```
import random
import numpy as np
class Network(object):
    def _init_(self, sizes):
     '''''参数sizes表示每一层神经元的个数，如[2,3,1]，表示第一层有2个神经元，第二层有3个神经元，第三层有1个神经
        self.num_layers=len(sizes)
        self.sizes=sizes
        self.biases=[np.random.randn(y,1)for y in sizes[1:]]
       self.weight=[np.random.randn(y,x)
                    for x,y in zip(sizes[:-1], sizes[1:])]
    def feedforward(self, a):
      '''''前向传播'''''
       for b, w in zip(self.biases, self.weights):
           a= sigmoid(np.dot(w,a)+b)
       return a
  def SGD(self, tranning_data, epochs, minibatch-size, eta, test-date=None):
        '''''随机梯度下降'''''
       if text-data:
            n-text=len(test-data)
       n=len(training-data)
      for j in xrange(epochs):
          random:shuffle(training-date)
          mini-batches=[
```

```
[training_data[k:k+mini_batch_size]
    for k in xrange(0, n, mini_batch_size)]
for mini-batch in mini-batches:
    self.update-mini_batch(mini_batch, eta)
if test_data:
  print "Epoch{0}:{1}/{2}".format(j, self.evaluate(test_data), n_test)
else:
   print "Epoch{0}complete".format(j).

def update_mini_batch(self, mini_batch, eta):
  """使用反向传播法进行参数更新. mini_batch 是一个元组(x, y)的列表、eta是学习速率"""
  nabla_b=[np.zeros(b.shape) for b in self.biases]
  nabla_w=[np.zeros(w.shape) for w in self.weights]
  for x, y in mini_batch:
      delta_nabla_b, delta_nabla_w=self.backprop(x, y)
      nabla_b=[nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
      nabla_w=[nw+dnw. for nw, dnw in zip(nabla_w, delta_nabla_w)]
  self.weights=[w-(eta/len(mini_batch))*nw
          for w, nw. in zip(self.weights, nabla_w)]
  self.biases=[b-(eta/len(mini_batch))*nb
          for b, nb in zip(self.biases, nabla_b)].
```

```
    返回(nabla_b, nabla_w)代表损失函数的梯度"""
    nabla_b=[np.zeros(b.shape) for b in self.biases]
    nabla_w=[np.zeros(w.shape) for w in self.weights]
    #前向传播
    activation=x
    activations =[x] #list to store all the activations, layer by layer
    zs=[] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z=np.dot(w, activation)+b
        zs.append(z)
        activation=sigmoid(z)
        activations.append(activation)
    #backward pass
    delta=self.cost_derivative(activations[-1], y)*sigmoid_prime(zs[-1])
    nabla_b[-1]=delta
    nabla_w[-1]=np.dot(delta, activations[-2].transpose())
    """l=1表示最后一层神经元; l=2表示倒数第二层神经元, 依次类推"
    for l in xrange(2, self.num_layers):
        z=zs[-l]
        sp=sigmoid_prime(z)
        delta=np.dot(self.weights[-l+1].transpose(), delta)*sp
```

```
    nabla_b[-1]=delta
    nabla_w[-1]=np.dot(delta, activations[-1-1].transpose())

return(nabla_b, nabla_w)

def evaluate(self, test_data):
    """返回正确的分类个数"""
    test_results = [(np.argmax(self.feedforward(x)), y) for (x, y) in test_data]
    return sum(int(x==y) for (x, y) in test_results)

def cost_derivative(self, output_activations, y):
    return(output_activations-y)

def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """sigmoid函数的导数"""
    return sigmoid(z)*(1-sigmoid(z))
```

function	first derivative
Loss=(y-h2)^2	dLoss/dw2=-(y-h2)
h2=sigmoid(z2)	dh2/dz2=h2(1-h2)
z2=h1W2	dz2/dw2=h1
z2=h1W2	dz2/dh1=w2

$$Loss=\frac{1}{2N}\sum_{i=0}^{N}(y_i-h_i)^2+\frac{\lambda}{2N}\sum_{j=0}^{M}\cancel{\theta}w_j^2$$

$$Loss=\frac{1}{2N}\sum_{i=0}^{N}(y_i-h_i)^2+\frac{\lambda}{2N}\sum_{j=0}^{M}w_j^2$$

$$\frac{dLoss}{dw_2}=\frac{dLoss}{dh_2}\cdot\frac{\frac{dz_1}{dh_2}}{dz_2}\cdot\frac{dz_2}{dw_2}\quad\frac{dh_1}{dz_1}\quad\frac{dLoss}{dh_1}$$

$$-(y-h_2)\qquad h_2(1-h_2)\qquad\frac{dz_2}{dw_2}=h_1$$

$$Loss=\frac{1}{n}\sum_{i=1}^{n}(f_i-y_i)^2$$

$$J(\theta)=-\frac{1}{m}\left[\sum_{i=1}^{m}y^i\log(h_\theta(x^{(i)}))+(1-y^i)\log(1-h_\theta(x^{i}))\right]$$

$$Loss=\frac{1}{n}\sum_{i=1}^{n}|f_i-y_i|.$$

$$-\frac{1}{m}\left[\sum_{i=1}^{m}y^i\log(g(\theta^{(2)}a^{(2)}))+(1-y^i)\log(1-g(\theta^{(2)}a^{(2)}))\right]$$

其中y^i为实际值, $g(\theta^{(2)}a^{(2)})=a^{(3)}$

$$w_1^*=w_1-a\frac{dLoss}{dW_1}=\begin{bmatrix}-0.67740455 & 0.57769425 & -0.07365253\\-0.53663519. & 0.60920496 & 0.21149622\end{bmatrix}$$

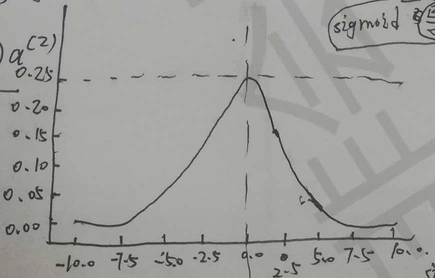
$$-0.001\begin{bmatrix}0.00723246 & 0.00546965 & 0.03647082\\0.00164061 & 0.00124073 & 0.00827303\end{bmatrix}$$

$$\frac{\partial J(\theta)}{\partial\theta^{(2)}}=\frac{1}{m}\sum_{i=1}^{m}(g(\theta^{(2)}a^{(2)})-y^i)a^{(2)}$$

$$=\frac{1}{m}\sum_{i=1}^{m}(a^{(3)}-y^i)a^{(2)}$$

$f(x)=\partial(wx+b)$

sigmoid ~~函数~~导数图像



激活函数

```
1 def sigmoid(z)
      return 1.0/(1.0+np.exp(-z))
```

sigmoid函数重要性质:
$f'(x)=f(x)(1-f(x))$ 即计算导数计

```
1. def sigmoid_prime(z):
2      return sigmoid(z)*(1-sigmo
```