《计算机模拟》

第11讲 –基于神经网络的模拟

胡贤良

浙江大学数学科学学院

梯度下降法 – 基本思想

通过多项式对函数进行逼近

$$f(x) = \frac{f(x_0)}{0!} + \frac{f'(x_0)}{1!}(x-x_0)^1 + \frac{f''(x_0)}{2!}(x-x_0)^2 + \cdots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n + R_n(x)$$

其中$R_n(x)$是余项

$$f(x+\Delta x) \simeq f(x) + \Delta x \nabla f(x)$$

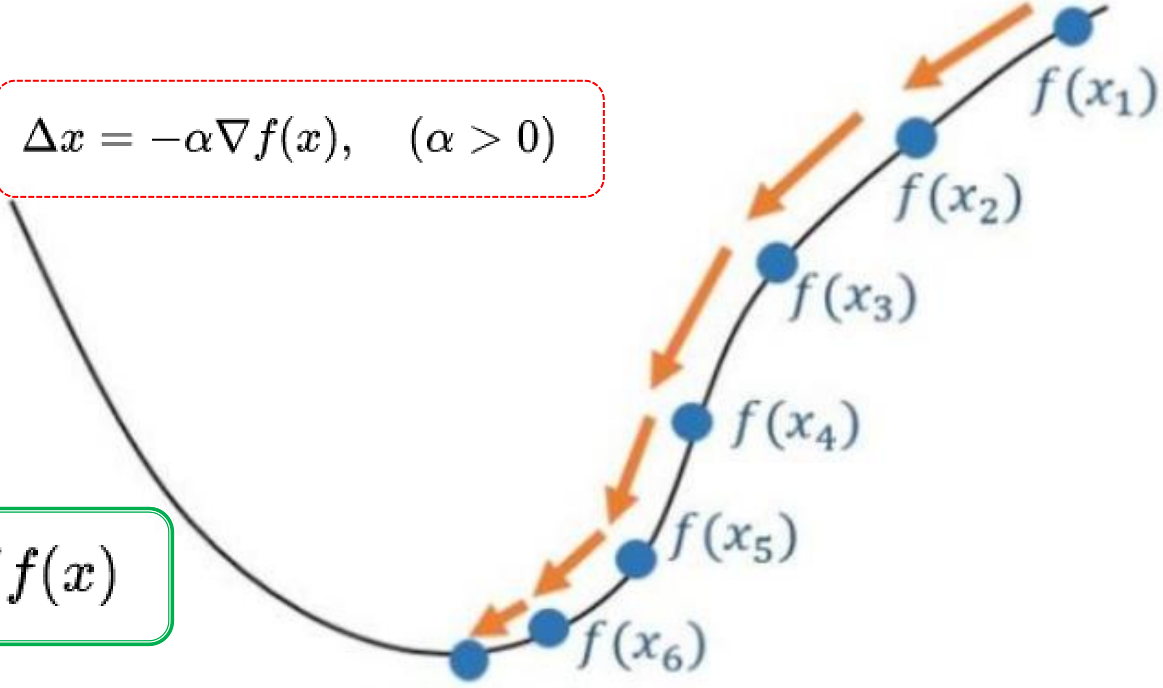
$$\Delta x = -\alpha \nabla f(x), \quad (\alpha > 0)$$

故

$$\Delta x = -\alpha \nabla f(x), \quad (\alpha > 0)$$

得梯度下降法:

$$\boxed{x_{i+1} = x_i - \alpha \nabla f(x)}$$



示例1：一元函数极小值

设一元函数为

$$J(\theta) = \theta^2$$

函数的微分为

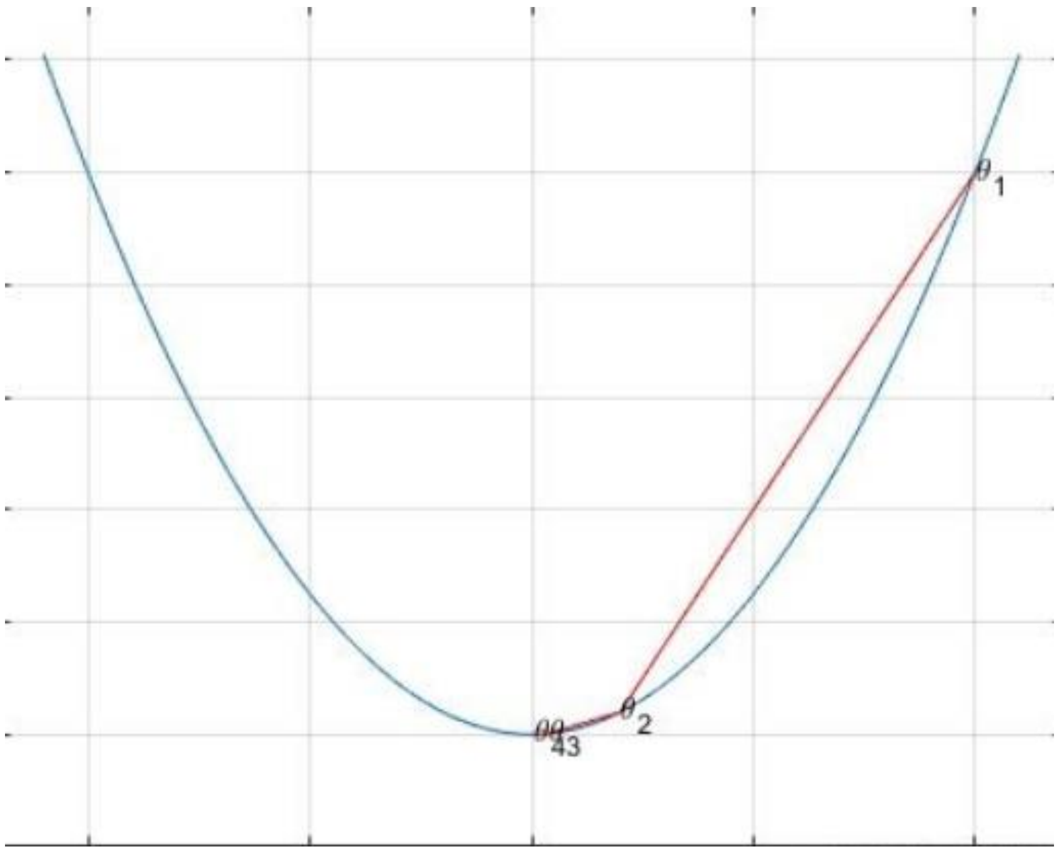
$$J'(\theta) = 2\theta$$

设起点为 $J(\theta) = \theta^2$ ，步长 $alpha = 0.4$,根据梯度下降的公式

$$\theta_1 = \theta_0 - \alpha \nabla J(\theta)$$

经过4次迭代：

$$\theta_0 = 1$$
$$\theta_1 = \theta_0 - 0.4 * 2 * 1 = 0.2$$
$$\theta_2 = \theta_1 - 0.4 * 2 * 0.2 = 0.04$$
$$\theta_3 = \theta_2 - 0.4 * 2 * 0.04 = 0.008$$



示例2：二元函数极小值

设二元函数为

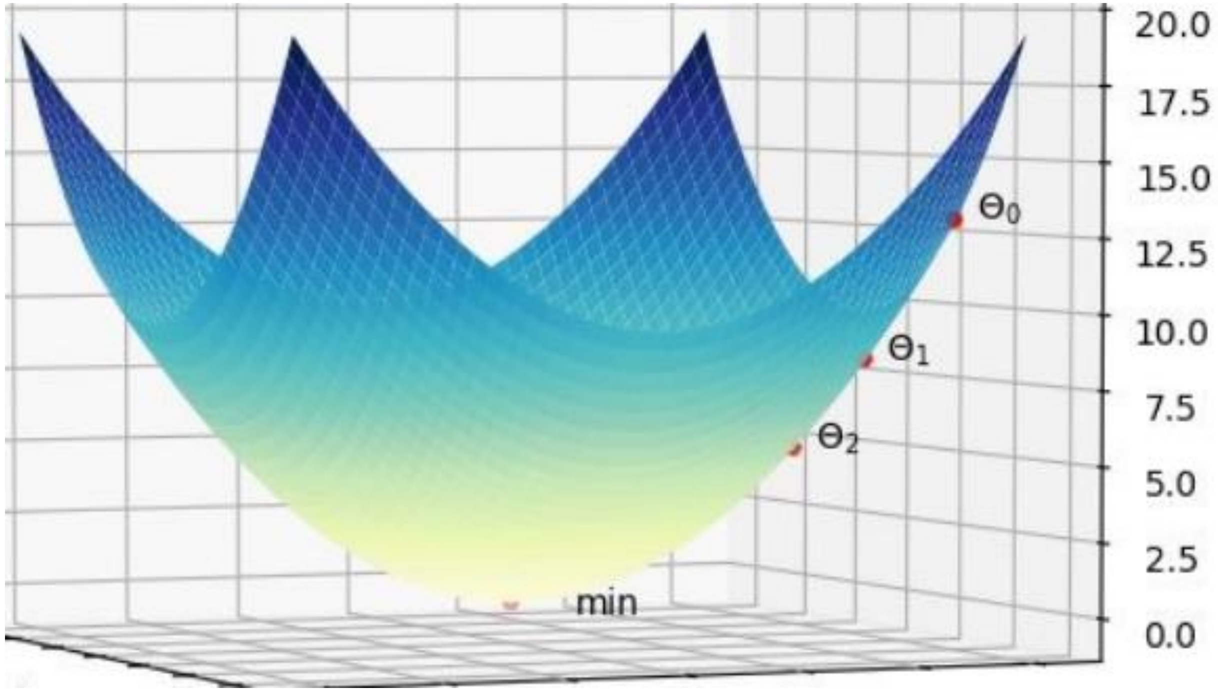
$$J(\Theta) = \theta_1^2 + \theta_2^2$$

函数的梯度为

$$\nabla J(\Theta) = (2\theta_1, 2\theta_2)$$

设起点为(2,3)，步长 $\alpha = 0.1$,根据梯度下降的公式,经过多次迭代后，

$$\Theta_0 = (2,3)$$
$$\Theta_1 = \Theta_0 - 0.1 * (2*2, 2*3) = (1.6, 2.4)$$
$$\Theta_2 = \Theta_1 - 0.1 * (2*1.6, 2*2.4) = (1.28, 1.92)$$
$$\vdots$$
$$\Theta_{99} = \Theta_{98} - 0.1 * \Theta_{98} = (6.36e-10, 9.55e-10)$$
$$\Theta_{100} = \Theta_{99} - 0.1 * \Theta_{99} = (5.09e-10, 7.64e-10)$$



梯度下降法： 简短的Python实现

$$\theta_{n+1} = \theta_n - \eta \nabla_\theta f(\theta_n)$$

```
import numpy as np
import matplotlib.pyplot as plt

def cost_function(theta, X, y):
    diff = np.dot(X, theta) - y
    return (1./2*m) * np.dot(np.transpose(diff), diff)

def gradient_function(theta, X, y):
    diff = np.dot(X, theta) - y
    return (1./m) * np.dot(np.transpose(X), diff)

def gradient_descent(X, y, eta):
    theta = np.array([1, 1]).reshape(2, 1)
    gradient = gradient_function(theta, X, y)
    while not np.all(np.absolute(gradient) <= 1e-5):
        theta = theta - eta* gradient
        gradient = gradient_function(theta, X, y)
    return theta
```

```
m = 18    # sample length
X0 = np.ones((m, 1))
X1 = np.arange(1, m+1).reshape(m, 1)
X = np.hstack((X0, X1))
# matrix y
y = np.array([2,3,3,5,8,10,10,13,15,15,16,
19,19,20,22,22,25,28])
y = y.reshape(m,1)
eta = 0.01
[theta0, theta1] = gradient_descent(X, y, eta)
plt.figure()
plt.scatter(x,y)
plt.scatter(X1,y)
plt.plot(X1, theta0 + theta1*X1, color='r')
plt.title('基于梯度下降算法的线性回归拟合')
plt.grid(True)
plt.show()
```

案例1：最小二乘拟合

参考SGA-lsFit-gd目录下matlab代码

进一步我们来讨论：已知二维数据 (x_i, y_i), $i=1,2,\cdots,m$. 我们希望寻找：

$$y=f(x)=\sum_{j=1}^{n} w_j\varphi_j(x),$$

其中 $w_j\in\mathbb{R}$ 是待定系数，$\varphi_j(x):\mathbb{R}^2\to\mathbb{R}$ 是一组基底函数（它未必是基函数，只是用于构建 y 的基础材料），$j=1,2,\cdots,n$. 这里一般有 $m\gg n$.

虽然此问题从线性方程组的角度看是超定的，不存在解，但我们可以转而求优化问题：

$$\min_{w_j} R := \|y-\hat{y}\|_2^2$$

其中 $\hat{y}=(y_1,y_2,\cdots,y_m)^T$. $y=(f(x_1),f(x_2),\cdots,f(x_m))^T$.

即

$$\min_{w_j} R(w)=\frac{1}{2}\sum_{i=1}^{m}\Big[\sum_{j=1}^{n} w_j\varphi_j(x_i)-y_i\Big]^2 \quad (\text{为何}\tfrac{1}{2}\downarrow)$$

其中 $w=(w_1,w_2,\cdots,w_n)^T$

不论线性还是非线性，我们都可以用梯度下降法来求解：

$$(*)\begin{cases}\nabla R_k=\left(\dfrac{\partial R}{\partial w_j}\right)^T=\left(\displaystyle\sum_{i=1}^{m}\Big[\sum_{j=1}^{n} w_j\varphi_j(x_i)-y_i\Big]\cdot\varphi_j(x_i)\right)^T_{j=1,2,\cdots,n}\Bigg|_{w=w_k}\\ w_{k+1}=w_k+\alpha\nabla R_k\end{cases}$$

这个方法在机器学习等领域被称为批量梯度下降法（Batch Gradient Descent, BGD). 它的实际效果不错，但问题是它每一步迭代，都要用到全部拟合数据 (x_i, y_i)，而在机器学习等领域中，共同点是：

i) 数据量极大，m 可能是百万级甚至更大；

ii) x_i 是高维向量，维数可能是上万维甚至更高；

iii) y_i 不准确，y_i 中混杂了各种噪声数据，只有概率意义上的

2. 批梯度下降法(BGD)求极小

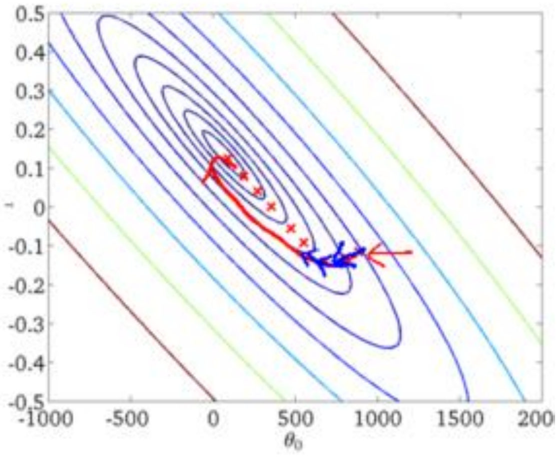
- 批梯度下降法(Batch Gradient Descent)针对的是整个数据集，通过对**所有的样本**的计算来求解梯度的方向。
- 每迭代一步，都要用到训练集所有的数据，如果样本数目很大，迭代速度就会很慢。优点是迭代次数较少
- 缺点：每更新一个参数的时候，要用到所有的样本，训练速度会随着样本数量的增加而变得非常缓慢。

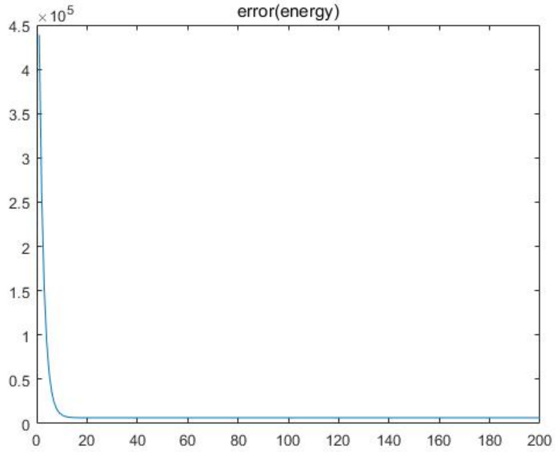
repeat{

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

(for every j=0, ... , n)

}





3. 改进 - 小批量(mini batch)梯度下降法

综合批梯度下降和随机梯度下降，提出小批量随机梯度下降。可以减小随机梯度下降的方差同时不至于使参数更新过慢。

- 迭代公式：

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

- 伪代码：

尝试基于前几页的代码实现

Repeat{

for i=1, 11, 21, 31, ... , 991{

$$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$$

(for every j=0, ... , n)

}

}

4. 随机梯度下降法(Stochastic GD)

- SGD 利用**每个/逐个**样本的损失函数对θ求偏导得到对应的梯度来更新θ。迭代公式:

$$\theta = \theta - \eta \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

- 缺点：由于频繁更新数据，方差较大，损失函数会产生波动。在解空间中体现为搜索较为盲目。但大体上仍向最优解方向移动

1. Randomly shuffle dataset ;

2. **repeat**{

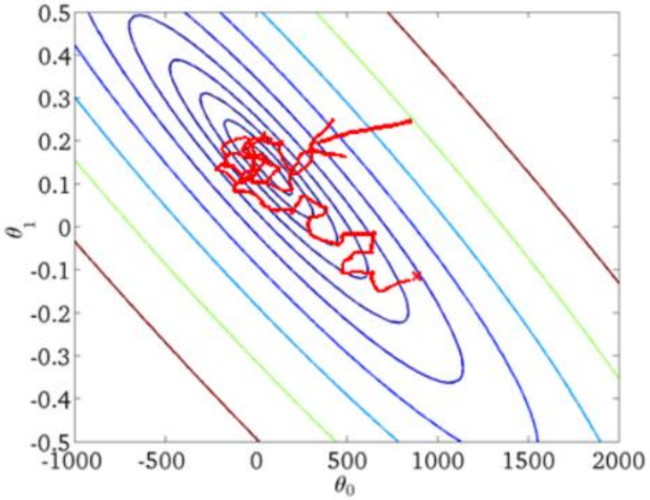
    for i=1, ... , m{

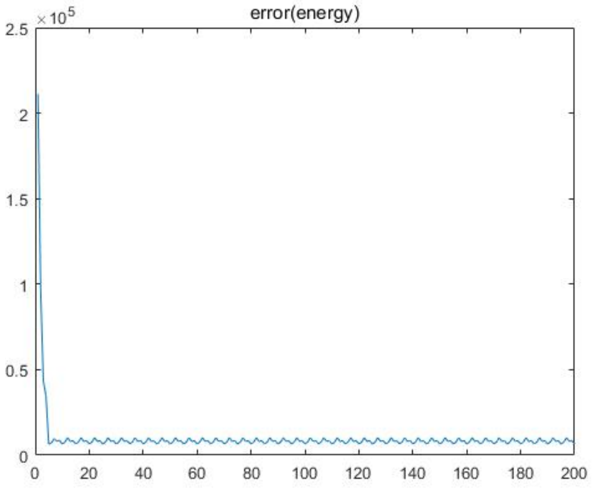
    $$\theta_j' = \theta_j + (y^i - h_\theta(x^i))x_j^i$$

    (for j=0, ... , n)

    }

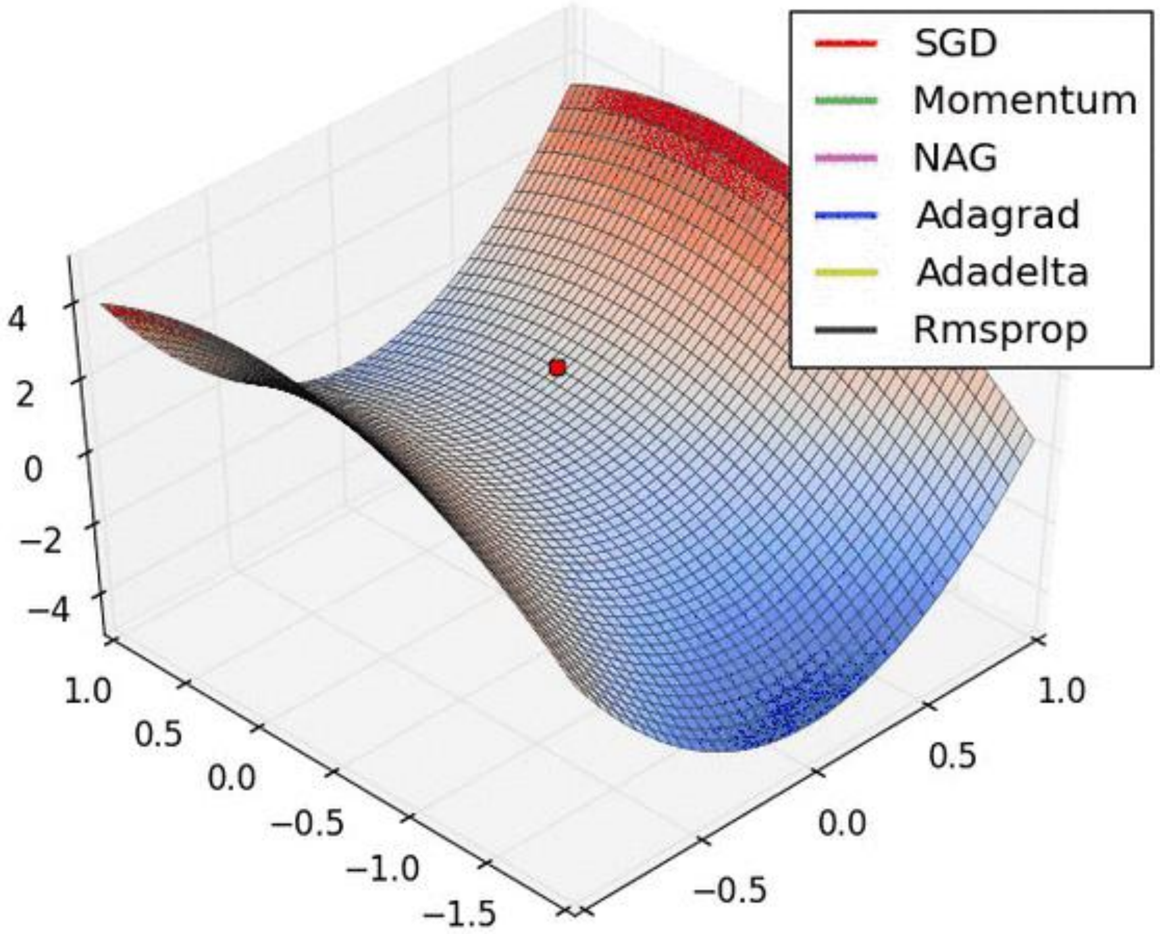
}





梯度下降法的推广

1. Momentum
2. Nesterov accelerated gradient
3. Adagrad
4. Adadelta
5. RMSprop
6. Adam
7. AdaMax
8. Nadam



参考论文：An overview of gradient descent optimization algorithms
地址：https://arxiv.org/pdf/1609.04747.pdf

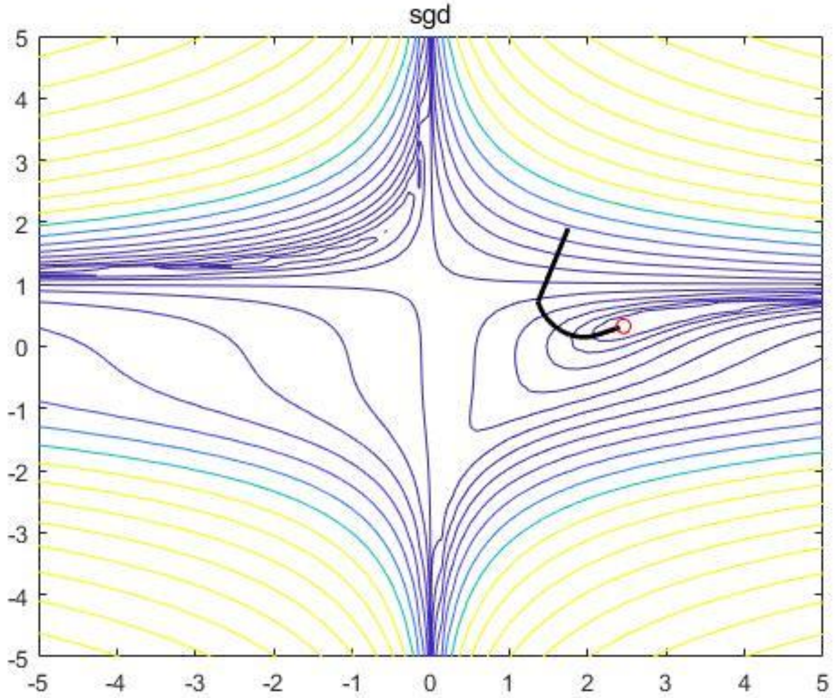
案例：给定表达式求极小点

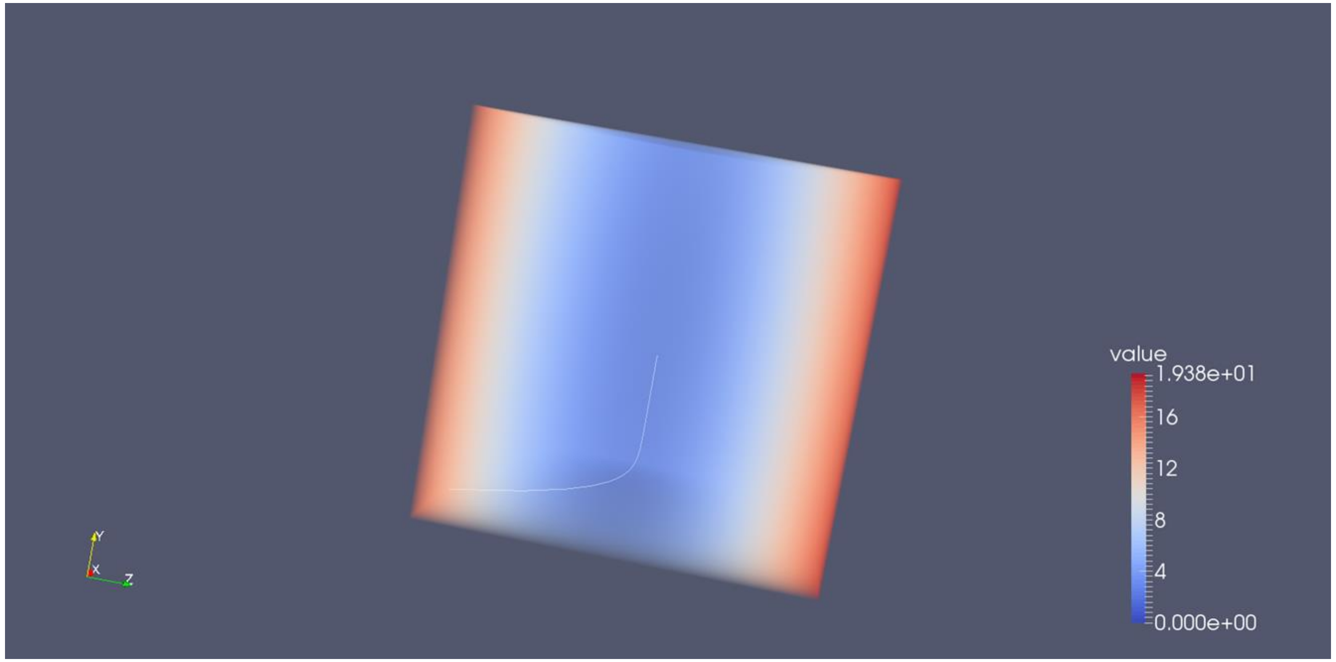
二维函数表达式：$f(x,y) = (1.5 - xy)^2 + \left(2.25 - x + xy^2\right)^2 + \left(2.65 - x + xy^3\right)^3$
三维函数表达式：$f(x,y,z) = x^2 + 0.1y^2 + 2z^2$

参数设置：

二维：η = 0.002，起始点 [1.75 , 1.9]

三维：η = 0.1，起始点 [-2,-2,-2]





- 代码参考：**gradient descent_exe2d下的sgd.m**

1. Momentum

论文：Learning representations by back-propagating errors
地址：https://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf

- Momentum是对梯度下降法的改进，使用了过去的梯度信息。Momentum可以在不牺牲易用性和局部性的前提下显著加速收敛。
- 迭代公式：

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

$$\theta = \theta - v_t$$

- 参数γ通常设为0.9， η为学习率

伪代码：

f = f(x_1, x_2,⋯, x_n)

g = gradient(f)

初始化：起始点x， 学习率η， x点负梯度d,γ

for n from 1 to N:

    d = γd- η g(x)

    x = x+d

end

算例展示

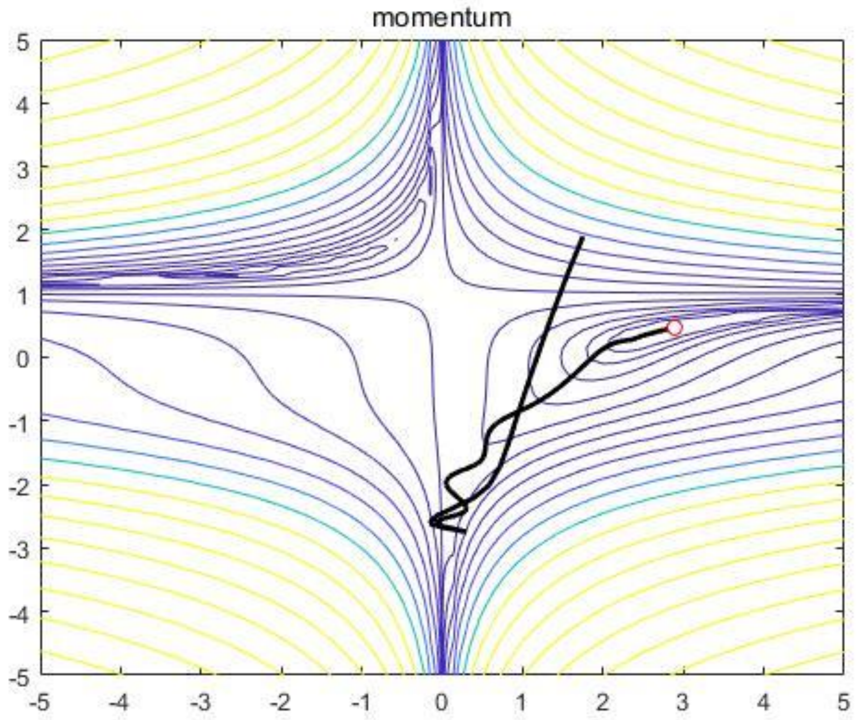
二维函数表达式： $f(x,y) = (1.5 - xy)^2 + \left(2.25 - x + xy^2\right)^2 + \left(2.65 - x + xy^3\right)^3$

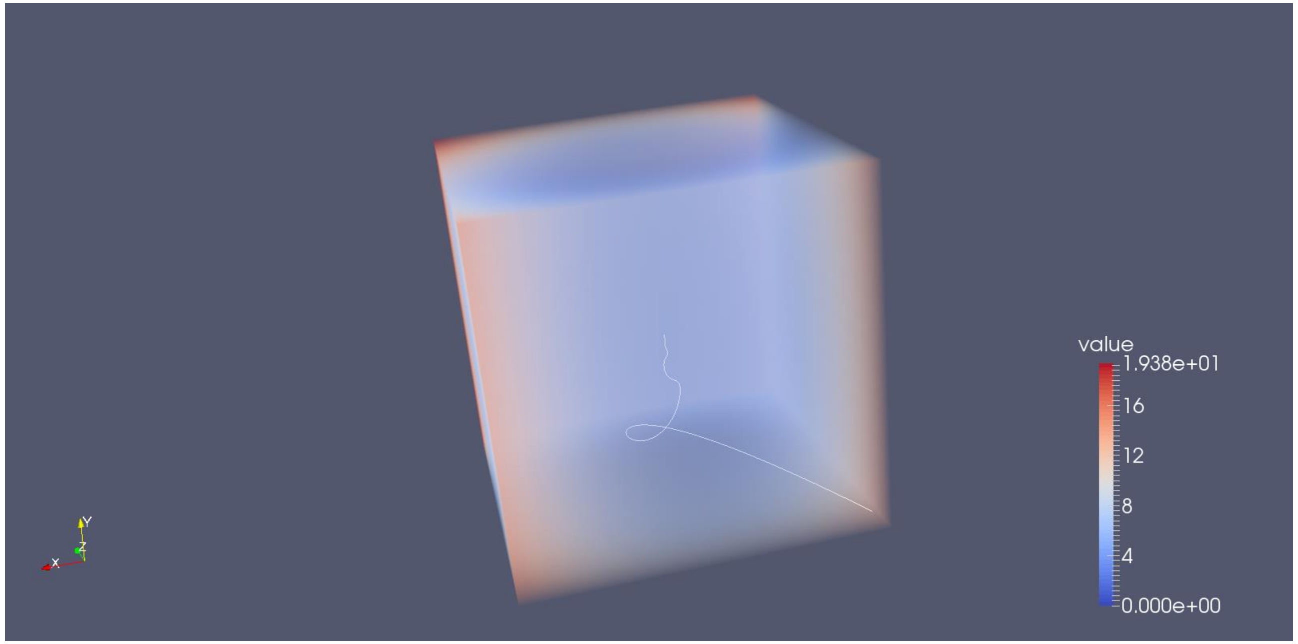
三维函数表达式： $f(x,y,z) = x^2 + 0.1y^2 + 2z^2$

参数设置：

二维： $\eta = 0.001$， γ=0.9，起始点 [1.75 , 1.9]

三维： $\eta = 0.01$ ， γ=0.9，起始点 [-2,-2,-2]





- 代码参考： gradient descent_exe2d

2. Nesterov accelerated gradient

论文：A method of solving a convex programming problem with convergence rate O(1/k^2)
地址： http://mpawankumar.info/teaching/cdt-big-data/nesterov83.pdf

- 基于momentum的改进：每次迭代使用的梯度为未来参数位置的预测值的梯度，能够在函数值上升之前减小梯度。
- 迭代公式(某种预估-校正)：

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

$$\theta=\theta-v_t$$

伪代码：

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

g = gradient(f)

初始化：起始点x，学习率η，x点梯度d, γ

for n from 1 to N:

d = γd- η g(x - γd)

x = x - d

end

return x

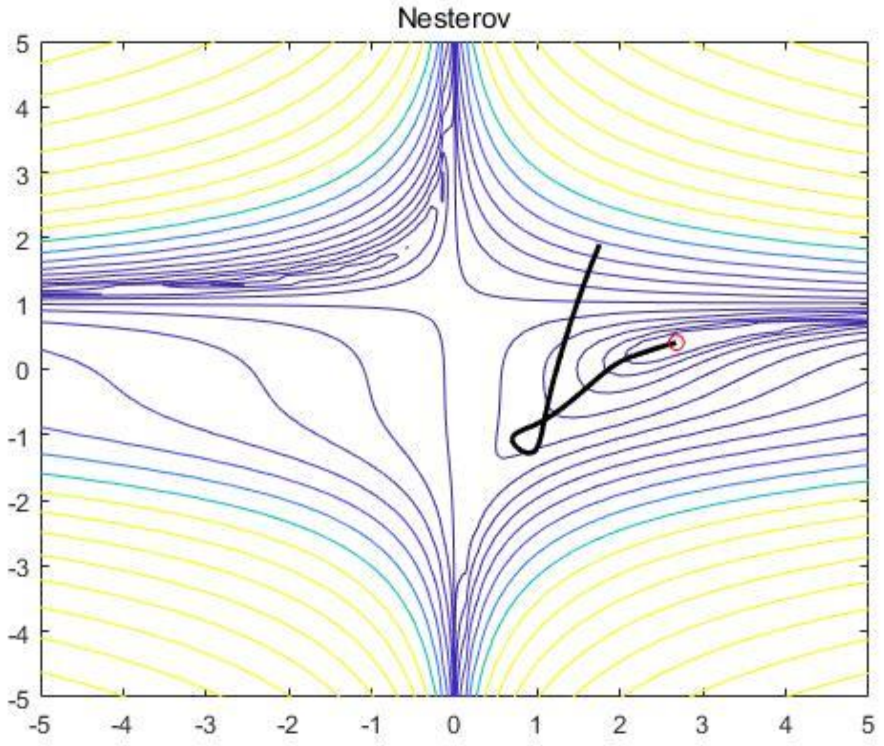
二维函数表达式： $f(x,y) = (1.5 - xy)^2 + \left(2.25 - x + xy^2\right)^2 + \left(2.65 - x + xy^3\right)^3$

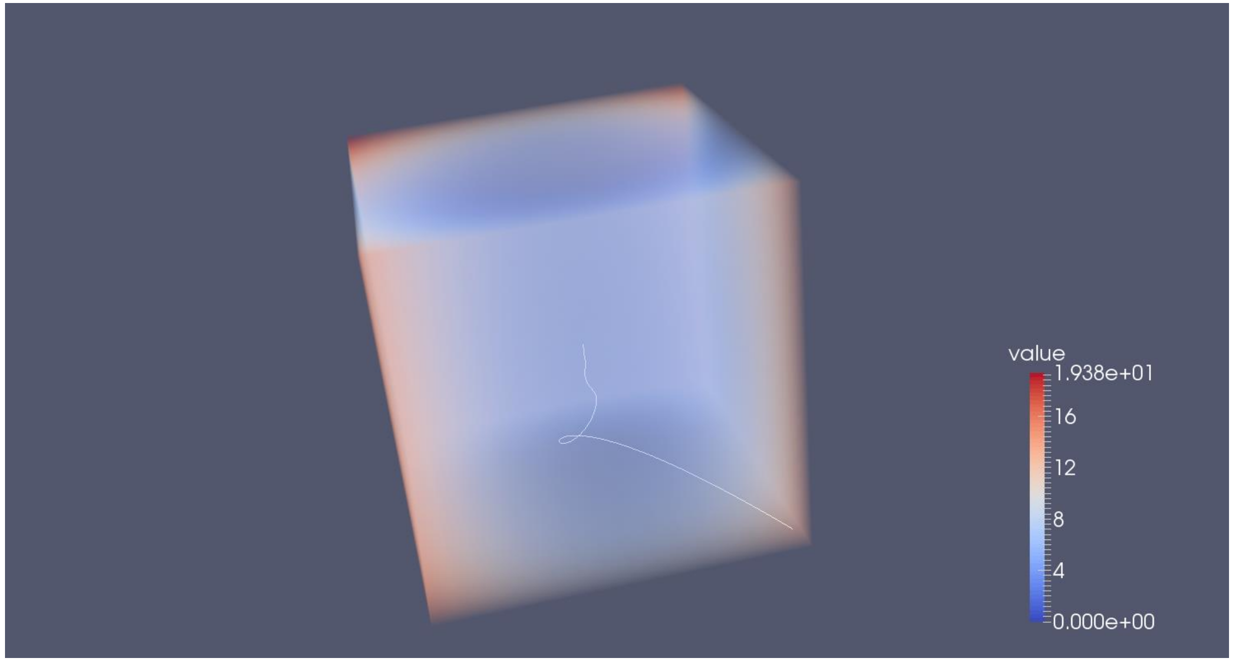
三维函数表达式： $f(x,y,z) = x^2 + 0.1y^2 + 2z^2$

参数设置：

二维： $\eta = 0.0005$， γ=0.9，起始点 [1.75 , 1.9]

三维： $\eta = 0.01$ ， γ=0.9，起始点 [-2,-2,-2]





3. Adagrad

- Adagrad实现了根据参数调整学习率。
- 迭代公式： $g_{t,i} = \nabla_\theta J(\theta_{t,i})$

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

- Adagrad的缺点在于分母上平方梯度的增加，导致学习速率下降。

论文： Adaptive Subgradient Methods for Online Learning and Stochastic Optimization
地址： http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf

伪代码：

f = f(x_1, x_2,⋯, x_n)

g = gradient(f)

初始化：起始点x，学习率η，G，ε

for n from 1 to N:

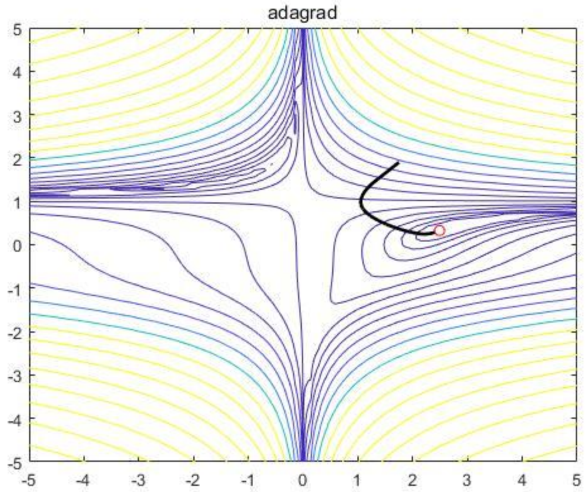
d = -g(x)

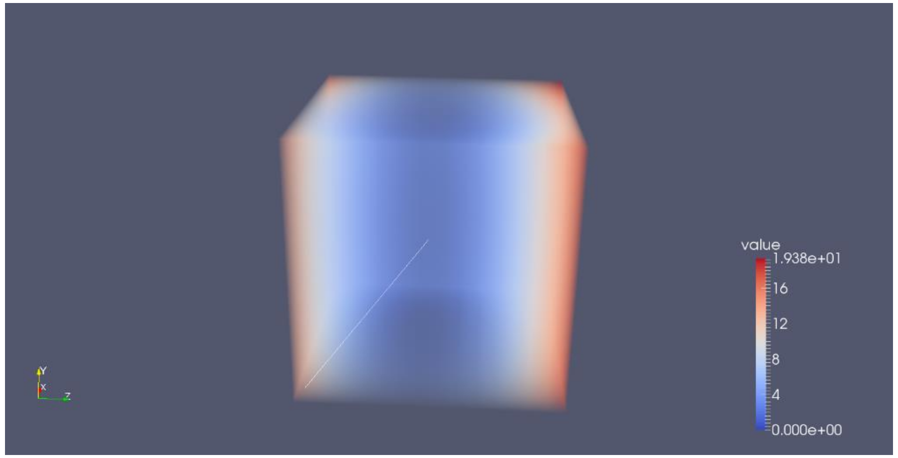
G += d^2

x = x+$\frac{\eta}{\sqrt{G+\varepsilon}}$d

end

return x





参数设置：
二维：
η 0.5
ε 1e-6
起始点 [1.75 , 1.9]
三维：
η 0.5
ε 1e-6
起始点 [-2,-2,-2]

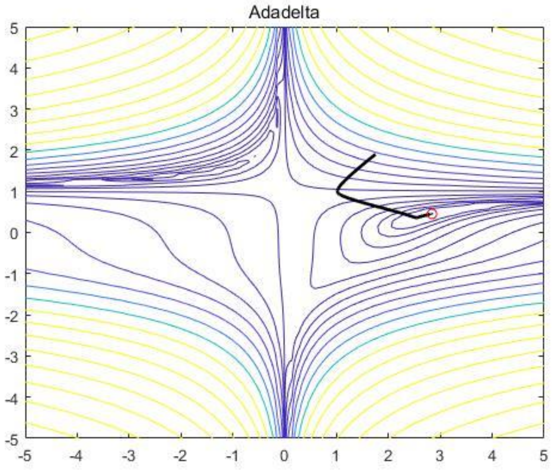
4. Adadelta

论文：ADADELTA: An Adaptive Learning Rate Method
地址： https://arxiv.org/pdf/1212.5701.pdf

- Adadelta是Adagrad的扩展，对Adagrad两处改进:
  1. 对梯度累计方式进行改进，使学习率不会趋于0
  2. 采用二阶优化的思想进行参数更新
- 迭代公式:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$
$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma) \Delta\theta_t^2$$

with $\Delta\theta_t = -\frac{\sqrt{E[\Delta\theta^2]_{t-1}+\varepsilon}}{\sqrt{E[g^2]_t+\varepsilon}} g_t$ and $\theta_{t+1} = \theta_t + \Delta\theta_t$



二维:
- γ 0.9
- ε 1e-6
- 起始点 [1.75 , 1.9]

三维:
- γ 0.9
- ε 1e-5
- 起始点 [-2,-2,-2]

伪代码:

f = f(x_1, x_2,⋯, x_n)

g = gradient(f)

初始化：起始点x，$E[g^2]_t$，$E[\Delta\theta^2]_t$，γ，ε

for n from 1 to N:

  d = -g(x)

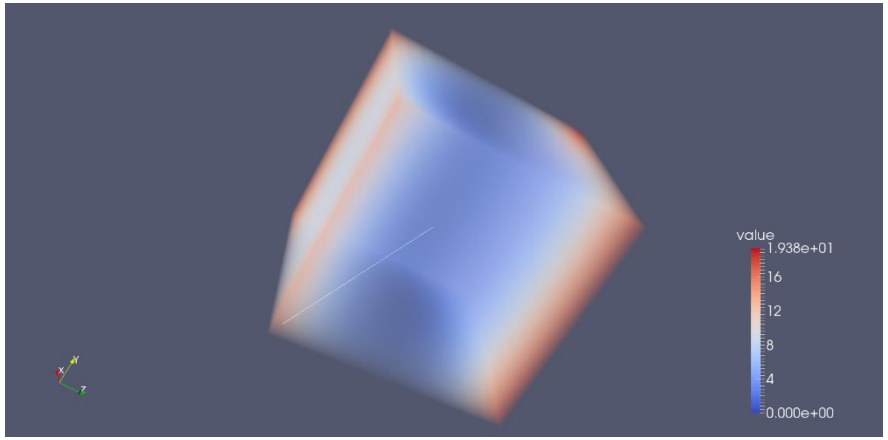
  $E[g^2]_t = \gamma E[g^2]_t + (1-\gamma) d^2$

  x = x+$\frac{\sqrt{E[\Delta\theta^2]_t+\varepsilon}}{\sqrt{E[g^2]_t+\varepsilon}}$d

  $E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_t + (1-\gamma) \left(\frac{\sqrt{E[\Delta\theta^2]_t+\varepsilon}}{\sqrt{E[g^2]_t+\varepsilon}} d\right)^2$

end

return x



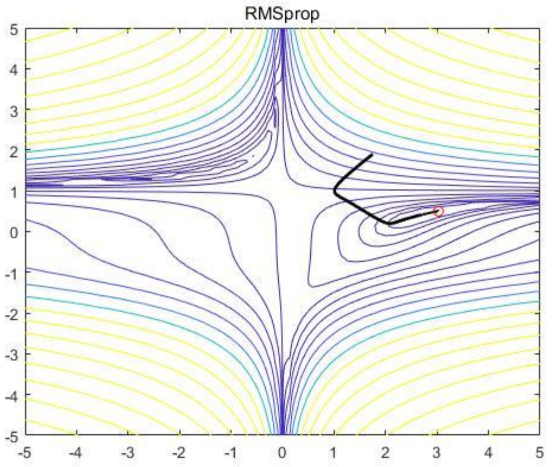
5. RMSprop

http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

RMSprop是与Adadelta同时提出的Adagrad改进算法，未以论文的形式进行发表，而是在Geoff Hinton 教授的课程中被提及，结合梯度平方的指数移动平均数来调节学习率的变化。能够在不稳定的目标函数情况下进行很好地收敛。迭代公式：

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t$$



二维：

η 0.01

ε 1e-8

γ 0.9

起始点 [1.75 , 1.9]

三维：

η 0.01

ε 1e-8

γ 0.9

起始点 [-2,-2,-2]

f = f(x_1, x_2,⋯, x_n)

g = gradient(f)

初始化：起始点x，学习率η，$E[g^2]_t$，γ，ε

for n from 1 to N:

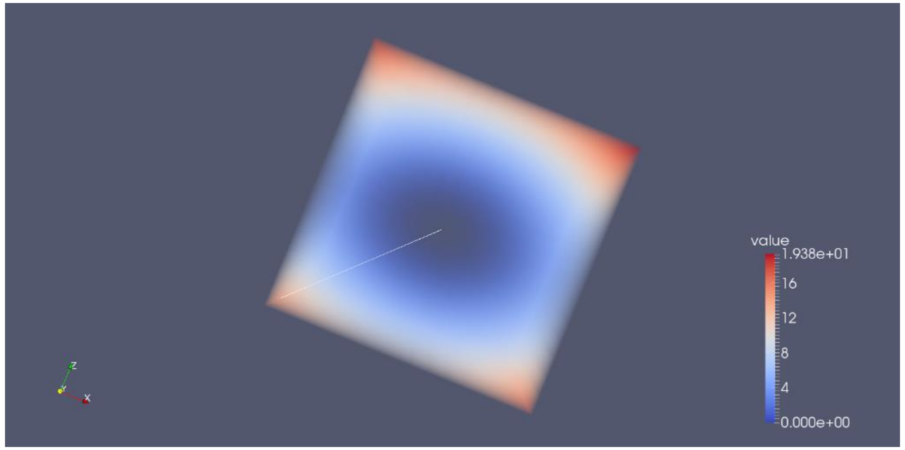
  d = -g(x)

  $E[g^2]_t = \gamma E[g^2]_t + (1-\gamma) d^2$

  x = x+$\frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}}$d

end

return x



6. Adam

论文： Adam: A Method for Stochastic Optimization
地址： https://arxiv.org/pdf/1412.6980.pdf

2014年，Kingma和Lei Ba提出了Adam优化器，结合AdaGrad和RMSProp两种优化算法的优点。对梯度的一阶矩估计（First Moment Estimation，梯度均值）和二阶矩估计（Second Moment Estimation，梯度未中心化的方差）进行综合考虑，计算出更新步长：

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

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

with $\hat{m_t} = \frac{m_t}{1-\beta_2^n}, \hat{v_t} = \frac{v_t}{1-\beta_1^n}$ and $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v_t}+\varepsilon}} \hat{m_t}$

f = f(x_1, x_2,⋯, x_n)

g = gradient(f)

初始化：x，η，β_1，β_2，m_t，v_t

for n from 1 to N:

$$m_t = \beta_1 m_t + (1-\beta_1) g(x)$$

$$v_t = \beta_2 v_t + (1-\beta_2) g(x)^2$$

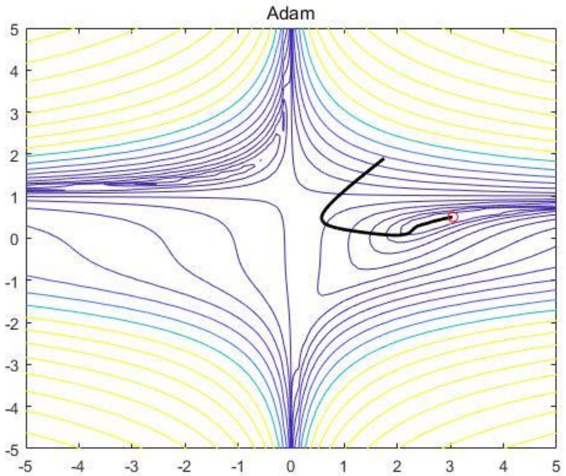
$$\hat{m_t} = \frac{m_t}{1-\beta_2^n}$$

$$\hat{v_t} = \frac{v_t}{1-\beta_1^n}$$

$$x = x - \frac{\eta}{\sqrt{\hat{v_t}+\varepsilon}} \hat{m_t}$$

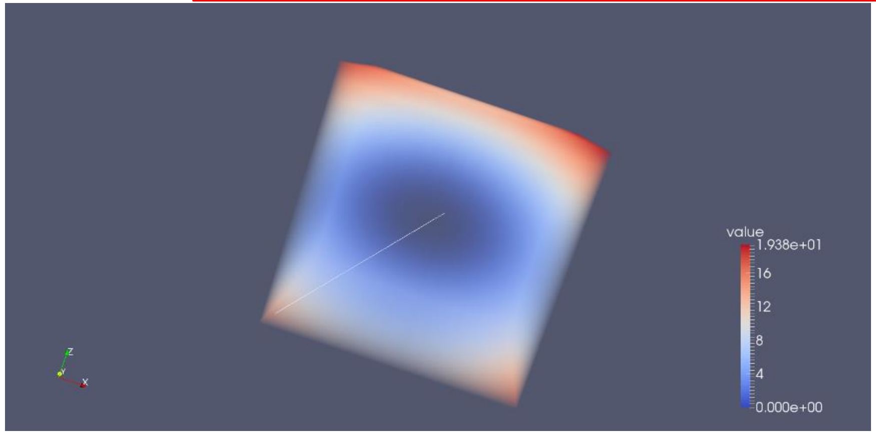
end

return x



二维：
β_1 0.9， β_2 0.97
ε 1e-6
起始点 [1.75 , 1.9]

三维：
β_1 0.9， β_2 0.999
ε 1e-6
起始点 [-2,-2,-2]



7. AdaMax

论文： Adam: A Method for Stochastic Optimization
地址： https://arxiv.org/pdf/1412.6980.pdf

AdaMax将Adam中将基于 L_2 范数的更新规则泛化到基于 L_p范数的更新规则中。虽然这样会因为 p 的值较大而在数值上变得不稳定，但令 p → ∞则会得出一个极其稳定和简单的算法：

$$v_t = \max(\beta_2 \cdot v_{t-1}, |g_t|)$$

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

f = f(x_1, x_2,⋯, x_n)

g = gradient(f)

初始化： x， η ， β_1， β_2， m_t， v_t

for n from 1 to N:

$$m_t = \beta_1 m_t + (1 - \beta_1) g(x)$$

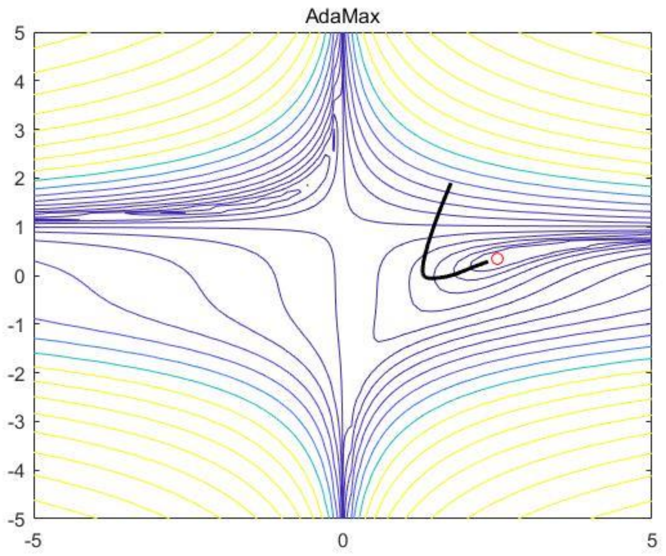
$$v_t = \max(\beta_2 \cdot v_t, |g_t|)$$

$$\hat{m_t} = \frac{m_t}{1-\beta_2^n}$$

$$x = x - \frac{\eta}{v_t} \hat{m_t}$$

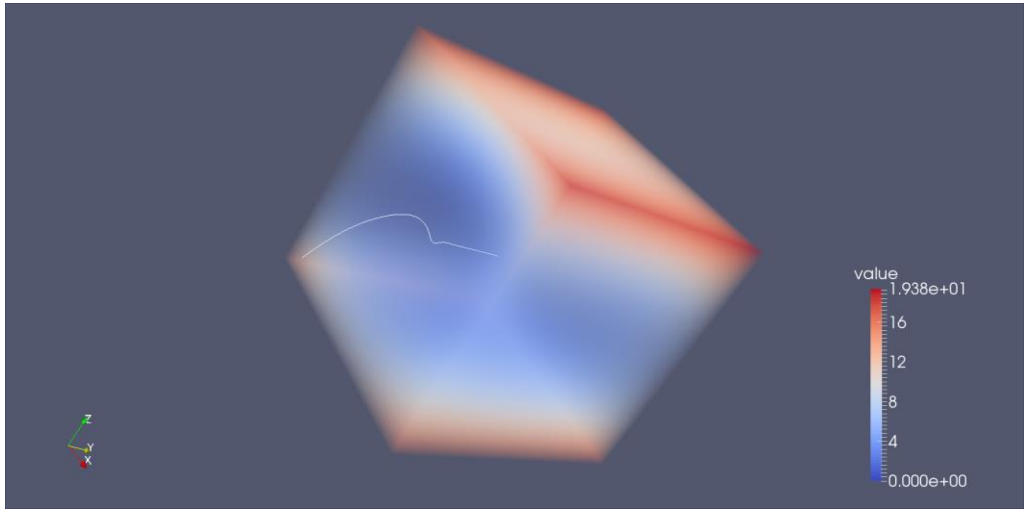
end

return x



二维：
β_1 0.9, β_2 0.999
ε 1e-6, η 0.5
起始点 [1.75 , 1.9]

三维：
β_1 0.9, β_2 0.999
ε 1e-6, η 0.1
起始点 [-2,-2,-2]



8. Nadam

论文：Incorporating Nesterov Momentum into Adam
地址：https://openreview.net/pdf?id=OM0jvwB8jIp57ZJjtNEZ

Nadam将Nesterov加速方法应用在Adam算法上：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v_t} + \varepsilon}}\left(\beta_1 \hat{m_t} + \frac{(1-\beta_1)g_t}{1-\beta_1^t}\right)$$

f = f(x_1, x_2,⋯, x_n),　　g = gradient(f)

初始化：x，η，β_1，β_2，m_t，v_t，γ，ε

for n from 1 to N:

$$m_t = \beta_1 m_t + (1-\beta_1)g(x)$$

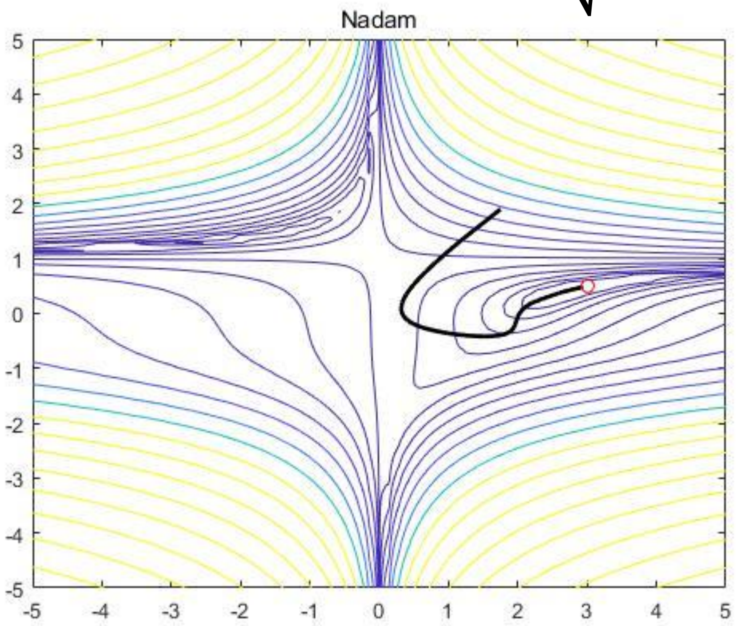
$$v_t = \beta_2 v_t + (1-\beta_2)g(x)^2$$

$$\hat{m_t} = \frac{m_t}{1-\beta_2^n},\quad \hat{v_t} = \frac{v_t}{1-\beta_1^n}$$

$$x = x - \frac{\eta}{\sqrt{\hat{v_t}+\varepsilon}}\left(\beta_1 \hat{m_t} + \frac{(1-\beta_1)g(x)}{1-\beta_1^n}\right)$$

end

return x

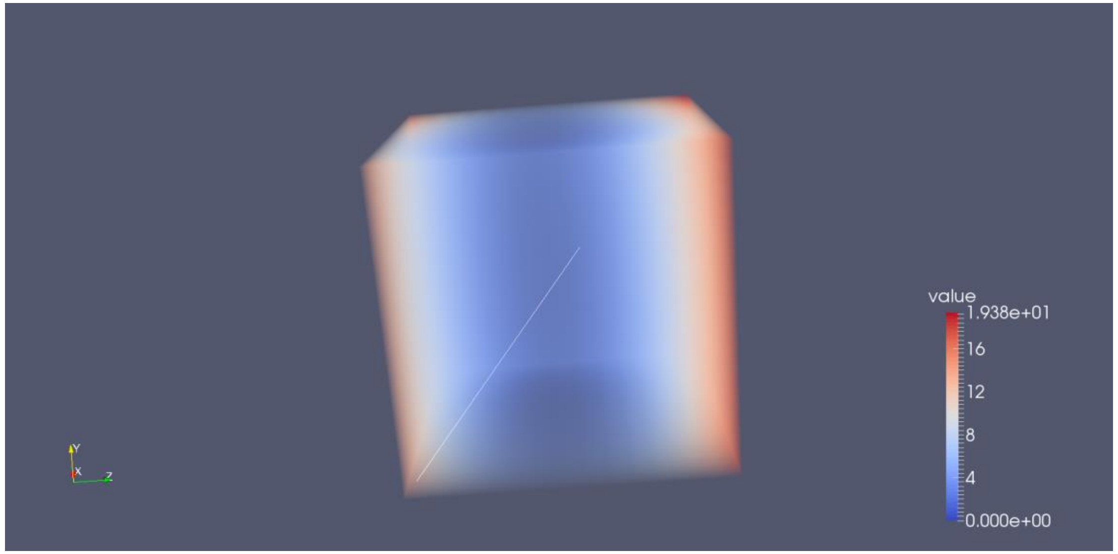


二维：

β_1 0.9，β_2 0.999
ε 1e-6，η 0.3
起始点 [1.75 , 1.9]

三维：

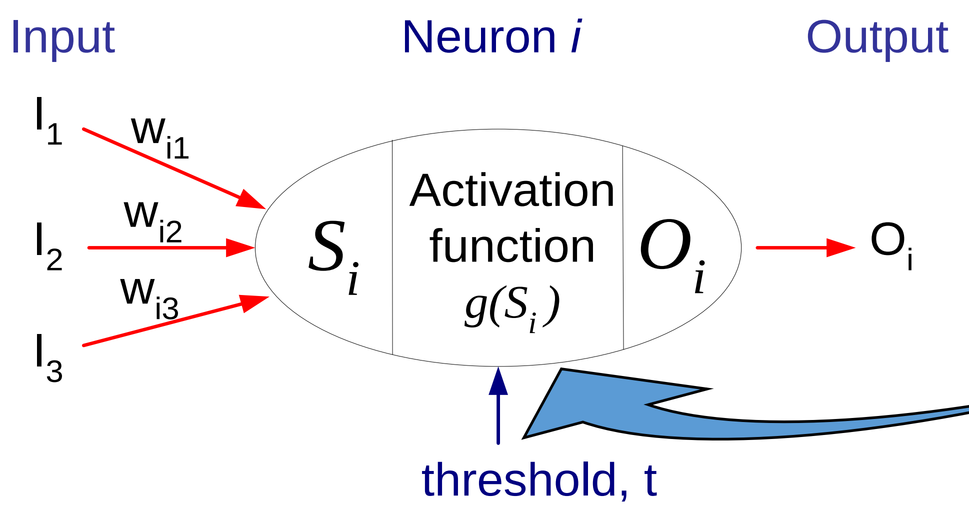
β_1 0.9，β_2 0.999
ε 1e-6，η 0.3
起始点 [-2,-2,-2]

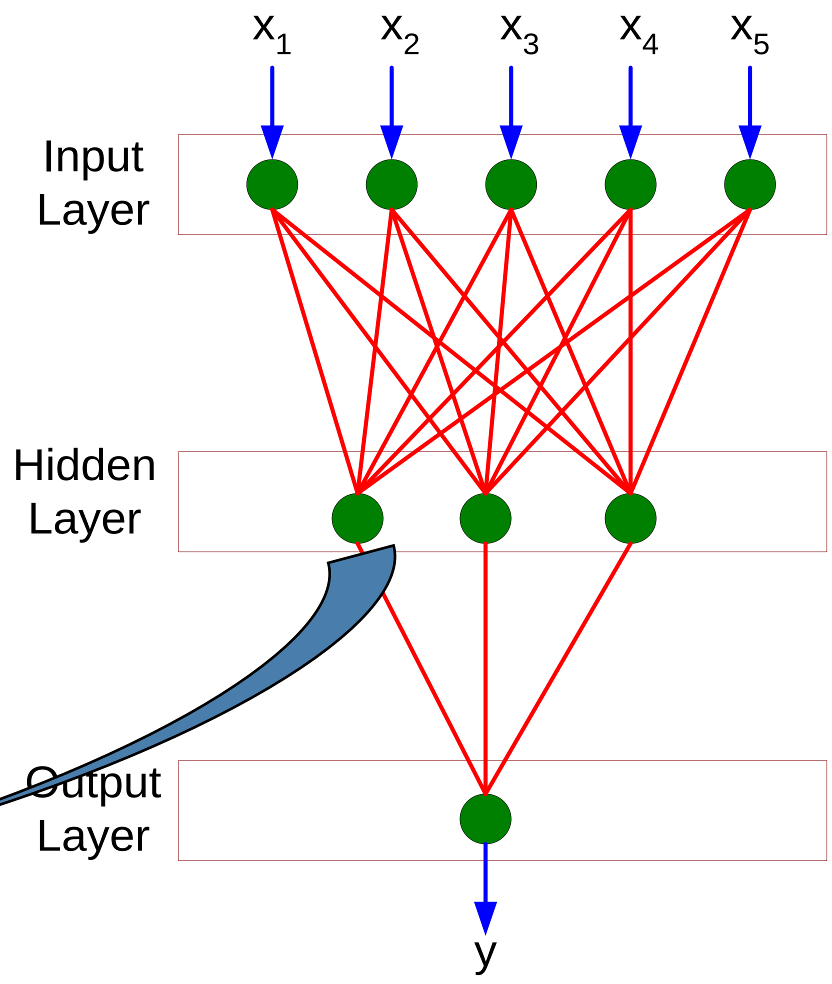


案例3：利用多层神经网络逼近函数

求$f(x) = x_1^2 + x_2^2 + x_3^2$最小值

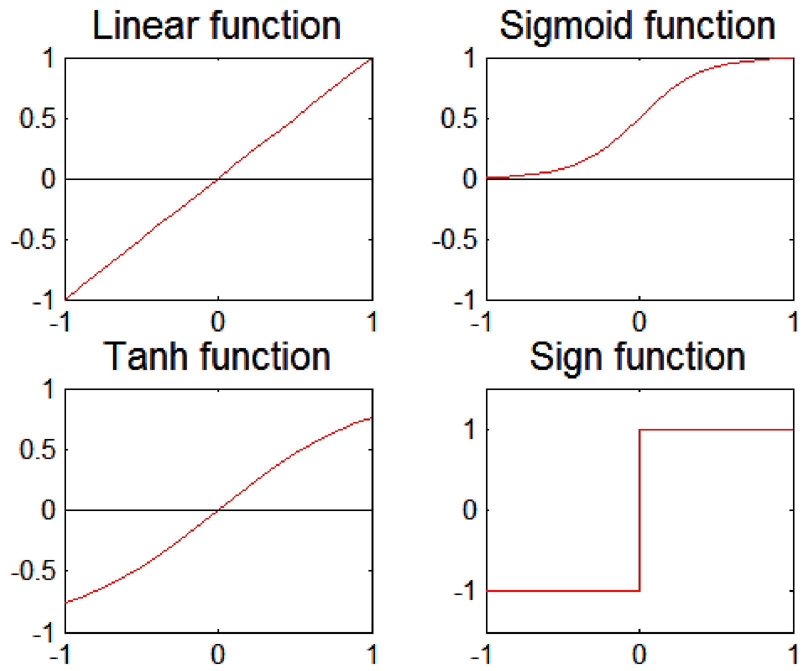
- 四层神经网络，神经元个数为 1，4，2，3
- 输入值为**1**，输出为函数取最小值的自变量值
- 主程序见下一页（完整代码参考**NNminfun/main.m**）





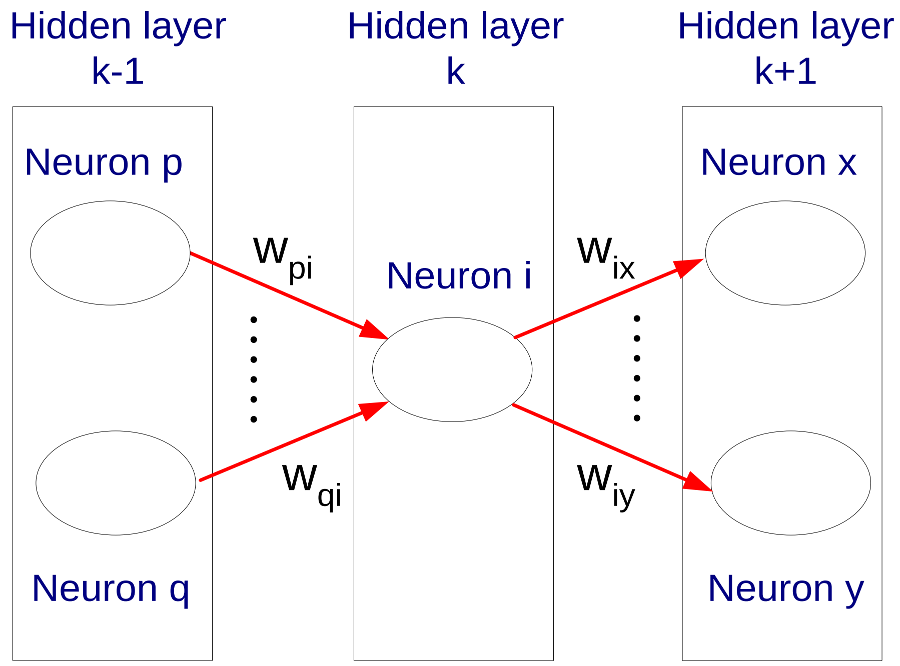
多层神经网络的训练策略

- 误差函数: $E=\frac{1}{2}\sum_{i=1}^{N}\left(y_i-f(\sum_j w_j x_{ij})\right)^2$
- 权值更新: $\boxed{w_j^{(k+1)}=w_j^{(k)}-\lambda\frac{\partial E}{\partial w_j}}$
- 激活函数必须是可微的(S型函数或双曲正切)



- 随机梯度下降的反向传播机制

$$w_j^{(k+1)}=w_j^{(k)}+\lambda\sum_i(t_i-o_i)o_i(1-o_i)x_{ij}$$



方案一： Matlab实现

NNminfun/main.m

```
Clear ;  Clc
%formula of real function
F = @(x1,x2,x3)x1^2+ x2^2 + x3^2;
Nabla_f = [2,2,2]';

%inputs
Inputs = 1;
%begin a NN
%set networks
[in,~] = size(inputs);
[out,~] = size(nabla_f);
n_levels = [in,4,2,out];

%feedforward NN
%define w0 and theta0
W = create_w(n_levels);
theta = create_theta(n_levels);

%set loop parameters
np = 1;    perf = 1;
Iter = 100;
Perf = zeros(1,Iter);
step =  0.1;
n_l = length(n_levels);
n_w = numel(W);
n = numel(inputs);
```

```
while(np < Iter && perf > 1e-7)
  z =  cell(n_w,1);
  DW = cell(n_w,1);     DTheta = cell(n_w,1);
  K = cell(n_w,1);
  for i = 1:n_w
    if i == 1
      z{i} = W{i}*inputs + theta{i};
      K{i} = inputs;
    else
      z{i} = W{i}*sigmoid(z{i-1}) + theta{i};
      K{i} = sigmoid(z{i-1});
    end
  end
  outputs = BP_predict(inputs, W, theta);
  perf = f(outputs(1),outputs(2),outputs(3))
  Perf(np) = perf;
  % backpropagation
  for i = n_w:-1:1
   if i == n_w
      delta{i} = nabla_f.*Dsigmoid(z{i});
   else
      delta{i} = W{i+1}'*delta{i+1}.*Dsigmoid(z{i});
   end
  end
  % update W and theta
  for i = 1:n_w
     W{i} = W{i} - step*delta{i}*K{i}';
     theta{i} = theta{i} - step * delta{i};
  end
end
plot(Perf);
outputs = BP_predict(inputs, W, theta)
minfun  = f(outputs(1),outputs(2),outputs(3))
```

实现方案二：基于Numpy

以手写体数字识别问题为例：

- 输入：28x28的图片灰度值矩阵，转换为764x1的向量
- 输出：十维单位向量，1的位置表示分类的数字
- 训练集：5000张带标记的28x28的图片以及相应的数字标记
- 测试集：另外1000张28x28的数字图片
- 采用的神经网络：三层全连接网络[784,28,10]
- 激活函数：sigmoid函数
- 优化算法：GD，SGD，ADAM

神经网络Python实现

```
#Construct Neural Networks
class NeuralNetwork:
    def __init__(self, layers, activation, opt_alg):
        """
        layers: layers of NNs
        Activation:activation function
        opt_alg:optimization algorithm
        """
        if activation == 'tanh':
            self.activation = tanh
            self.activation_deriv = tanh_deriv
        elif activation == 'sigmoid':
            self.activation = sigmoid
            self.activation_deriv = sigmoid_deriv
        elif activation == 'RELU':
            self.activation = RELU
            self.activation_deriv = RELU_deriv
        elif activation == 'RELU3':
            self.activation_deriv = RELU3_deriv
```

```
if opt_alg == 'GD':
            self.opt = self.GD
        elif opt_alg == 'SGD':
            self.opt = self.SGD
        elif opt_alg == 'ADAM':
            self.opt = self.ADAM
            #initial parameters of ADAM
            self.mw = []
            self.mtheta = []
            self.vw = []
            self.vtheta = []
            for i in range(len(layers)-1):
                self.mw.append(np.zeros((layers[i+1], layers[i])))
                self.mtheta.append(np.zeros(layers[i+1]))
                self.mtheta[i] = np.mat(self.mtheta[i]).T

            for i in range(len(layers)-1):
                self.vw.append(np.zeros((layers[i+1], layers[i])))
                self.vtheta.append(np.zeros(layers[i+1]))
                self.vtheta[i] = np.mat(self.vtheta[i]).T

        self.weights = []
        self.thetas = []
        for i in range(len(layers)-1):
            self.weights.append(2*(np.random.rand(layers[i+1], layers[i]))-1)
            self.thetas.append(2*(np.random.random(layers[i+1]))-1)
            self.thetas[i] = np.mat(self.thetas[i]).T
        self.layers = layers
```

前向传播与反向传播

```
def propagation(self, x, k):
    temp = x
    for i in range(len(self.weights)):
        temp =self.activation(np.dot(self.weights[i],temp) + self.thetas[i])
    z = k * temp
    return z

def backpropagation(self, x, error):
    """
    Compute the back propagation
    x:input
    return: back propagation
    K:output of each layer
    Delta:the propagation of each layer
    """
    n_w = len(self.weights)
    z = []
    K = []
    dweights = []
    dthetas = []
    delta = []
    for i in range(n_w):
        if i == 0:
            z.append(np.dot(self.weights[i], x) + self.thetas[i])
            K.append(x)
            delta.append(x)
        else:
            z.append(np.dot(self.weights[i], self.activation(z[i-1])) + self.thetas[i])
            K.append(self.activation(z[i-1]))
            delta.append(self.activation(z[i-1]))

    for i in range(n_w-1, -1, -1):
        if i == n_w-1:
            delta[i] = np.multiply(error, self.activation_deriv(z[i]))
        else:
            delta[i] = np.multiply(np.dot(self.weights[i+1].T, delta[i+1]), self.activation_deriv(z[i]))

    for i in range(n_w):
        dweights.append(np.dot(delta[i], K[i].T))
        dthetas.append(delta[i])
    return dweights, dthetas
```

优化算法

```
def GD(self, X, Y, k, learning_rate, epochs):
    perf = 0
    ddweights = []
    ddthetas = []
    layers = self.layers
    for i in range(len(layers)-1):
        ddweights.append(np.zeros((layers[i+1], layers[i])))
        ddthetas.append(np.zeros(layers[i+1]))
        ddthetas[i] = np.mat(ddthetas[i]).T

    for j in range(int(X.shape[1])):
        input = np.mat(X[:,j]).T
        output = self.propagation(input, k)
        y = np.mat(Y[:,j]).T
        error = y - output
        perf += 1.0/2 * np.sum(np.multiply(error, error))

        dweights, dthetas = self.backpropagation(input, error)
        for i in range(len(self.weights)):
            ddweights[i] += k * 1.0/len(X)*learning_rate * dweights[i]
            ddthetas[i] += k * 1.0/len(X)*learning_rate * dthetas[i]
            #self.weights[i] += k * 1.0*learning_rate * dweights[i]
            #self.thetas[i] += k * 1.0*learning_rate * dthetas[i]

    for i in range(len(self.weights)):
        self.weights[i] += ddweights[i]
        self.thetas[i] += ddthetas[i]
    return perf
```

```
def ADAM(self, X, Y, k, learning_rate , epochs):
    perf = 0
    layers = self.layers
    beta2 = 0.999
    beta1 = 0.9
    epsilon = 0.00000001
    ddweights = []
    ddthetas = []
    for i in range(len(layers)-1):
        ddweights.append(np.zeros((layers[i+1], layers[i])))
        ddthetas.append(np.zeros(layers[i+1]))
        ddthetas[i] = np.mat(ddthetas[i]).T
    rand_X = np.arange(X.shape[1])
    np.random.shuffle(rand_X)
    n = int(X.shape[1]/10)
    for j in range(n):
        input = np.mat(X[:,rand_X[j]]).T
        output = self.propagation(input, k)
        y = np.mat(Y[:,rand_X[j]]).T
        error = y - output
        perf += 1.0/2 * np.sum(np.multiply(error, error))

        dweights, dthetas = self.backpropagation(input, error)
        for i in range(len(self.weights)):
            ddweights[i] += k * 1.0/n * dweights[i]
            ddthetas[i] += k * 1.0/n * dthetas[i]

    for j in range(len(self.weights)):
        self.mw[j] = beta1*self.mw[j] + (1-beta1)*ddweights[j]
        self.vw[j] = beta2*self.vw[j] + (1-beta2)*np.multiply(ddweights[j],ddweights[j])
        self.mtheta[j] = beta1*self.mtheta[j] + (1-beta1)*ddthetas[j]
        self.vtheta[j] = beta2*self.vtheta[j] + (1-beta2)*np.multiply(ddthetas[j],ddthetas[j])
        mwHat = self.mw[j]/(1 - beta1**epochs)
        vwHat = self.vw[j]/(1 - beta2**epochs)
        mtHat = self.mtheta[j] / (1-beta1**epochs)
        vtHat = self.vtheta[j] / (1-beta2**epochs)
        self.weights[j] += learning_rate * np.multiply(mwHat, 1.0/(np.sqrt(vwHat) + epsilon))
        self.thetas[j] += learning_rate * np.multiply(mtHat, 1.0/(np.sqrt(vtHat) + epsilon))
    return perf
```

算例结果 GD & SGD

```
perf: 1151.717221082434 epochs: 9967 predict_true: 3456 precision: 0.6912
perf: 1151.7520690405495 epochs: 9968 predict_true: 3457 precision: 0.6914
perf: 1151.7268360006374 epochs: 9969 predict_true: 3457 precision: 0.6914
perf: 1151.737901898233 epochs: 9970 predict_true: 3457 precision: 0.6914
perf: 1151.7183555250201 epochs: 9971 predict_true: 3457 precision: 0.6914
perf: 1151.7078286233686 epochs: 9972 predict_true: 3457 precision: 0.6914
perf: 1151.7455567021823 epochs: 9973 predict_true: 3456 precision: 0.6912
perf: 1151.7292076041917 epochs: 9974 predict_true: 3456 precision: 0.6912
perf: 1151.7282099598415 epochs: 9975 predict_true: 3457 precision: 0.6914
perf: 1151.719959860039 epochs: 9976 predict_true: 3457 precision: 0.6914
perf: 1151.6755991201467 epochs: 9977 predict_true: 3457 precision: 0.6914
perf: 1151.6803086769437 epochs: 9978 predict_true: 3457 precision: 0.6914
perf: 1151.65095034867 epochs: 9979 predict_true: 3457 precision: 0.6914
perf: 1151.6722665448053 epochs: 9980 predict_true: 3457 precision: 0.6914
perf: 1151.6486455708905 epochs: 9981 predict_true: 3457 precision: 0.6914
perf: 1151.6618234982925 epochs: 9982 predict_true: 3457 precision: 0.6914
perf: 1151.663934855332 epochs: 9983 predict_true: 3458 precision: 0.6916
perf: 1151.6594579574469 epochs: 9984 predict_true: 3459 precision: 0.6918
perf: 1151.656087433337 epochs: 9985 predict_true: 3459 precision: 0.6918
perf: 1151.6402211049058 epochs: 9986 predict_true: 3459 precision: 0.6918
perf: 1151.6025620124294 epochs: 9987 predict_true: 3459 precision: 0.6918
perf: 1151.6574820039716 epochs: 9988 predict_true: 3459 precision: 0.6918
perf: 1151.670972740334 epochs: 9989 predict_true: 3459 precision: 0.6918
perf: 1151.599820279128 epochs: 9990 predict_true: 3459 precision: 0.6918
perf: 1151.5874068696567 epochs: 9991 predict_true: 3459 precision: 0.6918
perf: 1151.573963689703 epochs: 9992 predict_true: 3459 precision: 0.6918
perf: 1151.5816908132958 epochs: 9993 predict_true: 3460 precision: 0.692
perf: 1151.626513247648 epochs: 9994 predict_true: 3460 precision: 0.692
perf: 1151.564730155813 epochs: 9995 predict_true: 3460 precision: 0.692
perf: 1151.5427160377014 epochs: 9996 predict_true: 3459 precision: 0.6918
perf: 1151.5245993468843 epochs: 9997 predict_true: 3459 precision: 0.6918
perf: 1151.5361285780034 epochs: 9998 predict_true: 3459 precision: 0.6918
perf: 1151.4948603403739 epochs: 9999 predict_true: 3459 precision: 0.6918
```

```
perf: 7.341692123196259 epochs: 9966 predict_true: 3955 precision: 0.791
perf: 8.223755231679604 epochs: 9967 predict_true: 3957 precision: 0.7914
perf: 9.579641652290364 epochs: 9968 predict_true: 3957 precision: 0.7914
perf: 6.114119047257799 epochs: 9969 predict_true: 3955 precision: 0.791
perf: 8.314916494454025 epochs: 9970 predict_true: 3958 precision: 0.7916
perf: 8.550272513617722 epochs: 9971 predict_true: 3959 precision: 0.7918
perf: 7.878488553364032 epochs: 9972 predict_true: 3961 precision: 0.7922
perf: 9.544017749534385 epochs: 9973 predict_true: 3956 precision: 0.7912
perf: 6.981533952372505 epochs: 9974 predict_true: 3960 precision: 0.792
perf: 8.828661327789305 epochs: 9975 predict_true: 3954 precision: 0.7908
perf: 11.470910347928829 epochs: 9976 predict_true: 3950 precision: 0.79
perf: 7.2590667818852195 epochs: 9977 predict_true: 3946 precision: 0.7892
perf: 6.3452150416625175 epochs: 9978 predict_true: 3945 precision: 0.789
perf: 6.883654619156194 epochs: 9979 predict_true: 3943 precision: 0.7886
perf: 7.838274253272724 epochs: 9980 predict_true: 3952 precision: 0.7904
perf: 7.500713027480227 epochs: 9981 predict_true: 3951 precision: 0.7902
perf: 6.988023567477538 epochs: 9982 predict_true: 3948 precision: 0.7896
perf: 8.715132384583043 epochs: 9983 predict_true: 3953 precision: 0.7906
perf: 7.75041410303785 epochs: 9984 predict_true: 3954 precision: 0.7908
perf: 9.042222521378422 epochs: 9985 predict_true: 3947 precision: 0.7894
perf: 8.141650316270624 epochs: 9986 predict_true: 3949 precision: 0.7898
perf: 7.955961704505201 epochs: 9987 predict_true: 3946 precision: 0.7892
perf: 11.584425705382154 epochs: 9988 predict_true: 3955 precision: 0.791
perf: 8.60868174342482 epochs: 9989 predict_true: 3956 precision: 0.7912
perf: 8.22857393115915 epochs: 9990 predict_true: 3945 precision: 0.789
perf: 7.79985004179325 epochs: 9991 predict_true: 3946 precision: 0.7892
perf: 11.731010138358858 epochs: 9992 predict_true: 3949 precision: 0.7898
perf: 5.9108459608978485 epochs: 9993 predict_true: 3951 precision: 0.7902
perf: 7.4783986805230676 epochs: 9994 predict_true: 3953 precision: 0.7906
perf: 8.148068809418668 epochs: 9995 predict_true: 3952 precision: 0.7904
perf: 8.688817693198848 epochs: 9996 predict_true: 3947 precision: 0.7894
perf: 9.46231089903901 epochs: 9997 predict_true: 3946 precision: 0.7892
perf: 9.76619232814428 epochs: 9998 predict_true: 3947 precision: 0.7894
perf: 8.529349238501243 epochs: 9999 predict_true: 3944 precision: 0.7888
```

算例结果 SGD & ADAM

```
perf: 7.341692123196259 epochs: 9966 predict_true: 3955 precision: 0.791
perf: 8.223755231679604 epochs: 9967 predict_true: 3957 precision: 0.7914
perf: 9.579641652290364 epochs: 9968 predict_true: 3957 precision: 0.7914
perf: 6.114119047257799 epochs: 9969 predict_true: 3955 precision: 0.791
perf: 8.314916494454025 epochs: 9970 predict_true: 3958 precision: 0.7916
perf: 8.550272513617722 epochs: 9971 predict_true: 3959 precision: 0.7918
perf: 7.878488553364032 epochs: 9972 predict_true: 3961 precision: 0.7922
perf: 9.544017749534385 epochs: 9973 predict_true: 3956 precision: 0.7912
perf: 6.981533952372505 epochs: 9974 predict_true: 3960 precision: 0.792
perf: 8.828661327789305 epochs: 9975 predict_true: 3954 precision: 0.7908
perf: 11.470910347928829 epochs: 9976 predict_true: 3950 precision: 0.79
perf: 7.2590667818852195 epochs: 9977 predict_true: 3946 precision: 0.7892
perf: 6.3452150416625175 epochs: 9978 predict_true: 3945 precision: 0.789
perf: 6.883654619156194 epochs: 9979 predict_true: 3943 precision: 0.7886
perf: 7.838274253272724 epochs: 9980 predict_true: 3952 precision: 0.7904
perf: 7.500713027480227 epochs: 9981 predict_true: 3951 precision: 0.7902
perf: 6.988023567477538 epochs: 9982 predict_true: 3948 precision: 0.7896
perf: 8.715132384583043 epochs: 9983 predict_true: 3953 precision: 0.7906
perf: 7.75041410303785 epochs: 9984 predict_true: 3954 precision: 0.7908
perf: 9.042222521378422 epochs: 9985 predict_true: 3947 precision: 0.7894
perf: 8.141650316270624 epochs: 9986 predict_true: 3949 precision: 0.7898
perf: 7.955961704505201 epochs: 9987 predict_true: 3946 precision: 0.7892
perf: 11.584425705382154 epochs: 9988 predict_true: 3955 precision: 0.791
perf: 8.60868174342482 epochs: 9989 predict_true: 3956 precision: 0.7912
perf: 8.22857393115915 epochs: 9990 predict_true: 3945 precision: 0.789
perf: 7.79985004179325 epochs: 9991 predict_true: 3946 precision: 0.7892
perf: 11.731010138358858 epochs: 9992 predict_true: 3949 precision: 0.7898
perf: 5.9108459608978485 epochs: 9993 predict_true: 3951 precision: 0.7902
perf: 7.4783986805230676 epochs: 9994 predict_true: 3953 precision: 0.7906
perf: 8.148068809418668 epochs: 9995 predict_true: 3952 precision: 0.7904
perf: 8.688817693198848 epochs: 9996 predict_true: 3947 precision: 0.7894
perf: 9.46231089903901 epochs: 9997 predict_true: 3946 precision: 0.7892
perf: 9.76619232814428 epochs: 9998 predict_true: 3947 precision: 0.7894
perf: 8.529349238501243 epochs: 9999 predict_true: 3944 precision: 0.7888
```

```
perf: 40.257105847397504 epochs: 9968 predict_true: 4490 precision: 0.898
perf: 40.12481526431358 epochs: 9969 predict_true: 4491 precision: 0.8982
perf: 38.47324660560l844 epochs: 9970 predict_true: 4490 precision: 0.898
perf: 44.13612099897375 epochs: 9971 predict_true: 4488 precision: 0.8976
perf: 41.63369112737642 epochs: 9972 predict_true: 4488 precision: 0.8976
perf: 38.01004994465526 epochs: 9973 predict_true: 4489 precision: 0.8978
perf: 41.16930772905113 epochs: 9974 predict_true: 4484 precision: 0.8968
perf: 48.86802981728974 epochs: 9975 predict_true: 4483 precision: 0.8966
perf: 43.899974014171896 epochs: 9976 predict_true: 4482 precision: 0.8964
perf: 40.16922643218518 epochs: 9977 predict_true: 4482 precision: 0.8964
perf: 39.627301612455064 epochs: 9978 predict_true: 4482 precision: 0.8964
perf: 39.73576005099163 epochs: 9979 predict_true: 4483 precision: 0.8966
perf: 40.683501372723015 epochs: 9980 predict_true: 4483 precision: 0.8966
perf: 44.99131991688551 epochs: 9981 predict_true: 4481 precision: 0.8962
perf: 45.73617925344857 epochs: 9982 predict_true: 4480 precision: 0.896
perf: 40.54960500774615 epochs: 9983 predict_true: 4482 precision: 0.8964
perf: 52.72879452449752 epochs: 9984 predict_true: 4481 precision: 0.8962
perf: 44.09604803245771 epochs: 9985 predict_true: 4482 precision: 0.8964
perf: 36.94358883164008 epochs: 9986 predict_true: 4481 precision: 0.8962
perf: 40.90821803346045 epochs: 9987 predict_true: 4481 precision: 0.8962
perf: 42.77301808455376 epochs: 9988 predict_true: 4480 precision: 0.896
perf: 48.35476120024695 epochs: 9989 predict_true: 4481 precision: 0.8962
perf: 48.396250985287686 epochs: 9990 predict_true: 4481 precision: 0.8962
perf: 42.93236589313248 epochs: 9991 predict_true: 4481 precision: 0.8962
perf: 42.737546508024806 epochs: 9992 predict_true: 4482 precision: 0.8964
perf: 43.6248755867181 epochs: 9993 predict_true: 4485 precision: 0.897
perf: 44.17281304499997 epochs: 9994 predict_true: 4485 precision: 0.897
perf: 38.33770280263736 epochs: 9995 predict_true: 4486 precision: 0.8972
perf: 43.79242160079971 epochs: 9996 predict_true: 4489 precision: 0.8978
perf: 41.90986565716443 epochs: 9997 predict_true: 4490 precision: 0.898
perf: 39.77771777846409 epochs: 9998 predict_true: 4490 precision: 0.898
```

Homework 12

1. 请详细考察上述梯度下降法中的任意1-2种，并利用它们寻找某二元、三元或多元函数(建议自行构造)的极小值点，比较并记录你的运算结果。
2. 选用合适的梯度下降方案，用于计算文件ex1.dat (三列数据分别为欧氏空间的x,y,z坐标)中所给**散点数据**的线性(z=a+bx+cy)最小二乘拟合。
3. 请用一个四层（即包含两个隐含层）MLP拟合上一题中的散点数据。
4. 针对手写体数字识别任务，尝试复现比较GD，SGD以及ADAM的效率。

注：题1-2不必给出全部代码，简要记录思路、主要代码片段以及结果，作为本讲作业。题3-4选作，鼓励上机课讨论。