Machine Learning - Week 5

赵燕

目录

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 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（$K \geqslant 3$）;
- K output units

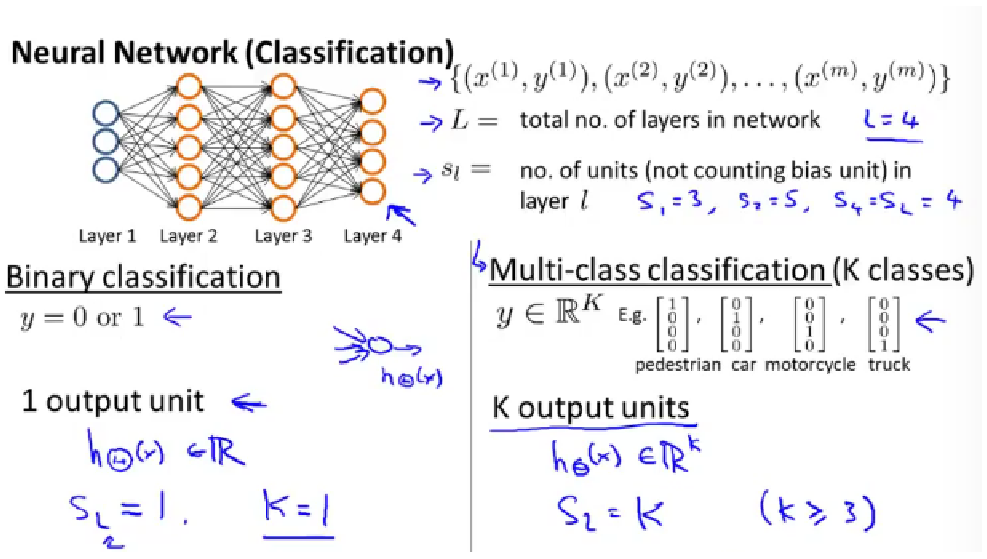

Neural Network (Classification)
$\{(x^{(1)},y^{(1)}),(x^{(2)},y^{(2)}),\dots,(x^{(m)},y^{(m)})\}$
$L =$ total no. of layers in network
$L=4$
$s_l =$ no. of units (not counting bias unit) in layer l
$s_1=3$, $s_2=5$, $s_4=s_L=4$
Layer 1
Layer 2
Layer 3
Layer 4
Binary classification
$y = 0$ or 1
$h_\Theta(x)$
1 output unit
$h_\Theta(x) \in \mathbb{R}$
$s_L = 1$, $K=1$
Multi-class classification (K classes)
$y \in \mathbb{R}^K$ E.g.
pedestrian car motorcycle truck
K output units
$h_\Theta(x) \in \mathbb{R}^K$
$s_L = K$
$(K \geq 3)$

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 kth output. Our cost function for neural networks is going to be a generalization of the one we used for logistic regression.

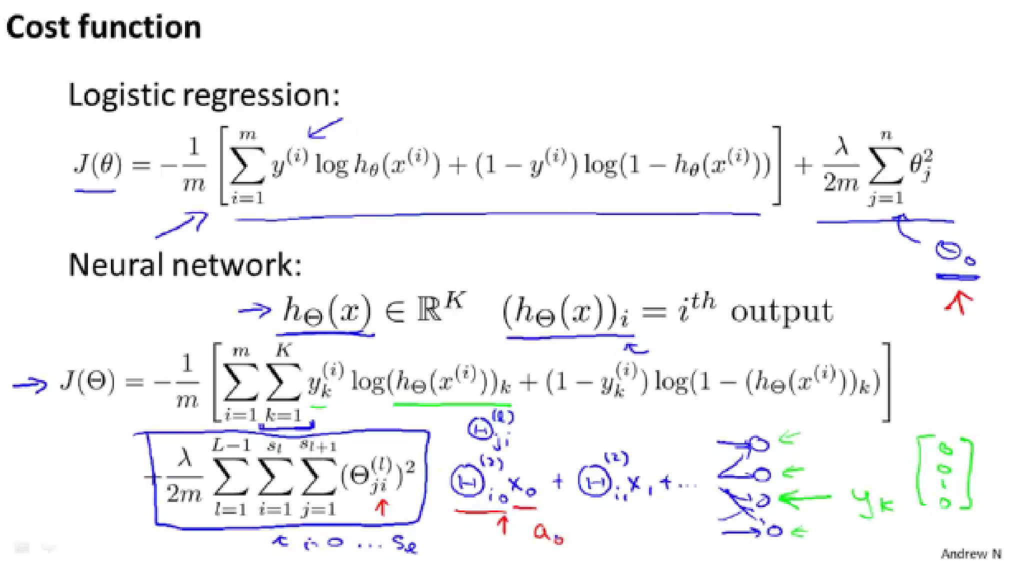
Recall that the cost function for regularized logistic regression was:

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

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:

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



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:

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

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

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Θ_0后，每一层的Θ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 Θs 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：

$$a^{(1)} = x \tag{3}$$
$$z^{(2)} = \Theta^{(1)}a^{(1)} \tag{4}$$
$$a^{(2)} = g(z^{(2)})(add \quad a_0^{(2)}) \tag{5}$$
$$z^{(3)} = \Theta^{(2)}a^{(2)} \tag{6}$$
$$a^{(3)} = g(z^{(3)})(add \quad a_0^{(3)}) \tag{7}$$
$$z^{(4)} = \Theta^{(3)}a^{(3)} \tag{8}$$
$$a^{(4)} = h_\Theta(x) = g(z^{(4)}) \tag{9}$$

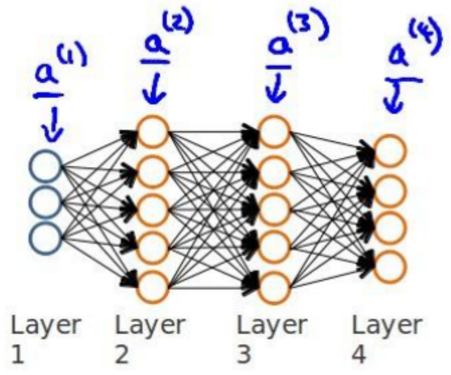


图 3: 前向传播算法求激励函数

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

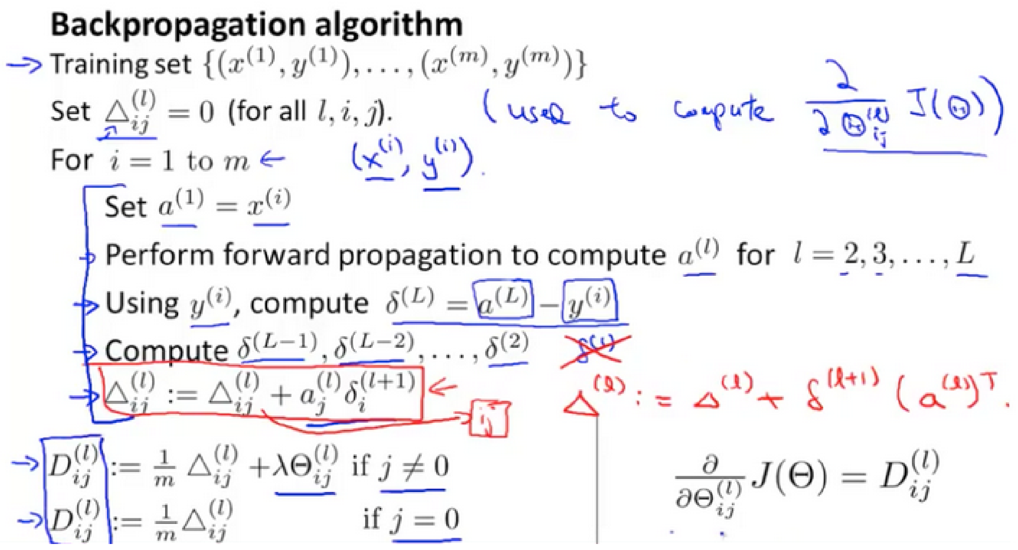
”Backpropagation” is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute:

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

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 $J(\Theta)$:

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

To do so, we use the following algorithm:



Backpropagation algorithm

→ Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

Set $\triangle_{ij}^{(l)} = 0$ (for all l, i, j). $\left(\text{use to compute } \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)\right)$

For $i = 1$ to m ← $(x^{(i)}, y^{(i)})$.

→ Set $a^{(1)} = x^{(i)}$

→ Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$

→ Using $y^{(i)}$, compute $\delta^{(L)} = \boxed{a^{(L)}} - \boxed{y^{(i)}}$

→ Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ $\cancel{\delta^{(1)}}$

→ $\boxed{\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}}$ ← $\triangle^{(l)} := \triangle^{(l)} + \delta^{(l+1)} (a^{(l)})^T.$

→ $D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ if $j \neq 0$

→ $D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)}$ if $j = 0$

$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$

图 4: 后向传播算法

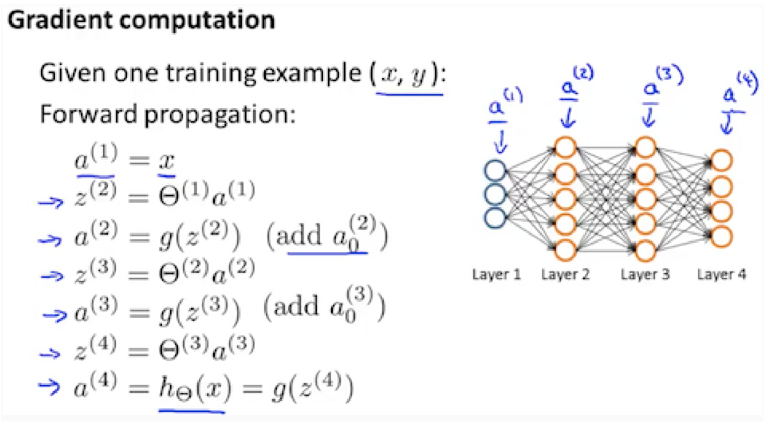
Given training set $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$

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



Gradient computation

Given one training example (x, y):

Forward propagation:

$a^{(1)} = x$

→ $z^{(2)} = \Theta^{(1)} a^{(1)}$

→ $a^{(2)} = g(z^{(2)})$ (add $a_0^{(2)}$)

→ $z^{(3)} = \Theta^{(2)} a^{(2)}$

→ $a^{(3)} = g(z^{(3)})$ (add $a_0^{(3)}$)

→ $z^{(4)} = \Theta^{(3)} a^{(3)}$

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

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）Compute $\delta^{(L-1)}$,$\delta^{(L-2)}$,...,$\delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T\delta^{(l+1)}).*a^{(l)}.*(1-a^{(l)})$

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:

$$g'(z^{(l)}) = a^{(l)}.*(1-a^{(l)}) \tag{12}$$

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

Hence we update our new Δ matrix.

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

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

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):

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

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

我们用δ来表示误差，则（向量化）：

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

向量的维度：单元数×1

我们利用这个误差来计算前一层的误差：

$$\delta^{(3)} = (\Theta^{(3)})^T\delta^{(4)}.*g'(z^{(3)}) \tag{17}$$

其中$g'(z^{(3)})$是S形函数的导数：

$$g'(z^{(3)}) = a^{(3)}.*(1-a^{(3)}) \tag{18}$$

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：

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} . * g'(z^{(2)}) \tag{19}$$

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：

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

注释：

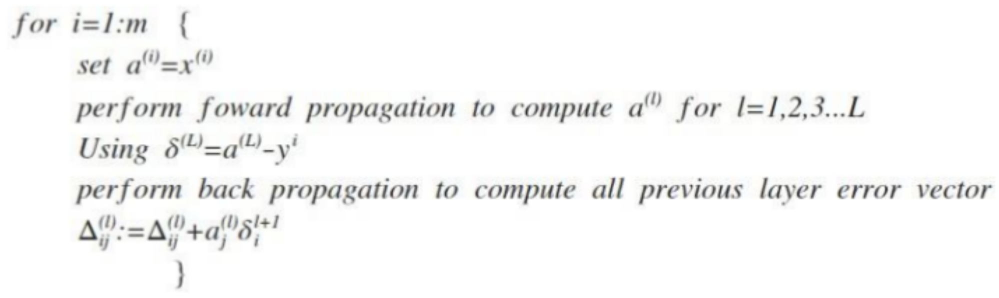
l=目前所计算的是第几层；

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 foward 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
    $\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)}\delta_i^{l+1}$
        }
```

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，我们便可以计算低价函数的偏导数了，计算方法如下：

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

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

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，Theat2，Theat3，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));
Theta1=reshape(thetaVec(221:231,1,11));
```

1.3 Backpropagation Intuition

2 Backpropagation in Practice

2.1 Implementation Note:Unrolling Parameters

2.2 Gradient Checking

2.3 Random Initialization

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。同理,如果我们初始所有的参数都为一个非 0 的数,结果也是一样的。

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

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

2.4 Putting it Together

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。

训练神经网络:

（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