

# Machine Learning - Week 5

赵燕

## 目录

<b>1</b>	<b>Cost Function and Backpropagation</b>	<b>2</b>
1.1	Cost Function . . . . .	2
1.2	Backpropagation Algorithm . . . . .	4
1.3	Backpropagation Intuition . . . . .	8
<b>2</b>	<b>Backpropagation in Practice</b>	<b>10</b>
2.1	Implementation Note:Unrolling Parameters . . . . .	10
2.2	Gradient Checking . . . . .	11
2.3	Random Initialization . . . . .	14
2.4	Putting it Together . . . . .	15
<b>3</b>	<b>Application of Neural Networks</b>	<b>17</b>
3.1	Autonomous Driving . . . . .	17

# 1 Cost Function and Backpropagation

## 1.1 Cost Function

首先引入一些便于稍后讨论的新标记方法:

假设神经网络的训练样本有 $m$ 个, 每个包含一组输入 $x$ 和输出信号 $y$ ,  $L$ 表示神经网络层数,  $s_l$ 表示每层的neuron的个数,  $s_L$ 表示最后一层中处理单元的个数。

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

- $L$  = total number of layers in the network
- $s_l$  = number of units (not counting bias unit) in layer  $l$
- $K$  = number of output units/classes

将神经网络的分类定义为两种情况: 二类分类和多类分类。

二类分类 (Binary classification) :

- $s_L = 1, y = 0 \text{ or } 1$ , 表示哪一类;
- 1 output unit

K类分类 (Multi-class classification (K classes) ) :

- $s_L = K, y_i = 1$ , 表示分到第 $i$ 类 ( $K \geq 3$ ) ;
- $K$  output units

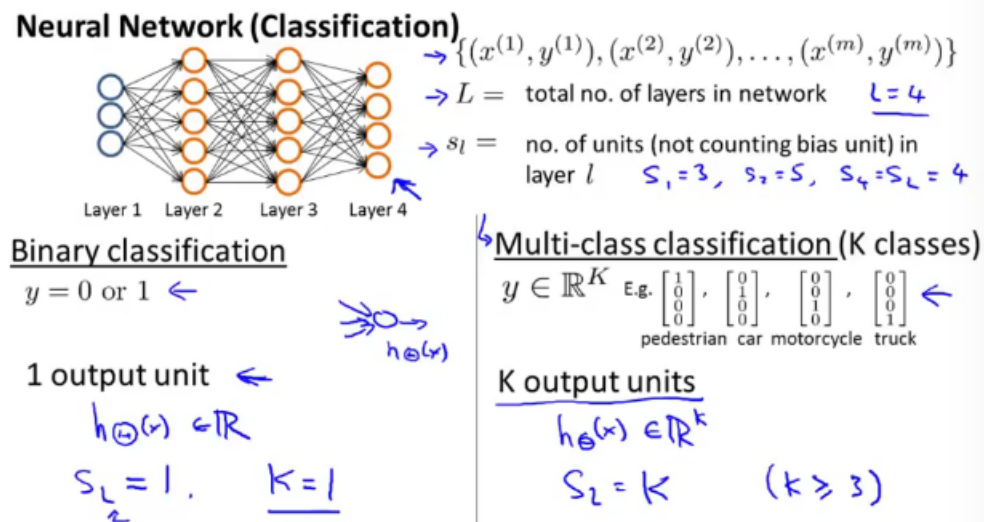


图 1: 二类分类和K类分类

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:

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

在逻辑回归中，只有一个输出变量，又称标量（scalar），也只有一个因变量y，但是在神经网络中，可以有很多输出变量， $h_{\Theta}(x)$ 是一个K维度的向量，并且训练集中的因变量也是同样维度的一个向量，因此神经网络的代价函数会比逻辑回归更加复杂一些。

For Neural Networks:

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

## Cost function

Logistic regression:

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

Neural network:

$\rightarrow h_{\Theta}(x) \in \mathbb{R}^K$   $(h_{\Theta}(x))_i = i^{th}$  output

$$\rightarrow J(\Theta) = -\frac{1}{m} \left[ \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log((h_{\Theta}(x^{(i)}))_k) + (1 - y_k^{(i)}) \log(1 - (h_{\Theta}(x^{(i)}))_k) \right]$$

$+\frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{j,i}^{(l)})^2$

Diagram illustrating the neural network structure and cost function components. The diagram shows a neural network with layers  $s_0, s_1, \dots, s_L$ . The input layer  $s_0$  has nodes  $x_0, x_1, \dots$ . The hidden layers  $s_1, \dots, s_{L-1}$  have nodes  $x_0, x_1, \dots$ . The output layer  $s_L$  has nodes  $y_1, y_2, \dots, y_K$ . The cost function is shown with nested summations over samples  $i$ , output nodes  $k$ , and layers  $l$ . The regularization term is shown as a sum of squared weights  $\Theta_{j,i}^{(l)}$  across all layers  $l$  and nodes  $i, j$ .

图 2: 逻辑回归和神经网络的代价函数

这个看起来复杂很多的代价函数背后的思想还是一样的，希望通过代价函数来观察算法预测的结果与真实情况的误差有多大，唯一不同的是，对于每一行特征，我们都会给出K个预测，基本上可以利用循环，对每一行特征都预测K个不同的结果，然后利用循环在K个预测中选择可能性最高的一个，将其与y中的实际数据进行比较。

正则化的那一项只是排除了每一层的 $\Theta_0$ 后，每一层的 $\Theta$ 矩阵的和，最里层的循环j循环所有的行（由 $s_l + 1$ 层的激活单元数决定），循环i则循环所有的列，由该层（ $s_l$ 层）的激活单元数所决定。即： $h_{\Theta}(x)$ 与真实值之间的距离为每个样本，每个类输出的加和，对参数进行regularization的bias项处理所有参数的平方和。

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.

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  $i$

## 1.2 Backpropagation Algorithm

之前在计算神经网络预测结果时采用的是一种正向传播方法，从第一层开始正向一层一层进行计算，直到最后一层的  $h_{\Theta}(x)$ 。

现在，为了计算代价函数的偏导数  $\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta)$ ，我们采用一种反向传播算法，也就是首先计算一层的误差，然后再一层一层反向求出各层的误差，直到倒数第二层。

举例说明反向传播算法：

假设我们的训练集只有一个实例  $(x^{(1)}, y^{(1)})$ ，我们的神经网络是一个四层的神经网络，其中  $K=4$ ， $S_L = 4$ ， $L=4$ ：

前向传播算法：

$$a^{(1)} = x \quad (3)$$

$$z^{(2)} = \Theta^{(1)} a^{(1)} \quad (4)$$

$$a^{(2)} = g(z^{(2)}) (\text{add } a_0^{(2)}) \quad (5)$$

$$z^{(3)} = \Theta^{(2)} a^{(2)} \quad (6)$$

$$a^{(3)} = g(z^{(3)}) (\text{add } a_0^{(3)}) \quad (7)$$

$$z^{(4)} = \Theta^{(3)} a^{(3)} \quad (8)$$

$$a^{(4)} = h_{\Theta}(x) = g(z^{(4)}) \quad (9)$$

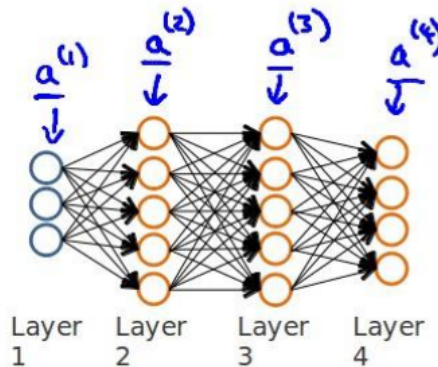


图 3: 前向传播算法求激励函数

计算偏导数项利用反向传播算法: Backpropagation Algorithm

”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:

$$\min_{\Theta} J(\Theta) \quad (10)$$

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  $J(\Theta)$ :

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta) \quad (11)$$

To do so, we use the following algorithm:

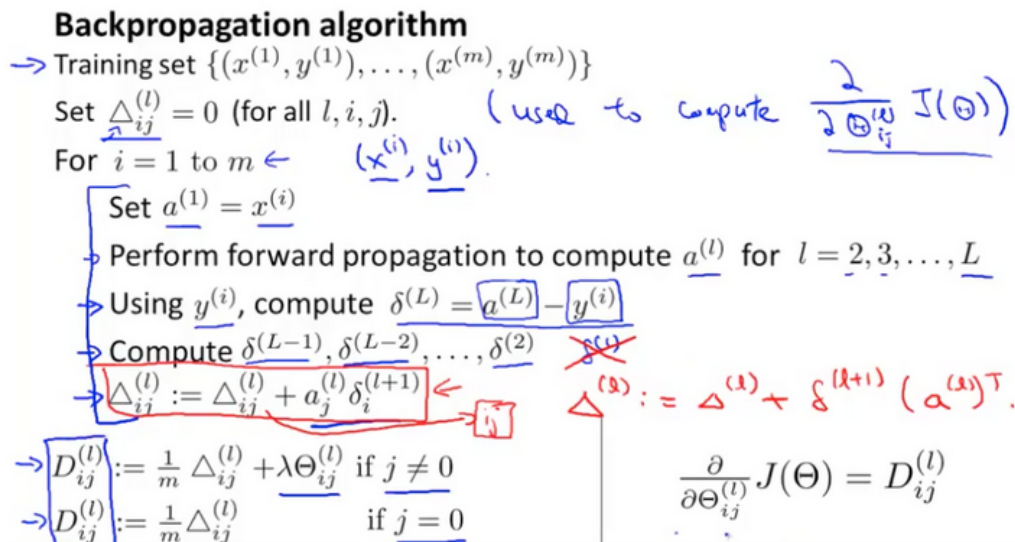


图 4: 后向传播算法

Given training set  $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})$

Set  $\Delta_{i,j}^{(l)} := 0$  for all  $(l, i, j)$ , (hence you end up having a matrix full of zeros)

For training example  $t = 1$  to  $m$ :

- (1) Set  $a^{(1)} := x^{(t)}$
- (2) Perform forward propagation to compute  $a^{(l)}$  for  $l = 2, 3, \dots, L$

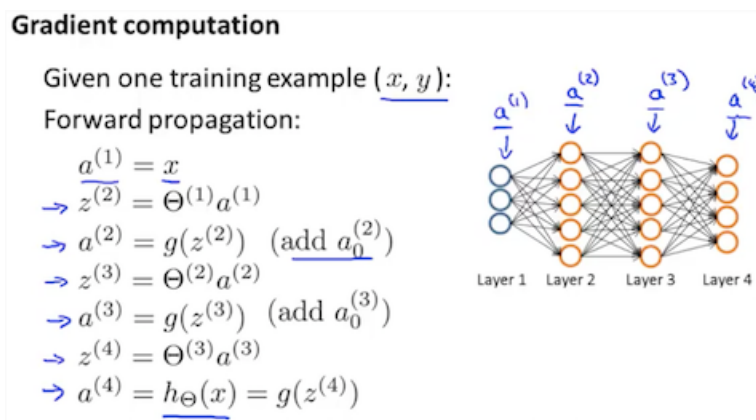


图 5: 前向传播算法

- (3) Using  $y^{(t)}$ , compute  $\delta^{(L)} = a^{(L)} - y^{(t)}$

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:

$$(4) \text{ Compute } \delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)} \text{ using } \delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) \cdot a^{(l)} \cdot (1 - a^{(l)})$$

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:

$$g'(z^{(l)}) = a^{(l)} \cdot (1 - a^{(l)}) \quad (12)$$

$$(5) \Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)} \text{ or with vectorization, } \Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$$

Hence we update our new  $\Delta$  matrix.

$$D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \quad \text{if } j \neq 0 \quad (13)$$

$$D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} \quad \text{if } j = 0 \quad (14)$$

The capital-delta matrix  $D$  is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative. Thus we get  $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$

Intuition:  $\delta_j^{(i)}$  = "error" of node  $j$  in layer  $l$ .

$a_j^{(i)}$ : 第 $l$ 层第 $j$ 个单元 (节点) 的激励值。

我们从最后一层的误差开始计算，误差是激活单元的预测 ( $a_j^{(4)}$ ) 与实际值 ( $y_j$ ) 之间的误差， $j=1:L$ 。

For each output unit (layer  $L=4$ ):

$$\delta_j^{(4)} = a_j^{(4)} - y_j \quad (15)$$

这里  $\delta_j^{(4)}$  相当于  $(h_{\Theta}(x))_j$

我们用  $\delta$  来表示误差，则 (向量化) :

$$\delta^{(4)} = a^{(4)} - y \quad (16)$$

向量的维度: 单元数  $\times 1$

我们利用这个误差来计算前一层的误差:

$$\delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} \cdot g'(z^{(3)}) \quad (17)$$

其中  $g'(z^{(3)})$  是 S 形函数的导数:

$$g'(z^{(3)}) = a^{(3)} \cdot (1 - a^{(3)}) \quad (18)$$

而  $(\Theta^{(3)})\delta^{(4)}$  则是权重导数的误差的和。

下一步是继续计算第二层的误差:

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} \cdot g'(z^{(2)}) \quad (19)$$

因为第一层是输入变量, 不存在误差。我们有了所有的误差的表达式后, 便可以计算代价函数的偏导数了, 假设 $\lambda = 0$ , 即我们不做任何正则化处理时:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{l+1} \quad (20)$$

注释:

$l$ =目前所计算的是第几层;

$j$ =目前计算层中的激活单元的下标, 也将是下一层的第 $j$ 个输入变量的下标;

$i$ =下一层中误差单元的下标, 是受到权重矩阵中第 $i$ 行影响的下一层中的误差单元的下标。

如果考虑正则化处理, 并且训练集是一个特征矩阵而非向量。在上面特殊的情况中, 我们需要计算每一层的误差单元来计算代价函数的偏导数。在更为一般的情况中, 同样需要计算每一层的误差单元, 但是需要为整个训练集计算误差单元, 此时的误差单元也是一个矩阵, 我们用 $\Delta_{ij}^{(l)}$ 来表示整个误差矩阵。第 $l$ 层的第 $i$ 个激活单元受到第 $j$ 个参数影响而导致的误差。

算法表示:

```
for i=1:m {
    set a(i)=x(i)
    perform forward propagation to compute a(l) for l=1,2,3...L
    Using  $\delta^{(L)}=a^{(L)}-y^i$ 
    perform back propagation to compute all previous layer error vector
     $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l+1)} + a_j^{(l)} \delta_i^{l+1}$ 
}
```

图 6: 算法表示

即首先用正向传播方法计算出每一层的激活单元, 利用训练集的结果与神经网络预测的结果求出最后一层的误差, 然后利用该误差运用反向传播算法计算出直至第二层的所有误差。

在求出了 $\Delta_{ij}^{(l)}$ 之后, 我们便可以计算代价函数的偏导数了, 计算方法如下:

$$D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} \quad \text{if } j \neq 0 \quad (21)$$

$$D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} \quad \text{if } j = 0 \quad (22)$$

在Octave中, 如果我们要使用fminuc这样的优化算法来求解出权重矩阵, 我们需要将矩阵首先展开成向量, 再利用算法求出最优解后再重新转换回矩阵。

假设我们有三个权重矩阵，Theta1, Theta2, Theta3，尺寸分别为和，下面的代码可以实现这样的转换：

```
thetaVec=[Theta1(:);Theta2(:);Theta3(:)]
...optimization using functions like fminuc...
Theta1=reshape(thetaVec(1:110,10,11));
Theta2=reshape(thetaVec(111:220,10,11));
Theta3=reshape(thetaVec(221:231,1,11));
```

### 1.3 Backpropagation Intuition

回顾之前所学的前向传播算法：

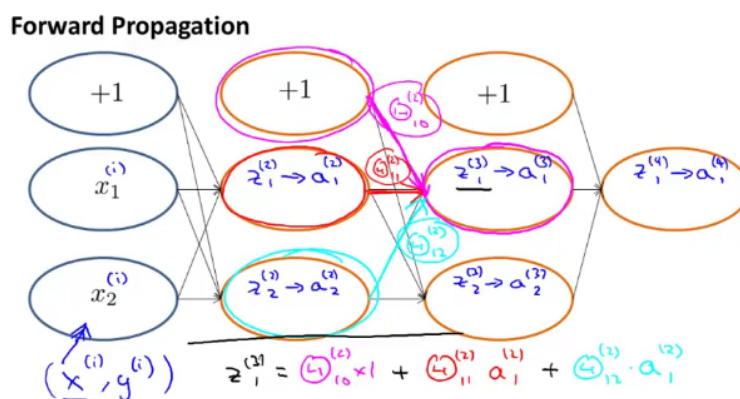


图 7: 回顾前向传播算法

Recall that the cost function for a neural network is:

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K [y_k^{(i)} \log((h_{\Theta}(x^{(i)}))_k) + (1 - y_k^{(i)}) \log(1 - (h_{\Theta}(x^{(i)}))_k)] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{j,i}^{(l)})^2 \quad (23)$$

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

$$\text{cost}(t) = y^{(t)} \log(h_{\Theta}(x^{(ty)})) + (1 - y^{(t)}) \log(1 - h_{\Theta}(x^{(t)})) \quad (24)$$

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:

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(t) \quad (25)$$

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  $\delta_j^{(l)}$ :

后向传播算法做的是：



What is backpropagation doing?

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

$(x^{(i)}, y^{(i)})$

Focusing on a single example  $x^{(i)}, y^{(i)}$ , the case of 1 output unit, and ignoring regularization ( $\lambda = 0$ ),

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

(Think of  $\text{cost}(i) \approx (h_{\Theta}(x^{(i)}) - y^{(i)})^2$ )

i.e. how well is the network doing on example  $i$ ?

图 8: 后向传播算法做什么

后向传播算法:

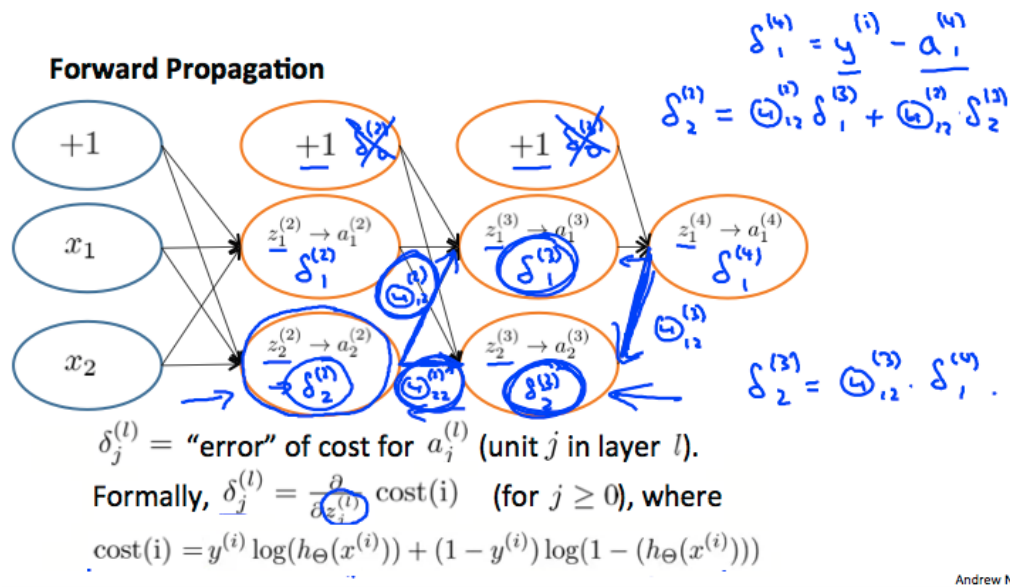
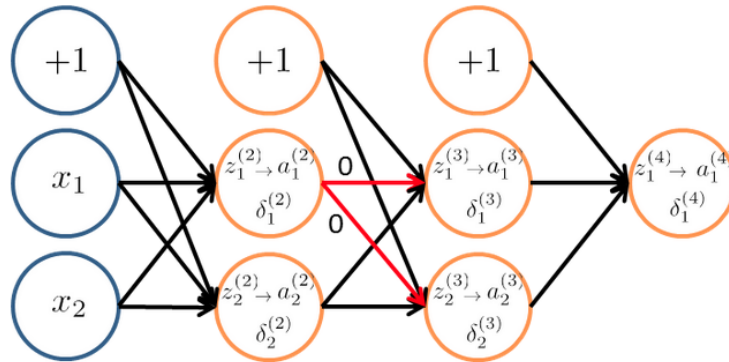


图 9: 后向传播算法

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  $\delta_2^{(3)} = \Theta_{12}^{(3)} * \delta_1^{(4)}$ .

练习题:

Consider the following neural network:



Suppose both of the weights shown in red ( $\Theta_{11}^{(2)}$  and  $\Theta_{21}^{(2)}$ ) are equal to 0. After running backpropagation, what can we say about the value of  $\delta_1^{(3)}$ ?

- ☐  $\delta_1^{(3)} > 0$
- ☐  $\delta_1^{(3)} = 0$  only if  $\delta_1^{(2)} = \delta_2^{(2)} = 0$ , but not necessarily otherwise
- ☐  $\delta_1^{(3)} \leq 0$  regardless of the values of  $\delta_1^{(2)}$  and  $\delta_2^{(2)}$
- ☒ There is insufficient information to tell

正确

图 10: 后向传播算法习题

## 2 Backpropagation in Practice

### 2.1 Implementation Note: Unrolling Parameters

在上一段视频中，我们谈到了怎么使用反向传播算法计算代价函数的导数，在这段视频中，介绍细节的实现过程，怎样把参数从矩阵展开成向量，以便我们在高级最优化步骤中的使用需要。

#### Advanced optimization

```
function [jVal, gradient] = costFunction(theta)
    ...
    optTheta = fminunc(@costFunction, initialTheta, options)
```

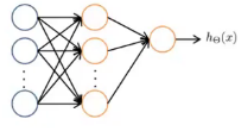
Neural Network (L=4):

- $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$  - matrices (Theta1, Theta2, Theta3)
- $D^{(1)}, D^{(2)}, D^{(3)}$  - matrices (D1, D2, D3)

"Unroll" into vectors

### Example

$s_1 = 10, s_2 = 10, s_3 = 1$   
 $\rightarrow \Theta^{(1)} \in \mathbb{R}^{10 \times 11}, \Theta^{(2)} \in \mathbb{R}^{10 \times 11}, \Theta^{(3)} \in \mathbb{R}^{1 \times 11}$   
 $\rightarrow D^{(1)} \in \mathbb{R}^{10 \times 11}, D^{(2)} \in \mathbb{R}^{10 \times 11}, D^{(3)} \in \mathbb{R}^{1 \times 11}$   
 $\rightarrow \text{thetaVec} = [\text{Theta1}(:); \text{Theta2}(:); \text{Theta3}(:)];$   
 $\rightarrow \text{DVec} = [\text{D1}(:); \text{D2}(:); \text{D3}(:)];$   
 $\text{Theta1} = \text{reshape}(\text{thetaVec}(1:110), 10, 11);$   
 $\rightarrow \text{Theta2} = \text{reshape}(\text{thetaVec}(111:220), 10, 11);$   
 $\rightarrow \text{Theta3} = \text{reshape}(\text{thetaVec}(221:231), 1, 11);$



Suppose D1 is a 10x6 matrix and D2 is a 1x11 matrix. You set:

`DVec = [D1(:); D2(:)];`

Which of the following would get D2 back from DVec?

- ☐ `reshape(DVec(60:71), 1, 11)`
- ☐ `reshape(DVec(61:72), 1, 11)`
- ☒ `reshape(DVec(61:71), 1, 11)`
- ☐ `reshape(DVec(60:70), 11, 1)`

正确

### Learning Algorithm

$\rightarrow$  Have initial parameters  $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ .  
 $\rightarrow$  Unroll to get `initialTheta` to pass to  
 $\rightarrow \text{fminunc}(@\text{costFunction}, \text{initialTheta}, \text{options})$   
  
`function [jval, gradientVec] = costFunction(thetaVec)`  
 $\rightarrow$  From `thetaVec`, get  $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$  *reshape*  
 $\rightarrow$  Use forward prop/back prop to compute  $D^{(1)}, D^{(2)}, D^{(3)}$  and  $J(\Theta)$ .  
 Unroll  $D^{(1)}, D^{(2)}, D^{(3)}$  to get `gradientVec`.

## 2.2 Gradient Checking

当我们对一个较为复杂的模型（例如神经网络）使用梯度下降算法时，可能存在一些不容易察觉的错误，意味着，虽然表面看起来一直在工作，代价看上去不断的减少，但是最终的结果可能并不是最优解。

为了避免这个问题，采取一种叫做梯度的数值检验（Numerical Gradient Checking）的方法，思想是通过估计梯度值来检验我们计算的导数值是否真的是我们要求的。

对梯度的估计方法：在代价函数上沿着切线的方向选择两个非常近的点，然后再计算两个点的平均值，以估计梯度。

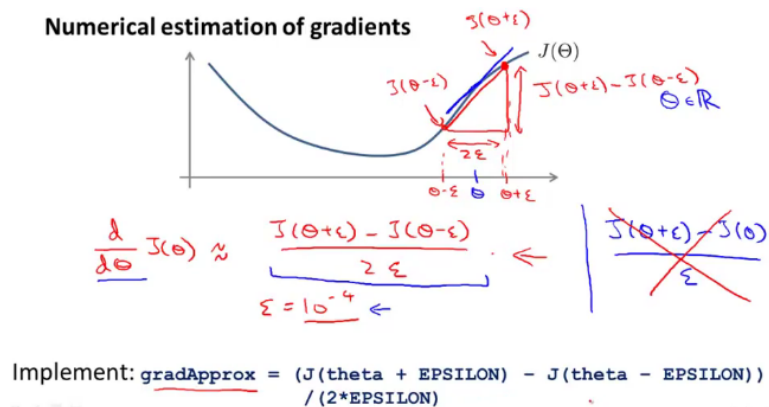


图 11: 梯度检验法

Let  $J(\theta) = \theta^3$ . Furthermore, let  $\theta = 1$  and  $\epsilon = 0.01$ . You use the formula:

$$\frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2\epsilon}$$

to approximate the derivative. What value do you get using this approximation?  
(When  $\theta = 1$ , the true, exact derivative is  $\frac{d}{d\theta} J(\theta) = 3$ ).

☐ 3.0000

☒ 3.0001

正确

☐ 3.0301

☐ 6.0002

即对于某个特定的 $\theta$ ，我们计算出在 $\theta - \epsilon$ 处和 $\theta + \epsilon$ 的代价函数值（ $\epsilon$ 是一个非常小的值，通常选取0.0001），然后求两个代价的平均，用以估计在 $\theta$ 处的代价值。

### Parameter vector $\theta$

→  $\theta \in \mathbb{R}^n$  (E.g.  $\theta$  is "unrolled" version of  $\underline{\theta}^{(1)}, \underline{\theta}^{(2)}, \underline{\theta}^{(3)}$ )

→  $\theta = [\theta_1, \theta_2, \theta_3, \dots, \theta_n]$

→  $\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1 + \epsilon, \theta_2, \theta_3, \dots, \theta_n) - J(\theta_1 - \epsilon, \theta_2, \theta_3, \dots, \theta_n)}{2\epsilon}$

→  $\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1, \theta_2 + \epsilon, \theta_3, \dots, \theta_n) - J(\theta_1, \theta_2 - \epsilon, \theta_3, \dots, \theta_n)}{2\epsilon}$

⋮

→  $\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, \dots, \theta_n + \epsilon) - J(\theta_1, \theta_2, \theta_3, \dots, \theta_n - \epsilon)}{2\epsilon}$

图 12: 向量 $\theta$

Octave中的代码如下:

```
gradApprox=(J(theta+eps)-J(theta-eps))/(2*eps)
```

当 $\theta$ 是一个向量时,则需要对偏导数进行检验。因为代价函数的偏导数检验只针对一个参数的改变进行检验,下面是一个只针对 $\theta_1$ 进行检验的示例:

$$\frac{\partial}{\partial \theta_1} = \frac{J(\theta_1 + \epsilon_1, \theta_2, \theta_3, \dots, \theta_n) - J(\theta_1 - \epsilon_1, \theta_2, \theta_3, \dots, \theta_n)}{2\epsilon} \quad (26)$$

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

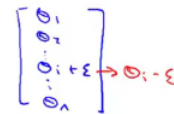
Once you have verified once that your backpropagation algorithm is correct, you don't need to compute gradApprox again. The code to compute gradApprox can be very slow.

```

for i = 1:n, ←
    thetaPlus = theta;
    thetaPlus(i) = thetaPlus(i) + EPSILON;
    thetaMinus = theta;
    thetaMinus(i) = thetaMinus(i) - EPSILON;
    gradApprox(i) = (J(thetaPlus) - J(thetaMinus))
                  / (2*EPSILON);
end;

```

Check that  $\text{gradApprox} \approx \text{DVec}$  ←  
 ↑  
 From backprop.



$\frac{\partial}{\partial \theta_i} J(\theta)$

最后我们还需要对通过反向传播方法计算出的偏导数进行检验。

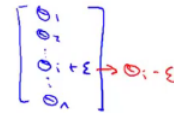
根据上面的算法,计算出的偏导数存储在矩阵  $D_{ij}^{(l)}$  中。检验时,将该矩阵展开成为向量,同时我们也将 $\Theta$ 矩阵展开为向量,我们针对每一个 $\Theta$ 都计算一个近似的梯度值,将这些值存储于一个近似梯度矩阵中,最终将得出的这个矩阵同 $D_{ij}^{(l)}$ 进行比较。

```

for i = 1:n, ←
    thetaPlus = theta;
    thetaPlus(i) = thetaPlus(i) + EPSILON;
    thetaMinus = theta;
    thetaMinus(i) = thetaMinus(i) - EPSILON;
    gradApprox(i) = (J(thetaPlus) - J(thetaMinus))
                  / (2*EPSILON);
end;

```

Check that  $\text{gradApprox} \approx \text{DVec}$  ←  
 ↑  
 From backprop.



$\frac{\partial}{\partial \theta_i} J(\theta)$

What is the main reason that we use the backpropagation algorithm rather than the numerical gradient computation method during learning?

- ☐ The numerical gradient computation method is much harder to implement.
- ☒ The numerical gradient algorithm is very slow.
- ☐ Backpropagation does not require setting the parameter EPSILON.
- ☐ None of the above.

正确

## 2.3 Random Initialization

### Initial value of $\Theta$

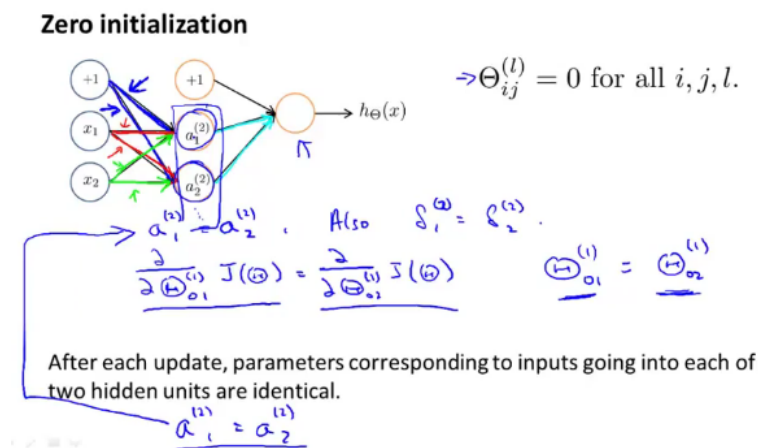
For gradient descent and advanced optimization method, need initial value for  $\Theta$ .

```
optTheta = fminunc(@costFunction,  
    initialTheta, options)
```

Consider gradient descent

Set initialTheta = zeros(n,1) ?

任何优化算法都需要一些初始的参数。到目前为止我们都是初始所有参数为 0, 这样的初始方法对于逻辑回归来说是可行的, 但是对于神经网络来说是不可行的。如果我们令所有的初始参数都为 0, 这将意味着我们第二层的所有激活单元都会有相同的值 (对称化)。同理, 如果我们初始所有的参数都为 0 的数, 结果也是一样的。



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: 通常初始参数为正负  $\epsilon$  之间的随机值, 假设我们要随机初始一个尺寸为  $10 \times 11$  的参数矩阵, 代码如下:

```
Theta1 = rand(10, 11) * (2*eps) - eps
```

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)

**Random initialization: Symmetry breaking**

→ Initialize each  $\Theta_{ij}^{(l)}$  to a random value in  $[-\epsilon, \epsilon]$   
(i.e.  $-\epsilon \leq \Theta_{ij}^{(l)} \leq \epsilon$ )

E.g.

→  $\text{Theta1} = \text{rand}(10, 11) * (2 * \text{INIT\_EPSILON}) - \text{INIT\_EPSILON};$   $[-\epsilon, \epsilon]$

→  $\text{Theta2} = \text{rand}(1, 11) * (2 * \text{INIT\_EPSILON}) - \text{INIT\_EPSILON};$

*Handwritten notes: "Random 10x11 matrix (betw. 0 and 1)" with an arrow pointing to the rand(10,11) function.*

Consider this procedure for initializing the parameters of a neural network:

1. Pick a random number  $r = \text{rand}(1,1) * (2 * \text{INIT\_EPSILON}) - \text{INIT\_EPSILON};$
2. Set  $\Theta_{ij}^{(l)} = r$  for all  $i, j, l$ .

Does this work?

- ☐ Yes, because the parameters are chosen randomly.
- ☐ Yes, unless we are unlucky and get  $r=0$  (up to numerical precision).
- ☐ Maybe, depending on the training set inputs  $x(i)$ .
- ☒ No, because this fails to break symmetry.

正确

## 2.4 Putting it Together

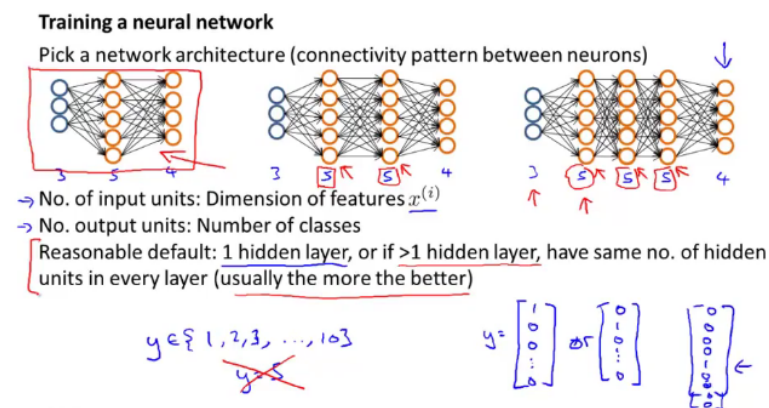


图 13: 训练神经网络

使用神经网络的步骤:

网络结构: 第一件要做的事是选择网络结构, 即决定选择多少层以及决定每层分别有多少个单元。

第一层的单元数即是我们训练集的特征数量。



最后一层的单元数是我们训练集的结果的类的数量。

如果隐藏层数大于1，确保每个隐藏层的单元个数相同，通常情况下隐藏层单元的个数越多越好。

我们真正要决定的是隐藏层的层数和每个中间层的单元数。

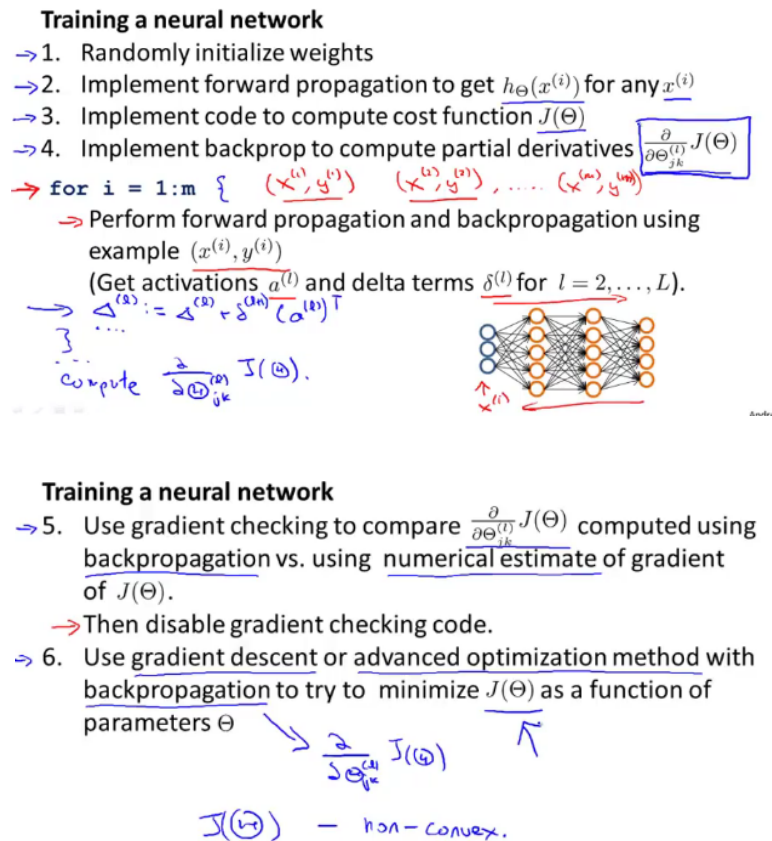


图 14: 训练神经网络步骤

注意:  $J(\Theta)$ 是一个non-convex函数（非凸函数），理论上都能都优化到最小值。

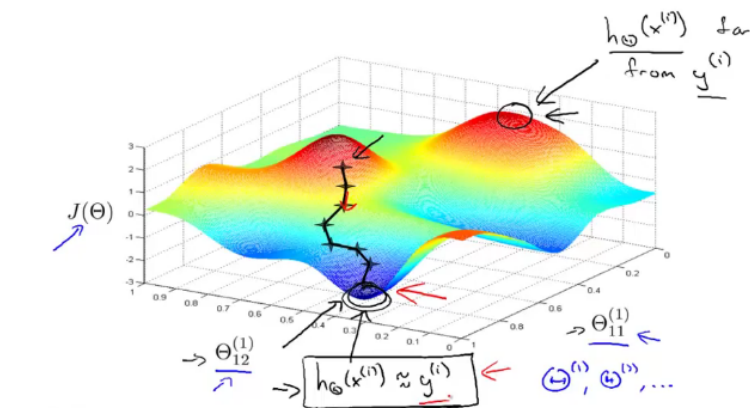


图 15: 神经网络的梯度下降算法的实现



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.

Suppose you are using gradient descent together with backpropagation to try to minimize  $J(\Theta)$  as a function of  $\Theta$ . Which of the following would be a useful step for verifying that the learning algorithm is running correctly?

- ☐ Plot  $J(\Theta)$  as a function of  $\Theta$ , to make sure gradient descent is going downhill.
  - ☐ Plot  $J(\Theta)$  as a function of the number of iterations and make sure it is increasing (or at least non-decreasing) with every iteration.
  - ☒ Plot  $J(\Theta)$  as a function of the number of iterations and make sure it is decreasing (or at least non-increasing) with every iteration.
- 正确
- ☐ Plot  $J(\Theta)$  as a function of the number of iterations to make sure the parameter values are improving in classification accuracy.

训练神经网络的步骤:

- (1) 参数的随机初始化
- (2) 利用正向传播方法计算所有的 $h_{\Theta}(x)$
- (3) 编写计算代价函数 $J(\Theta)$ 的代码
- (4) 利用反向传播算法计算所有偏导数
- (5) 利用数值检验方法检验这些偏导数
- (6) 使用优化算法来最小化代价函数

## 3 Application of Neural Networks

### 3.1 Autonomous Driving