

# Cost Function

Let's first define a few variables that we will need to use:

- $L$  = total number of layers in the network  $L$  表示神经网络的层数
- $s_l$  = number of units (not counting bias unit) in layer  $l$
- $K$  = number of output units/classes  $K$  表示最终结果的元素数

Recall that in neural networks, we may have many output nodes. We denote  $h_{\Theta}(x)_k$  as being a hypothesis that results in the  $k^{th}$  output. Our cost function for neural networks is going to be a generalization of the one we used for logistic regression. Recall that the cost function for regularized logistic regression was:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(h_{\theta}(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\theta}(x^{(i)}))] + \frac{\lambda}{2m} \sum_{j=1}^n \theta_j^2$$

For neural networks, it is going to be slightly more complicated:

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K \left[ y_k^{(i)} \log((h_{\Theta}(x^{(i)}))_k) + (1 - y_k^{(i)}) \log(1 - (h_{\Theta}(x^{(i)}))_k) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}}$$

We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.

In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

theta的列数等于当前层元素数（包括偏差项），theta的行数等于下一层元素数（不包括偏差项）

Note:

- the double sum simply adds up the logistic regression costs calculated for each cell in the output layer
- the triple sum simply adds up the squares of all the individual  $\Theta$ s in the entire network.

- the i in the triple sum does **not** refer to training example i

# Backpropagation Algorithm 反向传递算法

"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute:

$$\min_{\Theta} J(\Theta)$$

That is, we want to minimize our cost function J using an optimal set of parameters in theta. In this section we'll look at the equations we use to compute the partial derivative of J(θ):

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta)$$

In back propagation we're going to compute for every node:

$$\delta_j^{(l)} = \text{"error" of node } j \text{ in layer } l \text{ delta计算与y的误差}$$

Recall that  $a_j^{(l)}$  is activation node j in layer l. For the **last layer**, we can compute the vector of delta values with:

$$\delta^{(L)} = y - a^{(L)}$$

Where L is our total number of layers and  $a^{(L)}$  is the vector of outputs of the activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y. To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left:

$$\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) .* g'(z^{(l)})$$

用后一层计算出的误差值 (delta) 反向推出前一层的误差值，因此叫反向传递算法

The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with a function called  $g'$ , or g-prime, which is the derivative of the activation function g evaluated with the input values given by  $z^{(l)}$ .

The g-prime derivative terms can also be written out as:

$$g'(z^{(l)}) = a^{(l)} .* (1 - a^{(l)})$$

This can be shown and proved in calculus :

$$g(z) = \frac{1}{1+e^{-z}}$$

$$\frac{\partial g(z)}{\partial z} = -\left(\frac{1}{1+e^{-z}}\right)^2 \frac{\partial}{\partial z} (1 + e^{-z})$$

The derivation and proofs are complicated and involved, but you can still implement the above equations to do back propagation without knowing the details. If you are familiar with calculus you can try to finish the derivation, else do not worry about it. The full back propagation equation for the inner nodes is then:

$$\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) . * a^{(l)} . * (1 - a^{(l)})$$

We can compute our partial derivative terms by multiplying our activation values and our error values for each training example t:

$$\frac{\partial J(\Theta)}{\partial \Theta_{i,j}^{(l)}} = \frac{1}{m} \sum_{t=1}^m a_j^{(t)(l)} \delta_i^{(t)(l+1)}$$

This however ignores regularization, which we'll deal with later.

Note:  $\delta^{l+1}$  and  $a^{l+1}$  are vectors with  $s_{l+1}$  elements. Similarly,  $a^{(l)}$  is a vector with  $s_l$  elements. Multiplying them produces a matrix that is  $s_{l+1}$  by  $s_l$  which is the same dimension as  $\Theta^{(l)}$ . That is, the process produces a gradient term for every element in  $\Theta^{(l)}$ .

We can now take all these equations and put them together into a backpropagation algorithm:

### **Back propagation Algorithm**

Given training set  $\{(x^{(1)}, y^{(1)}) \dots (x^{(m)}, y^{(m)})\}$

- Set  $\Delta_{i,j}^{(l)} := 0$  for all  $(l,i,j)$

For training example t = 1 to m:

- Set  $a^{(1)} := x^{(t)}$

- Perform forward propagation to compute  $a^{(l)}$  for  $l=2,3,\dots,L$
- Using  $y^{(t)}$ , compute  $\delta^{(L)} = a^{(L)} - y^{(t)}$
- Compute  $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$  using  $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) . * a^{(l)} . * (1 - a^{(l)})$
- $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$  or with vectorization,  $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$
- $D_{i,j}^{(l)} := \frac{1}{m} (\Delta_{i,j}^{(l)} + \lambda \Theta_{i,j}^{(l)})$ , if  $j \neq 0$ .
- $D_{i,j}^{(l)} := \frac{1}{m} \Delta_{i,j}^{(l)}$  if  $j = 0$

The capital-delta matrix is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative.

The actual proof is quite involved, but, the  $D_{i,j}^{(l)}$  terms are the partial derivatives and the results we are looking for:

$$D_{i,j}^{(l)} = \frac{\partial J(\Theta)}{\partial \Theta_{i,j}^{(l)}}.$$

此处l为神经网络的层数， i表示第i个元素， j表示第i元素的第j个参数

i和j的数字根据当前计算中a的位置为theta注明下标，而不是根据前一列元素的位置，

并且此处theta的标记方法在前向传递和反向传递中都是一样的

另外，偏差项一般被忽略因为永远等于1

## Backpropagation Intuition

Recall that the cost function for a neural network is:

$$J(\Theta) = -\frac{1}{m} \sum_{t=1}^m \sum_{k=1}^K \left[ y_k^{(t)} \log(h_\Theta(x^{(t)}))_k + (1 - y_k^{(t)}) \log(1 - h_\Theta(x^{(t)}))_k \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\cdot)$$

If we consider simple non-multiclass classification ( $k = 1$ ) and disregard regularization, the cost is computed with:

$$cost(t) = y^{(t)} \log(h_\Theta(x^{(t)})) + (1 - y^{(t)}) \log(1 - h_\Theta(x^{(t)}))$$

Intuitively,  $\delta_j^{(l)}$  is the "error" for  $a_j^{(l)}$  (unit j in layer l). More formally, the delta values are actually the derivative of the cost function:

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(t)$$

这段可以理解为反向传递就是一个计算误差值的方法，和代价函数没有本质区别

Recall that our derivative is the slope of a line tangent to the cost function, so the steeper the slope the more incorrect we are. Let us consider the following neural network below and see how we could calculate some  $\delta_j^{(l)}$ .



In the image above, to calculate  $\delta_2^{(2)}$ , we multiply the weights  $\Theta_{12}^{(2)}$  and  $\Theta_{22}^{(2)}$  by their respective  $\delta$  values found to the right of each edge. So we get  $\delta_2^{(2)} = \Theta_{12}^{(2)} * \delta_1^{(3)} + \Theta_{22}^{(2)} * \delta_2^{(3)}$ . To calculate every single possible  $\delta_j^{(l)}$ , we could start from the right of our diagram. We can think of our edges as our  $\Theta_{ij}$ . Going from right to left, to calculate the value of  $\delta_j^{(l)}$ , you can just take the over all sum of each weight times the  $\delta$  it is coming from. Hence, another example would be  $\delta_2^{(3)} = \Theta_{12}^{(3)} * \delta_1^{(4)}$ .

这里以相反的方向计算delta值，使用的theta与前向传递略有区别，使用时应予以注意（上下标的标记方法相同）

## Implementation Note: Unrolling Parameters

With neural networks, we are working with sets of matrices:

$$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}, \dots$$

$$D^{(1)}, D^{(2)}, D^{(3)}, \dots$$

In order to use optimizing functions such as "fminunc()", we will want to "unroll" all the elements and put them into one long vector:

```
1 thetaVector = [ Theta1(:); Theta2(:); Theta3(:); ]
2 deltaVector = [ D1(:); D2(:); D3(:) ]
```

If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11, then we can get back our original matrices from the "unrolled" versions as follows:

```

1 Theta1 = reshape(thetaVector(1:110),10,11)
2 Theta2 = reshape(thetaVector(111:220),10,11)
3 Theta3 = reshape(thetaVector(221:231),1,11)
4

```

由于theta在神经网络中变成了向量，这里主要讲解了怎么用Octave计算神经网络的参数theta  
计算时先将theta和结果D转换成一个超长矢量，然后再用reshape函数还原（需要描述元素数）

## Gradient Checking

思路：用theta的值同时减去和加上一个值（通常是 $10^{-4}$ 次方）并计算连接此两点的直线的斜率（直角边相除），

再将之与theta点的切线斜率相比较，以此来保证反向传递函数的计算没有出现问题

另外，确认反向传递函数没有问题后应该关闭梯度检验，不然会导致计算非常缓慢

Gradient checking will assure that our backpropagation works as intended. We can approximate the derivative of our cost function with:

$$\frac{\partial}{\partial \Theta} J(\Theta) \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}$$

With multiple theta matrices, we can approximate the derivative **with respect to  $\Theta_j$**  as follows:

$$\frac{\partial}{\partial \Theta_j} J(\Theta) \approx \frac{J(\Theta_1, \dots, \Theta_j + \epsilon, \dots, \Theta_n) - J(\Theta_1, \dots, \Theta_j - \epsilon, \dots, \Theta_n)}{2\epsilon}$$

A small value for  $\epsilon$  (epsilon) such as  $\epsilon = 10^{-4}$ , guarantees that the math works out properly. If the value for  $\epsilon$  is too small, we can end up with numerical problems.

Hence, we are only adding or subtracting epsilon to the  $\Theta_j$  matrix. In octave we can do it as follows:

```

1 epsilon = 1e-4;
2 for i = 1:n,
3     thetaPlus = theta;
4     thetaPlus(i) += epsilon;
5     thetaMinus = theta;
6     thetaMinus(i) -= epsilon;
7     gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
8 end;
9

```

We previously saw how to calculate the deltaVector. So once we compute our gradApprox vector, we can check that  $\text{gradApprox} \approx \text{deltaVector}$ .

Once you have verified **once** that your backpropagation algorithm is correct, you don't need to compute gradApprox again. The code to compute gradApprox can be very slow.

## Random Initialization

theta的初始值用0的话会导致神经网络效果降低，因为每一个a都是一样的结果，变成了单纯的梯度下降

Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our  $\Theta$  matrices using the following method:



Hence, we initialize each  $\Theta_{ij}^{(l)}$  to a random value between  $[-\epsilon, \epsilon]$ . Using the above formula guarantees that we get the desired bound. The same procedure applies to all the  $\Theta$ 's. Below is some working code you could use to experiment.

```
1 If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11.  
2  
3 Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;  
4 Theta2 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;  
5 Theta3 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;  
6
```

rand(x,y) is just a function in octave that will initialize a matrix of random real numbers between 0 and 1.

(Note: the epsilon used above is unrelated to the epsilon from Gradient Checking)

## Putting it Together

First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers in total you want to have.

- Number of input units = dimension of features  $x^{(i)}$
- Number of output units = number of classes
- Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
- Defaults: 1 hidden layer. If you have more than 1 hidden layer, then it is recommended that you have the same number of units in every hidden layer.

- ①输入的x数决定维度
- ②输出的元素数取决于种类数
- ③隐藏层的元素越多越好，但理论上每一隐藏层拥有的元素数都相同

## Training a Neural Network

1. Randomly initialize the weights
2. Implement forward propagation to get  $h_{\Theta}(x^{(i)})$  for any  $x^{(i)}$
3. Implement the cost function
4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

When we perform forward and back propagation, we loop on every training example:

```

1 for i = 1:m,
2     Perform forward propagation and backpropagation using example (x(i),y(i))
3     (Get activations a(l) and delta terms d(l) for l = 2,...,L)

```

The following image gives us an intuition of what is happening as we are implementing our neural network:



Ideally, you want  $h_{\Theta}(x^{(i)}) \approx y^{(i)}$ . This will minimize our cost function. However, keep in mind that  $J(\Theta)$  is not convex and thus we can end up in a local minimum instead.