# **Regularized Linear Regression**

# Cost Function

If we have overfitting from our hypothesis function, we can reduce the weight that some of the terms in our function carry by increasing their cost.

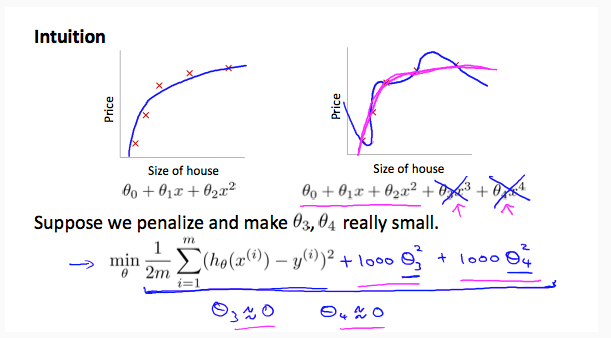
Say we wanted to make the following function more quadratic:



We'll want to eliminate the influence of \theta\_3x^3*θ*3​*x*3 and \theta\_4x^4*θ*4​*x*4 . Without actually getting rid of these features or changing the form of our hypothesis, we can instead modify our **cost function**:



We've added two extra terms at the end to inflate the cost of *θ*3​ and *θ*4​. Now, in order for the cost function to get close to zero, we will have to reduce the values of *θ*3​ and *θ*4​ to near zero. This will in turn greatly reduce the values of *θ*3\*x^3 and  *θ*4\*x^4 in our hypothesis function. As a result, we see that the new hypothesis (depicted by the pink curve) looks like a quadratic function but fits the data better due to the extra small terms \theta\_3x^3*θ*3​*x*3 and \theta\_4x^4*θ*4​*x*4.



We could also regularize all of our theta parameters in a single summation as:

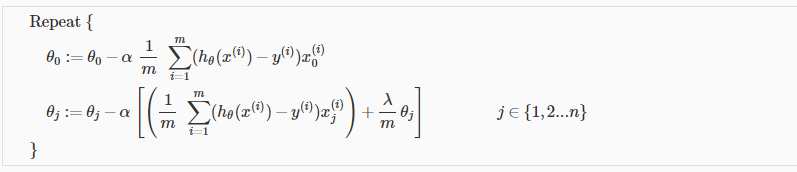
The λ, or lambda, is the **regularization parameter**. It determines how much the costs of our theta parameters are inflated.

Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If lambda is chosen to be too large, it may smooth out the function too much and cause underfitting. Hence, what would happen if \lambda = 0*λ*=0 or is too small ?



Gradient Descent

We will modify our gradient descent function to separate out \theta\_0*θ*0​ from the rest of the parameters because we do not want to penalize \theta\_0*θ*0​.



The term \frac{\lambda}{m}\theta\_j*mλ*​*θj*​ performs our regularization. With some manipulation our update rule can also be represented as:

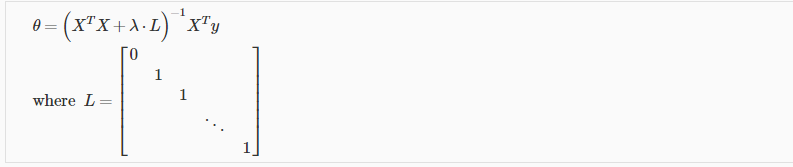
\theta\_j := \theta\_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum\_{i=1}^m(h\_\theta(x^{(i)}) - y^{(i)})x\_j^{(i)}*θj*​:=*θj*​(1−*αmλ*​)−*αm*1​∑*i*=1*m*​(*hθ*​(*x*(*i*))−*y*(*i*))*xj*(*i*)​

The first term in the above equation, 1 - \alpha\frac{\lambda}{m}1−*αmλ*​ will always be less than 1. Intuitively you can see it as reducing the value of \theta\_j*θj*​ by some amount on every update. Notice that the second term is now exactly the same as it was before.

### ****Normal Equation****

Now let's approach regularization using the alternate method of the non-iterative normal equation.

To add in regularization, the equation is the same as our original, except that we add another term inside the parentheses:

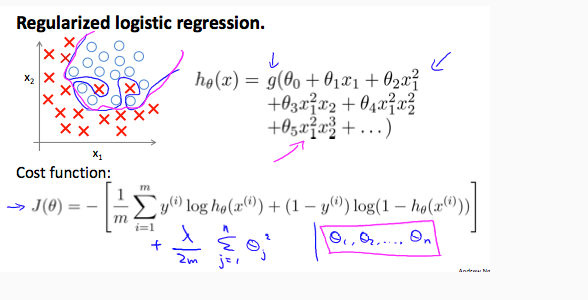


L is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else. It should have dimension (n+1)×(n+1). Intuitively, this is the identity matrix (though we are not including x\_0*x*0​), multiplied with a single real number λ.

Recall that if m < n, then X^TX*XTX* is non-invertible. However, when we add the term λ⋅L, then X^TX*XTX* + λ⋅L becomes invertible.

# **Regularized Logistic Regression**

We can regularize logistic regression in a similar way that we regularize linear regression. As a result, we can avoid overfitting. The following image shows how the regularized function, displayed by the pink line, is less likely to overfit than the non-regularized function represented by the blue line:



### Cost Function

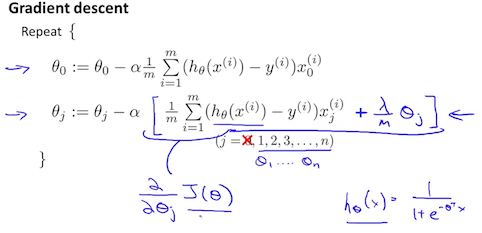
Recall that our cost function for logistic regression was:

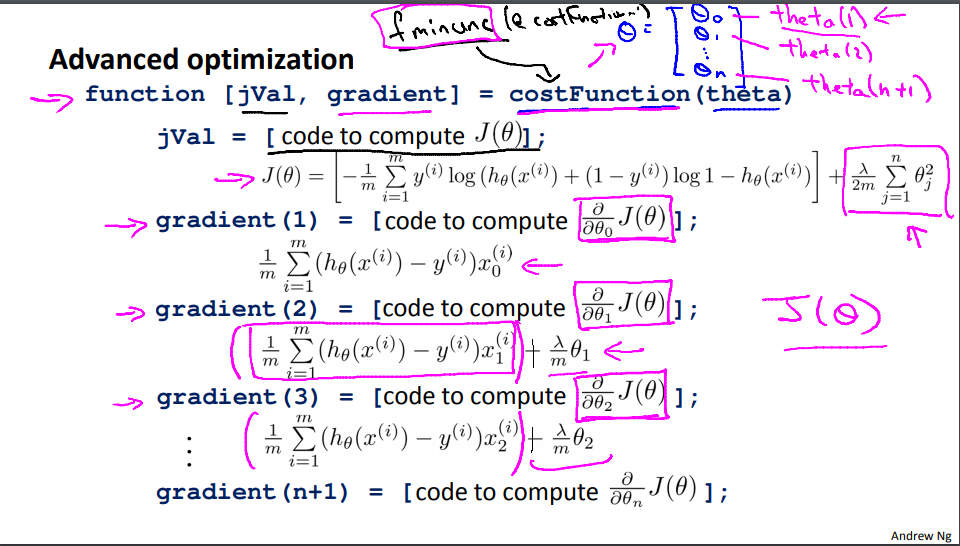


We can regularize this equation by adding a term to the end:



The second sum, \sum\_{j=1}^n \theta\_j^2∑*j*=1*n*​*θj*2​ **means to explicitly exclude** the bias term, \theta\_0*θ*0​. I.e. the θ vector is indexed from 0 to n (holding n+1 values, \theta\_0*θ*0​ through \theta\_n*θn*​), and this sum explicitly skips \theta\_0*θ*0​, by running from 1 to n, skipping 0. Thus, when computing the equation, we should continuously update the two following equations:

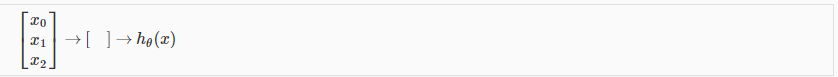




# **Model Representation I**

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\cdots x\_n*x*1​⋯*xn*​, and the output is the result of our hypothesis function. In this model our x\_0*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^Tx}}1+*e*−*θTx*1​, yet we sometimes call it a sigmoid (logistic) **activation** function. In this situation, our "theta" parameters are sometimes called "weights".

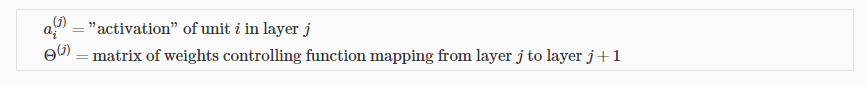
Visually, a simplistic representation looks like:



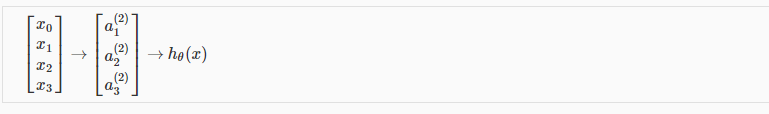
Our input nodes (layer 1), also known as the "input layer", go into another node (layer 2), which finally outputs the hypothesis function, known as the "output layer".

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

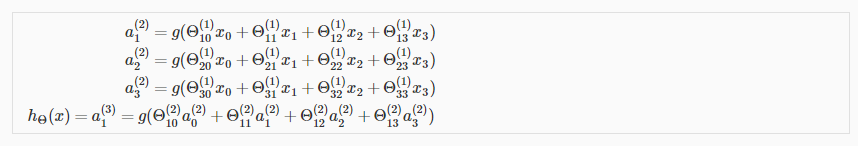
In this example, we label these intermediate or "hidden" layer nodes a^2\_0 \cdots a^2\_n*a*02​⋯*an*2​ and call them "activation units."



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



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



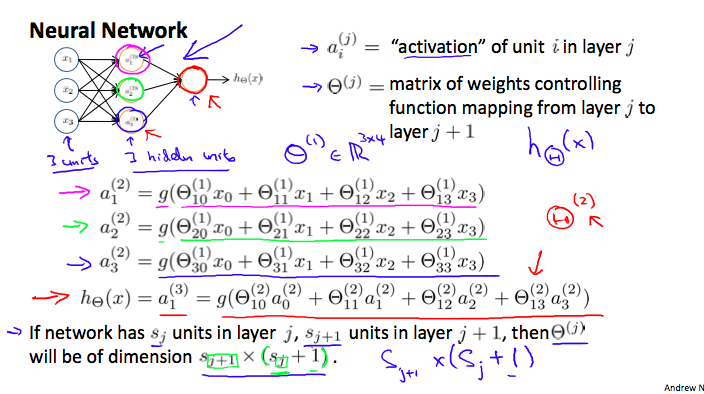
This is saying that we compute our activation nodes by using a 3×4 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied by yet another parameter matrix \Theta^{(2)}Θ(2) containing the weights for our second layer of nodes.

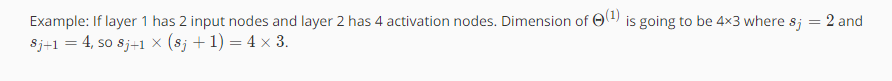
Each layer gets its own matrix of weights, \Theta^{(j)}Θ(*j*).

The dimensions of these matrices of weights is determined as follows:

\text{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 $s\_{j+1} \times (s\_j + 1)$.}If network has *sj*​ units in layer *j* and *sj*+1​ units in layer *j*+1, then Θ(*j*) will be of dimension *sj*+1​×(*sj*​+1).

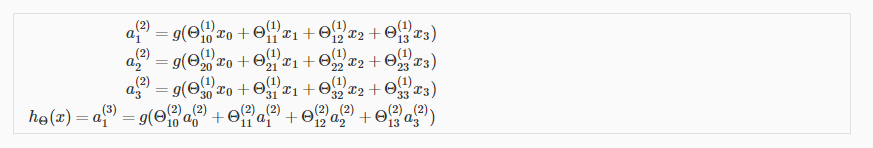
The +1 comes from the addition in \Theta^{(j)}Θ(*j*) of the "bias nodes," x\_0*x*0​ and \Theta\_0^{(j)}Θ0(*j*)​. In other words the output nodes will not include the bias nodes while the inputs will. The following image summarizes our model representation:



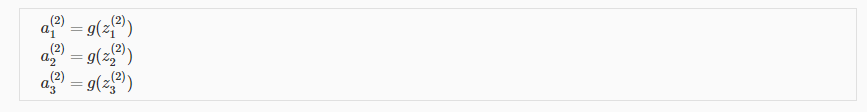


# **Model Representation II**

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



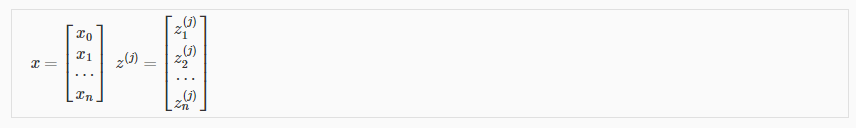
In this section we'll do a vectorized implementation of the above functions. We're going to define a new variable z\_k^{(j)}*zk*(*j*)​ that encompasses the parameters inside our g function. In our previous example if we replaced by the variable z for all the parameters we would get:



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

****

The vector representation of x and z^{j}*zj* is:

****

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

****

We are multiplying our matrix \Theta^{(j-1)}Θ(*j*−1) with dimensions s\_j\times (n+1)*sj*​×(*n*+1) (where s\_j*sj*​ is the number of our activation nodes) by our vector a^{(j-1)}*a*(*j*−1) with height (n+1). This gives us our vector z^{(j)}*z*(*j*) with height s\_j*sj*​. Now we can get a vector of our activation nodes for layer j as follows:

****

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

We can then add a bias unit (equal to 1) to layer j after we have computed a^{(j)}*a*(*j*). This will be element a\_0^{(j)}*a*0(*j*)​ and will be equal to 1. To compute our final hypothesis, let's first compute another z vector:

****

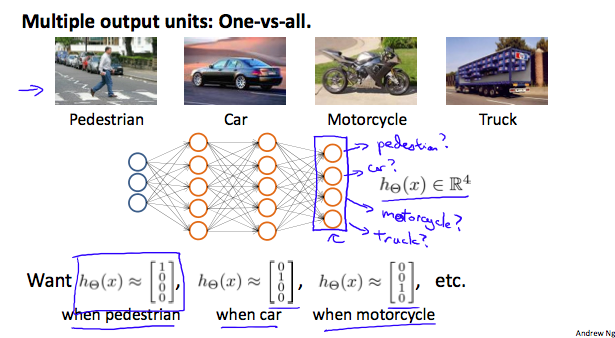
We get this final z vector by multiplying the next theta matrix after \Theta^{(j-1)}Θ(*j*−1) with the values of all the activation nodes we just got. This last theta matrix \Theta^{(j)}Θ(*j*) will have only **one row**which is multiplied by one column a^{(j)}*a*(*j*) so that our result is a single number.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



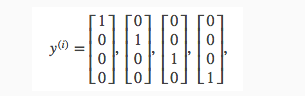
Notice that in this **last step**, between layer j and layer j+1, we are doing **exactly the same thing** as we did in logistic regression. Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypotheses.

# **Multiclass Classification**

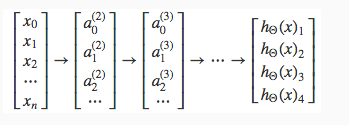
To classify data into multiple classes, we let our hypothesis function return a vector of values. Say we wanted to classify our data into one of four categories. We will use the following example to see how this classification is done. This algorithm takes as input an image and classifies it accordingly:



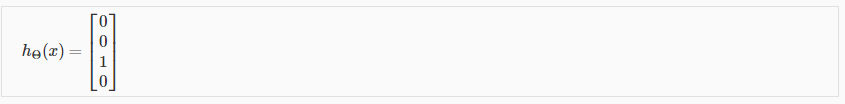
We can define our set of resulting classes as y:



Each y^{(i)}*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:



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



In which case our resulting class is the third one down, or h\_\Theta(x)\_3*h*Θ​(*x*)3​, which represents the motorcycle.

# **Cost Function**

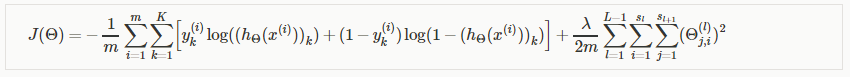
Let's first define a few variables that we will need to use:

* L = total number of layers in the network
* s\_l*sl*​ = 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*h*Θ​(*x*)*k*​ as being a hypothesis that results in the k^{th}*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:



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



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 Algorithm**

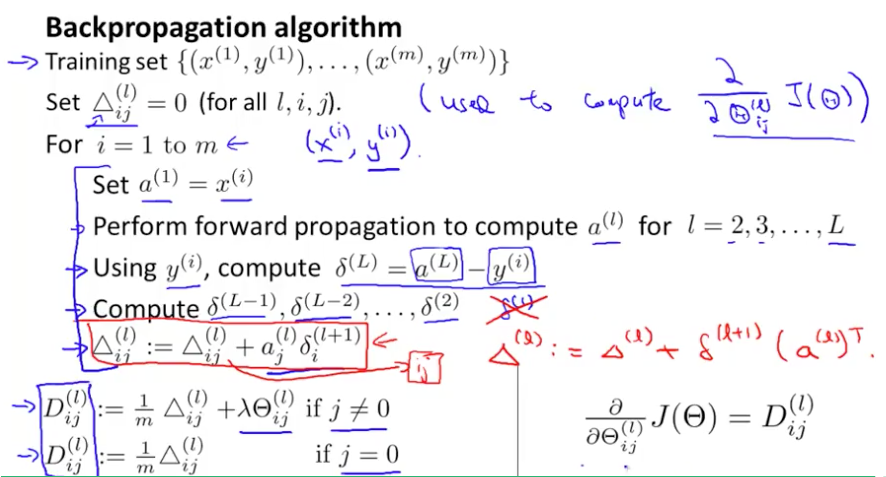
"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute:

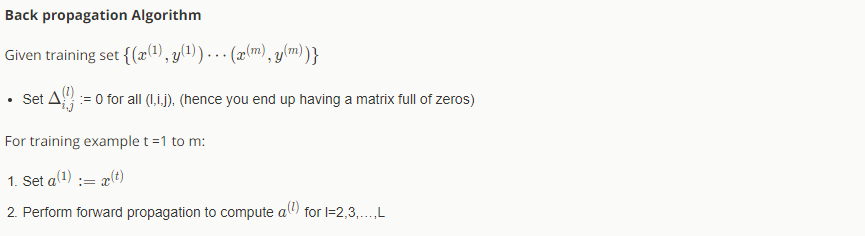


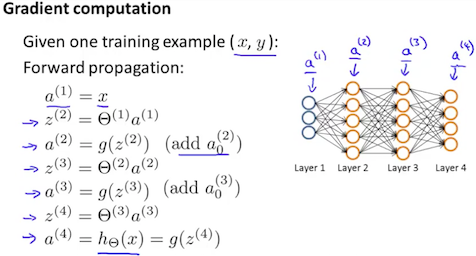
That is, we want to minimize our cost function J using an optimal set of parameters in theta. In this section we'll look at the equations we use to compute the partial derivative of J(Θ):

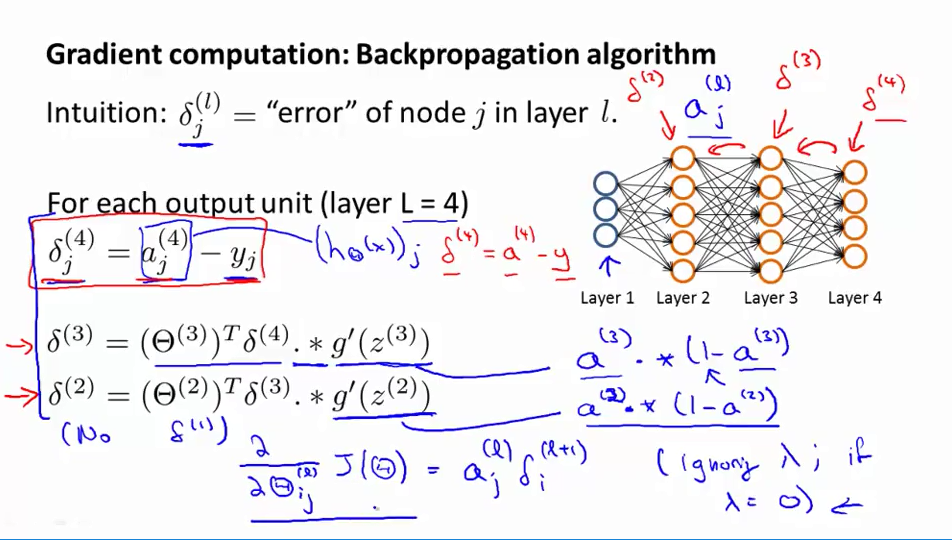


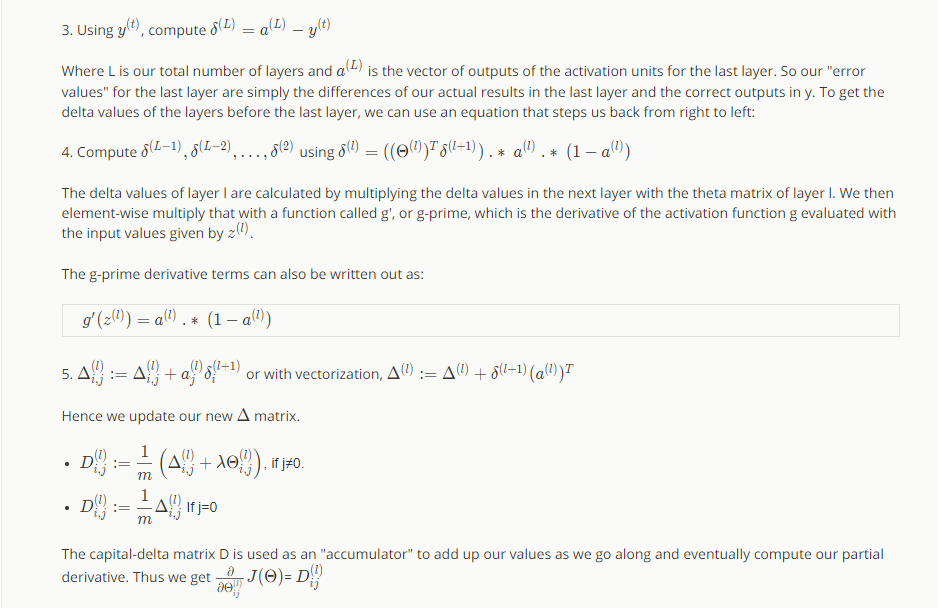
To do so, we use the following algorithm:

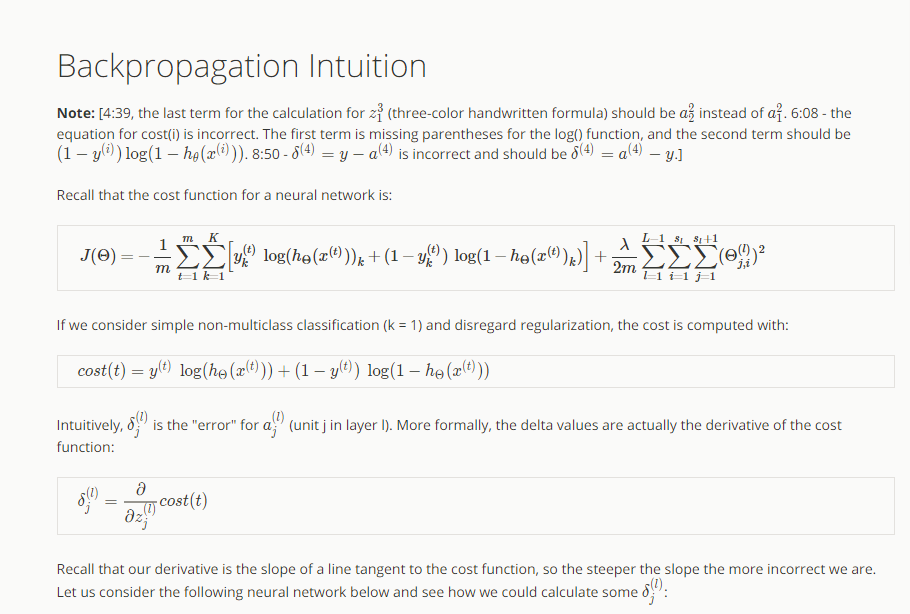


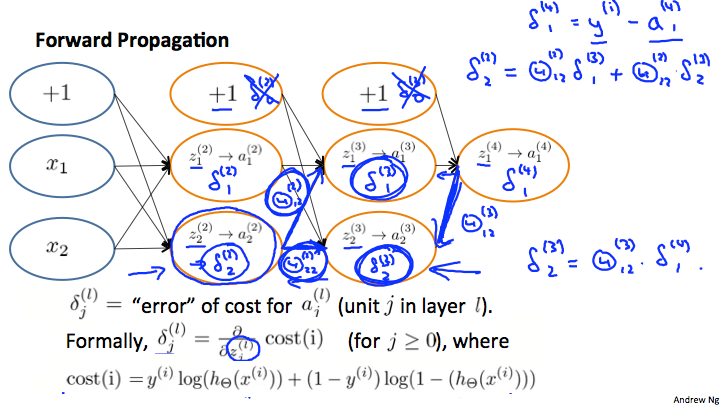


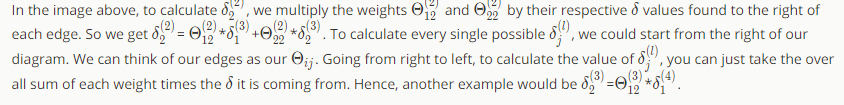






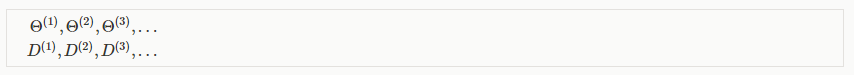






# **Implementation Note: Unrolling Parameters**

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



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



To summarize:

