

法律声明

□ 本课件包括演示文稿、示例、代码、题库、视频和声音等内容，小象学院和主讲老师拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意及内容，我们保留一切通过法律手段追究违反者的权利。

□ 课程详情请咨询

■ 微信公众号：小象

■ 新浪微博：ChinaHadoop



梯度下降与拟牛顿



小象学院
ChinaHadoop.cn

邹博

预备题目

□ 已知二次函数的一个点函数值和导数值，以及另外一个点的函数值，如果确定该函数的解析式？

■ 即：二次函数 $f(x)$ ，已知 $f(a)$ ， $f'(a)$ ， $f(b)$ ，求 $f(x)$

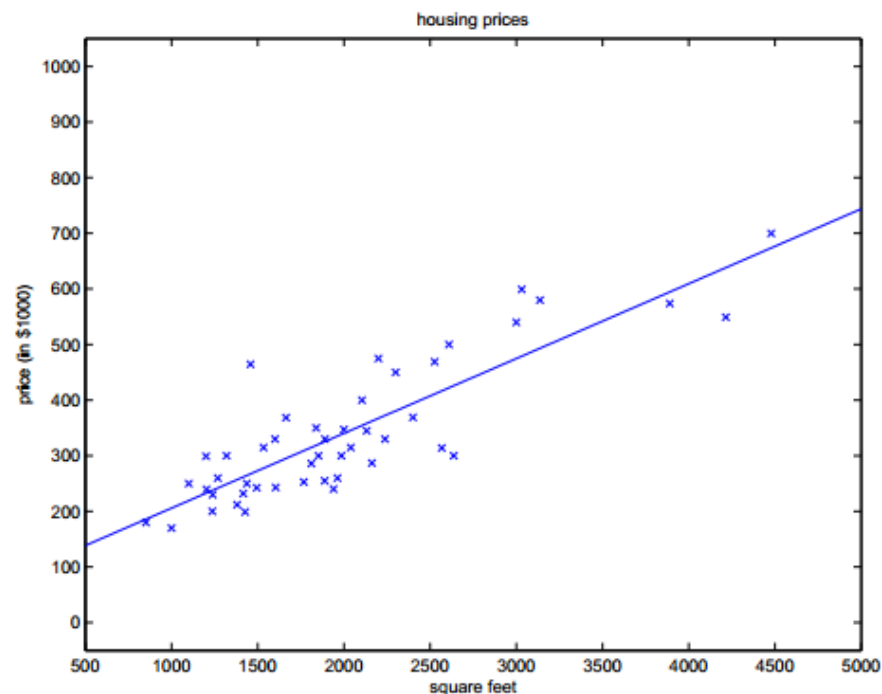
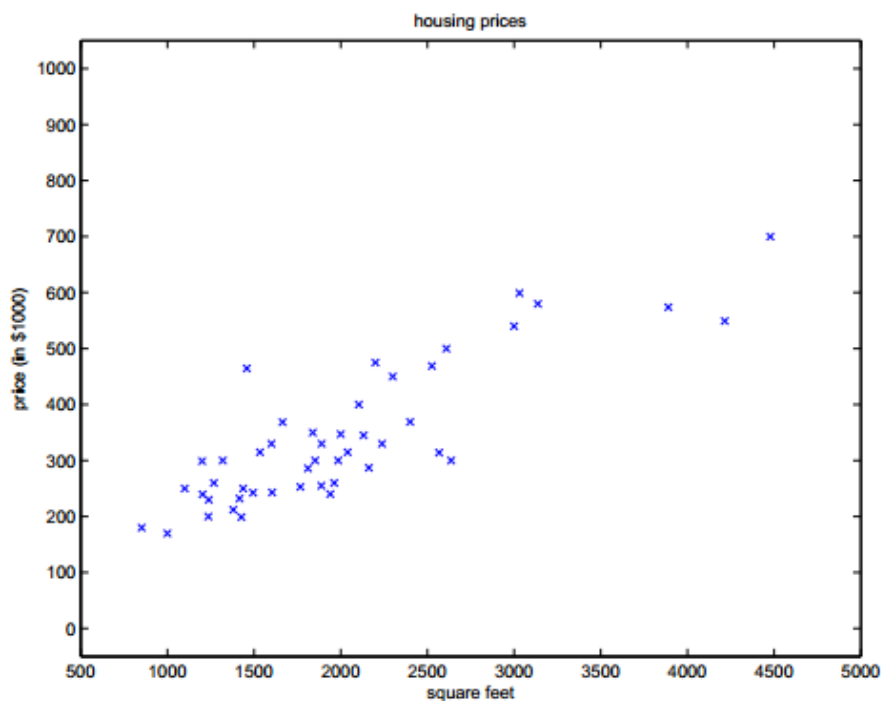
□ 特殊的，若 $a=0$ ，题目变成：

■ 对于二次函数 $f(x)$ ，已知 $f(0)$ ， $f'(0)$ ， $f(a)$ ，求 $f(x)$

$$f(x) = \frac{f(a) - f'(0)a - f(0)}{a^2}x^2 + f'(0)x + f(0)$$

从线性回归谈起

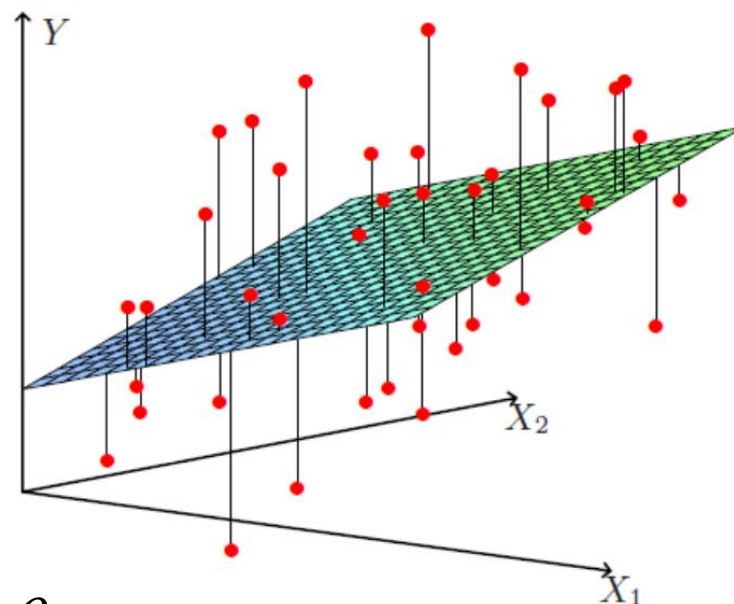
□ $y=ax+b=ax_1+bx_0$, 其中, $x_0\equiv 1$



多个变量的情形

□ 考虑两个变量

Living area (feet ²)	#bedrooms	Price (1000\$)
2104	3	400
1600	3	330
2400	3	369
1416	2	232
3000	4	540
⋮	⋮	⋮



$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

$$h_{\theta}(x) = \sum_{i=0}^n \theta_i x_i = \theta^T x$$

θ 的解析式的求解过程

□ 将M个N维样本组成矩阵X:

■ 其中，X的每一行对应一个样本，共M个样本

■ X的每一列对应样本的一个维度，共N维

□ 其实还有额外的一维常数项，全为1

□ 一个比较“符合常理”的误差函数为:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 = \frac{1}{2} (X\theta - y)^T (X\theta - y)$$

□ 已经论证：最小二乘建立的目标函数，是在高斯噪声的假设下，利用极大似然估计的方法建立的。

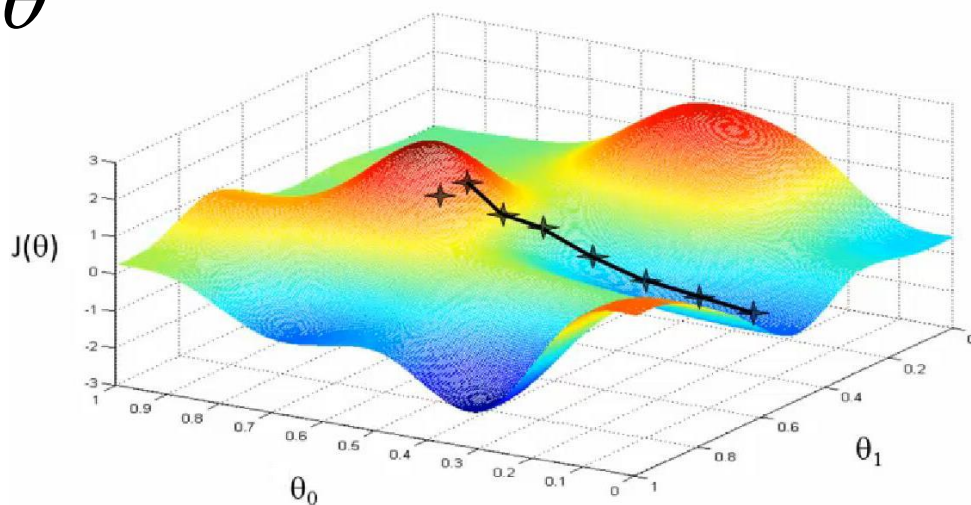
梯度下降算法 $J(\theta) = \frac{1}{2} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$

□ 初始化 θ (随机初始化)

□ 沿着负梯度方向迭代，更新后的 θ 使 $J(\theta)$ 更小

■ α : J 学习率、步长 $\theta_j = \theta_j - \alpha \cdot \frac{\partial J(\theta)}{\partial \theta}$

Gradient Descent



线性回归目标函数的梯度方向计算

$$\begin{aligned}\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} (h_{\theta}(x) - y)^2 \\ &= 2 \cdot \frac{1}{2} (h_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} (h_{\theta}(x) - y) \\ &= (h_{\theta}(x) - y) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{i=0}^n \theta_i x_i - y \right) \\ &= (h_{\theta}(x) - y) x_j\end{aligned}$$

问题似乎完美解决

□ 算法描述+凸函数极值

Repeat until convergence {

$$\theta_j := \theta_j + \alpha \sum_{i=1}^m (y^{(i)} - h_{\theta}(x^{(i)})) x_j^{(i)}$$

}

gradient descent. Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima; thus gradient descent always converges (assuming the learning rate α is not too large) to the global minimum. Indeed, J is a convex quadratic function.

思考

□ 学习率 α 如何确定

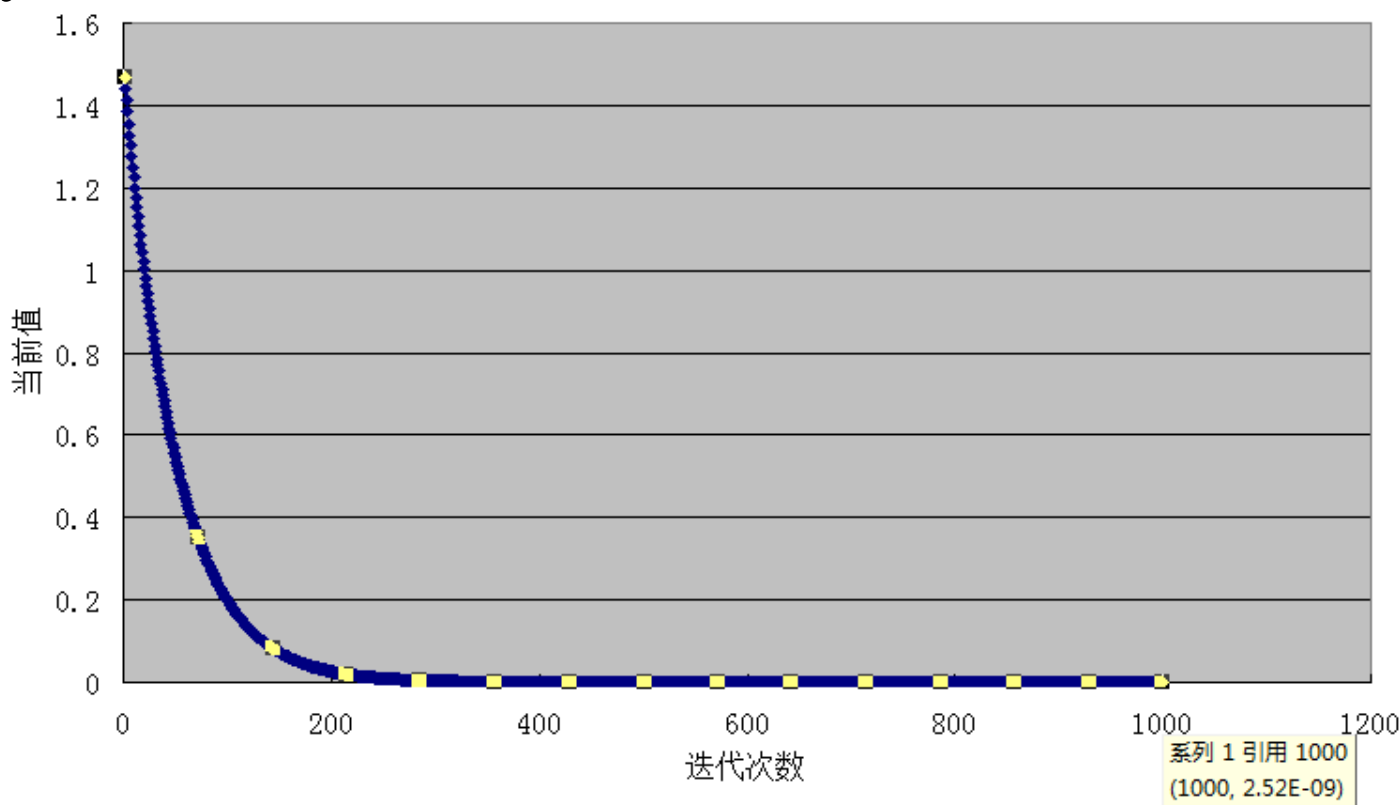
- 使用固定学习率还是变化学习率
- 学习率设置多大比较好?

□ 下降方向

- 处理梯度方向，其他方向是否可以?
- 可行方向和梯度方向有何关系?

实验：固定学习率的梯度下降

□ $y=x^2$ ，初值取 $x=1.5$ ，学习率使用0.01



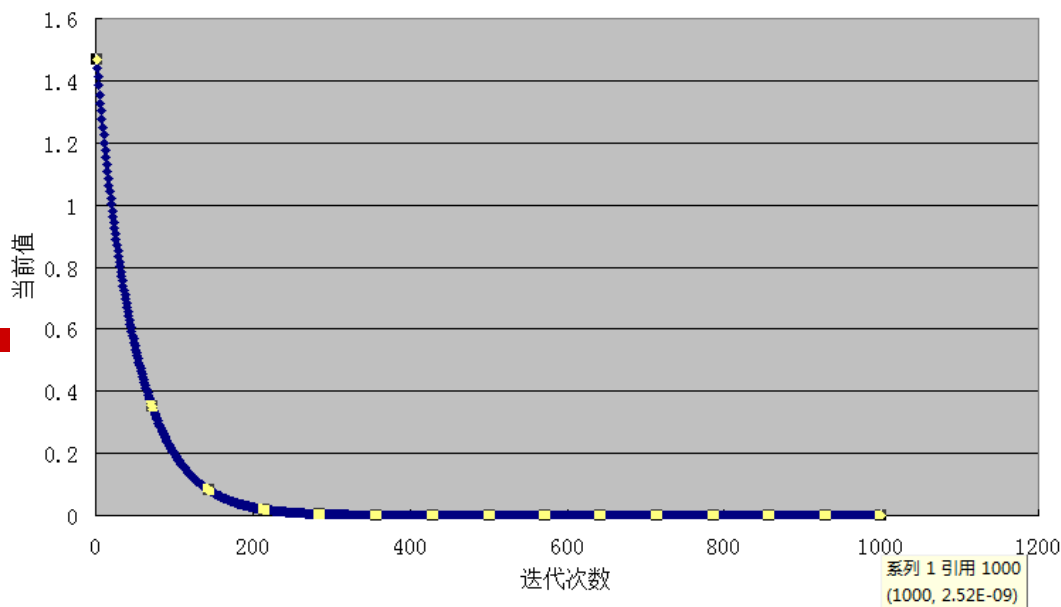
固定学习率

□ 分析：

□ 效果还不错

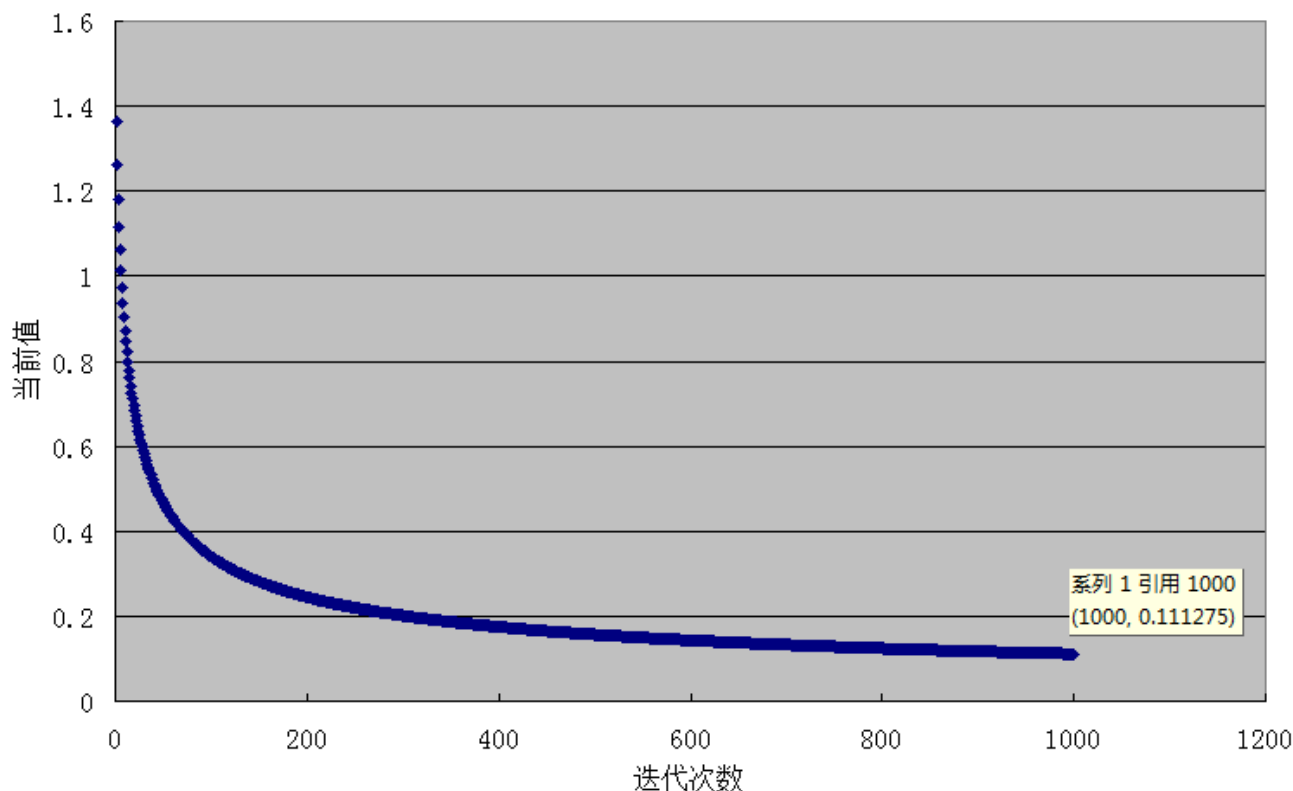
□ 经过200次迭代， $x=0.0258543$ ；

□ 经过1000次迭代， $x=2.52445 \times 10^{-9}$



实验2：固定学习率的梯度下降

□ $y=x^4$ ，初值取 $x=1.5$ ，学习率使用0.01

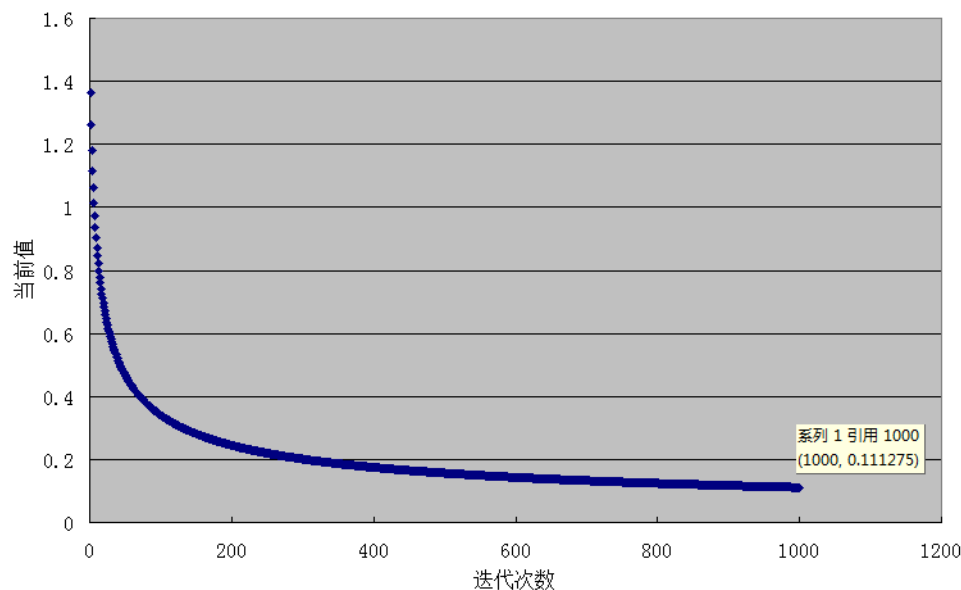


固定学习率

□ 分析：效果不理想

■ 经过200次迭代， $x=0.24436$ ；

■ 经过1000次迭代， $x=0.111275$



附：固定学习率实验的C代码

```
int _tmain(int argc, _TCHAR* argv[])
{
    double x = 1.5;
    double d;           //一阶导
    double a = 0.01;    //学习率
    for(int i = 0; i < 1000; i++)
    {
        d = g(x);
        x -= d * a;
        cout << i << '\t' << a << '\t' << x << '\n';
    }
    return 0;
}
```

优化学习率

□ 分析 “学习率 α ”在 $f(x)$ 中的意义

□ 调整学习率：

■ 在参数迭代的开始阶段，由于与最优值差距较大，可以使用较大的学习率；在迭代后期，使用较小的学习率增加稳定性和精度。

■ 如何构造学习率 α

梯度下降的运行过程分析

- $x_k=a$, 沿着负梯度方向, 移动到 $x_{k+1}=b$, 有:

$$b = a - \alpha \nabla F(a) \Rightarrow f(a) \geq f(b)$$

- 从 x_0 为出发点, 每次沿着当前函数梯度反方向移动一定距离 α_k , 得到序列:

$$x_0, x_1, x_2, \dots, x_n$$

- 对应的各点函数值序列之间的关系为:

$$f(x_0) \geq f(x_1) \geq f(x_2) \geq \dots \geq f(x_n)$$

- 当 n 达到一定值时, 函数 $f(x)$ 收敛到局部最小值

视角转换

- 记当前点为 \mathbf{x}_k ，当前搜索方向为 \mathbf{d}_k (如：负梯度方向)，因为学习率 α 是待考察的对象，因此，将下列函数 $f(\mathbf{x}_k + \alpha \mathbf{d}_k)$ 看做是关于 α 的函数 $h(\alpha)$ 。

$$h(\alpha) = f(x_k + \alpha d_k), \quad \alpha > 0$$

- 当 $\alpha=0$ 时， $h(0)=f(\mathbf{x}_k)$
- 导数 $\nabla h(\alpha) = \nabla f(x_k + \alpha d_k)^T d_k$

学习率 α 的计算标准

- 因为梯度下降是寻找 $f(x)$ 的最小值，那么，在 \mathbf{x}_k 和 \mathbf{d}_k 给定的前提下，即寻找函数 $f(\mathbf{x}_k + \alpha \mathbf{d}_k)$ 的最小值。即：

$$\alpha = \arg \min_{\alpha > 0} h(\alpha) = \arg \min_{\alpha > 0} f(x_k + \alpha d_k)$$

- 进一步，如果 $h(\alpha)$ 可导，局部最小值处的 α 满足：

$$h'(\alpha) = \nabla f(x_k + \alpha d_k)^T d_k = 0$$

学习率函数导数的分析 $h'(\alpha) = \nabla f(x_k + \alpha d_k)^T d_k = 0$

□ 将 $\alpha=0$ 带入：

$$h'(0) = \nabla f(x_k + 0 * d_k)^T d_k = \nabla f(x_k)^T d_k$$

□ 下降方向 d_k 可以选负梯度方向 $d_k = -\nabla f(x_k)$

■ 或者选择与负梯度夹角小于 90° 的某方向(后面会继续阐述搜索方向问题)

□ 从而： $h'(0) < 0$

□ 如果能够找到足够大的 α ，使得 $h'(\hat{\alpha}) > 0$

□ 则必存在某 α ，使得 $h'(\alpha^*) = 0$

□ α^* 即为要寻找的学习率。

线性搜索(Line Search)

□ 最简单的处理方式

■ 二分线性搜索(Bisection Line Search)

- 不断将区间 $[\alpha_1, \alpha_2]$ 分成两半，选择端点异号的一侧，知道区间足够小或者找到当前最优学习率。

回溯线性搜索(Backing Line Search)

- 基于Armijo准则计算搜索方向上的最大步长，其基本思想是沿着搜索方向移动一个较大的步长估计值，然后以迭代形式不断缩减步长，直到该步长使得函数值 $f(\mathbf{x}_k + \alpha \mathbf{d}_k)$ 相对与当前函数值 $f(\mathbf{x}_k)$ 的减小程度大于预设的期望值(即满足Armijo准则)为止。

$$f(x_k + \alpha d_k) \leq f(x_k) + c_1 \alpha \nabla f(x_k)^T d_k \quad c_1 \in (0,1)$$

回溯与二分线性搜索的异同

- 二分线性搜索的目标是求得满足 $h'(\alpha) \approx 0$ 的最优步长近似值，而回溯线性搜索放松了对步长的约束，只要步长能使函数值有足够大的变化即可。
- 二分线性搜索可以减少下降次数，但在计算最优步长上花费了不少代价；回溯线性搜索找到一个差不多的步长即可。

回溯线性搜索

- x 为当前值
- d 为 x 处的导数
- a 为输入学习率
- 返回调整后的学习率

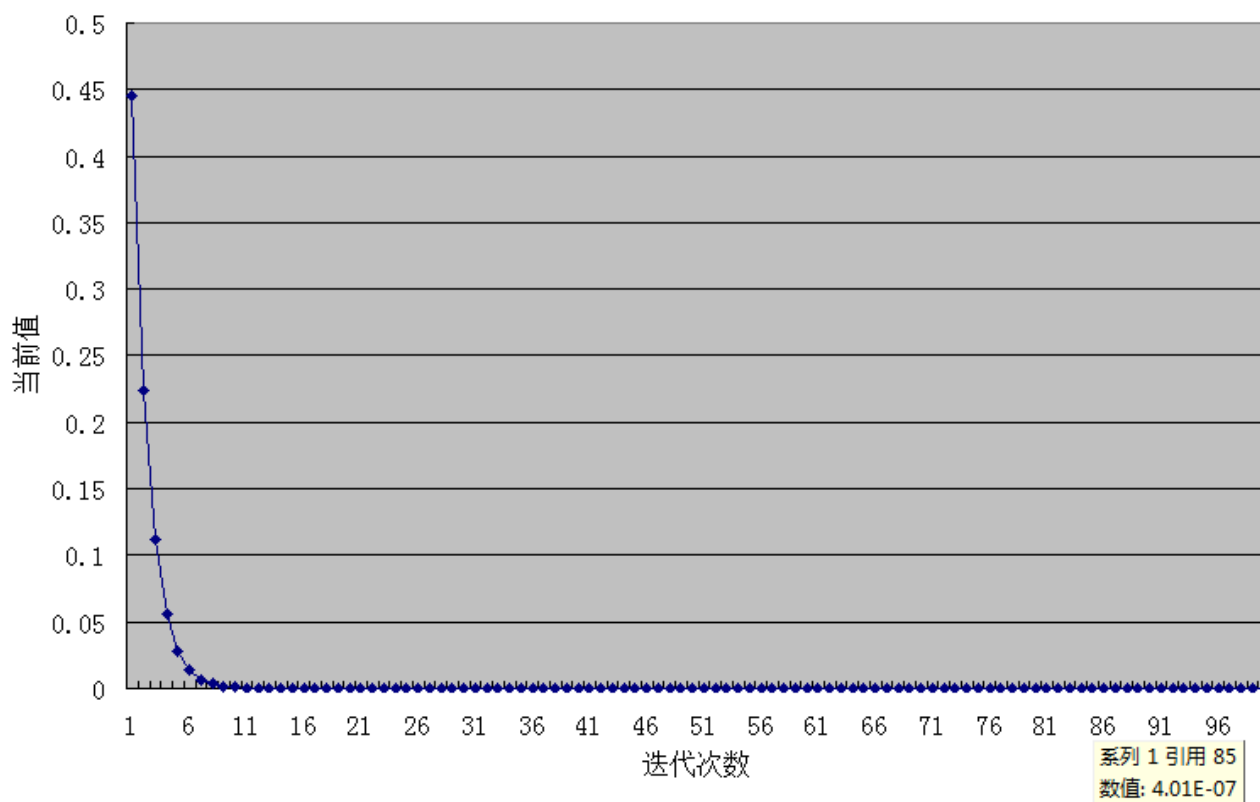
```
double GetA_ArmiJo(double x, double d, double a)
{
    double c1 = 0.3;
    double now = f(x);
    double next = f(x - a*d);

    int count = 30;
    while(next < now)
    {
        a *= 2;
        next = f(x - a*d);
        count--;
        if(count == 0)
            break;
    }

    count = 50;
    while(next > now - c1*a*d*d)
    {
        a /= 2;
        next = f(x - a*d);
        count--;
        if(count == 0)
            break;
    }
    return a;
}
```


实验：回溯线性搜索寻找学习率

□ $y=x^4$ ，初值取 $x=1.5$ ，回溯线性方法



回溯线性搜索

□ 分析：

□ 效果还不错

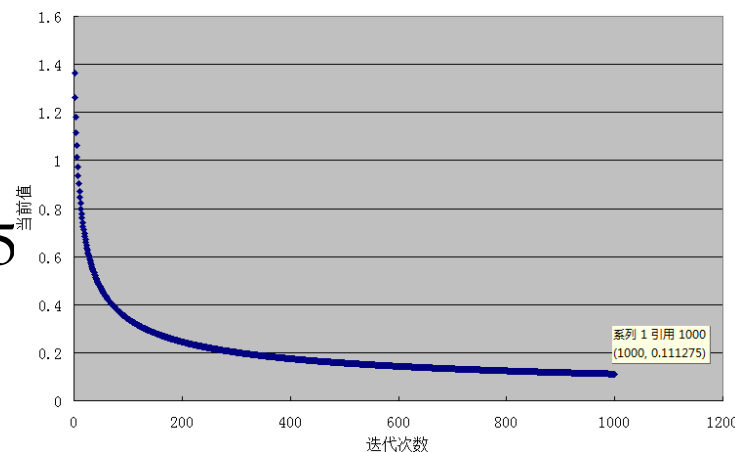
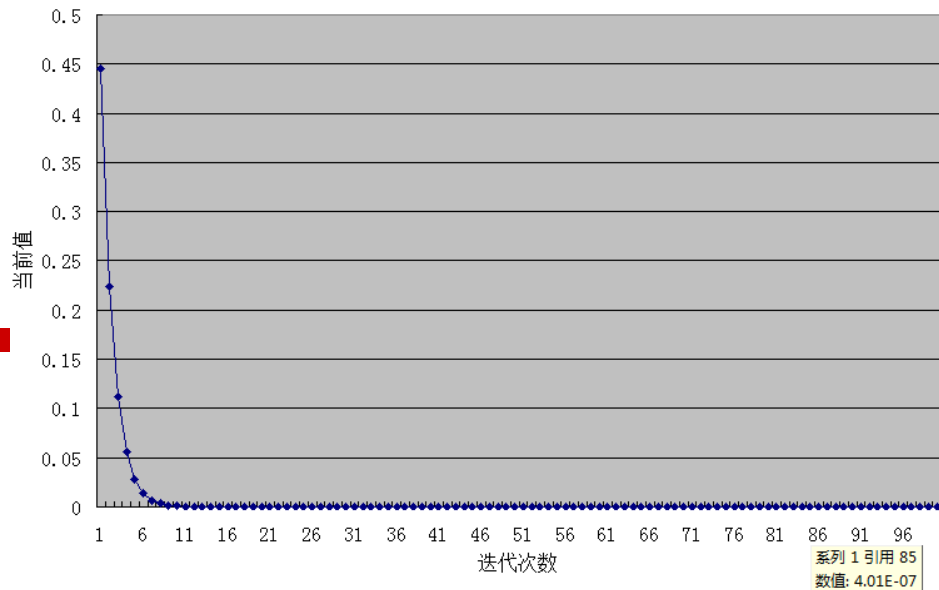
□ 经过12次迭代， $x=0.00010872$ ；

□ 经过100次迭代， $x=3.64905 \times 10^{-7}$

■ 试比较固定学习率时：

□ 经过200次迭代， $x=0.24436$ ；

□ 经过1000次迭代， $x=0.111275$



回溯线性搜索的思考：插值法

- 采用多项式插值法(Interpolation) 拟合简单函数，然后根据该简单函数估计函数的极值点，这样选择合适步长的效率会高很多。
- 现在拥有的数据为： x_k 处的函数值 $f(x_k)$ 及其导数 $f'(x_k)$ ，再加上第一次尝试的步长 α_0 。如果 α_0 满足条件，显然算法退出；若 α_0 不满足条件，则根据上述信息可以构造一个二次近似函数：
$$h_q(\alpha) = \frac{h(\alpha_0) - h'(0)\alpha_0 - h(0)}{\alpha_0^2} \alpha^2 + h'(0)\alpha + h(0)$$

二次插值法求极值 $h_q(\alpha) = \frac{h(\alpha_0) - h'(0)\alpha_0 - h(0)}{\alpha_0^2} \alpha^2 + h'(0)\alpha + h(0)$

□ 显然，导数为0的最优值为：

$$\alpha_1 = \frac{h'(0)\alpha_0^2}{2[h'(0)\alpha_0 + h(0) - h(\alpha_0)]}$$

□ 若 α_1 满足Armijo准则，则输出该学习率；否则，继续迭代。

二次插值法

- x 为当前值
- d 为 x 处的导数
- a 为输入学习率
- 返回调整后的学习率

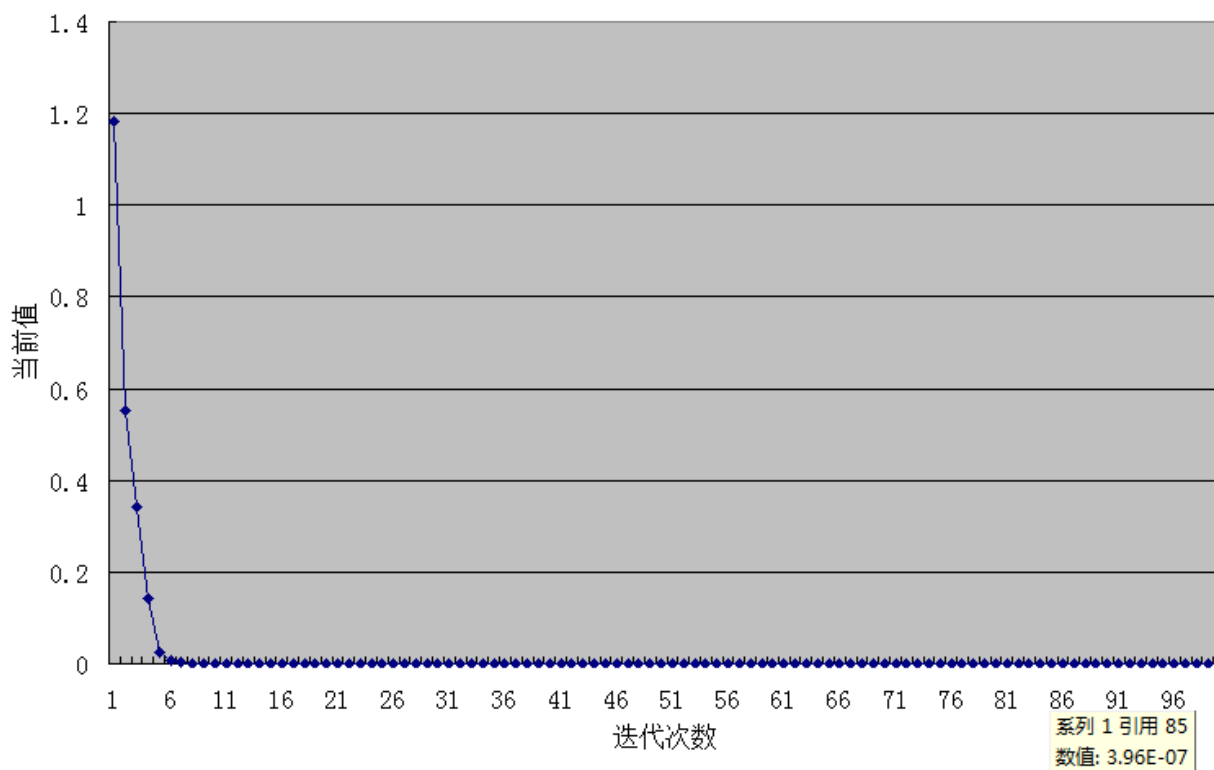
```
double GetA_Quad(double x, double d, double a)
{
    double c1 = 0.3;
    double now = f(x);
    double next = f(x - a*d);
```

```
int count = 30;
while(next < now)
{
    a *= 2;
    next = f(x - a*d);
    count--;
    if(count == 0)
        break;
}
```

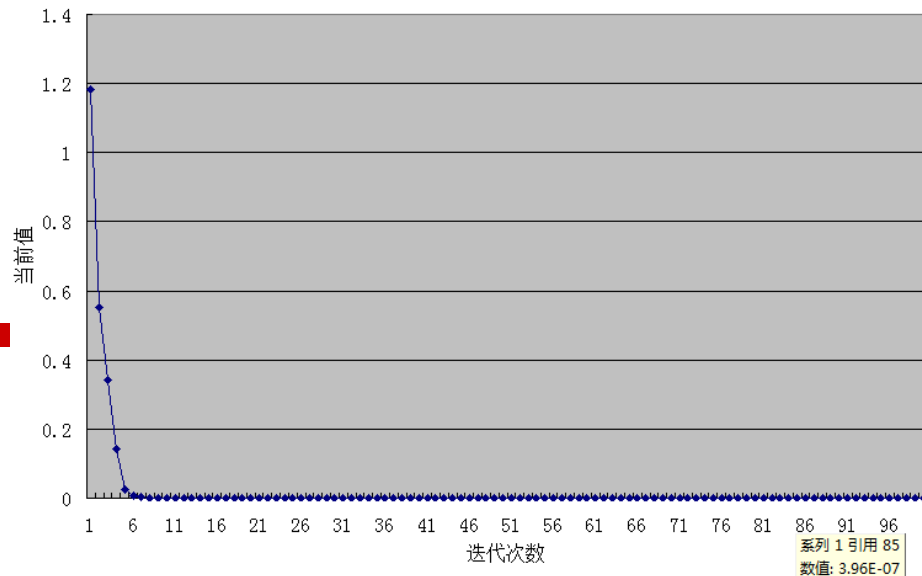
```
count = 50;
double b;
while(next > now - c1*a*d*d)
{
    b = d * a * a / (now + d * a - next);
    b /= 2;
    if(b < 0)
        a /= 2;
    else
        a = b;
    next = f(x - a*d);
    count--;
    if(count == 0)
        break;
}
return a;
```

实验：二次插值线性搜索寻找学习率

□ $y=x^4$ ，初值取 $x=1.5$ ，二次插值线性搜索方法



二次插值线性搜索



□ 分析：

□ 效果还不错

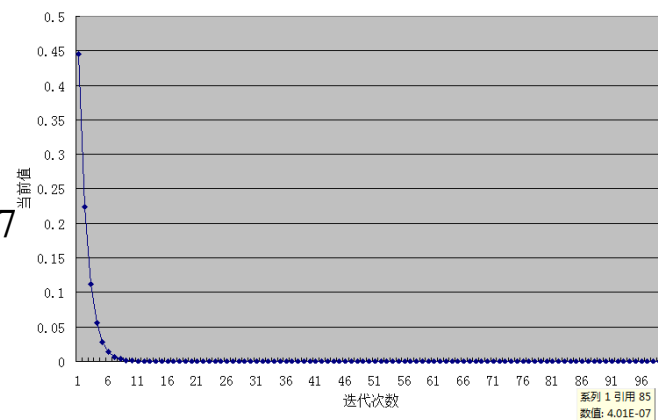
□ 经过12次迭代， $x=0.0000282229$ ；

□ 经过100次迭代， $x=3.61217 \times 10^{-7}$

■ 试比较回溯线性搜索时：

□ 经过12次迭代， $x=0.00010872$ ；

□ 经过1000次迭代， $x=3.649 \times 10^{-7}$



总结与思考

- 通过使用线性搜索的方式，能够比较好的解决学习率问题
- 一般的说，回溯线性搜索和二次插值线性搜索能够基本满足实践中的需要
- 问题：
 - 可否在搜索过程中，随着信息的增多，使用三次或者更高次的函数曲线，从而得到更快的学习率收敛速度？
 - 为避免高次产生的震荡，可否使用三次Hermite多项式，在端点保证函数值和一阶导都相等，从而构造更光顺的简单低次函数？

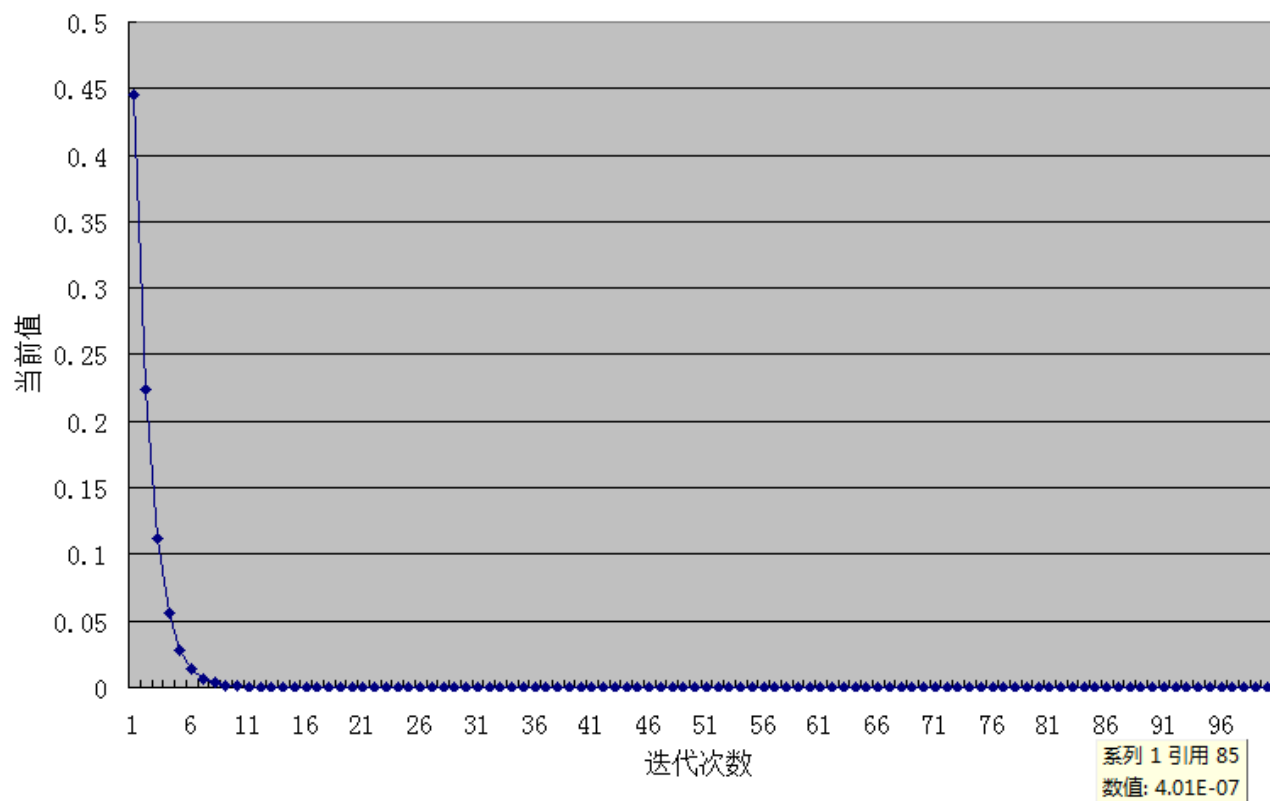
搜索方向

- 若搜索方向不是严格梯度方向，是否可以？
- 思考：
- 因为函数二阶导数反应了函数的凸凹性；二阶导越大，一阶导的变化越大。在搜索中，可否用二阶导做些“修正”？如：二者相除？

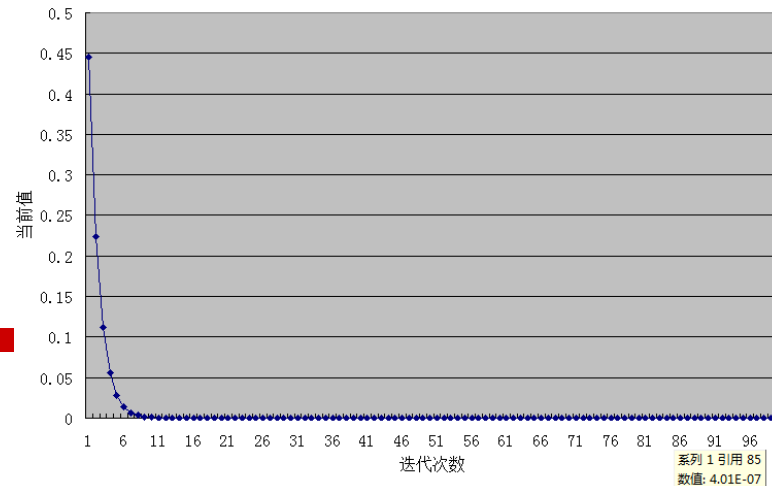
$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

实验：搜索方向的探索

□ $y=x^4$ ，初值取 $x=1.5$ ，负梯度除以二阶导



搜索方向的探索



- 效果出奇的好!
- 经过12次迭代, $x=0.00770735$;
- 经过100次迭代, $x=3.68948 \times 10^{-18}$
- 试比较二次插值线性搜索时:
 - 经过12次迭代, $x=0.0000282229$;
 - 经过1000次迭代, $x=3.61217 \times 10^{-7}$

平方根算法

□ 在任意点 x_0 处Taylor展开

$$\text{令 } f(x) = x^2 - a, \text{ 即 } f(x) = 0$$

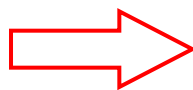
$$\Rightarrow f(x) = f(x_0) + f'(x_0)(x - x_0) + o(x)$$

$$\Rightarrow f(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

$$\Rightarrow 0 = f(x_0) + f'(x_0)(x - x_0)$$

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

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



将 $f(x_0) = x_0^2 - a$ 和 $f'(x_0) = 2x_0$ 带入,

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

$$\Rightarrow x = x_0 - \frac{x_0^2 - a}{2x_0}$$

$$\Rightarrow x = \frac{1}{2} \left(x_0 + \frac{a}{x_0} \right)$$

平方根

- 一般若干次(5、6次)迭代即可获得比较好的近似值。

```
float Calc(float x);
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    for(int i = 0; i <= 10; i++)
        cout << Calc((float)i) << '\n';
    return 0;
}

float Calc(float a)
{
    if(a < 1e-6) //负数或者0, 则直接返回0
        return 0;

    float x = a / 2;
    float t = a;
    while(fabs(x - t) > 1e-6)
    {
        t = x;
        x = (x + a/x) / 2;
    }
    return x;
}
```

试分析该代码的有效性和实践意义

```
double Reciprocal(double a)
{
    double x = 1;
    while(a * x >= 2)
    {
        if(a > 1)
            x /= 10;
        else
            x *= 10;
    }
    double t = a;
    while(fabs(x - t) > 1e-6)
    {
        t = x;
        x = x * (2 - a * x);
    }
    return x;
}
```

```
double Sqrt(double a)
{
    if(a < 1e-6)
        return 0;
    double x = 1;
    while(a * x * x >= 3)
        x *= 0.1;
    double t = a;
    while(fabs(x - t) > 1e-6)
    {
        t = x;
        x = (3 - a * t * t) * t / 2;
    }
    return Reciprocal(x);
}
```

分析上述结果的原因

□ 若 $f(x)$ 二阶导连续，将 $f(x)$ 在 x_k 处Taylor展开：

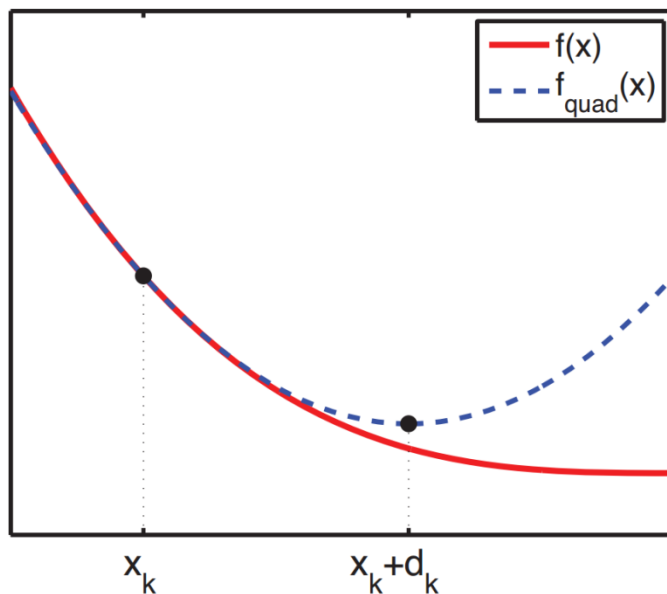
$$\varphi(x) = f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2} f''(x_k)(x - x_k)^2 + R_2(x)$$

$$\varphi'(x) \approx f'(x_k) + f''(x_k)(x - x_k)$$

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

牛顿法

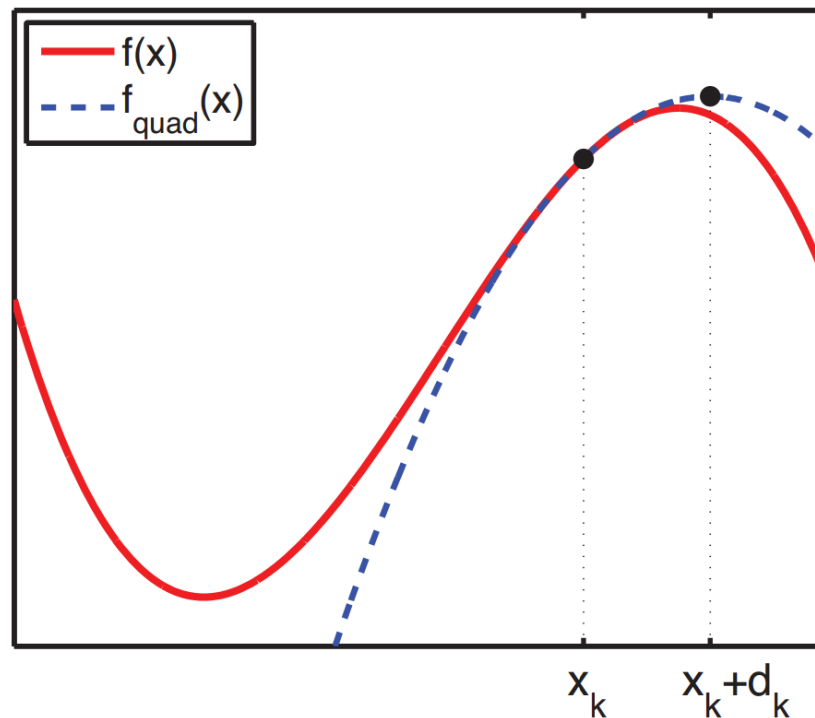
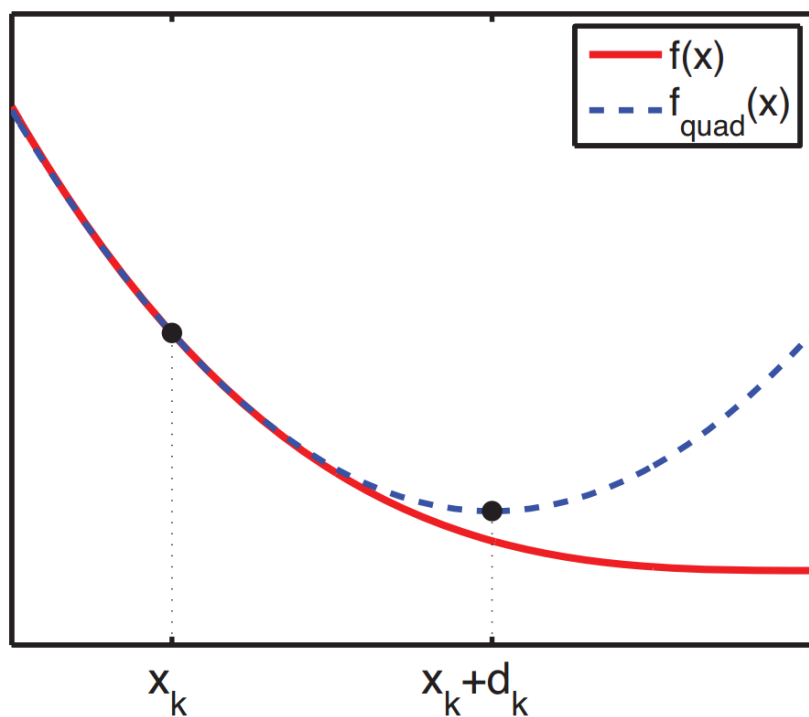
- 上述迭代公式，即牛顿法
- 该方法可以直接推广到多维：用方向导数代替一阶导，用Hessian矩阵代替二阶导



牛顿法的特点

- ❑ 经典牛顿法虽然具有二次收敛性，但是要求初始点需要尽量靠近极小点，否则有可能不收敛。
- ❑ 计算过程中需要计算目标函数的二阶偏导数的逆，时间复杂度较大。
- ❑ 目标函数的Hessian矩阵无法保持正定，会导致算法产生的方向不能保证是 f 在 x_k 处的下降方向，从而令牛顿法失效；
- ❑ 如果Hessian矩阵奇异，牛顿方向可能根本是不存在的。

二阶导非正定的情况(一维则为负数)



拟牛顿的思路

- 求Hessian矩阵的逆影响算法效率，同时，搜索方向并非严格需要负梯度方向或者牛顿方向；因此，可以用近似矩阵代替Hessian矩阵，只要满足该矩阵正定、容易求逆，或者可以通过若干步递推公式计算得到。
 - DFP: Davidon – Fletcher – Powell
 - BFGS: Broyden – Fletcher – Goldfarb - Shanno

二阶近似

□ 记函数 $f(\theta)$ 的梯度为 $g(\theta)$ ，二阶导函数为 $H(\theta)$

□ 在 θ_i 处 Taylor 展开：

$$f(\theta) = f(\theta_i) + (\theta - \theta_i)^T g_i + \frac{1}{2} (\theta - \theta_i)^T H_i (\theta - \theta_i) + o(\theta)$$

$$\Rightarrow f(\theta) \approx f(\theta_i) + (\theta - \theta_i)^T g(\theta_i) + \frac{1}{2} (\theta - \theta_i)^T H_i (\theta - \theta_i)$$

$$\Rightarrow g(\theta) = g(\theta_i) + H_i (\theta - \theta_i)$$

$$\Rightarrow g(\theta_{i-1}) = g(\theta_i) + H_i (\theta_{i-1} - \theta_i)$$

$$\Rightarrow g_{i-1} - g_i = H_i (\theta_{i-1} - \theta_i)$$

$$\Rightarrow H_i^{-1} (g_i - g_{i-1}) = \theta_i - \theta_{i-1}$$

$$\Rightarrow C_i \Delta g_i = \Delta \theta_i$$

$$\begin{cases} C_i = H_i^{-1} \\ \Delta g_i = g_i - g_{i-1} \\ \Delta \theta_i = \theta_i - \theta_{i-1} \end{cases}$$

DFP算法

$$C_i = C_{i-1} + a_{i-1}v_{i-1}v_{i-1}^T + b_{i-1}u_{i-1}u_{i-1}^T$$

$$\begin{cases} C_i \Delta g_i = \Delta \theta_i \\ C_i = C_{i-1} + a_{i-1}v_{i-1}v_{i-1}^T + b_{i-1}u_{i-1}u_{i-1}^T \end{cases} \Rightarrow (C_{i-1} + a_{i-1}v_{i-1}v_{i-1}^T + b_{i-1}u_{i-1}u_{i-1}^T) \Delta g_i = \Delta \theta_i$$

$$\begin{cases} C_i = H_i^{-1} \\ \Delta g_i = g_i - g_{i-1} \\ \Delta \theta_i = \theta_i - \theta_{i-1} \end{cases}$$

$$\Rightarrow C_{i-1} \Delta g_i + a_{i-1}v_{i-1}v_{i-1}^T \Delta g_i + b_{i-1}u_{i-1}u_{i-1}^T \Delta g_i = \Delta \theta_i$$

$$\xrightarrow{\text{def}} \begin{cases} v_{i-1} = \Delta \theta_i \\ u_{i-1} = C_{i-1} \Delta g_i \end{cases} \Rightarrow \begin{cases} a_{i-1}v_{i-1}^T \Delta g_i = 1 \\ b_{i-1}u_{i-1}^T \Delta g_i = -1 \end{cases} \Rightarrow \begin{cases} a_i = \frac{1}{v_{i-1}^T \Delta g_i} \\ b_i = -\frac{1}{u_{i-1}^T \Delta g_i} \end{cases}$$

$$\Rightarrow C_i = C_{i-1} + \frac{\Delta \theta_i \Delta \theta_i^T}{\Delta \theta_i^T \Delta g_i} - \frac{(C_{i-1} \Delta g_i)(C_{i-1} \Delta g_i)^T}{\Delta g_i^T C_{i-1} \Delta g_i}$$

DFP

$$C_i = C_{i-1} + \frac{\Delta\theta_i \Delta\theta_i^T}{\Delta\theta_i^T \Delta g_i} - \frac{(C_{i-1} \Delta g_i)(C_{i-1} \Delta g_i)^T}{\Delta g_i^T C_{i-1} \Delta g_i}$$

```
def hessian_r2(C, w, g):    # 这里的w和g是Δw和Δg
    n = w.size
    d = g.dot(w)
    if math.fabs(d) < 1e-4:  # 分母为0, 返回单位阵
        return np.identity(n)
    A = np.zeros((n, n))
    for i in range(n):
        A[i] = g[i] * g
    A /= d

    B = np.zeros((n, n))
    w2 = C.dot(w)
    for i in range(n):
        B[i] = w2[i] * w2
    d2 = w.dot(w2)
    if math.fabs(d2) < 1e-4:
        return np.identity(n)
    B /= d2
    return C + A - B
```

g 和 θ 的转换

$$\begin{cases} C_i = H_i^{-1} \\ \Delta g_i = g_i - g_{i-1} \\ \Delta \theta_i = \theta_i - \theta_{i-1} \end{cases}$$

□ 由DFP的迭代公式：

$$\begin{cases} C_i \Delta g_i = \Delta \theta_i \\ C_i = C_{i-1} + a_{i-1} v_{i-1} v_{i-1}^T + b_{i-1} u_{i-1} u_{i-1}^T \end{cases} \Rightarrow C_i = C_{i-1} + \frac{\Delta \theta_i \Delta \theta_i^T}{\Delta \theta_i^T \Delta g_i} - \frac{(C_{i-1} \Delta g_i)(C_{i-1} \Delta g_i)^T}{\Delta g_i^T C_{i-1} \Delta g_i}$$

□ 互换 g 和 θ ，得，

$$\begin{cases} \Delta g_i = H_i \Delta \theta_i \\ H_i = H_{i-1} + a_{i-1} v_{i-1} v_{i-1}^T + b_{i-1} u_{i-1} u_{i-1}^T \end{cases} \Rightarrow H_i = H_{i-1} + \frac{\Delta g_i \Delta g_i^T}{\Delta g_i^T \Delta \theta_i} - \frac{(H_{i-1} \Delta \theta_i)(H_{i-1} \Delta \theta_i)^T}{\Delta \theta_i^T H_{i-1} \Delta \theta_i}$$

■ 思考：如何方便的计算 H_i 的逆呢？

BFGS

- Sherman-Morrison公式：若A是n阶可逆矩阵，
u, v为n维列向量，若 $v^T A^{-1} u \neq -1$ 则

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^T A^{-1}}{1 + v^T A^{-1}u}$$

- 根据 $H_i = H_{i-1} + \frac{\Delta g_i \Delta g_i^T}{\Delta g_i^T \Delta \theta_i} - \frac{(H_{i-1} \Delta \theta_i)(H_{i-1} \Delta \theta_i)^T}{\Delta \theta_i^T H_{i-1} \Delta \theta_i}$
使用两次Sherman-Morrison公式，得，

$$C_i = \left(I - \frac{\Delta \theta_i \Delta g_i^T}{\Delta g_i^T \Delta \theta_i} \right) C_{i-1} \left(I - \frac{\Delta g_i \Delta \theta_i^T}{\Delta g_i^T \Delta \theta_i} \right) + \frac{\Delta \theta_i \Delta \theta_i^T}{\Delta g_i^T \Delta \theta_i}$$

Code

$$C_i = \left(I - \frac{\Delta\theta_i \Delta g_i^T}{\Delta g_i^T \Delta\theta_i} \right) C_{i-1} \left(I - \frac{\Delta g_i \Delta\theta_i^T}{\Delta g_i^T \Delta\theta_i} \right) + \frac{\Delta\theta_i \Delta\theta_i^T}{\Delta g_i^T \Delta\theta_i}$$

```
def lr_BFGS(data, alpha):
    n = len(data[0]) - 1
    w = np.zeros(n)      # 当前权值
    g = np.zeros(n)      # 当前梯度
    C = np.identity(n)   # Hessian阵的逆矩阵, 初始化为单位阵
    for times in range(10000):
        for d in data:
            x = np.array(d[:-1])    # 输入数据(n维列向量)
            y = d[-1]               # 期望输出(标量)
            g0 = g                  # 记录上一次的梯度
            w0 = w                  # 记录上一次的权值
            g = (y - pred(w, x)) * x # 梯度(n维列向量)
            w = w + alpha * C.dot(g)
            C = hessian_r(C, w-w0, g-g0)
        print times, w
    return w
```

```
def hessian_r(C, w, g):    # 这里的w和g是Δw和Δg
    n = w.size
    d = g.dot(w)
    if math.fabs(d) < 1e-5: # 分母为0, 返回单位阵
        return np.identity(n)
    A = np.zeros((n, n))
    B = np.zeros((n, n))
    for i in range(n):
        A[i] = w[i] * g
        B[i] = w[i] * w
    A /= d
    At = A.transpose()
    B /= d
    I = np.identity(n)
    return (I-A).dot(C).dot(I-At) + B
```

BFGS

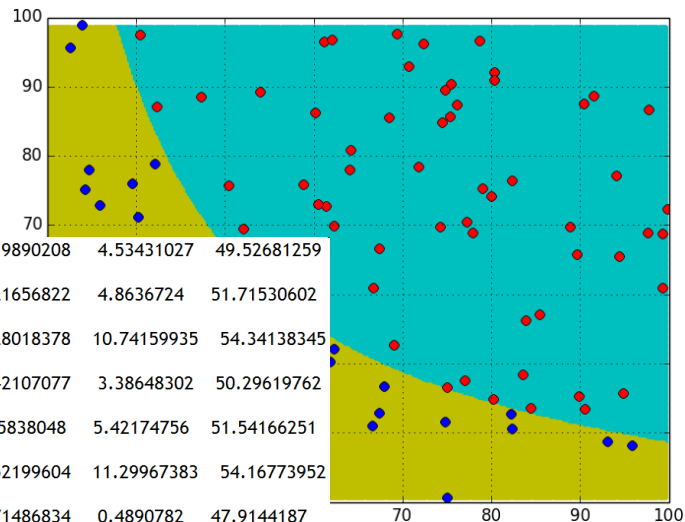
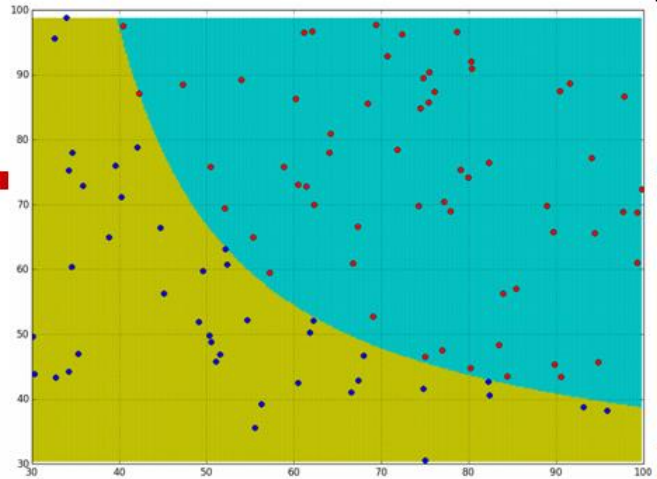
分类：Logistic回归

□ 沿似然函数正梯度上升

□ 维度提升

```
10863 [ -50.18428968 -1487.50027013 -1580.65133195 3.40403026 56.12350246, alpha, lamda):
0.78488539]
10864 [ -50.18628968 -1487.57017669 -1580.68814991 2.7820127 56.3139797 ]
4.40798887]
10865 [ -50.18928968 -1487.66404286 -1580.78099382 1.7918572 55.73429199
5.45247661]
10866 [ -50.19028968 -1487.69448744 -1580.80213304 2.06982364 55.61719486
7.81943088]
10867 [ -50.19328968 -1487.76663039 -1580.91904375 3.1291296 56.03309786
5.02662073]
10868 [ -50.19628968 -1487.86721884 -1581.00701802 2.48412901 54.7768604
6.88799219]
10869 [ -50.19928968 -1487.9576871 -1581.10371089 1.9179399 54.08841254
7.55408789]
10870 [ -50.20028968 -1487.99231076 -1581.18173559 0.7191421 51.38691213
1.4662352 ]
10871 [ -50.20028968 -1487.99231076 -1581.18173559 0.7191421 51.38691213
1.4662352 ]
10872 [ -50.20028968 -1487.99231076 -1581.18173559 0.7191421 51.38691213
1.4662352 ]
10873 [ -50.20028968 -1487.99231076 -1581.18173559 0.7191421 51.38691213
1.4662352 ]
10874 [ -50.20028968 -1487.99231076 -1581.18173559 0.7191421 51.38691213
1.4662352 ]
```

```
6261 [ -42.64589266 -1416.61455416 -1401.19890208 4.53431027 49.52681259
1.95847178]
6262 [ -42.64717761 -1416.64280187 -1401.21656822 4.8636724 51.71530602
4.08609934]
6263 [ -42.64904554 -1416.64378141 -1401.28018378 10.74159935 54.34138345
4.55963124]
6264 [ -42.65334977 -1416.82822531 -1401.42107077 3.38648302 50.29619762
4.55362734]
6265 [ -42.65444433 -1416.8431369 -1401.45838048 5.42174756 51.54166251
3.46387673]
6266 [ -42.65631226 -1416.84411645 -1401.52199604 11.29967383 54.16773952
3.93740837]
6267 [ -42.66165343 -1417.08426957 -1401.71486834 0.4890782 47.9144187
0.77858603]
6268 [ -42.66165343 -1417.08426957 -1401.71486834 0.4890782 47.9144187
0.77858603]
6269 [ -42.66165343 -1417.08426957 -1401.71486834 0.4890782 47.9144187
0.77858603]
6270 [ -42.66165343 -1417.08426957 -1401.71486834 0.4890782 47.9144187
0.77858603]
6271 [ -42.66165343 -1417.08426957 -1401.71486834 0.4890782 47.9144187
0.77858603]
6272 [ -42.66165343 -1417.08426957 -1401.71486834 0.4890782 47.9144187
0.77858603]
```



L-BFGS

- ❑ BFGS 需要存储 $n \times n$ 的方阵 C_k 用于近似 Hessian 阵的逆矩阵；而 L-BFGS 仅需要存储最近 m (m 约为 10, $m=20$ 足够) 个 $(\Delta x, \Delta(\nabla f(x)))$ 用于近似 C_k 即可。
- ❑ L-BFGS 的空间复杂度 $O(mn)$ ，若将 m 看做常数，则为线性，特别适用于变量非常多的优化问题。

总结与思考

□ 梯度下降算法是最重要的参数优化方法，尤其是随机梯度下降SGD，适用于在线环境，用途广泛。

■ 思考：SGD的优势有哪些？

□ 拟牛顿可以给出更好的下降方向，且没有牛顿法的异常情况，值得在实践中考虑使用。

■ 最重要的拟牛顿方法：BFGS

参考文献

- Prof. Andrew Ng. *Machine Learning*. Stanford University
- 李航, 统计学习方法, 清华大学出版社, 2012
- Stephen Boyd, Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004
- 中译本: 王书宁, 许鋆, 黄晓霖, 凸优化, 清华大学出版社, 2013

我们在这里

□ <http://wenda.ChinaHadoop.cn>

■ 视频/课程/社区

□ 微博

■ @ChinaHadoop

■ @邹博_机器学习

□ 微信公众号

■ 小象

■ 大数据分析挖掘



课程资源

- 直播课的入口
- 录播视频和讲义资料



搜索课程 搜索

首页 选课中心 小象问答 机器学习实训营 小象训练营 小象公开课

机器学习

算法推导+代码实现+参数调试+应用场景

开课时间：5月23日
主讲人：邹博•••

我要参团



《机器学习》第三期

★★★★★ (0评价)

承诺服务

试 问 疑 练 动

介绍 课程(2) 评价 话题 笔记



《机器学习算法基础》每周直播课

★★★★★



《机器学习》三期录屏回放与资料

★★★★★

感谢大家！

恳请大家批评指正！