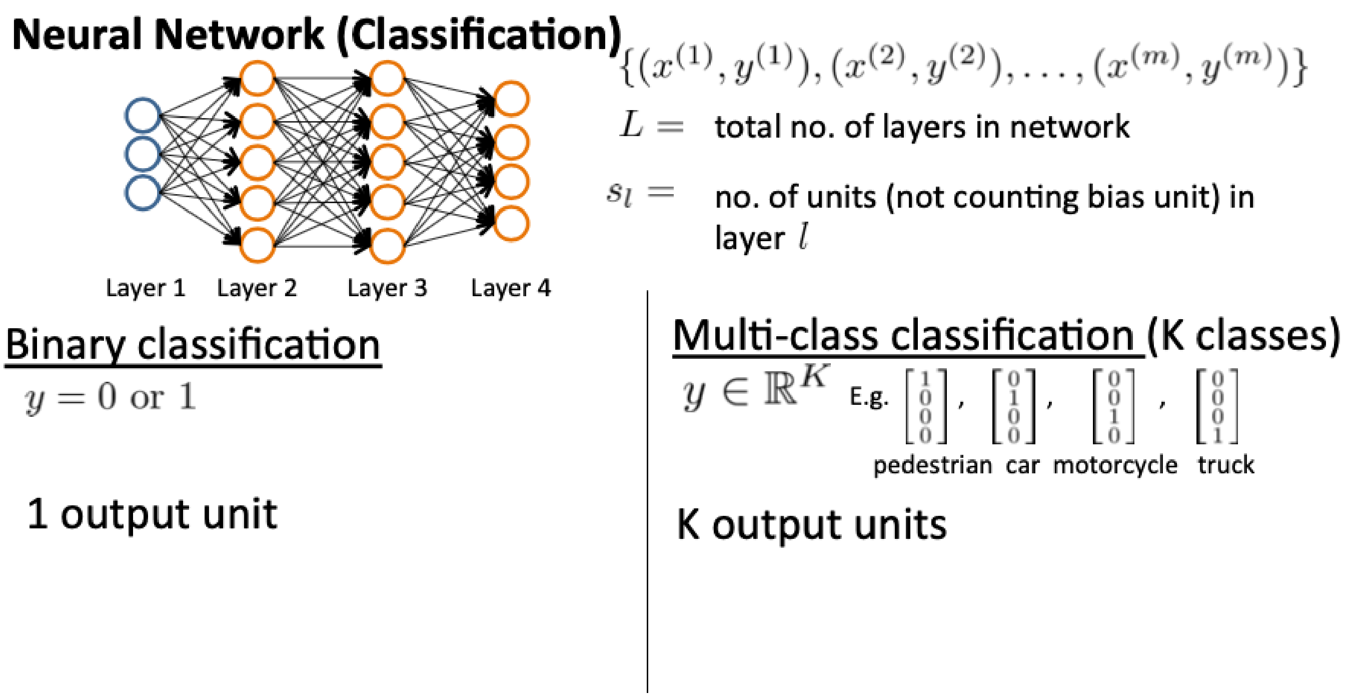
Week5

Table of Contents

Cost Function



Neural Network (Classification)

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

$L =$ total no. of layers in network

$s_l =$ no. of units (not counting bias unit) in layer l

Layer 1 Layer 2 Layer 3 Layer 4

Binary classification

$y = 0 \text{ or } 1$

1 output unit

Multi-class classification (K classes)

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

K output units

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

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:

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

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

Backpropagation Algorithms

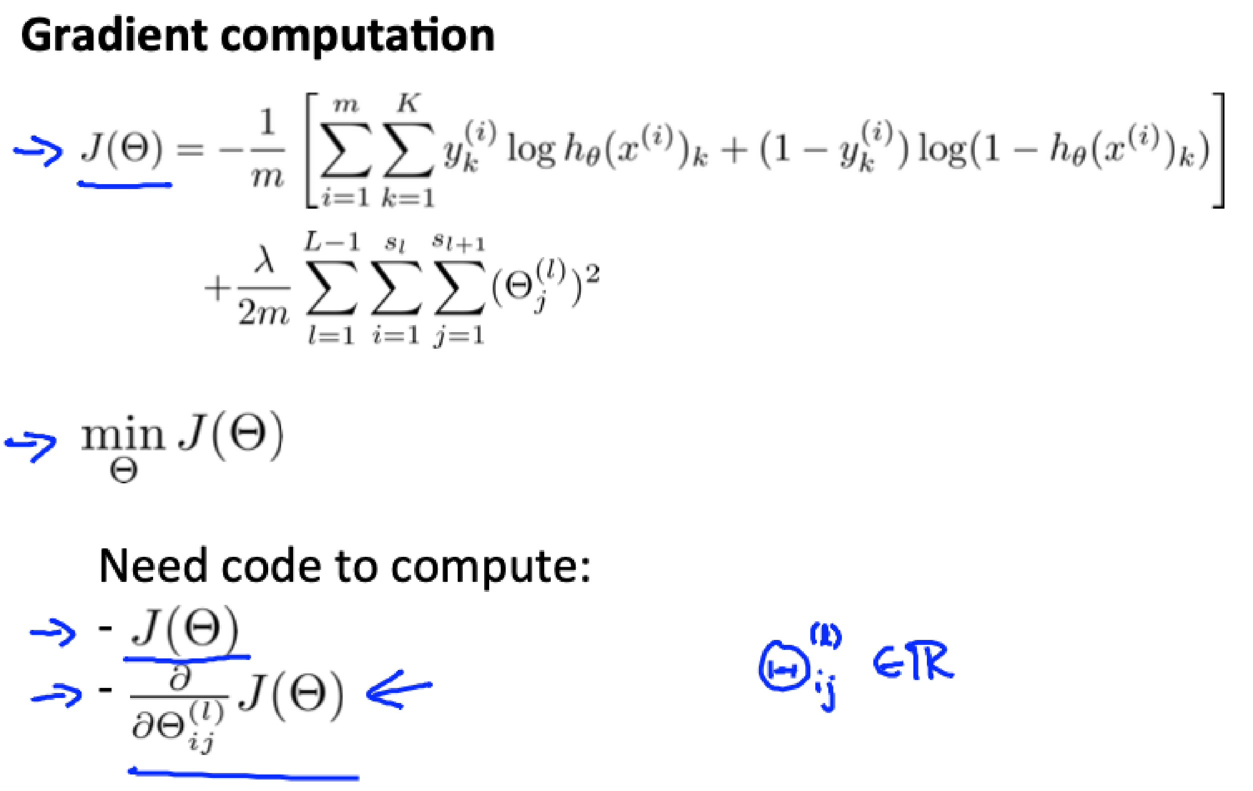
"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression.

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):

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



Gradient computation

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

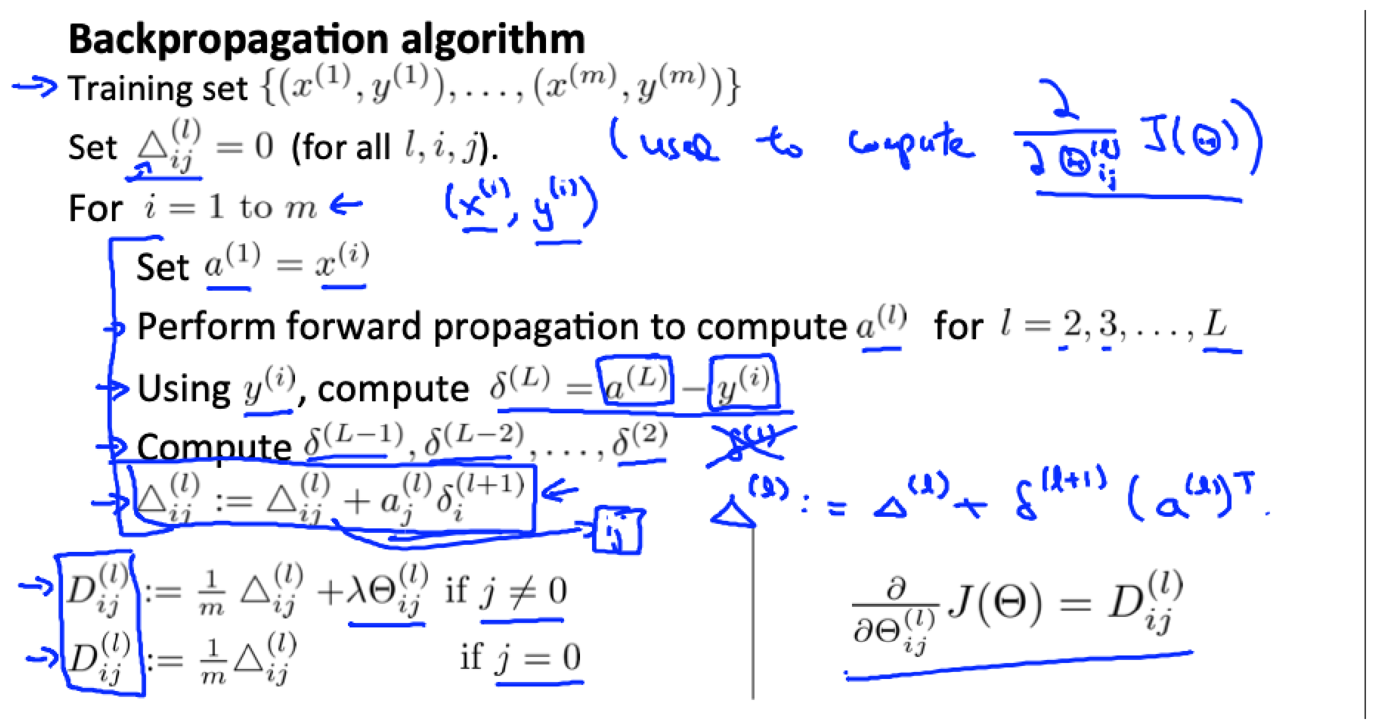
$$\rightarrow \min_{\Theta} J(\Theta)$$

Need code to compute:

- $\rightarrow$ - $\underline{J(\Theta)}$
- $\rightarrow$ - $\underline{\frac{\partial}{\partial\Theta_{ij}^{(l)}} J(\Theta)} \leftarrow$

$\Theta_{ij}^{(l)} \in \mathbb{R}$

We use following algorithms to achieve our goals.



Backpropagation algorithm

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

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

```
For $i = 1$ to $m$ $\leftarrow$ $(\underline{x^{(i)}}, \underline{y^{(i)}})$
    Set $\underline{a^{(1)}} = \underline{x^{(i)}}$
    $\rightarrow$ Perform forward propagation to compute $\underline{a^{(l)}}$ for $l = 2,3,\ldots,L$
    $\rightarrow$ Using $\underline{y^{(i)}}$, compute $\underline{\delta^{(L)} = \boxed{a^{(L)}} - \boxed{y^{(i)}}}$
    $\rightarrow$ Compute $\underline{\delta^{(L-1)}}, \underline{\delta^{(L-2)}}, \ldots, \underline{\delta^{(2)}}$  $\cancel{\delta^{(1)}}$
    $\rightarrow$ $\boxed{\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)}\delta_i^{(l+1)}} \leftarrow$ $\boxed{ij}$
```

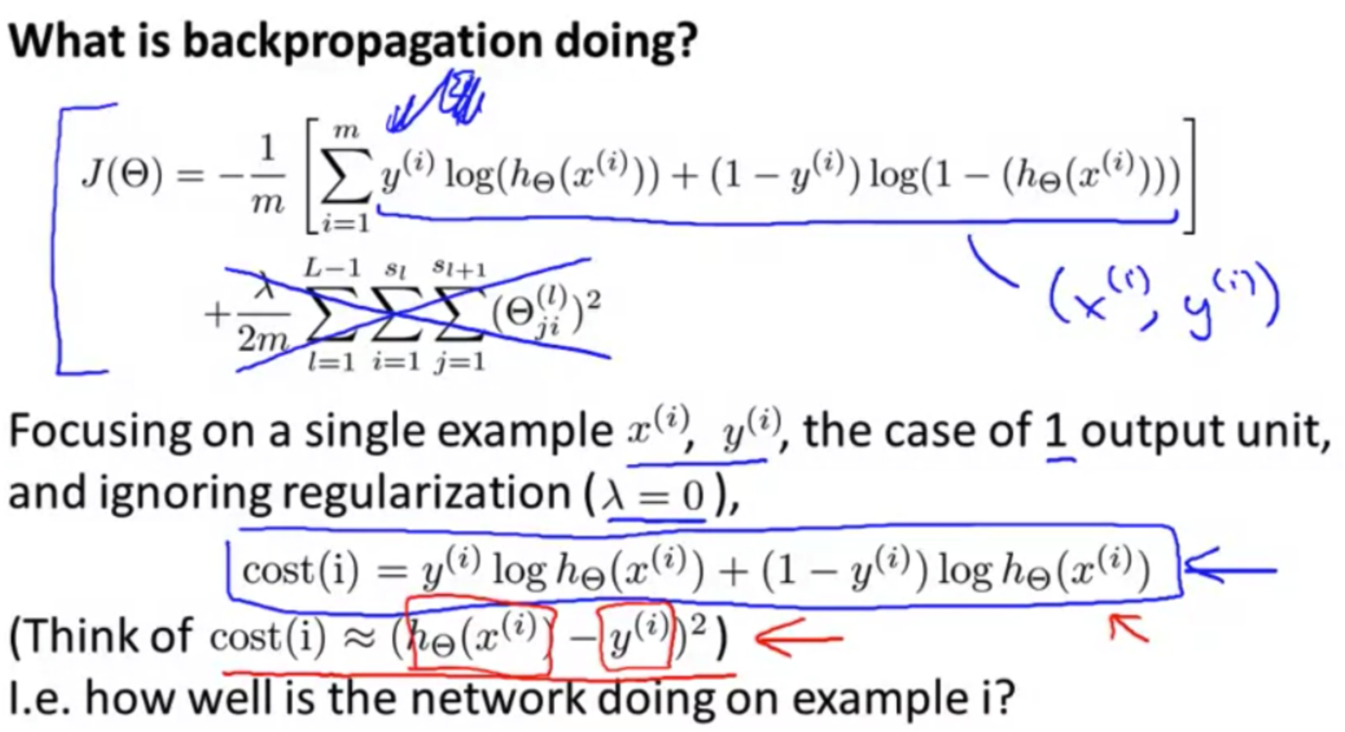
$\triangle^{(l)} := \triangle^{(l)} + \delta^{(l+1)}(a^{(l)})^T$

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

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

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

Intuition



What is backpropagation doing?

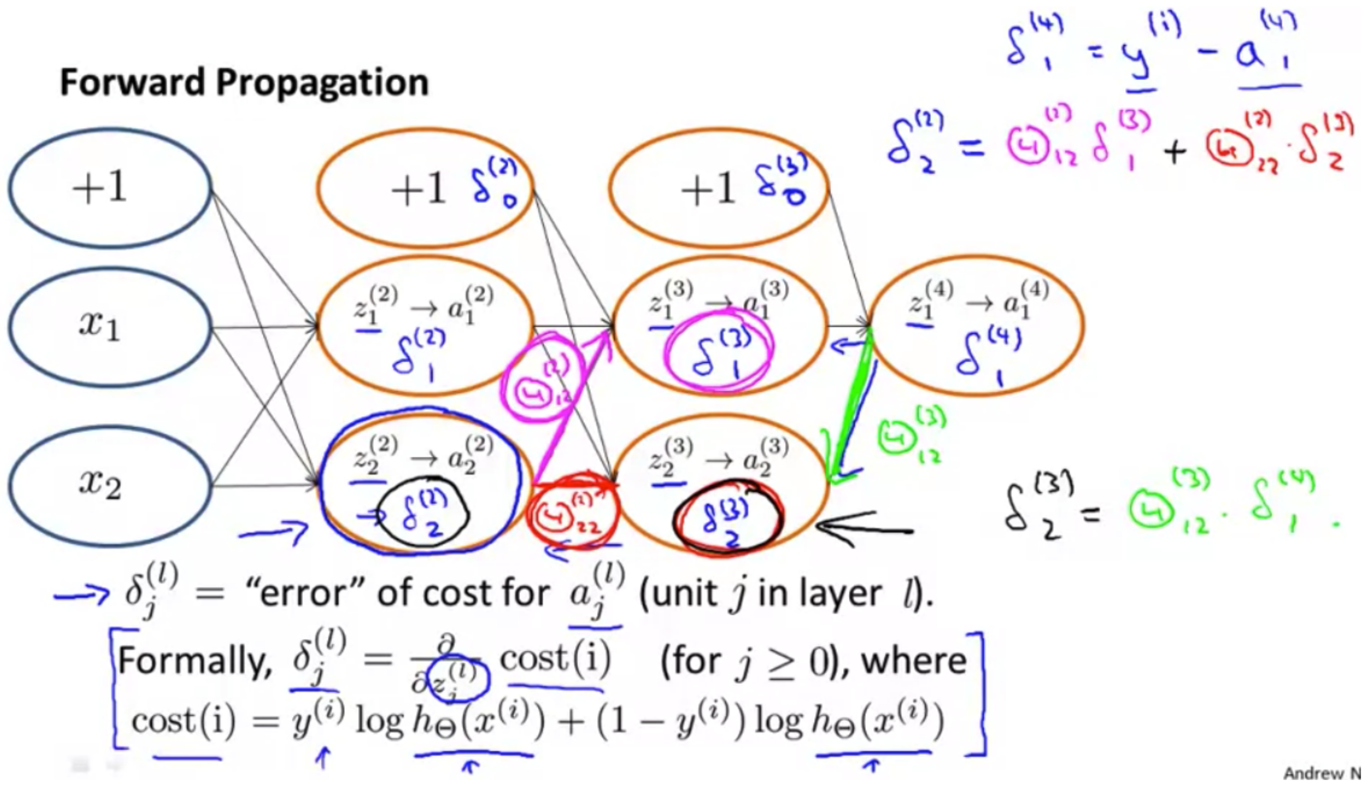
$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} \underbrace{y^{(i)}\log(h_\Theta(x^{(i)})) + (1-y^{(i)})\log(1-(h_\Theta(x^{(i)})))}_{(x^{(i)},\, y^{(i)})}\right]$$
$$\cancel{+\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2}$$

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

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

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

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



Forward Propagation
+1
x_1
x_2
+1 $\delta_0^{(2)}$
$z_1^{(2)} \to a_1^{(2)}$ $\delta_1^{(2)}$
$z_2^{(2)} \to a_2^{(2)}$ $\delta_2^{(2)}$
+1 $\delta_0^{(3)}$
$z_1^{(3)} \to a_1^{(3)}$ $\delta_1^{(3)}$
$z_2^{(3)} \to a_2^{(3)}$ $\delta_2^{(3)}$
$z_1^{(4)} \to a_1^{(4)}$ $\delta_1^{(4)}$
$\Theta_{12}^{(2)}$
$\Theta_{22}^{(2)}$
$\Theta_{12}^{(3)}$

$$\delta_1^{(4)} = \underline{y^{(i)}} - \underline{a_1^{(4)}}$$

$$\delta_2^{(2)} = \Theta_{12}^{(2)}\delta_1^{(3)} + \Theta_{22}^{(2)}\cdot\delta_2^{(3)}$$

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

$\rightarrow$ $\delta_j^{(l)}$ = "error" of cost for $\underline{a_j^{(l)}}$ (unit j in layer l).

Formally, $\underline{\delta_j^{(l)}} = \frac{\partial}{\partial z_j^{(l)}}\underline{\text{cost(i)}}$ (for $j \geq 0$), where
$\underline{\text{cost(i)}} = y^{(i)}\log \underline{h_\Theta(x^{(i)})} + (1-y^{(i)})\log \underline{h_\Theta(x^{(i)})}$

Andrew N

Implementation

Unrolling

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

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

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

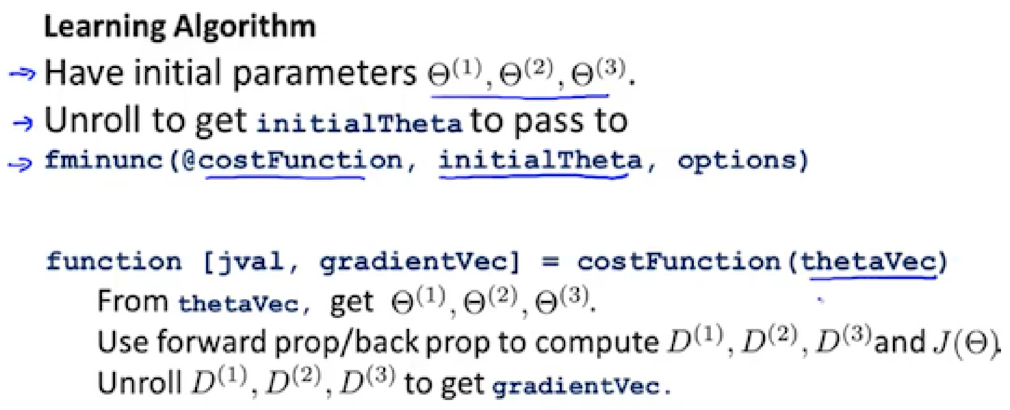
```
thetaVector = [ Theta1(:); Theta2(:); Theta3(:); ]
deltaVector = [ D1(:); D2(:); D3(:) ]
```

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:

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

```

To summarize:



Learning Algorithm

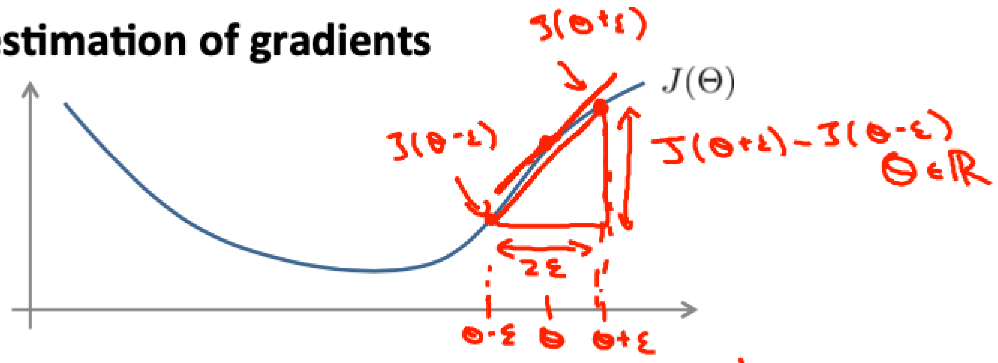
→ Have initial parameters $\underline{\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}}$.
→ Unroll to get `initialTheta` to pass to
→ `fminunc(@costFunction, initialTheta, options)`

```
function [jval, gradientVec] = costFunction(thetaVec)
    From thetaVec, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.
    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.
```

Gradient Checking

Assure out backpropagation works as intended.

Numerical estimation of gradients



$$\frac{d}{d\theta} J(\theta) \approx \underbrace{\frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon}} \leftarrow \quad \Big| \quad \cancel{\frac{J(\theta+\varepsilon)-J(\theta)}{\varepsilon}}$$

$\varepsilon = \underline{10^{-4}} \leftarrow$

Implement: `gradApprox = (J(theta + EPSILON) – J(theta – EPSILON)) / (2*EPSILON)`

Parameter vector θ

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

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

$$\left.\begin{array}{l}
\rightarrow \frac{\partial}{\partial\theta_1} J(\theta) \approx \frac{J(\boxed{\theta_1+\epsilon},\theta_2,\theta_3,\dots,\theta_n)-J(\boxed{\theta_1-\epsilon},\theta_2,\theta_3,\dots,\theta_n)}{2\epsilon} \\
\rightarrow \frac{\partial}{\partial\theta_2} J(\theta) \approx \frac{J(\theta_1,\boxed{\theta_2+\epsilon},\theta_3,\dots,\theta_n)-J(\theta_1,\boxed{\theta_2-\epsilon},\theta_3,\dots,\theta_n)}{2\epsilon} \\
\quad \vdots \\
\rightarrow \frac{\partial}{\partial\theta_n} J(\theta) \approx \frac{J(\theta_1,\theta_2,\theta_3,\dots,\boxed{\theta_n+\epsilon})-J(\theta_1,\theta_2,\theta_3,\dots,\boxed{\theta_n-\epsilon})}{2\epsilon}
\end{array}\right\}$$

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

```
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;
```

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

Check that $\boxed{\texttt{gradApprox}} \approx \boxed{\texttt{DVec}}$

From back prop.

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.

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

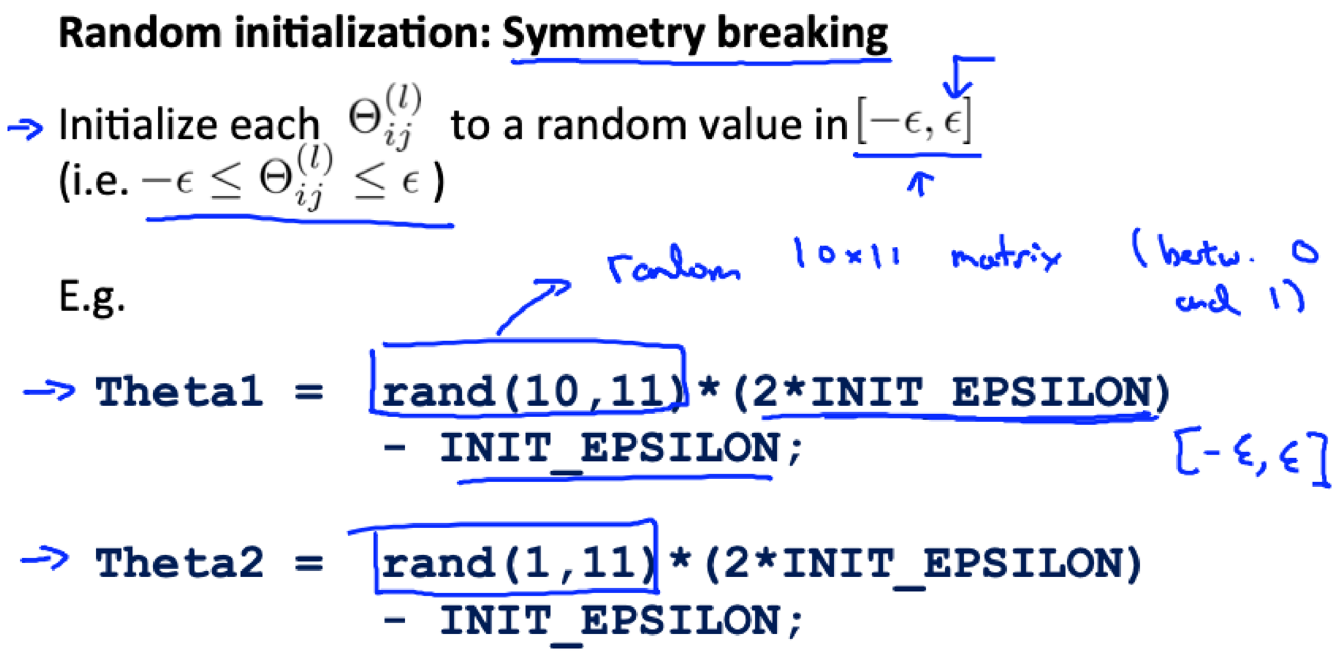
Important:

- 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.

```
epsilon = 1e-4;
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) += epsilon;
  thetaMinus = theta;
  thetaMinus(i) -= epsilon;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
end;
```

Random Initialization

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 Θ matrices using the following method.



Random initialization: Symmetry breaking

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

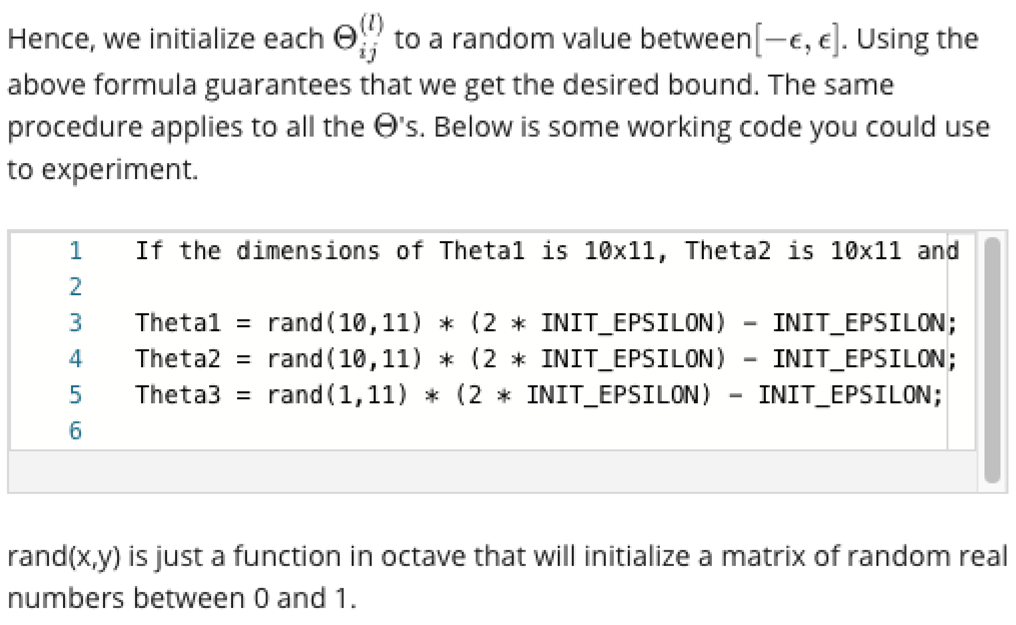
E.g.

random 10×11 matrix (betw. 0 and 1)

```
→ Theta1 =  rand(10,11)*(2*INIT_EPSILON)
            - INIT_EPSILON;
```

$[-\epsilon, \epsilon]$

```
→ Theta2 =  rand(1,11)*(2*INIT_EPSILON)
            - INIT_EPSILON;
```



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 Θ's. Below is some working code you could use to experiment.

```
If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and

Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta3 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;

```

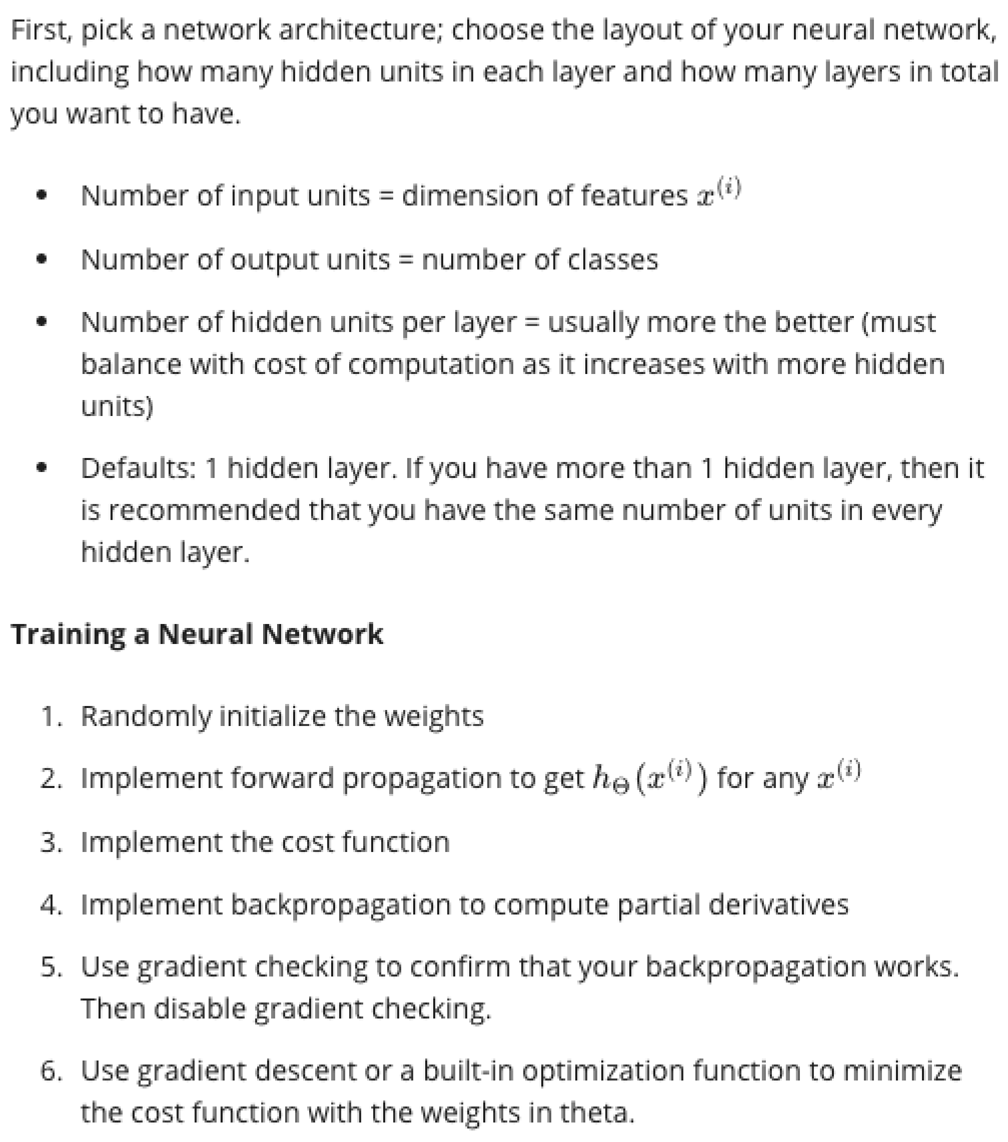
rand(x,y) is just a function in octave that will initialize a matrix of random real numbers between 0 and 1.

Implementation Note:

- Implement backprop to compute DVec
- Implement numerical gradient check to compute gradApprox.

- Make sure they give similar values.
- turn off gradient checking. Using backprop code for learning.

Put all 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.

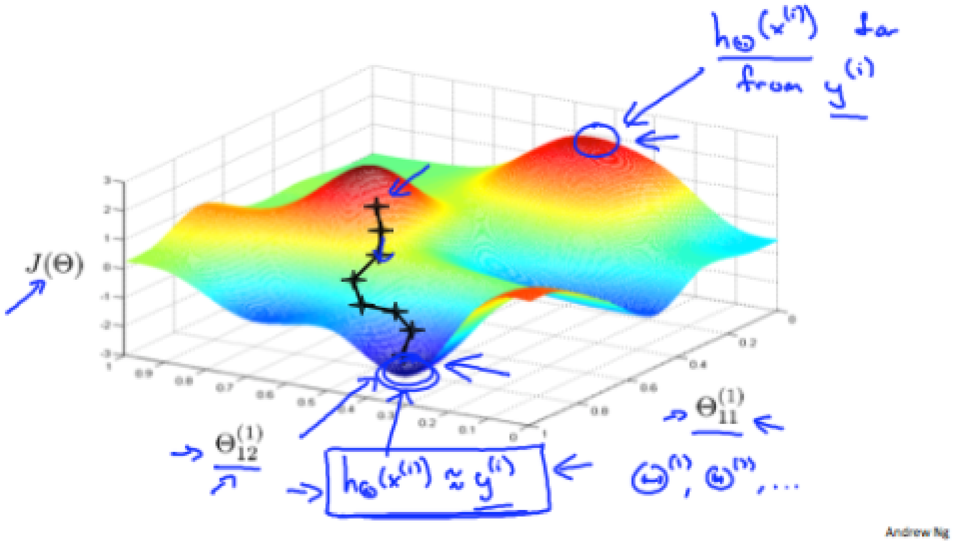
Training a Neural Network

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:

```
for i = 1:m,
   Perform forward propagation and backpropagation using example (x(i),y(i))
   (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.

Quiz

1.

1. You are training a three layer neural network and would like to use backpropagation to compute the gradient of the cost function. In the backpropagation algorithm, one of the steps is to update

   $$\Delta_{ij}^{(2)} := \Delta_{ij}^{(2)} + \delta_i^{(3)} * (a^{(2)})_j$$

   for every i, j. Which of the following is a correct vectorization of this step?

   - $\Delta^{(2)} := \Delta^{(2)} + (a^{(2)})^T * \delta^{(2)}$
   - [✓ $\Delta^{(2)} := \Delta^{(2)} + \delta^{(3)} * (a^{(2)})^T$]
   - $\Delta^{(2)} := \Delta^{(2)} + (a^{(2)})^T * \delta^{(3)}$
   - $\Delta^{(2)} := \Delta^{(2)} + \delta^{(2)} * (a^{(3)})^T$

根据反向传播的向量公式即可。

✓ **Correct**

This version is correct, as it takes the "outer product" of the two vectors $\delta^{(3)}$ and $a^{(2)}$ which is a matrix such that the (i, j)-th entry is $\delta_i^{(3)} * (a^{(2)})_j$ as desired.

2.

2. Suppose $\mathtt{Theta1}$ is a 5x3 matrix, and $\mathtt{Theta2}$ is a 4x6 matrix. You set $\mathtt{thetaVec} = [\mathtt{Theta1}(:); \mathtt{Theta2}(:)]$. Which of the following correctly recovers $\mathtt{Theta2}$?

- [✓ $\mathtt{reshape}(\mathtt{thetaVec}(16:39), 4, 6)$]
- $\mathtt{reshape}(\mathtt{thetaVec}(15:38), 4, 6)$
- $\mathtt{reshape}(\mathtt{thetaVec}(16:24), 4, 6)$
- $\mathtt{reshape}(\mathtt{thetaVec}(15:39), 4, 6)$
- $\mathtt{reshape}(\mathtt{thetaVec}(16:39), 6, 4)$

✓ **Correct**

This choice is correct, since $\mathtt{Theta1}$ has 15 elements, so $\mathtt{Theta2}$ begins at index 16 and ends at index 16 + 24 - 1 = 39.

3.

3. Let $J(\theta) = 2\theta^3 + 2$. Let $\theta = 1$, and $\epsilon = 0.01$. Use the formula $\frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}$ to numerically compute an approximation to the derivative at $\theta = 1$. What value do you get? (When $\theta = 1$, the true/exact derivati ve is $\frac{dJ(\theta)}{d\theta} = 6$.)

- 6
- 5.9998
- 8
- [✓ 6.0002]

✓ **Correct**

We compute $\frac{(2(1.01)^3+2)-(2(0.99)^3+2)}{2(0.01)} = 6.0002$.

✓ **Correct**

We compute $\frac{(2(1.01)^3+2)-(2(0.99)^3+2)}{2(0.01)} = 6.0002.$

4.

4. Which of the following statements are true? Check all that apply.

☐ Computing the gradient of the cost function in a neural network has the same efficiency when we use backpropagation or when we numerically compute it using the method of gradient checking.

☐ Gradient checking is useful if we are using one of the advanced optimization methods (such as in fminunc) as our optimization algorithm. However, it serves little purpose if we are using gradient descent.

☑ For computational efficiency, after we have performed gradient checking to

verify that our backpropagation code is correct, we usually disable gradient checking before using backpropagation to train the network.

✓ **Correct**

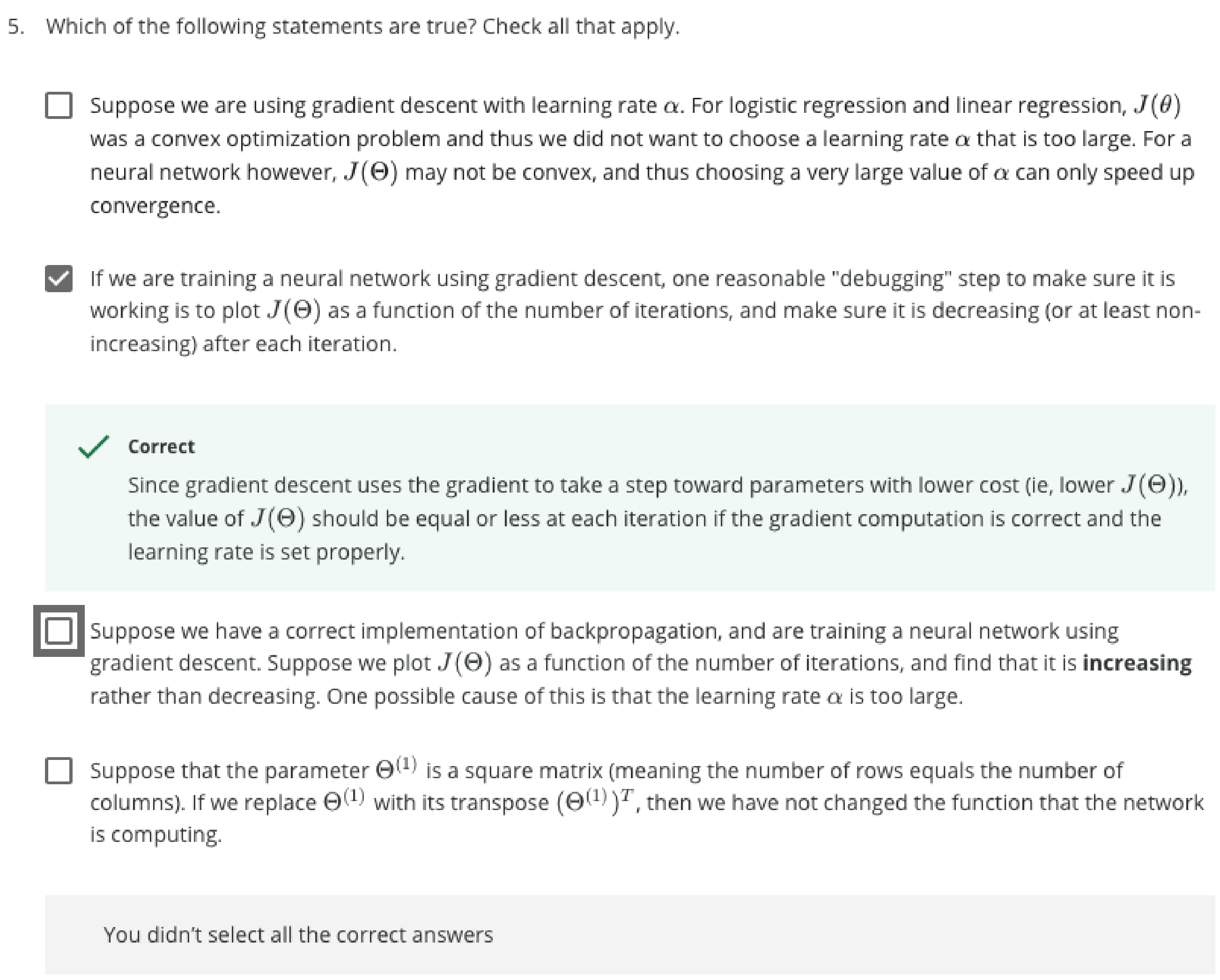
Checking the gradient numerically is a debugging tool: it helps ensure a corre

ct implementation, but it is too slow to use as a method for actually computing gradients.

☑ Using gradient checking can help verify if one's implementation of backpropagation is bug-free.

✓ **Correct**

If the gradient computed by backpropagation is the same as one computed numerically with gradient checking, this is very strong evidence that you have a correct implementation of backpropagation.



5. Which of the following statements are true? Check all that apply.

☐ Suppose we are using gradient descent with learning rate α. For logistic regression and linear regression, $J(\theta)$ was a convex optimization problem and thus we did not want to choose a learning rate α that is too large. For a neural network however, $J(\Theta)$ may not be convex, and thus choosing a very large value of α can only speed up convergence.

[✓] If we are training a neural network using gradient descent, one reasonable "debugging" step to make sure it is working is to plot $J(\Theta)$ as a function of the number of iterations, and make sure it is decreasing (or at least non-increasing) after each iteration.

✓ **Correct**
Since gradient descent uses the gradient to take a step toward parameters with lower cost (ie, lower $J(\Theta)$), the value of $J(\Theta)$ should be equal or less at each iteration if the gradient computation is correct and the learning rate is set properly.

☐ Suppose we have a correct implementation of backpropagation, and are training a neural network using gradient descent. Suppose we plot $J(\Theta)$ as a function of the number of iterations, and find that it is **increasing** rather than decreasing. One possible cause of this is that the learning rate α is too large.

☐ Suppose that the parameter $\Theta^{(1)}$ is a square matrix (meaning the number of rows equals the number of columns). If we replace $\Theta^{(1)}$ with its transpose $(\Theta^{(1)})^T$, then we have not changed the function that the network is computing.

You didn't select all the correct answers

5.

Choose B and C.

Reference：

神经网络15分钟入门！--反向传播到底是怎么传播的？