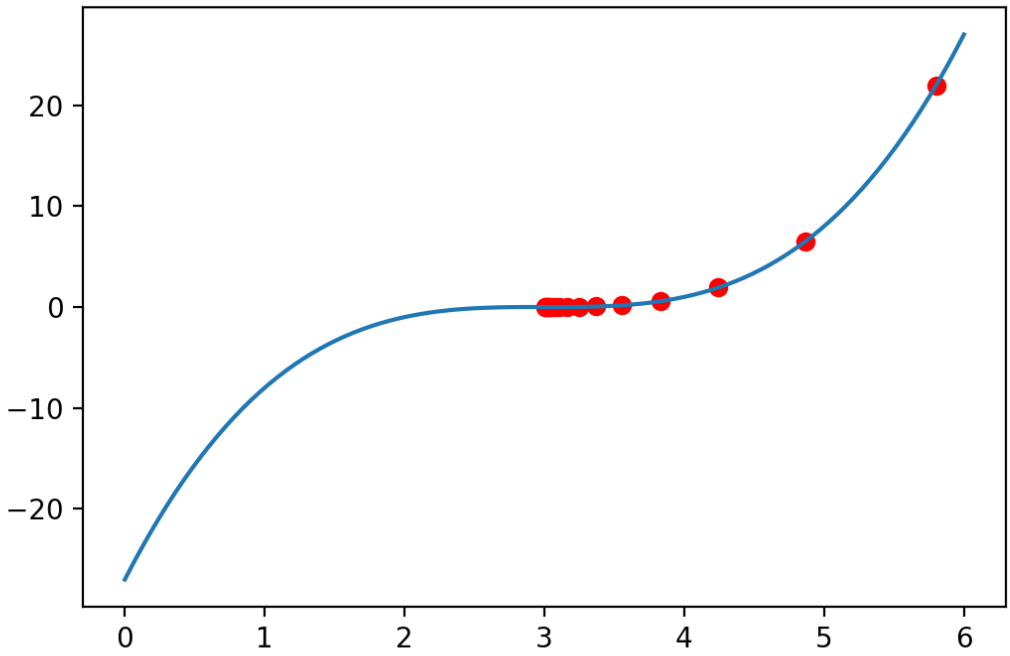
```
def f(x):
    return (x-3)**3         # 定义函数

def fd(x):
    return 3*((x-3)**2)     # 定义一阶导数

y = [5.8]
def newtonMethod(n,assum):
    count = n
    x = assum
    next_x = 0
    A = f(x)
    B = fd(x)
    print('A = %0.4f'%(A) + ',B = %0.4f'%(B) + ',time = %d'%(count))
    if f(x) == 0.0:
```

```
            return time,x
        else:
            next_x = x - A/B
            y.append(next_x)
            print('next_x = %0.4f'%(next_x))
        if abs(A - f(next_x)) < 1e-6:
            # 设置迭代跳出条件，同时输出满足f(x) = 0的x值
            print('方程的解为：  f(x) = 0,x = %0.4f'% (next_x))
        else:
            return newtonMethod(n+1,next_x)
# 设置从0开始计数，x0 = 4
newtonMethod(0,5.8)

x = np.linspace(0,6,100)
plt.plot(x,f(x))
plt.scatter(y,f(np.array(y)),color = 'red')
```



4.3、求解最优化问题

牛顿法最优化公式如下：

$$x_{k+1} = x_k - H_k^{-1} \cdot g_k$$

其中：

$$g_k = f'(x_k)$$

$$H_k = f''(x_k)$$

假设现在函数 $f(x)$ 迭代了 k 次的值为 x_k，则在 x_k 上进行二阶泰勒展开可近似得到以下公式：

$$f(x) = f(x_k) + f'(x_k) \cdot (x - x_k) + \frac{1}{2} f''(x_k) \cdot (x - x_k)^2$$

我们要求得 $f(x)$ 的极小值，则必要条件是 $f(x)$ 在极值点处的一阶导数为0，即：

$$\nabla f(x) = 0$$

因为我们把每轮迭代求得的满足目标函数极小值的 x 作为下一轮迭代的值，因此我们可以假设第 $k+1$ 轮的值就是最优解：

$$\nabla f(x_{k+1}) = 0$$

代入二阶泰勒展开并求导可得：

$$f'(x_k) + f''(x_k) \cdot (x_{k+1} - x_k) = 0$$

令：

$$g_k = f'(x_k)$$

$$H_k = f''(x_k)$$

其中H表示Hessian矩阵

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial^2 x_1}, & \frac{\partial^2 f}{\partial x_1 \partial x_2}, & \cdots, & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}, & \frac{\partial^2 f}{\partial^2 x_2}, & \cdots, & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \cdots, & \cdots, & \cdots, & \cdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}, & \frac{\partial^2 f}{\partial x_n \partial x_2}, & \cdots, & \frac{\partial^2 f}{\partial^2 x_n} \end{bmatrix}$$

可得最终的优化公式为：

$$x_{k+1} = x_k - H_k^{-1} \cdot g_k$$

复杂问题简单化，多元函数降级为一元函数，那么最优化牛顿法泰勒二阶展开的更新公式为：

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

梯度下降法只用到了一阶导数的信息，牛顿法既用到了一阶导数的信息，也用到了二阶导数的信息。梯度下降法是用线性函数来代替目标函数，牛顿法是用二次函数来代替目标函数，所以说牛顿法的收敛速度是更快的。

4.4、求解最优化代码演示

1、使用牛顿法求最优化（11步得到答案，2.0062）

```
import numpy as np
import matplotlib.pyplot as plt
# 构建方程
f = lambda x : (x - 2)**4 + 10
# 导函数
g1 = lambda x :4 * (x - 2)**3
g2 = lambda x :12 * (x - 2)**2
```

```
# 创建模拟数据并可视化
x = np.linspace(1,3,100)
y = f(x)
plt.plot(x,y)
y = [2.8]
def newtonMethod(n,assum):
    count = n
    x = assum
    next_x = 0
    A = g1(x)
    B = g2(x)
    print('A = %0.4f'%(A) + ',B = %0.4f'%(B) + ',time = %d'%(count))
    if f(x) == 0.0:
        return time,x
    else:
        next_x = x - A/B
        y.append(next_x)
        print('next_x = %0.4f'%(next_x),)
    if abs(f(x) - f(next_x)) < 1e-8:
        # 设置迭代跳出条件，同时输出满足f(x) = 0的x值
        print('方程的解为：  f(x) = 0,x = %0.4f'% (next_x))
    else:
        return newtonMethod(n+1,next_x)
# 设置从0开始计数，x0 = 4
newtonMethod(0,2.8)
plt.scatter(y,f(np.array(y)),color = 'red')
```

2、使用梯度下降求最优解（207步得到答案，2.04349）

```
import numpy as np
import matplotlib.pyplot as plt
# 构建方程
f = lambda x : (x - 2)**4 + 10
# 导函数
g1 = lambda x :4 * (x - 2)**3
eta = 0.3 # 学习率
# 随机（瞎蒙）,初始值
x = 2.8
# 多次while 循环，每次梯度下降，更新，记录一下上一次的值
# 比较精确度
# 一开始故意设置差异，目的为了有区分，不能一上来就停止
last_x = x + 0.1
# 精确度，人为设定
precision = 1e-4
print('-----------------随机x是：',x)
# 每次梯度下降，求解出来的x值，一开始随机给的
x_ = [x] # Python中列表
count = 0
while True:
    if np.abs(x - last_x) < precision: # 更新时，变化甚微，满足精确度，终止
        break
    # 更新，梯度下降
    # x是当前数值，赋值给上一个值
    last_x = x
    count += 1
    x = x - eta * g1(x) # 梯度下降公式
    x_.append(x)
    print('+++++++++++++++更新之后的x是：%0.5f'%(x))
print('+++++++++++++++梯度下降次数：',count)
```

```
# x1是NumPy数组
x1 = np.linspace(0,11.5,100)
y1 = f(x1)
plt.figure(figsize=(9,6))
plt.plot(x1,y1)

# 散点图
x_ = np.array(x_)
plt.scatter(x_, f(x_),color = 'red',s = 30)
```

4.5、拟牛顿法

如上节所说，牛顿法虽然收敛速度快，但是需要计算海塞矩阵的逆矩阵 H^{-1} ，而且有时目标函数的海塞矩阵无法保持正定（多元函数微分学），从而使得牛顿法失效。为了克服这两个问题，人们提出了拟牛顿法。这个方法的基本思想是：不用二阶偏导数而构造出可以近似海塞矩阵（或海塞矩阵的逆）的正定对称阵。不同的构造方法就产生了不同的拟牛顿法。

下面我们先推导一下拟牛顿条件，它给"对海塞矩阵（或海塞矩阵的逆）做近似"提供了理论指导，指出了用来近似的矩阵应该满足的条件。

对 $\nabla f(x)$ 做二阶泰勒展开我们得到了以下近似：

$$f(x) = f(x_k) + f'(x_k) \cdot (x - x_k) + \frac{1}{2} f''(x_k) \cdot (x - x_k)^2$$

$$\nabla f(x) = f'(x_k) + f''(x_k)(x - x_k)$$

令：

$$g_k = f'(x_k)$$

$$H_k = f''(x_k)$$

所以：

$$\nabla f(x) = g_k + H_k(x - x_k)$$

取 $x = x_{k+1}$ ，得到：

$$g_{k+1} = g_k + H_k(x - x_k)$$

$$g_{k+1} - g_k = H_k(x - x_k)$$

令 $y_k = g_{k+1} - g_k$ ， $\delta_k = x_{k+1} - x_k$ ，则：

$$y_k = H_k \delta_k$$

$$H_k^{-1} y_k = \delta_k$$

以上即为拟牛顿条件！

在拟牛顿法中，选择 G_k 作为 H_k^{-1} 的近似或选择 B_k 作为 H_k 的近似，并且使得它们满足上述拟牛顿条件即可。不同的拟牛顿法，区别就在于如何确定 G_k 或 B_k 。

常用的拟牛顿算法有：DFP、BFGS、L_BFGS算法。这里不做展开！

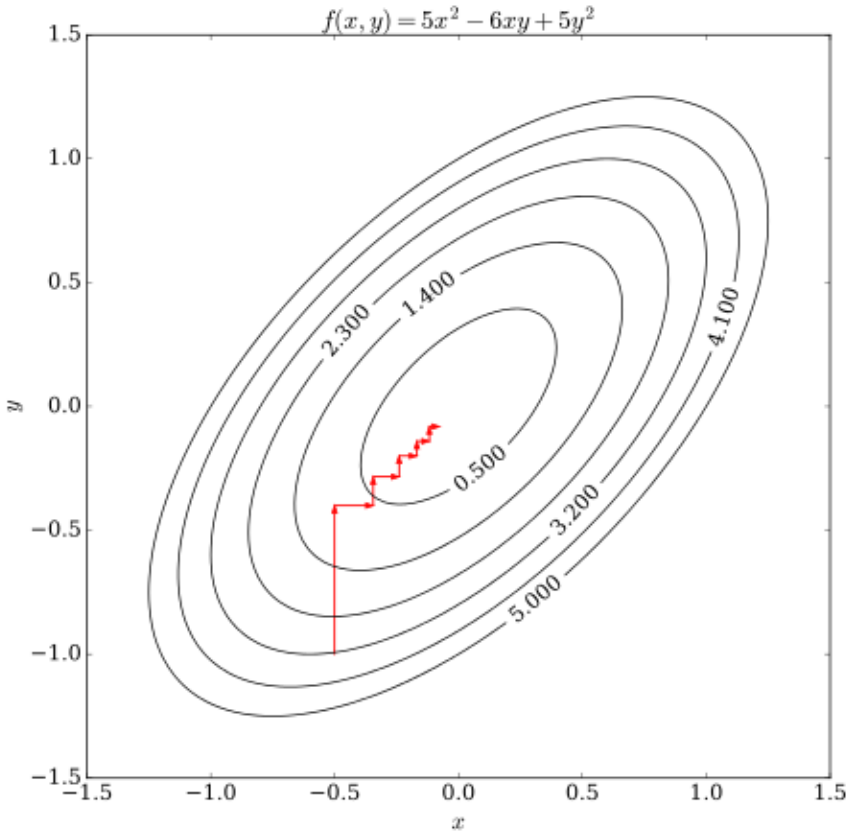
5、坐标下降法

坐标下降法就是分治法的思想。

$$min f(x),\ \ x = (x_1, x_2, \cdots, x_n)$$

它的思想是按住其它的不动，只优化其中一个比如 x_1 ，那就把多元函数求极值问题变成了一元函数求极值问题，这样优化的难度就小了很多，紧接着我们把其它的按住不动，再优化 x_2，一直到 x_n，一轮完了之后再回来优化 x_1 。至于一个变量怎么解，可以用梯度下降或者牛顿法等优化算法来求解，具体问题具体分析。

SVM 中的 SMO 算法就是把其中两个拿出来优化，其它的固定不动；liblinear 库也大量的使用坐标下降法来求解的。直观的看就好比：



我们把所有的变量 $x_1, x_2, \cdots x_n$ 都优化一遍，就好比梯度下降迭代一次，这种方法它也有很大好处，就是计算的工作量小很多。

6、最优化算法瓶颈

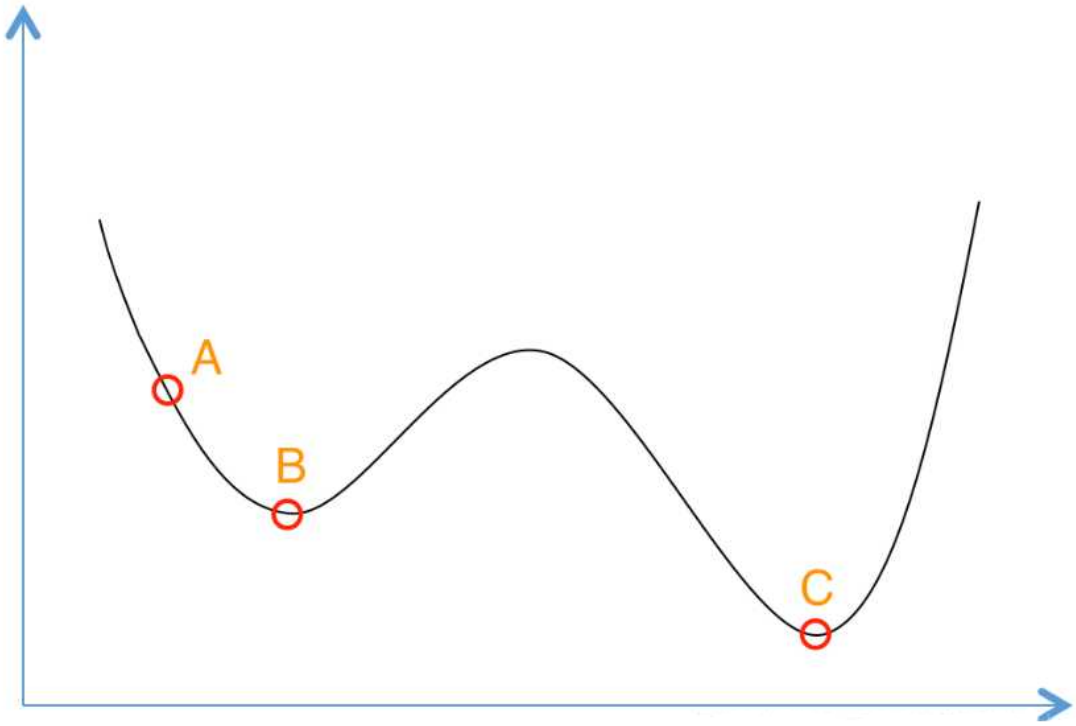
前面梯度下降法，还有牛顿法求极值的依据都是 $\nabla f(x_k) = 0$，而前面我们也说了函数在某一点的导数或梯度等于 0 就是驻点，它只是函数取得极值的必要条件，不是充分条件，也就是说无法推导出

$$\nabla f(x_k) = 0 \rightarrow \min x_k$$

所有我们前面介绍的数值优化算法，都会面临两个问题。

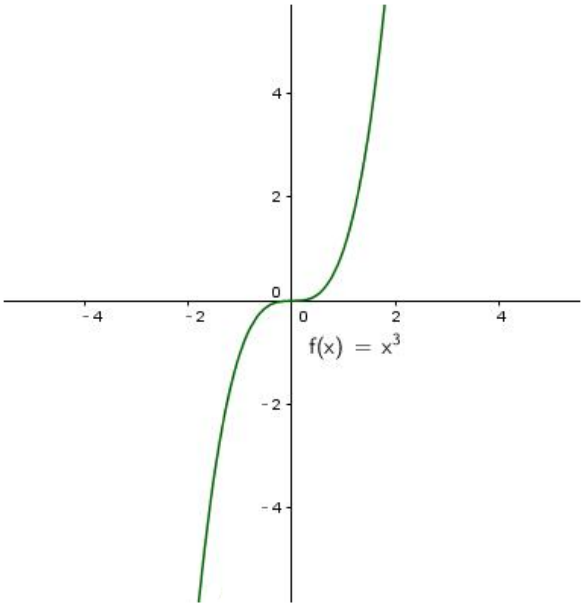
6.1、局部极值问题

这个问题好歹是个局部极值，只不过不是全局极值，理论上我们要求全局最优解的话，我们要把所有极小值找出来，你可能要不断的去设置不同的初始迭代点，反复的求解让它收敛到不同的局部最小值，然后来比较谁最小来找到一个全局最小值。

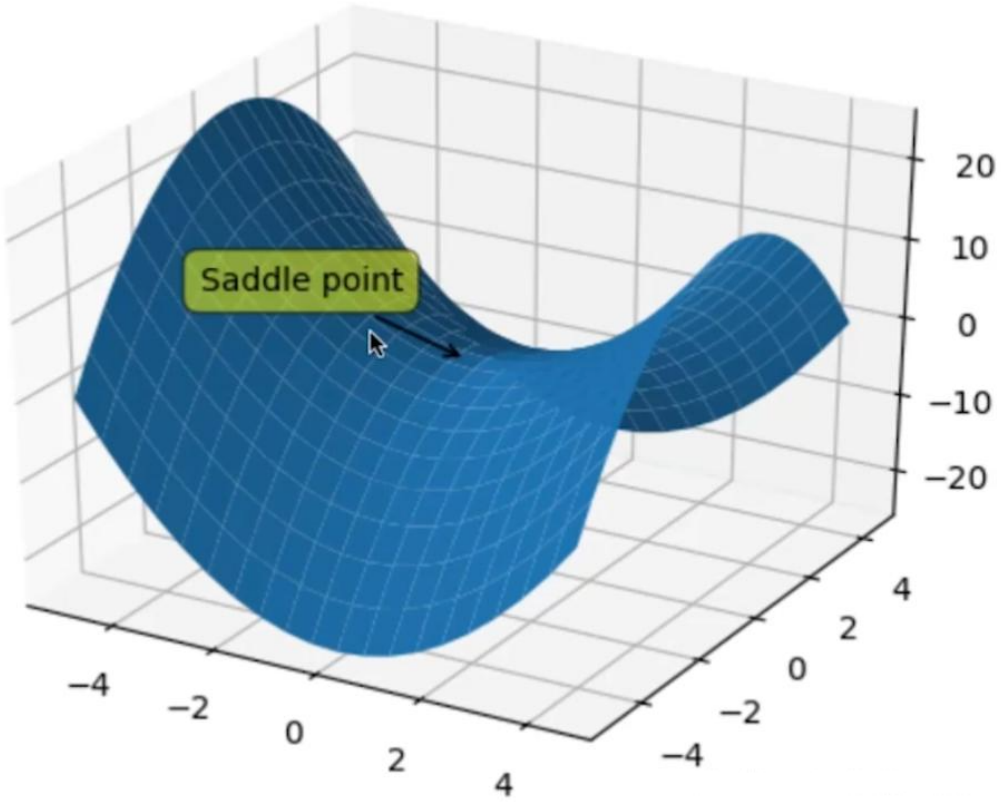


6.2、鞍点问题

前面我们说过一元函数 x^3 函数，在坐标为 0 处它是驻点，但是它连局部最小值都不是，对应多元函数来说，我们称之为鞍点（既不是极大值点也不是极小值点的临界点，叫做鞍点）。严格定义是，在这一点它的 Hessian 矩阵是不定的，既不正定也不负定，这样它就即不是极小值的点也不是极大值的点。



在物理上要广泛一些，指在一个方向是极大值，另一个方向是极小值的点。



7、凸优化问题

前面我们说过数值优化面临两个问题，一个是局部极值问题，和鞍点问题，我们能不能避免这两个问题呢？

只要我们对优化问题进行限定就可以，这类问题有两个限定条件

1、优化变量的可行域必须是凸集

2、优化函数必须是个凸函数

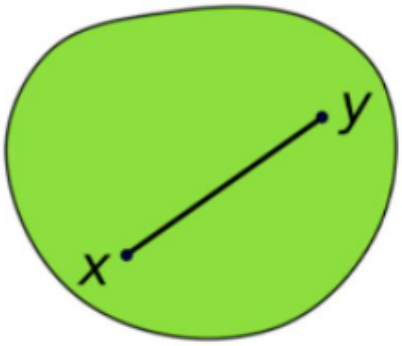
同时满足这两个限定条件的问题，叫做凸优化问题，从而才有局部极小值进而全局极小值。

8、凸集

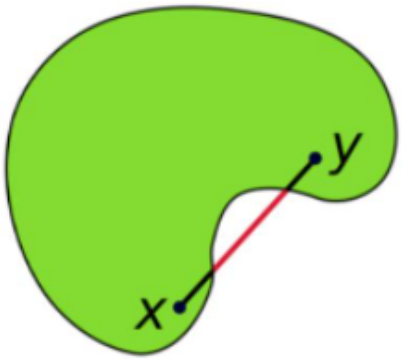
凸集的定义：对于一个点的集合 C，有 x,y 它都是属于 C 里面的两个点，它们两点的连线中任何一点也是属于集合 C 的。例如立方体就是凸集。

$$\theta x + (1-\theta)y \in C$$

$$0 \leq \theta \leq 1$$



凸集



非凸集

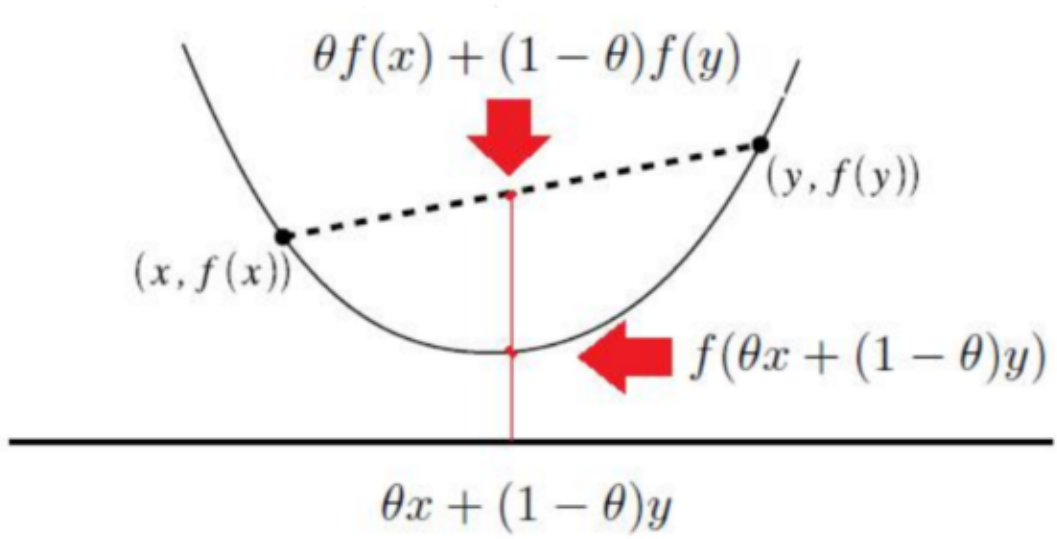
欧式空间 R^n

$$x, y \in R^n \rightarrow \theta x + (1-\theta)y \in R^n$$

它的意义在于，很多时候可行域就是欧式空间，那肯定是凸集。在凸集的前提下，才可以进行最优化问题求解。

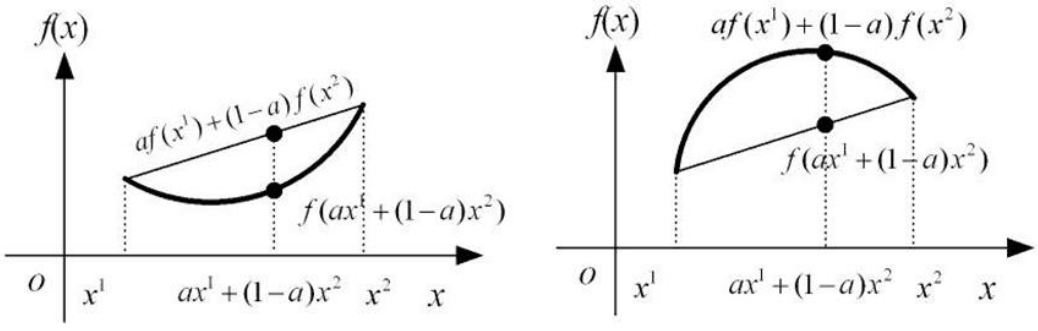
9、凸函数

凸函数在函数上任意找两点它们的连续就是割线上的值比对应的 $f(x)$ 的值要大



函数的二阶导数是和函数的凹凸性是有关系的，凹凸性怎么定义的?

先来做简单的介绍，这里先记住凸函数是向下凸的， 反正就是凹的，是否是凸函数可以通过二阶导数，如果二阶导数是大于 0 就是凸函数。$f''(x) > 0$



（a）凸函数　　（b）凹函数

如果一元函数，那么 $f''(x) \geq 0$ 就是凸函数。多元函数 Hessian 矩阵是半正定的，它就是凸函数；多元函数 Hessian 矩阵是正定的，它就是严格的凸函数。

如果每个函数 $f_i(x)$ 都是凸函数，那么它们的非负线性组合 $f(x) = \sum_{i=1}^{n} w_i f_i(x)$ if $\mathrm{w_i} \geq 0$ 也是凸函数。

10、凸优化表达形式

凸优化的定义是目标函数是凸函数，可行域是个凸集，如果有这两个限定条件的话，局部最优解一定是全局最优解。

凸优化一般表达形式：

$$\min f(x)$$

$x \in C$，其中 C 是凸集

带等式约束的表达形式：

$$\min f(x) \quad s.t \quad h_i(x) = 0 \quad i = 1, 2, \cdots, n$$

带不等式约束的表达形式：

$$\min f(x) \quad s.t \quad h_i(x) \leq 0 \quad i = 1, 2, \cdots, n$$

往往我们需要去证明的一些机器学习算法它们都是凸优化问题，比如逻辑回归，SVM，线性回归等，证明它的可行域是凸集，目标函数是凸函数就可以了，凸优化规避了局部最小值问题，而且也规避了鞍点问题，$f(x)$ 是凸函数，它的 Hessian 矩阵是半正定的，它是半正定的也就规避了鞍点问题。

11、拉格朗日乘子法

高等数学和微积分的时候我相信大家都学过，用来求解等式约束下的极值问题的

$$\min f(x) \quad s.t \quad h_i(x) = 0 \quad i = 1, 2, \cdots, n$$

拉格朗日乘子法，把对 x 带约束条件的优化问题，转化为不带约束条件的优化问题

$$L(x,\lambda)=f(x)+\sum_{i=1}^{n}\lambda_i h_i(x)$$

$L(x,\lambda)$ 叫做 Lagrange 函数（拉格朗日函数），λ 叫做拉格朗日乘子（其实就是系数）。求 $L(x,\lambda)$ 对 x 的偏导数，对 λ 求偏导数为，令导数为0，求解出 x,λ 的值，那么 x 就是函数 $f(x)$ 在附加条件 $h(x)$ 下的极值点。

以上就是拉格朗日乘数法，通俗理解拉格朗日乘数法就是将含有等式条件约束优化问题转换成了无约束优化问题构造出拉格朗日函数 $L(x,\lambda)$ 。

参考文章

12、KKT条件

假设我们面对的是不等式条件约束优化问题，如下：

$$\min f(x) \quad s.t \quad h_i(x)\le 0 \quad i=1,2,\cdots,n$$

针对上式，显然是一个不等式约束最优化问题，不能再使用拉格朗日乘数法，因为拉格朗日乘数法是针对等式约束最优化问题。

拉格朗日乘数法的扩展，用来解决带不等式约束条件的一种理论结果：

$$L(x,\lambda)=f(x)+\sum_{i=1}^{n}\lambda_i h_i(x)$$

约束条件为：

- $\nabla L(x,\lambda)=0$
- $\lambda\ge 0$
- $\lambda h(x)=0$

上面条件就称为KKT条件。

参考文章

13、拉格朗日对偶

凸优化，非线性规划问题，甚至是运筹学里面的线性规划问题都会涉及这个概念，它的意义是把原始问题转化为另外一个问题来求解，但是转化之后的问题要容易求解一点，拉格朗日乘数法的扩展，用来解决既带等式约束条件，又带不等式约束条件的一种方法通过把原问题转换为对偶问题来求解，很多时候对偶问题比原问题更容易求解。

$$\min f(x)$$

约束条件如下

$h_i(x)\le 0 \quad i=1,2,\cdots,n$

构建一个广义的拉格朗日函数，所谓广义就是还包括不等式约束条件。在下面式子中你会发现对 x 的约束没有了，虽然有个对 μ 的约束：

$$L(x,\lambda)=f(x)+\sum_{i=1}^{n}\lambda_i h_i(x) \quad \lambda_j\ge 0$$

原始问题

$$\min_{x}\max_{\lambda} L(x,\lambda) \quad s.t \quad \lambda \geq 0$$

我们定义对偶问题为（对上面方程的求解等效求解下面方程）：

$$\max_{\lambda}\min_{x} L(x,\lambda) \quad s.t \quad \lambda \geq 0$$

其实就是把 min 和 max 对调了一下，当然对应的变量也要变换。

对偶问题有什么好处呢？对于原问题，我们要先求里面的 max，再求外面的 min。而对于对偶问题，我们可以先求里面的 min。有时候，先确定里面关于 x 的函数最小值，比原问题先求解关于 λ 的最大值，要更加容易解。

但是原问题跟对偶问题并不是等价的，这里有一个强对偶性、弱对偶性的概念，弱对偶性是对于所有的对偶问题都有的一个性质。

所有的下凸函数都满足强对偶性,如果两个问题是强对偶的，那么这两个问题其实是等价的问题:

$$\min_{x}\max_{\lambda} L(x,\lambda) = \max_{\lambda}\min_{x} L(x,\lambda) \quad s.t \quad \lambda \geq 0$$

这里给出一个弱对偶性的推导过程：

$$\max_{\lambda}\min_{x} L(x,\lambda) = \min_{x} L(x,\lambda^*) \leq L(x^*,\lambda^*) \leq \max_{\lambda} L(x^*,\lambda) = \min_{x}\max_{\lambda} L(x,\lambda)$$

其中 x^*, λ^* 是函数取最大值最小值的时候对应的最优解，也就是说，原问题始终大于等于对偶问题:

$$\min_{x}\max_{\lambda} L(x,\lambda) \geq \max_{\lambda}\min_{x} L(x,\lambda) \quad s.t \quad \lambda \geq 0$$