Neural Networks

Contents

1 Model Representation I

Let's examine how we will represent a hypothesis function using neural networks. At a very simple level, neurons are basically computational units that take inputs (**dendrites**) as electrical inputs (called "spikes") that are channeled to outputs (**axons**). In our model, our dendrites are like the input features $x_1 ... x_n$, and the output is the result of our hypothesis function. In this model our x_0 input node is sometimes called the "bias unit." It is always equal to 1. In neural networks, we use the same logistic function as in classification, $\frac{1}{1+e^{-\theta^T x}}$, yet we sometimes call it a sigmoid (logistic) **activation** function. In this situation, our "theta" parameters are sometimes called "weights".

Visually, a simplistic representation looks like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} \quad \end{bmatrix} \rightarrow h_\theta(x)$$

Our input nodes (layer 1), also known as the "input layer", go into another node (layer 2), which finally outputs the hypothesis function, known as the "output layer".

We can have intermediate layers of nodes between the input and output layers called the "hidden layers."

In this example, we label these intermediate or "hidden" layer nodes $a_0^2 \dots a_n^2$ and call them "activation units."

$a_i^{(j)}$ = "activation" of unit i in layer j
$\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j to layer $j+1$

If we had one hidden layer, it would look like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} \rightarrow h_\theta(x)$$

The values for each of the "activation" nodes is obtained as follows:

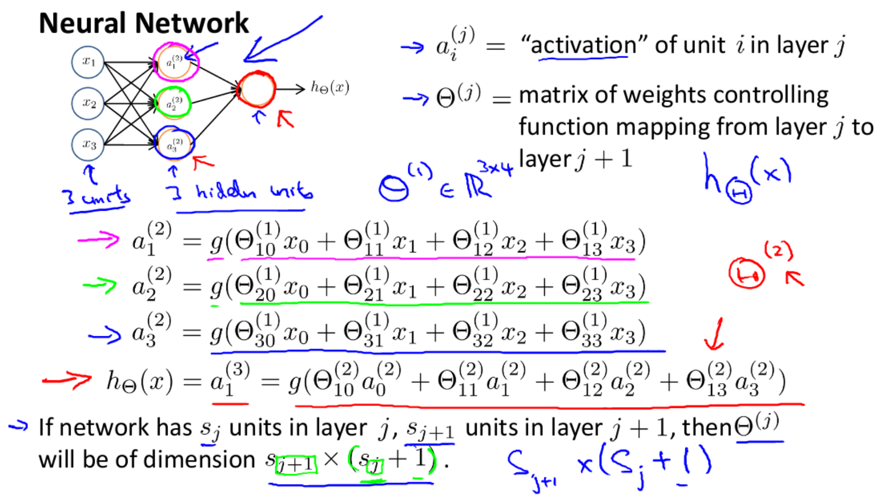
$$\begin{aligned} a_1^{(2)} &= g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3) \\ a_2^{(2)} &= g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3) \\ a_3^{(2)} &= g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3) \\ h_\Theta(x) = a_1^{(3)} &= g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)}) \end{aligned}$$

This is saying that we compute our activation nodes by using a 3x4 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied by yet another parameter matrix $\Theta^{(2)}$ containing the weights for our second layer of nodes.

Each layer gets its own matrix of weights, $\Theta^{(j)}$.

The dimensions of these matrices of weights is determined as follows:
If network has s_j units in layer j and s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension s_{j+1} x (s_j+1).

The +1 comes from the addition in $\Theta^{(j)}$ of the "bias nodes," x_0 end $\Theta_0^{(j)}$. In other words the output nodes will not include the bias nodes while the inputs will. The following image summarizes our model representation:



Neural Network

$\rightarrow a_i^{(j)} =$ "activation" of unit i in layer j

$\rightarrow \Theta^{(j)} =$ matrix of weights controlling function mapping from layer j to layer $j+1$

3 units

3 hidden units

$\Theta^{(1)} \in \mathbb{R}^{3\times 4}$

$h_\Theta(x)$

$\Theta^{(2)}$

$$\rightarrow a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$
$$\rightarrow a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$
$$\rightarrow a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$
$$\rightarrow h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

$\rightarrow$ If network has s_j units in layer j, s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$. $\quad S_{j+1} \times (S_j + 1)$

Example: if layer 1 has 2 input nodes and layer 2 has 4 activation nodes. Dimension of $\Theta^{(1)}$ is going to be 4x3 where $s_j = 2$ and $s_{j+1} = 4$, so s_{j+1} x (s_j+1)= 4 x 3.

2 Model Representation II

To re-iterate, the following is an example of a neural network:

$$\begin{aligned}
a_1^{(2)} &= g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)\\
a_2^{(2)} &= g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)\\
a_3^{(2)} &= g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)\\
h_\Theta(x) = a_1^{(3)} &= g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})
\end{aligned}$$

In this section we'll do a vectorized implementation of the above functions. We're going to define a new variable $z_k^{(j)}$ that encompasses the parameters inside our g function. In our previous example if we replaced by the variable z for all the parameters we would get:

$$a_1^{(2)} = g(z_1^{(2)})$$
$$a_2^{(2)} = g(z_2^{(2)})$$
$$a_2^{(2)} = g(z_3^{(2)})$$

In other words, for layer j=2 and node k, the variable z will be:

$$z_k^{(2)} = \Theta_{k,0}^{(1)} x_0 + \Theta_{k,1}^{(1)} x_1 + \cdots + \Theta_{k,n}^{(1)} x_n$$

The vector representation of x and z^j is:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix} \; z^{(j)} = \begin{bmatrix} a_1^{(j)} \\ a_2^{(j)} \\ \cdots \\ a_n^{(j)} \end{bmatrix}$$

Setting x = $a^{(1)}$, we can rewrite the equation as:

$$z^{(j)} = \Theta^{(j-1)} a^{(j-1)}$$

We are multiplying our matrix $\Theta^{(j-1)}$ with dimensions s_j x $(n+1)$ (where s_j is the number of our activation nodes) by our vector $a^{(j-1)}$ with height (n+1). This gives us our vector $z^{(j)}$ with height s_j. Now we can get a vector of our activation nodes for layer j as follows:

$$a^{(j)} = g(z^{(j)})$$

Where our function g can be applied element-wise to our vector $z^(j)$.

We can then add a bias unit (equal to 1) to layer j after we have computed $a^{(j)}$. This will be element $a_0^{(j)}$ and will be equal to 1. To compute our final hypothesis, let's first compute another z vector:

$$z^{(j+1)} = \Theta^{(j)} a^{(j)}$$

We get this final z vector by multiplying the next theta matrix after $\Theta^{(j-1)}$ with the values of all the activation nodes we just got. This last theta matrix $\Theta^{(j)}$ will have only **one row** which is multiplied by one column $a^{(j)}$ so that our result is a single number. We then get our final result with:

$$h_\Theta(x) = a^{(j+1)} = g(z^{(j+1)})$$

Notice that in this **last step**, between layer j and layer j+1, we are doing **exactly the same thing** as we did in logistic regression. Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypotheses.

3 Examples and Intuitions I

A simple example of applying neural networks is by predicting x_1 AND x_2, which is the logical 'and' operator and is only true if both x_1 and x_2 are 1. The graph of our functions will look like:

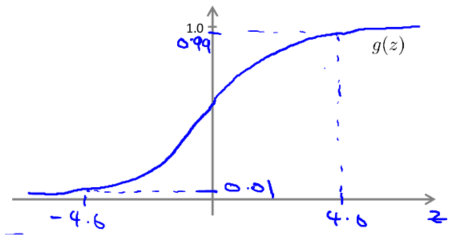
$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} g(z^{(2)}) \end{bmatrix} \rightarrow h_\theta(x)$$

Remember that x_0 is our bias variable and is always 1.
Let's set our first theta matrix as:

$$\Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \end{bmatrix}$$

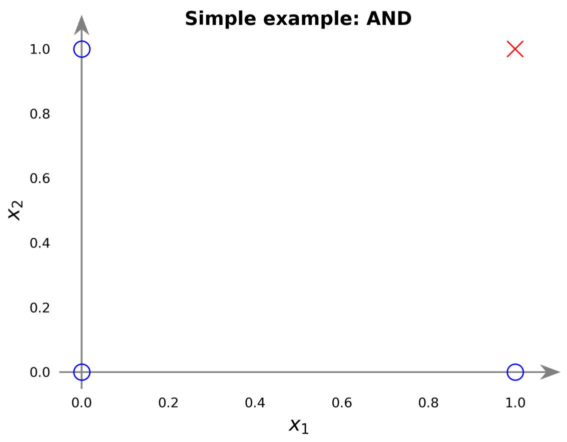
This will cause the output of our hypothesis to only be positive if both x_1 and x_2 are 1. In other words:

Where g(z) is the following:



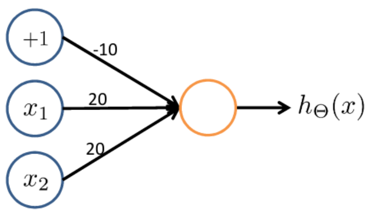
$$\begin{aligned}
&h_\Theta(x) = g(-30 + 20x_1 + 20x_2) \\
&x_1 = 0 \text{ and } x_2 = 0 \text{ then } g(-30) \approx 0 \\
&x_1 = 0 \text{ and } x_2 = 1 \text{ then } g(-10) \approx 0 \\
&x_1 = 1 \text{ and } x_2 = 0 \text{ then } g(-10) \approx 0 \\
&x_1 = 0 \text{ and } x_2 = 1 \text{ then } g(10) \approx 1
\end{aligned}$$

Which graphically can be represented as follows:



So we have constructed one of the fundamental operations in computers by using a small neural network rather than using an actual AND gate. Neural networks can also be used to simulate all the other logical gates. The following is an example of the logical operator 'OR', meaning either x_1 is true or x_2 is true, or both:

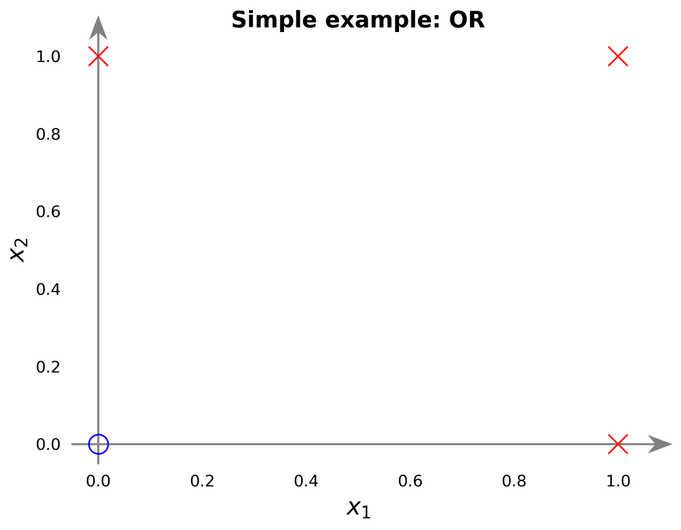
Example: OR function



$g(-10 + 20x_1 + 20x_2)$

x_1	x_2	$h_\Theta(x)$
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	≈ 1
1	1	≈ 1

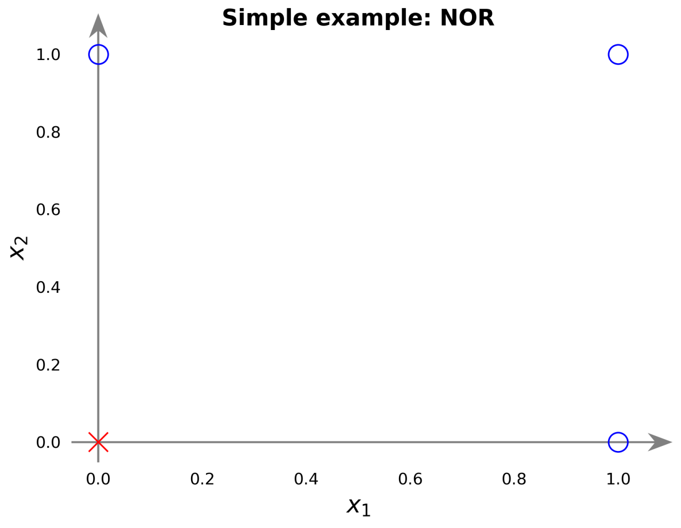
Which graphically can be represented as follows:



4 Examples and Intuitions II

The $\Theta^{(1)}$ matrices for AND, OR, and NOR are:

$$AND : \theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \end{bmatrix}$$
$$OR : \theta^{(1)} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix}$$
$$NOR : \theta^{(1)} = \begin{bmatrix} 10 & -20 & -20 \end{bmatrix}$$



We can combine these to get the XNOR logical operator (which gives 1 if x_1 and x_2 are both 0 or both 1).

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} a^{(3)} \end{bmatrix} \rightarrow h_\theta(x)$$

For the transition between the first and second layer, we'll use a $\Theta^{(1)}$ matrix that combines the values for AND and NOR:

$$\Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \\ 10 & -20 & -20 \end{bmatrix}$$

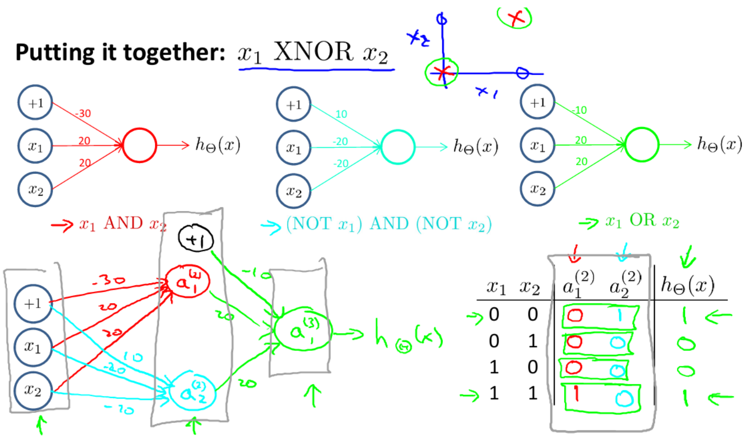
For the transition between the second and third layer, we'll use a $\Theta^{(2)}$ matrix that uses the value for OR:

$$\Theta^{(2)} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix}$$

Let's write out the values for all our nodes:

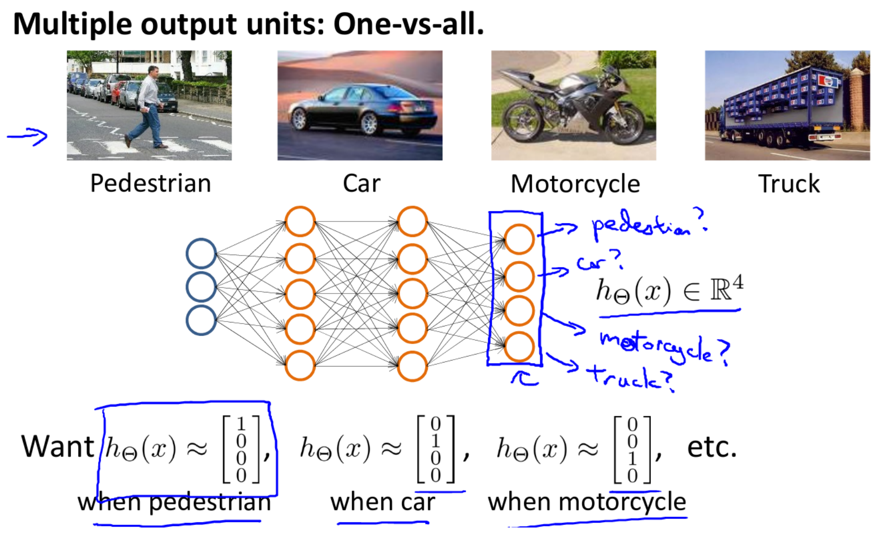
$$\begin{aligned} a^{(2)} &= g(\Theta^{(1)} * x) \\ a^{(3)} &= g(\Theta^{(2)} * a^{(2)}) \\ h_\Theta(x) &= a^{(3)}) \end{aligned}$$

And there we have the XNOR operator using a hidden layer with two nodes! The following summarizes the above algorithm:


Putting it together: x_1 XNOR x_2
x_1 AND x_2
(NOT x_1) AND (NOT x_2)
x_1 OR x_2
-30, 20, 20
10, -20, -20
-10, 20, 20
$h_\Theta(x)$
x_1 x_2 $a_1^{(2)}$ $a_2^{(2)}$ $h_\Theta(x)$
0 0 0 1 1
0 1 0 0 0
1 0 0 0 0
1 1 1 0 1

5 Multiclass Classification

To classify data into multiple classes, we let our hypothesis function return a vector of values. Say we wanted to classify our data into one of four categories. We will use the following example to see how this classification is done. This algorithm takes as input an image and classifies it accordingly:


Multiple output units: One-vs-all.
Pedestrian
Car
Motorcycle
Truck
pedestrian?
car?
$h_\Theta(x) \in \mathbb{R}^4$
motorcycle?
truck?
Want $h_\Theta(x) \approx \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $h_\Theta(x) \approx \begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, etc.
when pedestrian
when car
when motorcycle

We can define our set of resulting classes as y:

$$y^{(i)} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix},$$

Each $y^{(i)}$ represents a different image corresponding to either a car, pedestrian, truck, or motorcycle. The inner layers, each provide us with some new information which leads to our final hypothesis function. The setup looks like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \cdots \\ x_n \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(2)} \\ a_1^{(2)} \\ a_2^{(2)} \\ \cdots \end{bmatrix} \rightarrow \begin{bmatrix} a_0^{(3)} \\ a_1^{(3)} \\ a_2^{(3)} \\ \cdots \end{bmatrix} \rightarrow \cdots \rightarrow \begin{bmatrix} h_\theta(x)_1 \\ h_\theta(x)_2 \\ h_\theta(x)_3 \\ h_\theta(x)_4 \end{bmatrix}$$

Our resulting hypothesis for one set of inputs may look like:

$$h_\Theta(x) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

In which case our resulting class is the third one down, or $h_\Theta(x)_3$, which represents the motorcycle.

6 Cost Function

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

- L = total number of layers in the network
- s_i = 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. The cost function for the regularized logistic regression was:

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

For neural network, it is going to be slightly more complicated:

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

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 calcuated 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

7 Backpropagation Algorithm

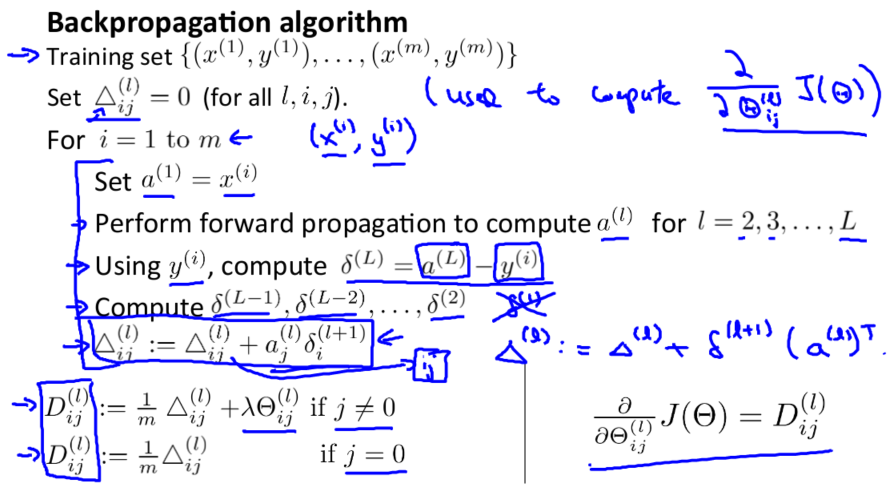
"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute:

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

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 $J(\Theta)$:

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

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 $l, i, j$).   (use 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 $\delta^{(L)} = a^{(L)} - y^{(i)}$
    Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$   $\cancel{\delta^{(1)}}$
    $\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$   $\triangle^{(l)} := \triangle^{(l)} + \delta^{(l+1)} (a^{(l)})^T$
$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$  if $j \neq 0$
$D_{ij}^{(l)} := \frac{1}{m} \triangle_{ij}^{(l)}$  if $j = 0$
```

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

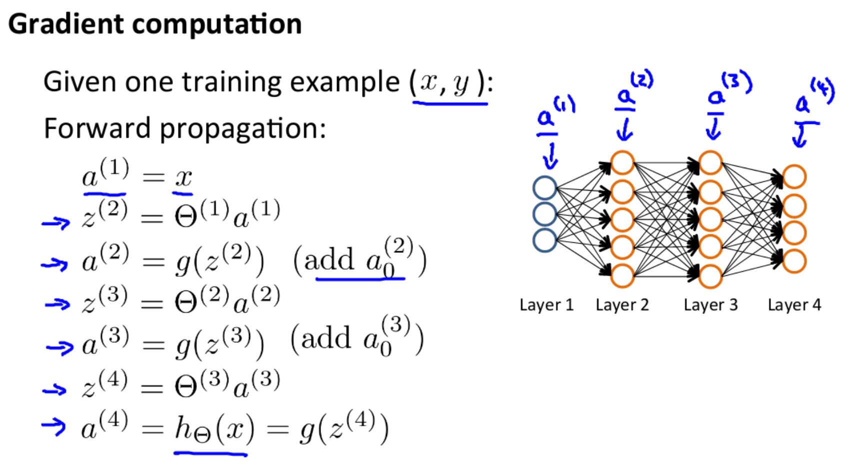
Back propagation Algorithm

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

- Set $\Delta^{(l)}_{i,j}$:= 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,...,L



Gradient computation

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

Forward propagation:

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

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 diffrences 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:

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

5.

$$\Delta^{(l)}_{i,j} := \Delta^{(l)}_{i,j} + a^{(l)}_j \delta^{(l+1)}_i$$

   or with vectorization,

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

   Hence we update our new Δ matrix.

- $D^{(l)}_{i,j} := \frac{1}{m}(\Delta^{(l)}_{i,j} + \lambda\Theta^{(l)}_{i,j})$, if $j \neq 0$.
- $D^{(l)}_{i,j} := \frac{1}{m}\Delta^{(l)}_{i,j}$,if $j = 0$

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

$$\frac{\partial}{\partial \Theta^{(l)}_{i,j}} J(\Theta) = D^{(l)}_{i,j}$$

8 Backpropagation Intuition

Recall that the cost function for a neural network is:

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

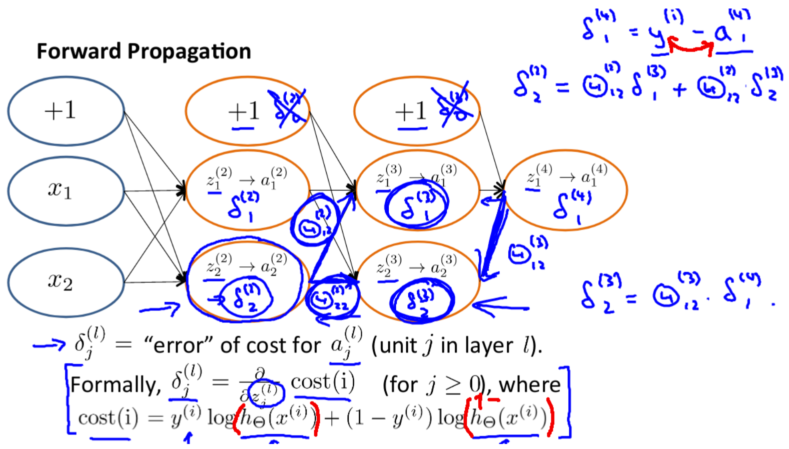
If we consider simple non-multiclass classification (k=1) and disregard regularization, the cost is computed with:

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

Intuitively, $\delta^{(l)}_j$ is the "error" for $a^{(l)}_j$ (unit j in layer l). More formally, the delta values are actually the derivative of the cost function:

$$\delta^{(l)}_j = \frac{\partial}{\partial z^{(l)}_j} cost(t)$$

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 and see how we could calculate some $\delta^{(l)}_j$:



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

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

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

In the image above, to calcuate $\delta_2^{(2)}$, we multiply the weights $\Theta_{12}^{(2)}$ and $\Theta_{22}^{(2)}$ by their respective δ 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 calcuate every single possible $\delta_j^{(l)}$, we could start from the right of our diagram. We can think of our edges as our Θ_{ij}. Going from right to left, to calcuate the value of $\delta_j^{(l)}$, you can just take the over all sum of each weight times the δ it is coming from. Hence, another example would be $\delta_2^{(3)} = \Theta_{12}^{(3)} * \delta_1^{(4)}$. Far most to the right, the initial $\delta_1^{(4)} = a_1^{(4)} - y^{(i)}$.

9 Implementation Note: Unrolling Parameters

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 advance optimization algorithms such us "optimize.minimize from scipy", we will want to 'unroll' all the elements and put them into one long vector:

```
import numpy as np

# Create a dictionary with thetas values in matrix representation
Thetas = {}
for i in range(3):
    Thetas['Theta{0}'.format(i)] = np.random.randint(5,
                                                     size=(2, 3))

# 'Unroll' to vector representation
thetaVector = []
thetaVector.extend((list(Thetas.get('Theta0').flatten()) +
                    list(Thetas.get('Theta1').flatten()) +
                    list(Thetas.get('Theta2').flatten())))

# Return to our original matrices from the 'unrolled' versions
Theta0 = np.reshape(thetaVector[0:6], (2, 3))
Theta1 = np.reshape(thetaVector[6:12], (2, 3))
Theta2 = np.reshape(thetaVector[12:18], (2, 3))
```

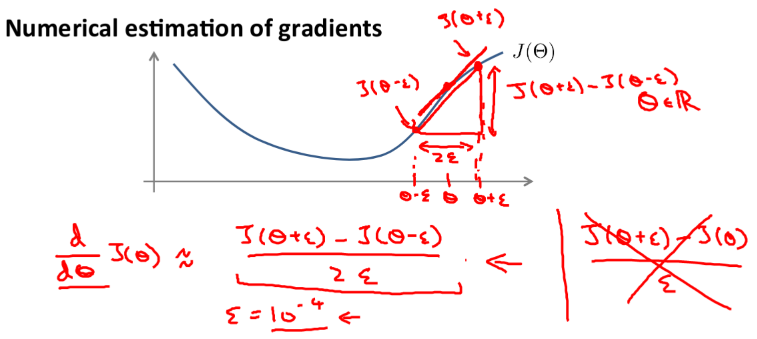
To summarize the Learning Algorithm procedure:

- Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$
- Unroll to get the initialTheta vectorized to pass though 'optimize.minimize'
- Call the costFunction (which returns $J(\Theta)$ and gradient descent) with the vectorized theta. Within the cost function:
  - Get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ from the vectorized theta.
  - Use forward propagation/back propagation to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$
  - Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get a vectorized gradient descent.

10 Gradient Checking

Gradient Checking will assure that our backpropagation works as intended. We can 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, we can approximate the derivative **with respect to** Θ_j as follows:

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

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 Python we can do it as follows:

```python
import numpy as np

# Cost function


def J(theta):
    '''
    J(theta) where theta is a list of vectorized theta.
    Raise the elements of a lsit to the cube
    '''
    return [elem**3 for elem in theta]


# Define epsilon
# Advised epsilon: 1e-4;
epsilon = 0.01

# 'Unrolled' vectorized theta
thetaVector = np.array([1,0], dtype=float)

# Calculate gradient checking
thetaPlus = np.copy(thetaVector) + epsilon
thetaMinus = np.copy(thetaVector) - epsilon
gradApprox = np.subtract(J(thetaPlus), J(thetaMinus))/(2*epsilon)
```

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

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

11 Random Initialization: Symmetry breaking

Initializing all theta weights to zero deos 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:

```
import numpy as np

# If the dimensions of Theta0 is 10x11, Theta1 is 10x11 and
# Theta3 is 1x11.

INIT_EPSILON = 0.0001

Theta0 = np.random.uniform(0, 1, size=((10, 11))) * \
    (2*INIT_EPSILON) - INIT_EPSILON
Theta1 = np.random.uniform(0, 1, size=((10, 11))) * \
    (2*INIT_EPSILON) - INIT_EPSILON
Theta2 = np.random.uniform(0, 1, size=((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.

random.uniform initialize a matrix of random real numbers between 0 and 1. (Note: the epsilon used above is unrelated to the epsilon from Gradient Checking).

12 Putting it 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.

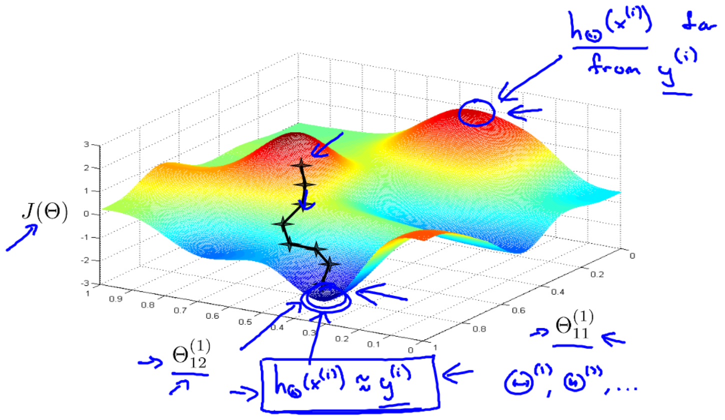
Training a Neural Network

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 build-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 in range(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.