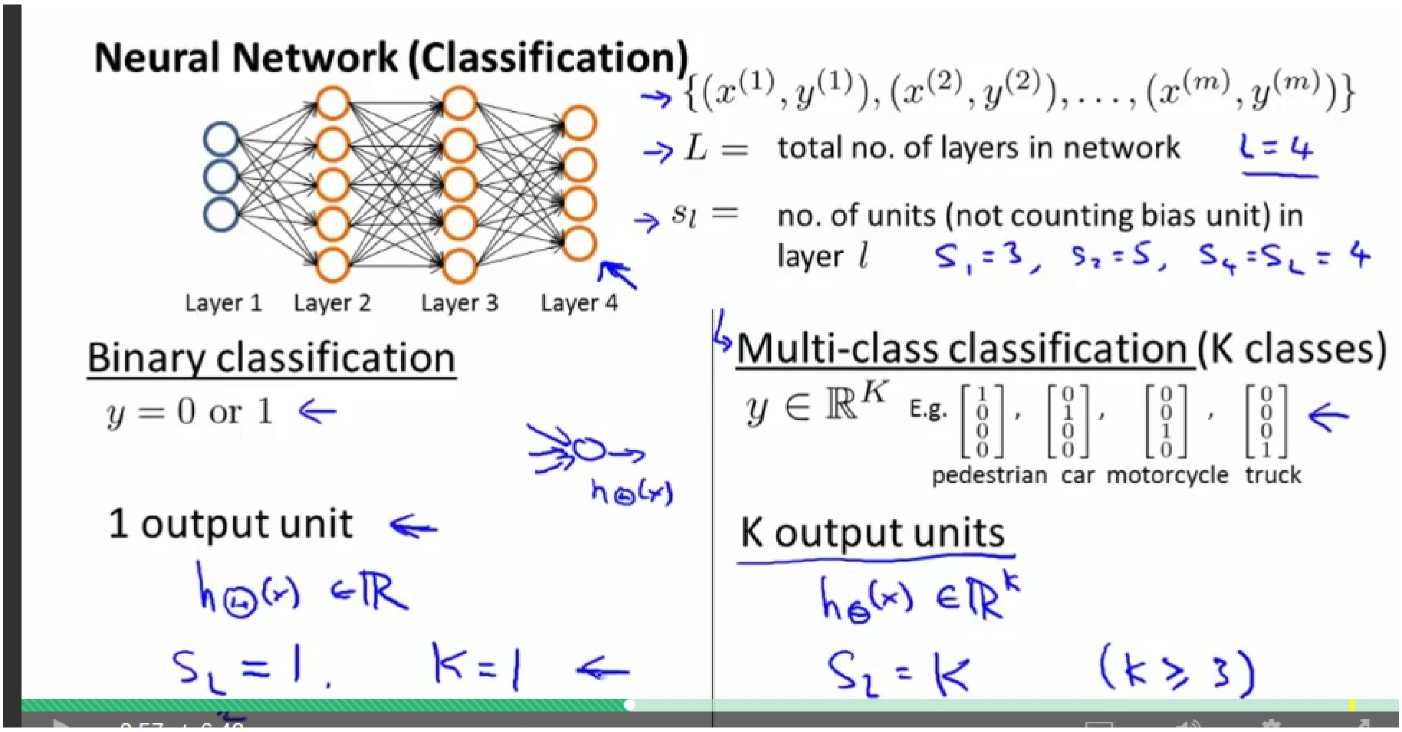
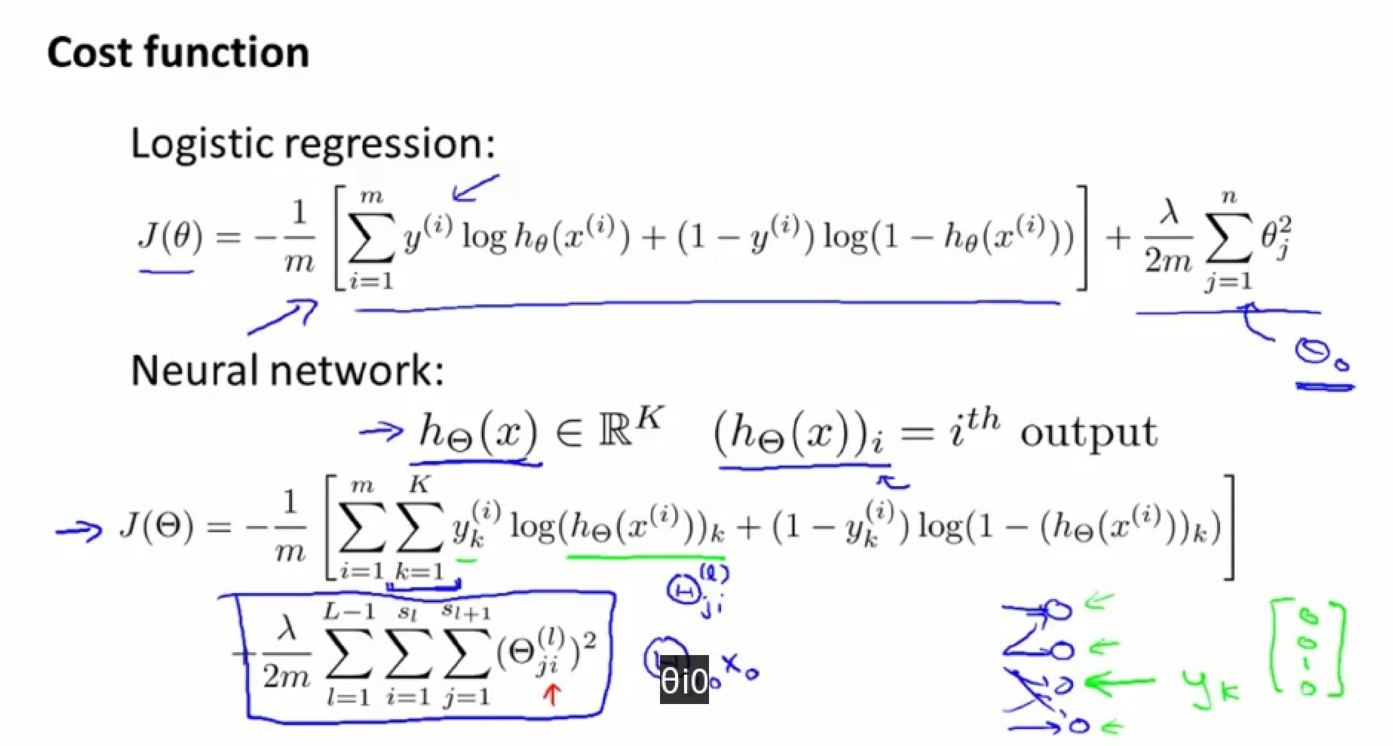
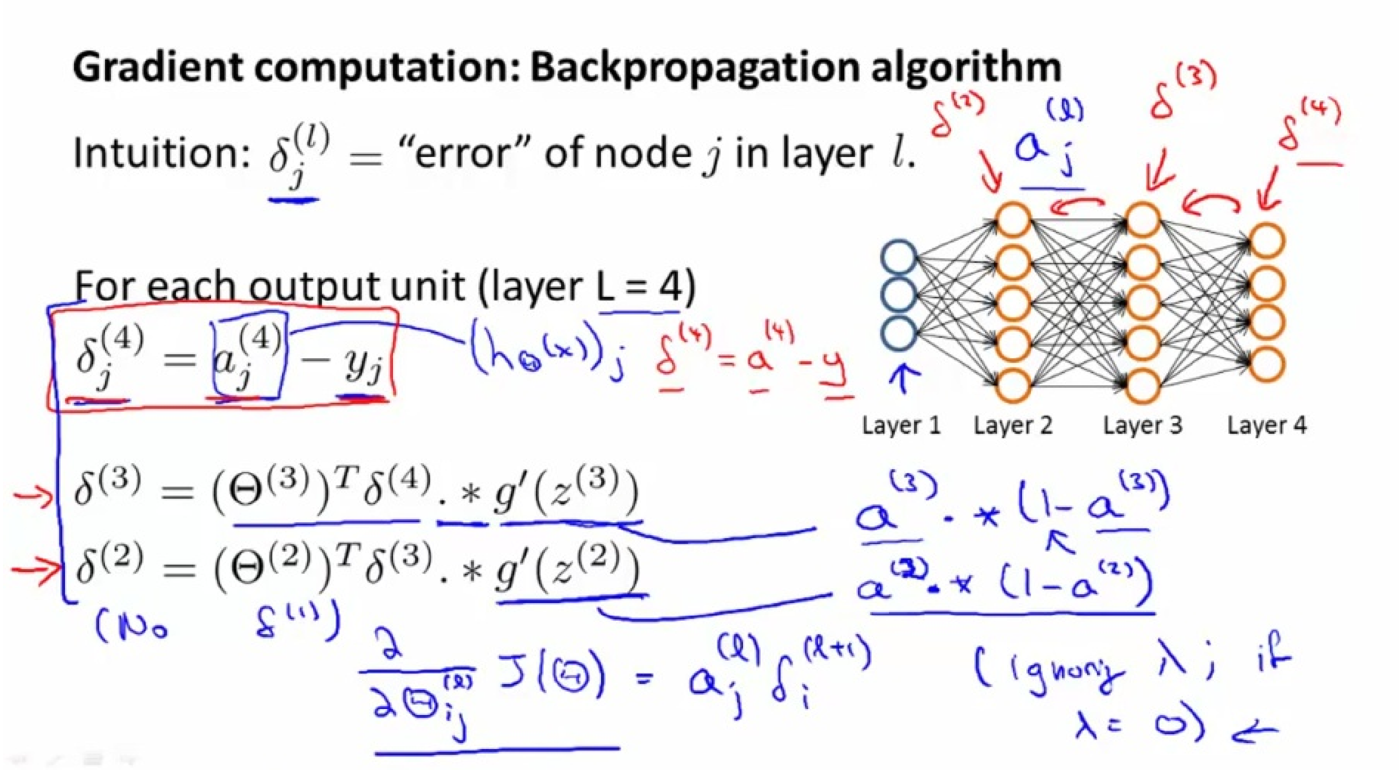
1. 神经网络符号约定：L=网络总层数; sl=网络中单元数。

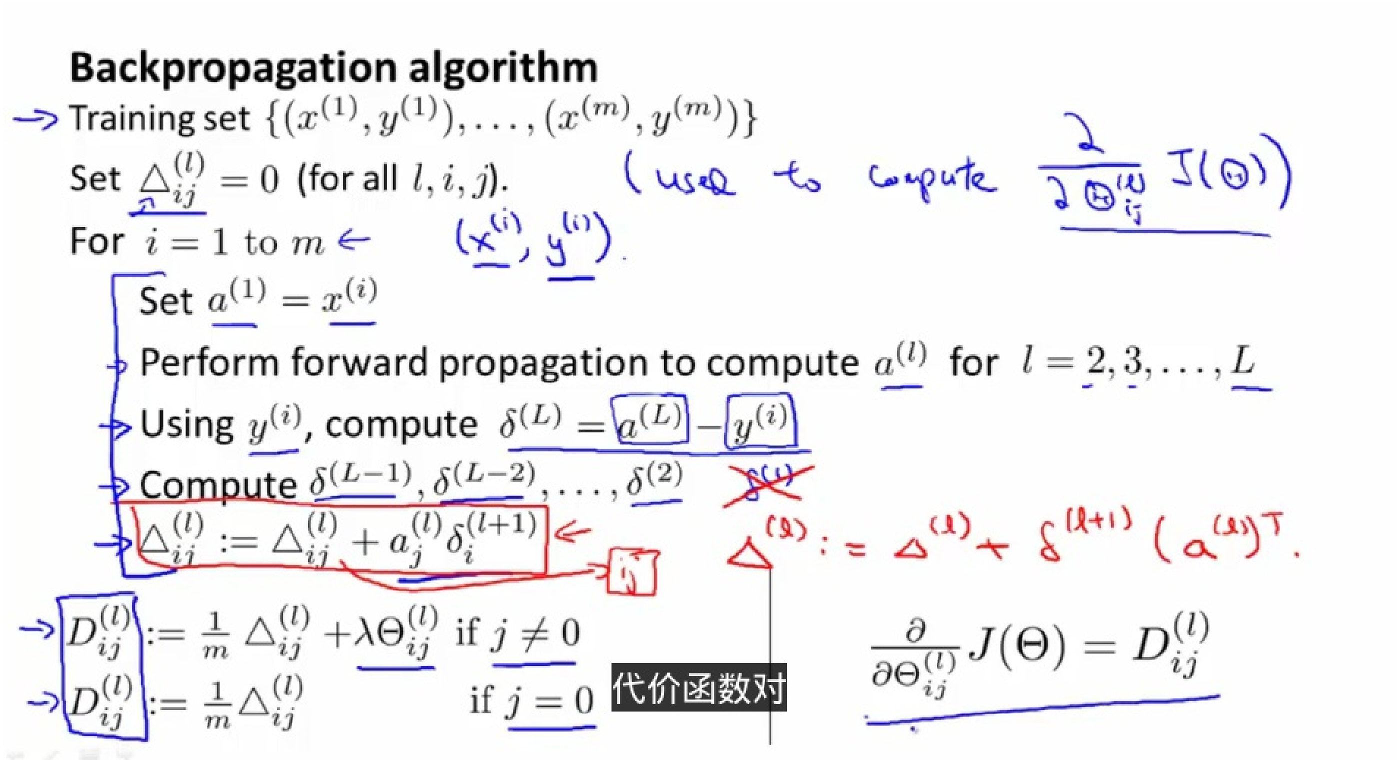


K输出单元代价函数：



2. 反向传播(Backpropagation)算法

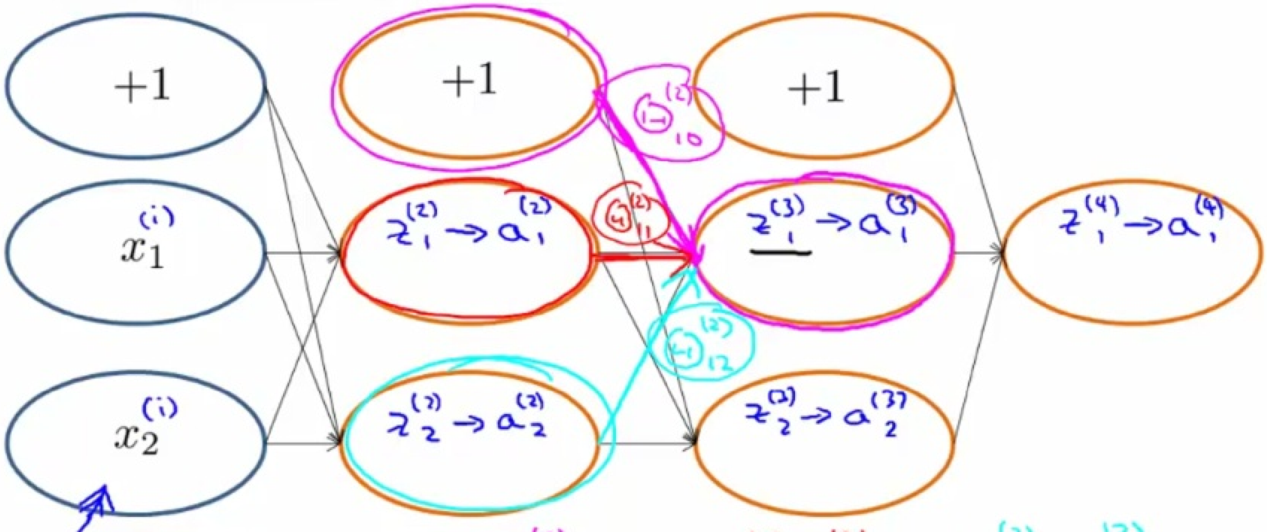




可以看出来，it is really 复杂.

直观理解：前向传播：

Forward Propagation



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

$$z_1^{(3)} = \Theta_{10}^{(2)} \times 1 + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} \cdot a_1^{(2)}$$

What is backpropagation doing?

What is backpropagation doing?

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

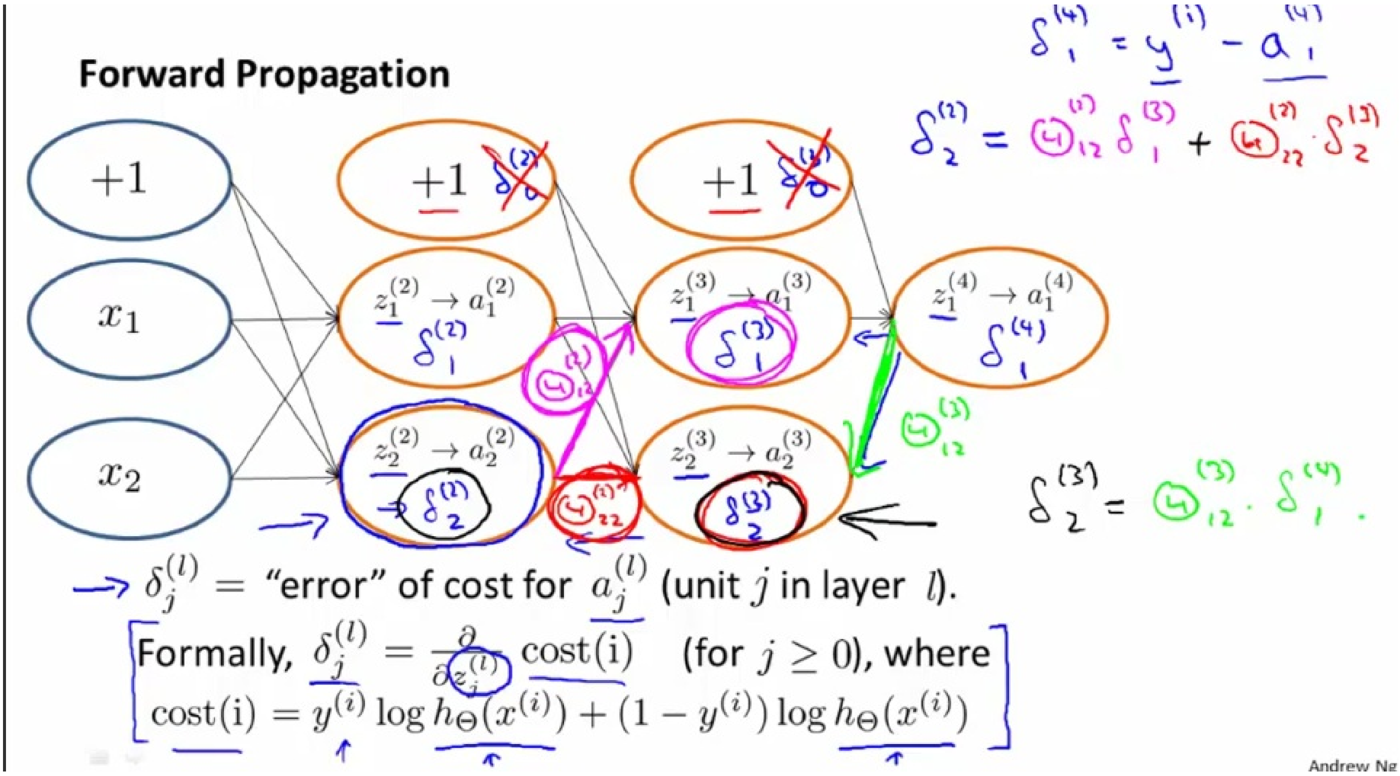
Focusing on a single example $x^{(i)}$, $y^{(i)}$, the case of 1 output unit, and ignoring regularization ($\lambda = 0$),

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

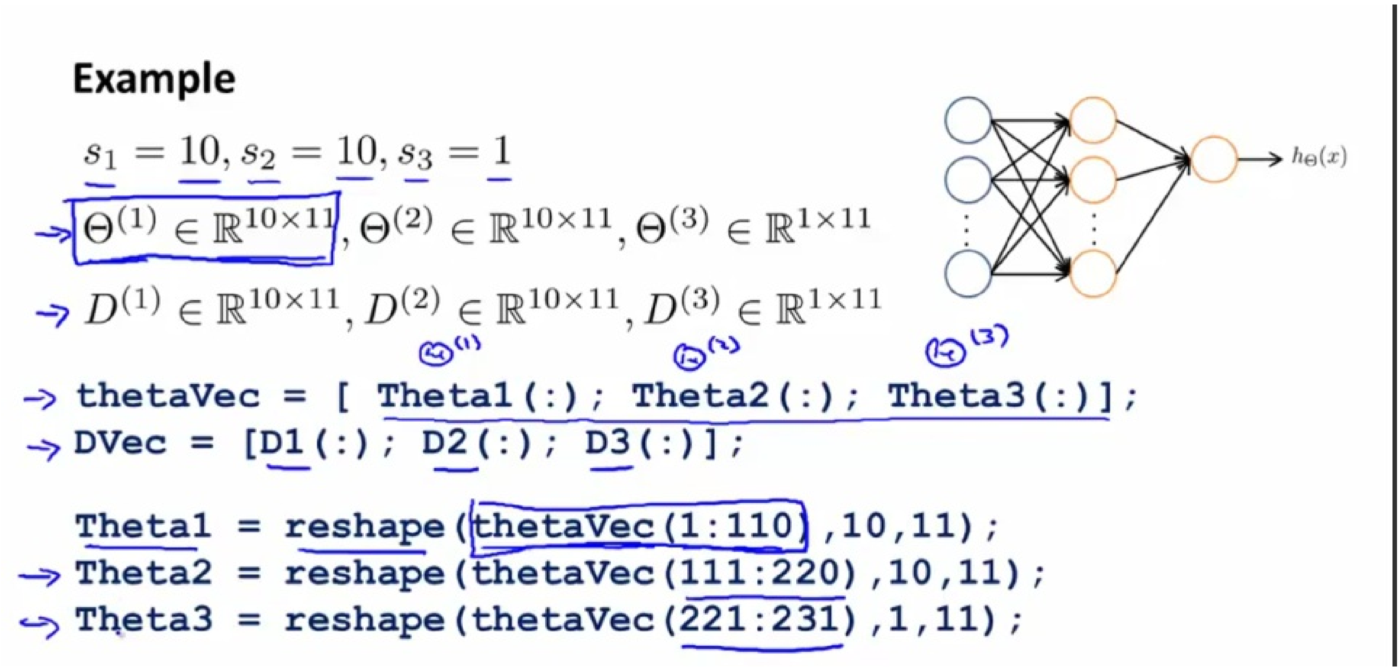
(Think of $\text{cost(i)} \approx (\boxed{h_\Theta(x^{(i)})} - \boxed{y^{(i)}})^2$)

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

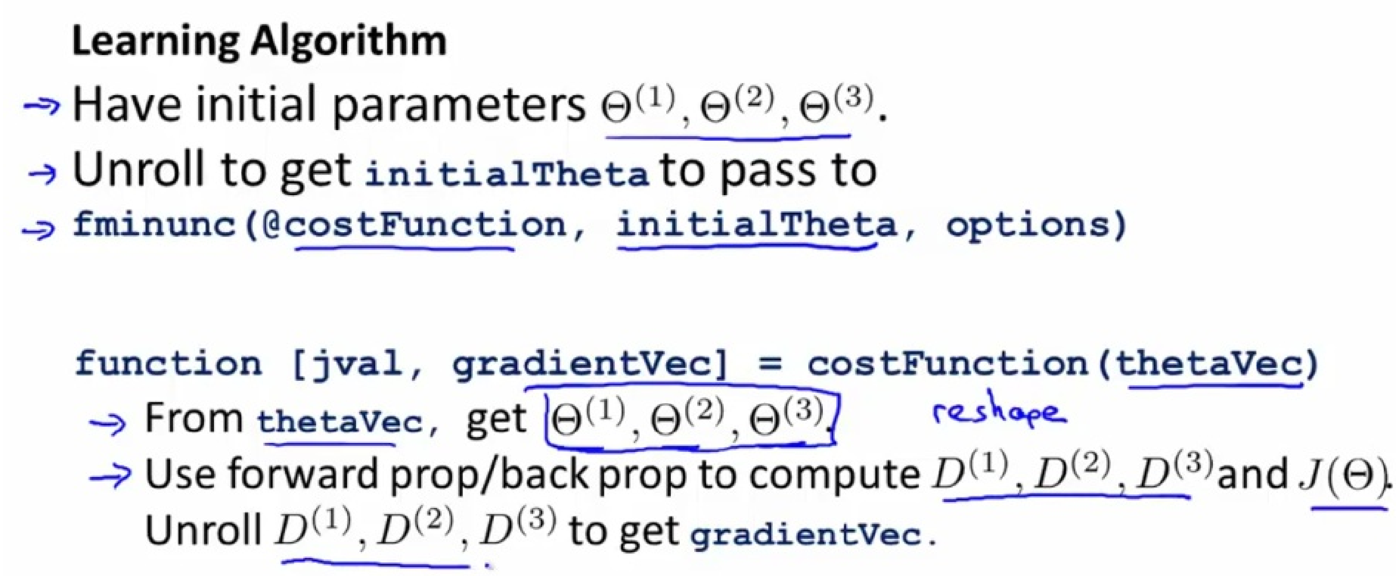
差的平方



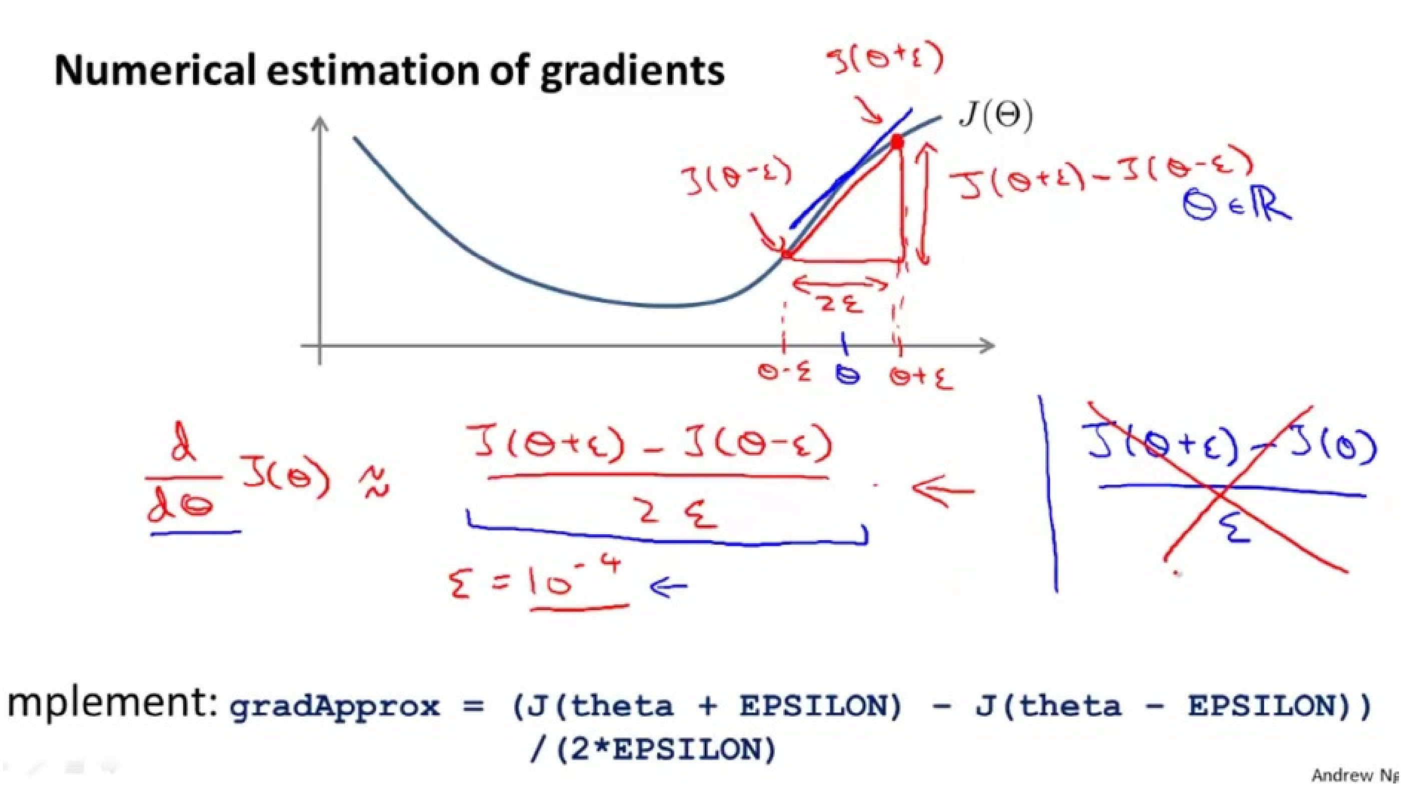
高级优化：把矩阵展开成向量：展开成列向量，还原时候使用reshape还原。



学习参数的过程：



梯度检验（Gradient Checking）：估计导数：



向量形式：

Parameter vector θ

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

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

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

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

$$\vdots$$

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

然后除以2ε

代码：（比较反向传播方法得到的梯度和估计的梯度）

```
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;
```

$\begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_i+\epsilon \\ \vdots \\ \theta_n \end{bmatrix} \rightarrow \theta_i - \epsilon$

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

Check that gradApprox ≈ DVec

From backprop.

使用反向传播的方法计算导数要比估计导数的方法快的多，所以在检验梯度计算的是正确之后，在实现中要关闭梯度估计，否则代码会运行的慢的多。

Implementation Note:

- Implement backprop to compute `DVec` (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$).
- Implement numerical gradient check to compute $\boxed{\texttt{gradApprox.}}$
- Make sure they give similar values.
- Turn off gradient checking. Using backprop code for learning.

$\to$ DVec

$\delta^{(4)}, \delta^{(3)}, \delta^{(2)}$

Important:

- Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent (or in the inner loop of `costFunction(…)`)your code will be very slow.

初始化参数的选择：随机初始化

Initial value of Θ

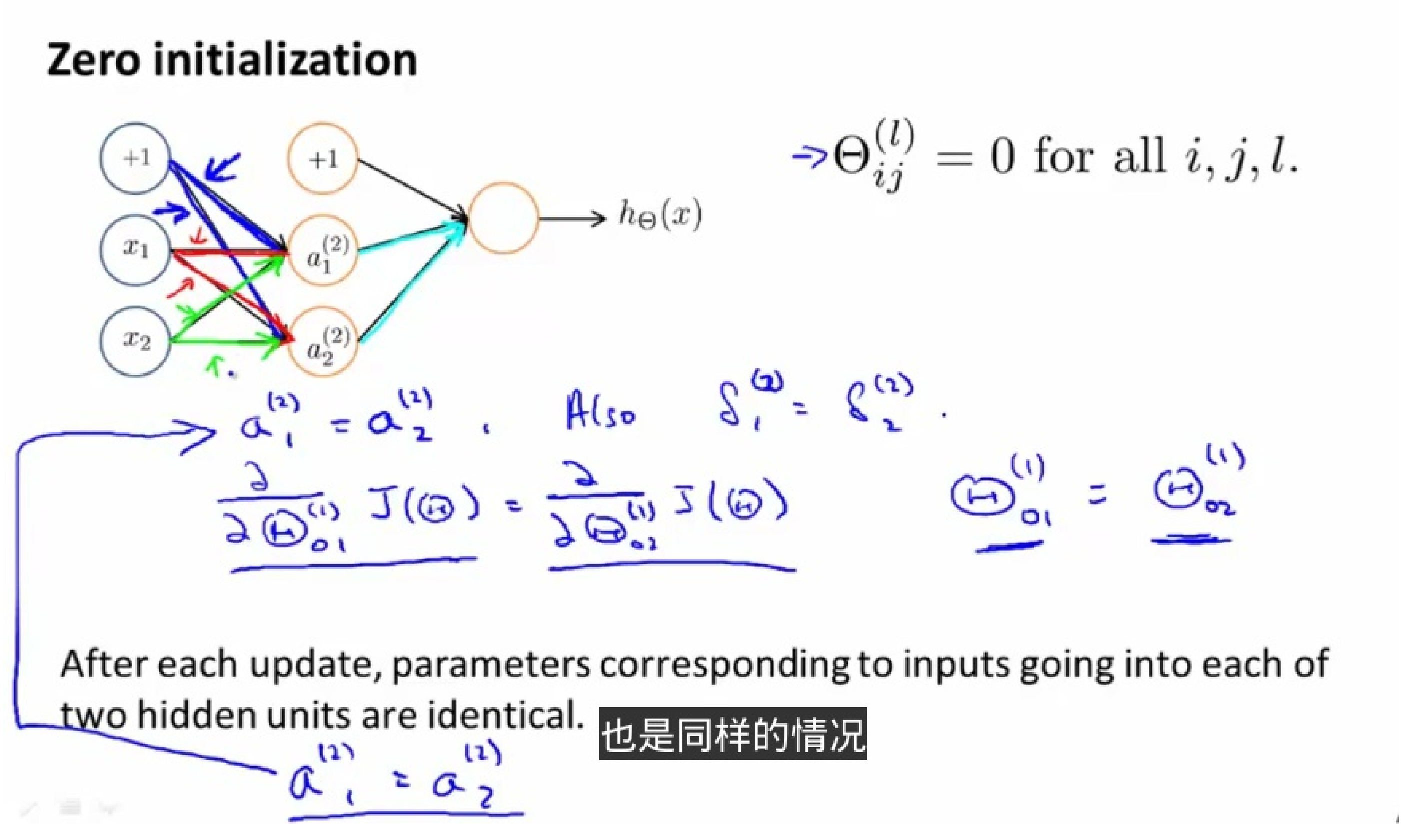
For gradient descent and advanced optimization method, need initial value for Θ.

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

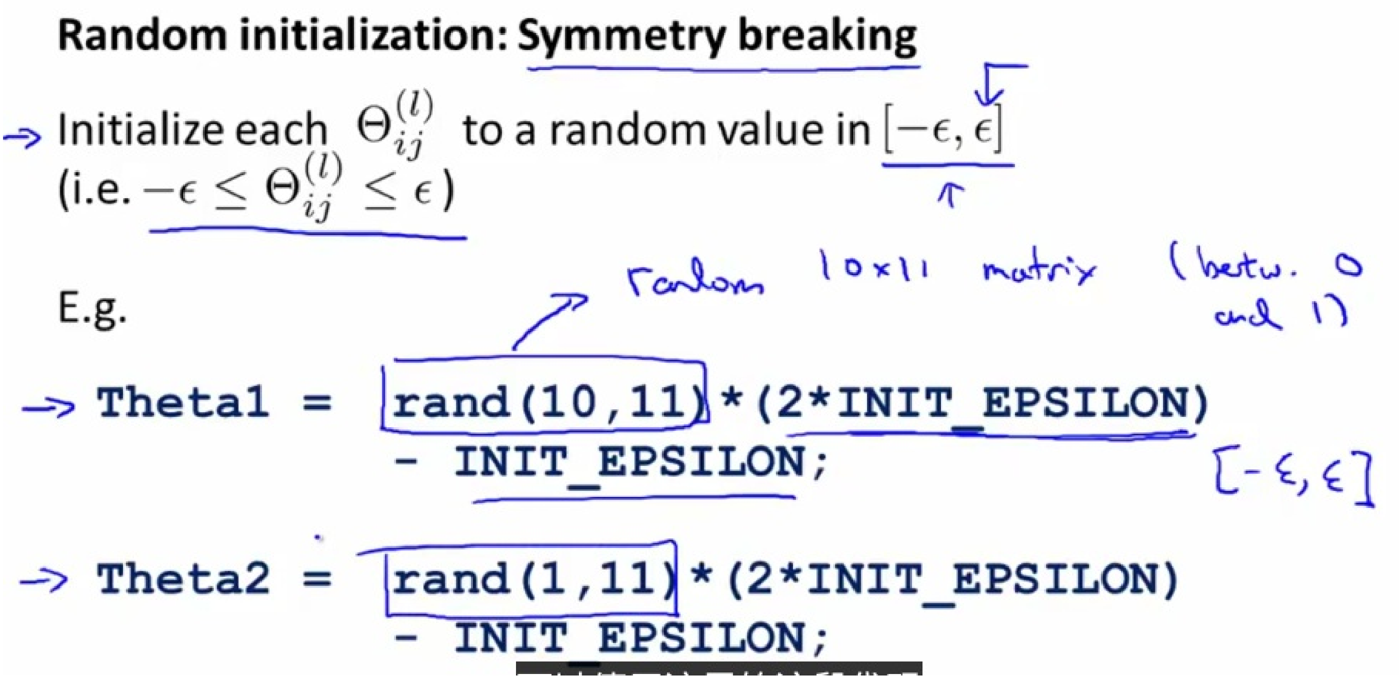
Consider gradient descent

Set `initialTheta = zeros(n,1)` ?

虽然在逻辑回归的方法中可以初始化theta为全0，但是训练神经网络的时候，不能初始化权重为0.



使用随机初始化的方式：----打破权重对称性（相同权重的情况）



注意这里的epsilon和梯度检验中的epsilon没有关系。

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

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

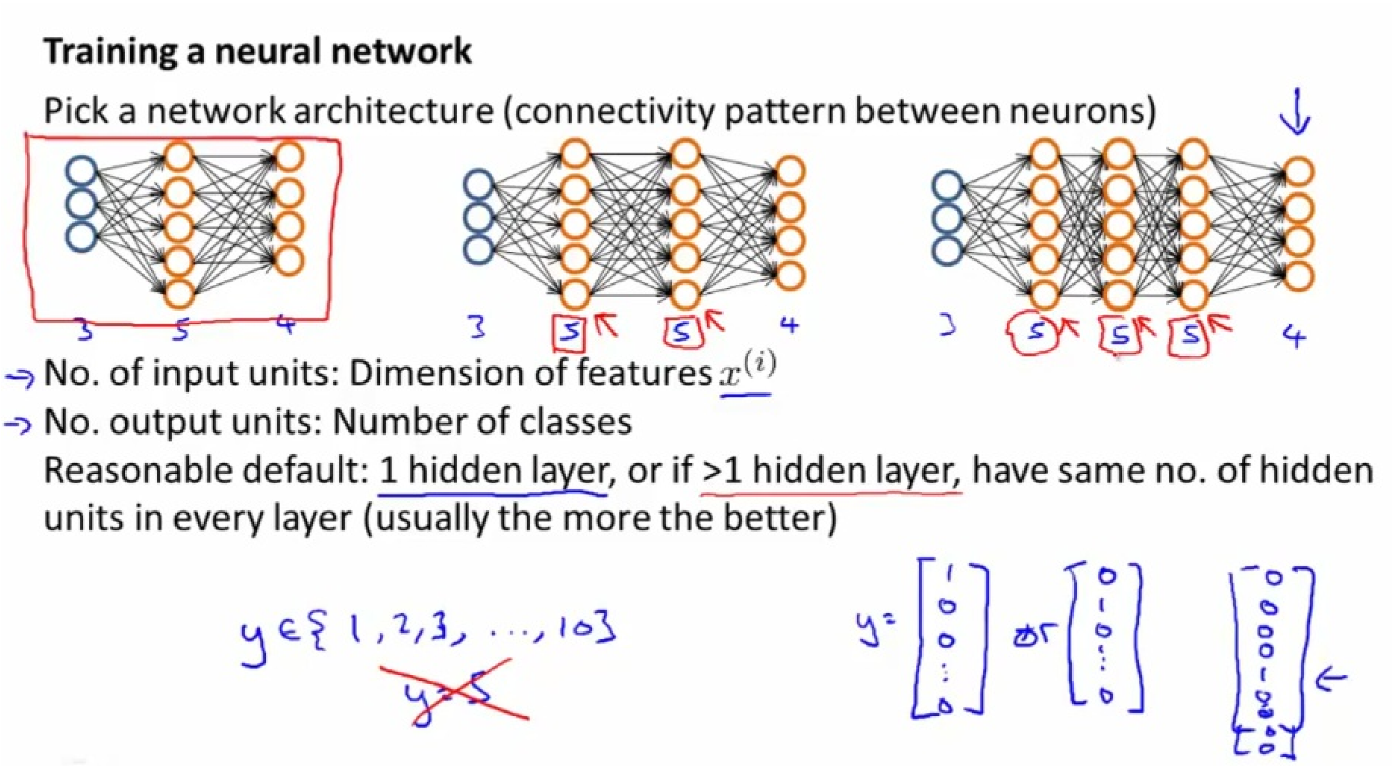
Does this work?

- ○ 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.

Correct Response

训练神经网络过程综述：

（1）选择网络架构：



Training a neural network

1. Randomly initialize weights
2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
3. Implement code to compute cost function $J(\Theta)$
4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial \Theta^{(l)}_{jk}} J(\Theta)$

```
for i = 1:m {
```
$(x^{(1)}, y^{(1)})$ $(x^{(2)}, y^{(2)})$, …… $(x^{(m)}, y^{(m)})$

Perform forward propagation and backpropagation using example $(x^{(i)}, y^{(i)})$

(Get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, \ldots, L$).

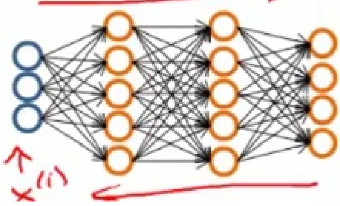
$\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

....

}

...

compute $\frac{\partial}{\partial \Theta^{(l)}_{jk}} J(\Theta)$.



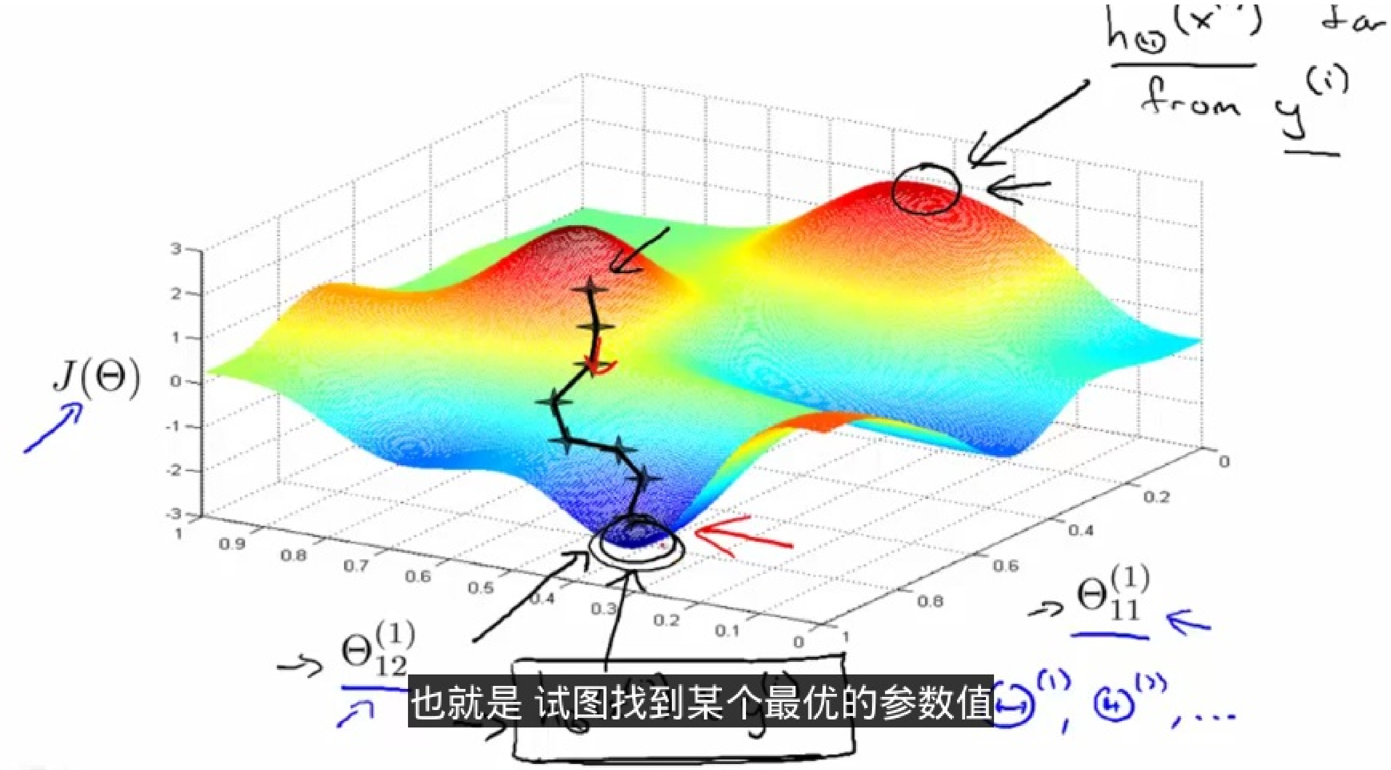
Training a neural network

5. Use gradient checking to compare $\frac{\partial}{\partial \Theta^{(l)}_{jk}} J(\Theta)$ computed using backpropagation vs. using numerical estimate of gradient of $J(\Theta)$.

   Then disable gradient checking code.
6. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters Θ

$\frac{\partial}{\partial \Theta^{(l)}_{jk}} J(\Theta)$

$J(\Theta)$ — non-convex.



反向传播的实现：

```
for i  = 1:m
    a1 = [1; X(i,:)'];
    z2 = Theta1 * a1;
    a2 = [1; sigmoid(z2)];
    z3 = Theta2 * a2;
    a3 = sigmoid(z3);
    yy = zeros(num_labels, 1);
    yy(y(i)) = 1;
    delta3 = a3 - yy;
    delta2 = Theta2(:,2:end)'*delta3.*sigmoidGradient(z2);
    Theta2_grad = Theta2_grad + delta3*(a2');
    Theta1_grad = Theta1_grad + delta2*(a1');
end
Theta1_grad = Theta1_grad ./ m;
Theta2_grad = Theta2_grad ./ m;
```

完整代价函数代码：

```
function [J grad] = nnCostFunction(nn_params, ...
                                   input_layer_size, ...
                                   hidden_layer_size, ...
                                   num_labels, ...
                                   X, y, lambda)
%NNCOSTFUNCTION Implements the neural network cost function for a two layer
%neural network which performs classification
%   [J grad] = NNCOSTFUNCTON(nn_params, hidden_layer_size, num_labels, ...
%   X, y, lambda) computes the cost and gradient of the neural network. The
%   parameters for the neural network are "unrolled" into the vector
%   nn_params and need to be converted back into the weight matrices.
%
%   The returned parameter grad should be a "unrolled" vector of the
%   partial derivatives of the neural network.
%

% Reshape nn_params back into the parameters Theta1 and Theta2, the weight matrices
% for our 2 layer neural network
Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), ...
                 hidden_layer_size, (input_layer_size + 1));

Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size + 1))):end), ...
                 num_labels, (hidden_layer_size + 1));

% Setup some useful variables
m = size(X, 1);
```

```

% You need to return the following variables correctly
J = 0;
Theta1_grad = zeros(size(Theta1));
Theta2_grad = zeros(size(Theta2));

% ====================== YOUR CODE HERE ======================
% Instructions: You should complete the code by working through the
%               following parts.
%
% Part 1: Feedforward the neural network and return the cost in the
%         variable J. After implementing Part 1, you can verify that your
%         cost function computation is correct by verifying the cost
%         computed in ex4.m
%
% Part 2: Implement the backpropagation algorithm to compute the gradients
%         Theta1_grad and Theta2_grad. You should return the partial derivatives of
%         the cost function with respect to Theta1 and Theta2 in Theta1_grad and
%         Theta2_grad, respectively. After implementing Part 2, you can check
%         that your implementation is correct by running checkNNGradients
%
%         Note: The vector y passed into the function is a vector of labels
%               containing values from 1..K. You need to map this vector into a
%               binary vector of 1's and 0's to be used with the neural network
%               cost function.
%
%         Hint: We recommend implementing backpropagation using a for-loop
%               over the training examples if you are implementing it for the
%               first time.
%
% Part 3: Implement regularization with the cost function and gradients.
%
%         Hint: You can implement this around the code for
%               backpropagation. That is, you can compute the gradients for
%               the regularization separately and then add them to Theta1_grad
%               and Theta2_grad from Part 2.
%

a2 = sigmoid([ones(m, 1), X]*Theta1');
h = sigmoid([ones(m, 1), a2]*Theta2');
for k = 1:num_labels
    J = J - [(y==k)',((1-(y==k)))']*[log(h(:,k));log(1-h(:,k))]./m;
end

t1 = ones(hidden_layer_size, input_layer_size);
t2 = ones(num_labels, hidden_layer_size);
t1 = Theta1(:,2:end).^2;
t2 = Theta2(:,2:end).^2;
J = J + (sum(t1(:))+sum(t2(:)))*lambda/(2*m);

a1     = zeros(input_layer_size+1, 1);
z2     = zeros(hidden_layer_size, 1);
a2     = zeros(hidden_layer_size+1, 1);
z3     = zeros(num_labels, 1);
a3     = zeros(num_labels, 1);
yy     = zeros(num_labels, 1);
delta3 = zeros(num_labels, 1);
delta2 = zeros(hidden_layer_size, 1);

for i  = 1:m
    a1 = [1; X(i,:)'];
    z2 = Theta1 * a1;
    a2 = [1; sigmoid(z2)];
    z3 = Theta2 * a2;
    a3 = sigmoid(z3);
    yy = zeros(num_labels, 1);
    yy(y(i)) = 1;
    delta3 = a3 - yy;
    delta2 = Theta2(:,2:end)'*delta3.*sigmoidGradient(z2);
    Theta2_grad = Theta2_grad + delta3*(a2');
    Theta1_grad = Theta1_grad + delta2*(a1');
end
Theta1_grad = Theta1_grad ./ m;
Theta2_grad = Theta2_grad ./ m;

```

```
Theta1_grad(:,2:end) = Theta1_grad(:,2:end) + Theta1(:,2:end).*lambda./m;
Theta2_grad(:,2:end) = Theta2_grad(:,2:end) + Theta2(:,2:end).*lambda./m;
% -------------------------------------------------------------

% =========================================================================

% Unroll gradients
grad = [Theta1_grad(:) ; Theta2_grad(:)];


end
```