F. Neural Network: Learning

I. Cost function and Backpropagation

1. Cost function

Let's first define a few variables:

+ 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

Denote $h_\Theta(x)_k$ as being a hypothesis that results in the kth output. The function for neural networks is going to be a generalization of the 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:

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

In the first part of the equation, before the square brackets, an additional nested summation that loops through the number of output nodes.

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

2. Backpropagation Algorithm

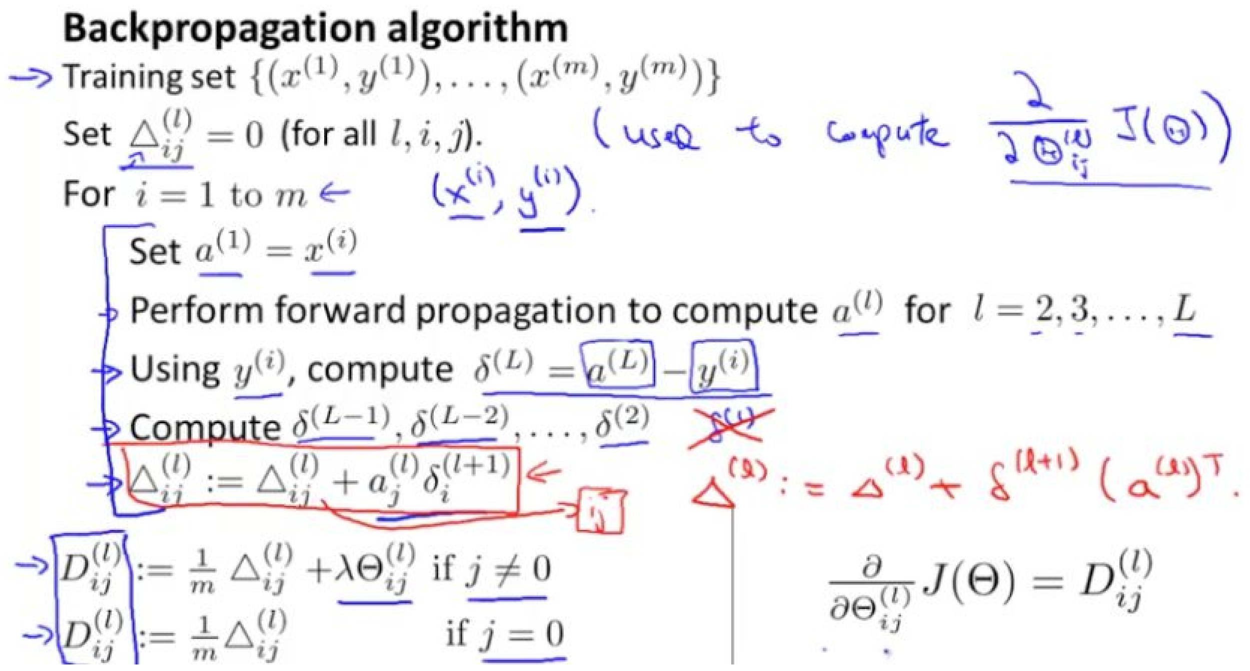
"Backpropagation" is neural-network terminology for minimizing cost function, just like gradient descent in logistic and linear regression. The goal is to compute:

$$\min_\Theta J(\Theta)$$

That is, minimize the cost function J using an optimal set of parameters in theta. In this section, the equations is used to compute the partial derivative of J(Θ):

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

The algorithm:



Backpropagation algorithm

```
Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
Set $\Delta_{ij}^{(l)} = 0$ (for all $l, i, j$).    (used to compute $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$)
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 $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$    $\cancel{\delta^{(1)}}$
    $\boxed{\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}}$    $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$.
$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ if $j \neq 0$
$D_{ij}^{(l)} := \frac{1}{m} \Delta_{ij}^{(l)}$ if $j = 0$
```

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

Backpropagation Algorithm:

Given a training set $\{(x^{(1)},y^{(1)}); \ldots; (x^{(m)},y^{(m)})\}$. Set $\Delta_{i,j}^{(l)} := 0$ for all (l, i, j) (hence end up having a matrix full of zeros)

For training example t =1 to m:

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

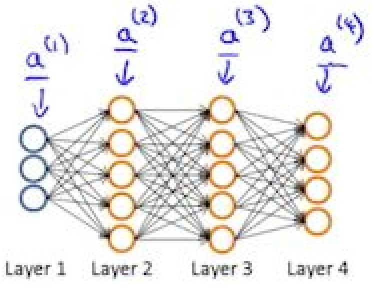
ii/ Perform forward propagation to compute $a^{(l)}$ for l=2,3,…,L

Gradient computation

Given one training example ($\underline{x, y}$):

Forward propagation:

$$\underline{a^{(1)}} = \underline{x}$$
$$\rightarrow z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$\rightarrow a^{(2)} = g(z^{(2)}) \quad (\underline{\text{add } a_0^{(2)}})$$
$$\rightarrow z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$\rightarrow a^{(3)} = g(z^{(3)}) \quad (\text{add } a_0^{(3)})$$
$$\rightarrow z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$\rightarrow a^{(4)} = \underline{h_\Theta(x)} = g(z^{(4)})$$



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

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

iv/ Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ 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. 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:

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

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

Hence update the new Δ matrix:

$$D_{i,j}^{(l)} := \frac{1}{m} \left(\Delta_{i,j}^{(l)} + \lambda \Theta_{i,j}^{(l)} \right), \text{ if j≠0.}$$

$$D_{i,j}^{(l)} := \frac{1}{m} \Delta_{i,j}^{(l)} \text{ If j=0}$$

The capital-delta matrix D is used as an "accumulator" to add up values as they go along and eventually compute partial derivatives. Thus:

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

3. Backpropagation intuition

Recall that the cost function for a neural network is:

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

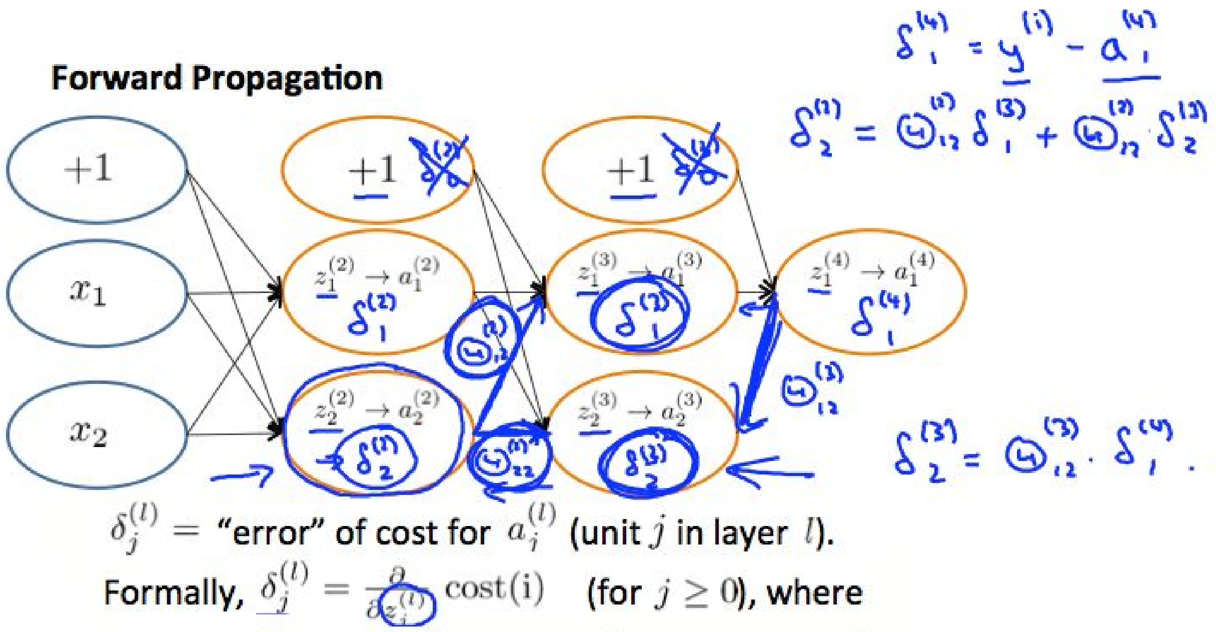
If consider simple non-multiclass classification (k = 1) and disregard regularization, the cost is computed with:

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

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:

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

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


Forward Propagation
$\delta_1^{(4)} = \underline{y}^{(i)} - \underline{a_1}^{(4)}$
$\delta_2^{(2)} = \Theta_{12}^{(2)} \delta_1^{(3)} + \Theta_{22}^{(2)} \delta_2^{(3)}$
$+1$
x_1
x_2
$\underline{+1}$ $\cancel{\delta_0^{(2)}}$
$\underline{+1}$ $\cancel{\delta_0^{(3)}}$
$z_1^{(2)} \to a_1^{(2)}$
$\delta_1^{(2)}$
$z_2^{(2)} \to a_2^{(2)}$
$\delta_2^{(2)}$
$\Theta_{12}^{(2)}$
$\Theta_{22}^{(2)}$
$z_1^{(3)} \to a_1^{(3)}$
$\delta_1^{(3)}$
$z_2^{(3)} \to a_2^{(3)}$
$\delta_2^{(3)}$
$\Theta_{12}^{(3)}$
$z_1^{(4)} \to a_1^{(4)}$
$\delta_1^{(4)}$
$\delta_2^{(3)} = \Theta_{12}^{(3)} \cdot \delta_1^{(4)}.$

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

Formally, $\underline{\delta_j^{(l)}} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(i)$ (for $j \geq 0$), where

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

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

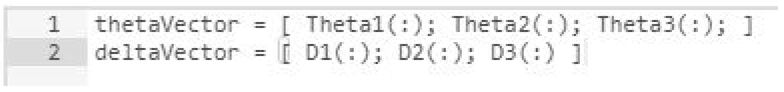
II. Backpropagaton in practice

1. Implementation note: Unrolling parameters

With neural networks, work with sets of matrices:

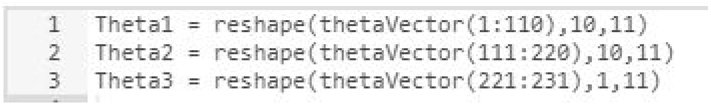
$$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}, \ldots$$
$$D^{(1)}, D^{(2)}, D^{(3)}, \ldots$$

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



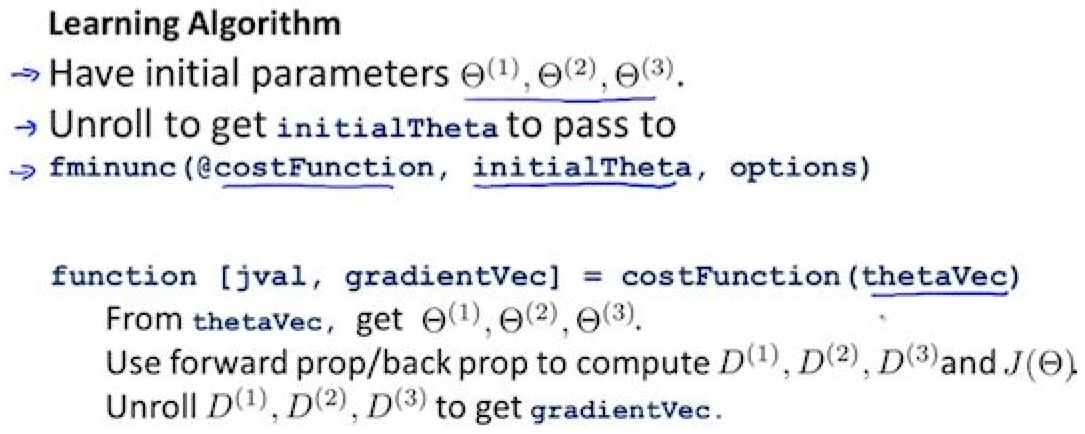
```
thetaVector = [ Theta1(:); Theta2(:); Theta3(:); ]
deltaVector = [ D1(:); D2(:); D3(:) ]
```

If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11, then 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 $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

→ 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`.

2. Gradient checking

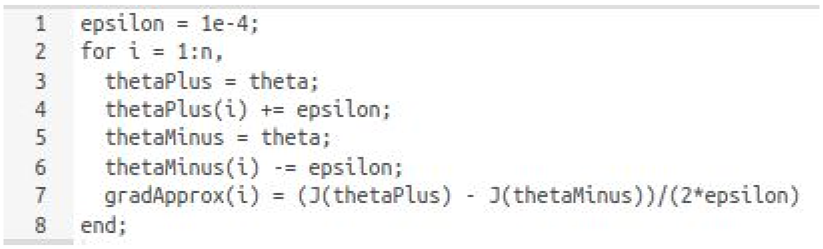
Gradient checking will assure that the backpropagation works as intended. Approximate the derivative of our cost function with:

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

With multiple theta matrices, approximate the derivative with respect to Θ_j as follows:

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

A small value for ϵ (epsilon) such as $\epsilon=10^{-4}$, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems. Hence, we are only adding or subtracting epsilon to the Θ_j matrix. In octave we can do it as follows:

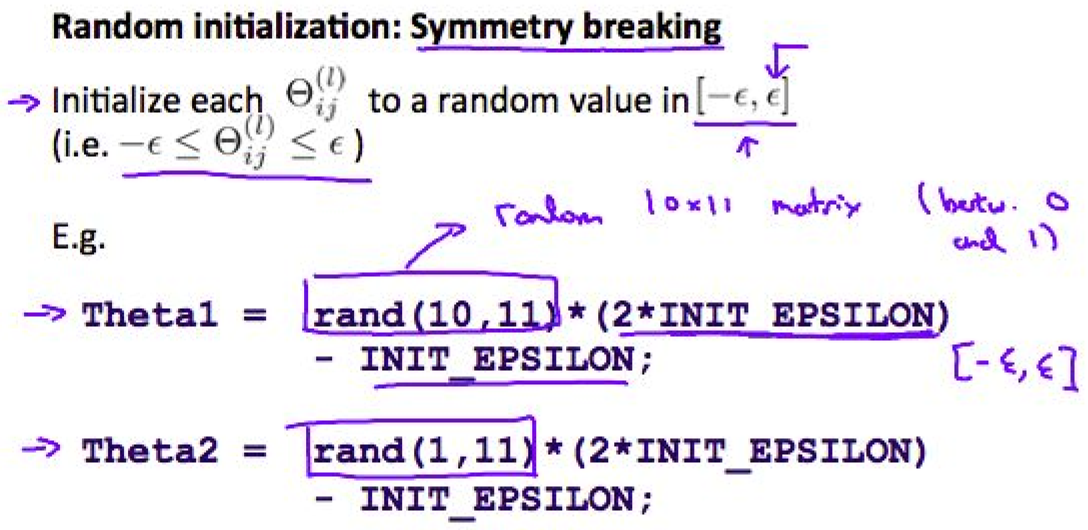


```
epsilon = 1e-4;
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) += epsilon;
  thetaMinus = theta;
  thetaMinus(i) -= epsilon;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
end;
```

Use the gradApprox vector to check that gradApprox ≈ deltaVector. When verified once that backpropagation algorithm is correct, you don't need to compute gradApprox again. The code to compute gradApprox can be very slow.

3. Random initialization

Initializing all theta weights to zero does not work with neural networks. When backpropagation, all nodes will update to the same value repeatedly. Instead can randomly initialize the 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. $-\epsilon \le \Theta_{ij}^{(l)} \le \epsilon$)

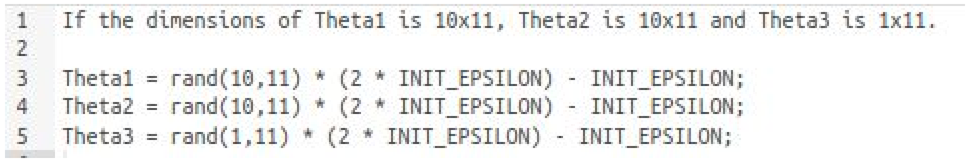
E.g.

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

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

random 10×11 matrix (betw. 0 and 1)
[-ε, ε]

Initialize each $\Theta_{ij}^{(l)}$ to a random value between [−ϵ,ϵ]. Using the above formula guarantees to get the desired bound. The same procedure applies to all the Θ's. Below is some working code to experiment.



```
If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11.

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

4. Putting it together

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

+ Number of input units = dimension of features x(i)x^{(i)}x(i)
+ Number of output units = number of classes

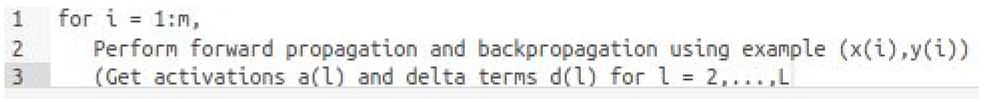
+ Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)

+ Defaults: 1 hidden layer. If it has more than 1 hidden layer, then it is recommended for the same number of units in every hidden layer.

Training a Neural Network

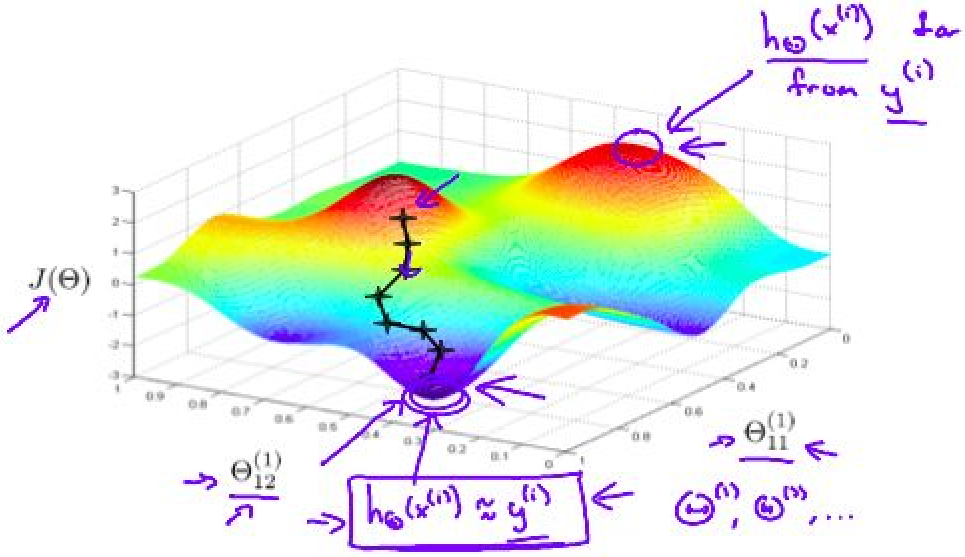
+ Randomly initialize the weights
+ Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$
+ Implement the cost function
+ Implement backpropagation to compute partial derivatives
+ Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
+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

When perform forward and back propagation, 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 an intuition of what is happening when implementing the neural network:



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