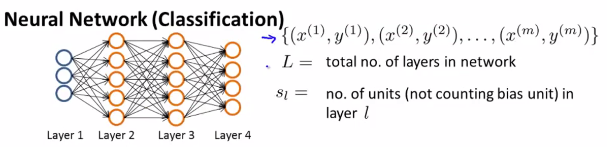
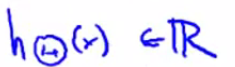
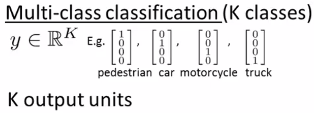
***Neural Networks – Learning: Cost Function and Backpropagation***

**I. COST FUNCTION**

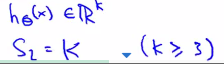
* NN are 1 of the most powerful learning algorithms we have today.
* We need a learning algorithm for fitting the parameters of a NN given a training set 🡪 cost function
* Let’s focus on the application of NNs to classification problems.
* 
* Suppose we have a network like this w/ a training set from x(1),y(1) to x(m)y(m), where L = the total number of layers in this network and S(L) = the number of units/neurons (excluding bias) in a layer
* For example, S(1) = 3 + S(2) = 5.
* We're going to consider 2 types of classification problems.
* The first is **Binary classification**, where the labels y are either 0 or 1
* In this case, we have 1 output unit that computes h(x), which is going to be a real number.

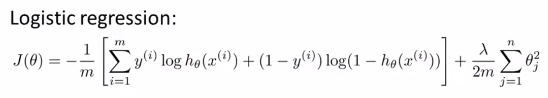


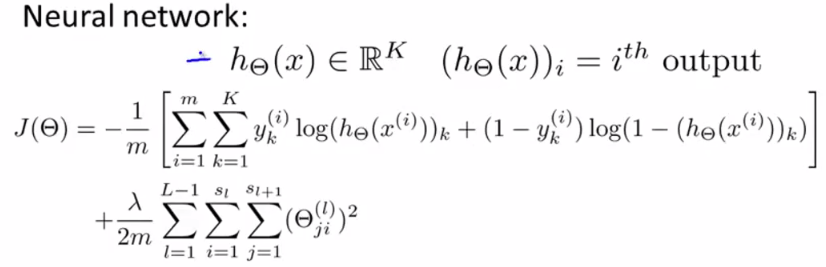
* Also, the number of output units, S(L), where L is the index of the final layer, is = 1
* To simplify notation later, set K = 1 + think of it as also denoting # of units in the output layer
* The second type of classification problem we'll consider will be **multi-class classification problem** w/ K distinct classes.



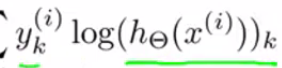
* See the representation for y if we have 4 classes, that we will have K output units, that our h(x) outputs vectors that are K-dimensional, + that the number of output units = K.



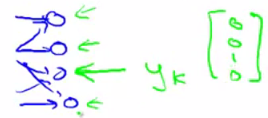
* We have a “K >= 3” constraint b/c if we had 2 classes, we don't need to use **the 1-vs-all method.**
* Use 1-vs-all method only if we have K < 3 classes b/c having only 2 classes requires the need of only 1 output unit.
* Now let's define the cost function for our NN, which is going to be a generalization of the one used for logistic regression.
* 
* For logistic regression we minimized the cost function J(ϴ) via minus 1/m multiplied by the cost function + an extra regularization term as a sum from J = 1 b/c we don’t regularize the bias term ϴ0.
* For a NN, our cost function is a generalization of this where instead of having just 1 logistic regression output unit, we may instead have K of them.



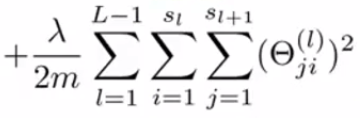
* Our NN outputs vectors in R(K) where R might be = 1 if we have a binary classification problem (i.e. K-dimensional)
* h(x)(i) denotes the ith output 🡪 h(x) is a K-dimensional vector so this subscript i selects out the ith element of the vector output by the NN
* The cost function J(ϴ) is then -1/m multiplied by a similar term to the one for logistic regression, except here we have the sum from k = 1 through K.
* This summation is basically a sum over the K output units
* So w/ 4 output units (in the final layer of NN), then this is a sum from k = 1 through 4 of basically the logistic regression algorithm's cost function over each of my 4 output units in turn.
* Notice, in particular, that this applies to y(k)h(k)



* This is b/c we're basically taking the Kth output unit + comparing that to the value of y(k), which is a vector saying what class should be.



* Finally, the 2nd term is the **regularization term**, similar to what we had for the logistic regression.



* This looks really complicated, but all it's doing is it's summing over these ϴ(j,i)^l for all values of i,j + l
* And we still don’t sum over the terms corresponding to the bias values like we have for logistic
* *Concretely, we don't sum over the terms corresponding to where i = 0.*
* This is b/c when computing the activation of a neuron, we have terms like these:



* The values w/a 0 corresponds to something that multiplies into x0 or a0.

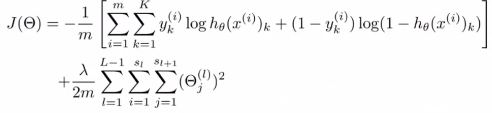


* So this is like a bias unit, and we won't sum over those terms in our regularization term b/c we don't want to regularize them + string their values as 0.
* But this is just *one possible convention*
* Even if you were to sum over i = 0 up to S(L), it would work about the same + doesn't make a big difference.
* But maybe this convention of NOT regularizing the bias term is just slightly more common.
* 

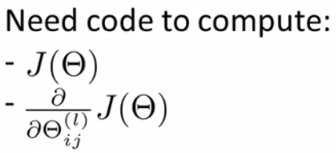
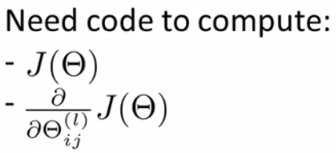
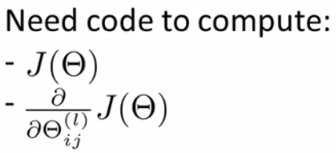


**II. BACKPROPOGATION ALGORITHM**

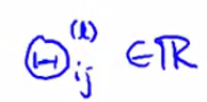
* The **backpropagation algorithm** is an algorithm for trying to minimize the NN cost function



* What we'd like to do is try to find parameters ϴ to try to minimize J(ϴ).
* In order to use either gradient descent or one of the advanced optimization algorithms, we need to write code that takes as input the parameters ϴ + computes J(ϴ) + the partial derivative terms.

 + 

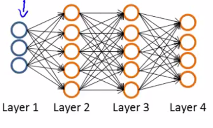
* Remember, the parameters ϴ in a NN is = ϴ(ij)l, which is real number, + can be seen in the partial derivative terms we need to compute.



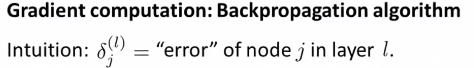
* Ex: Only 1 one training example: = a pair (x1,y1).
* The 1st thing we do is we apply **forward propagation** in order to compute whether a hypotheses actually outputs given the input x.

* Concretely, recall a(1) = the **activation values** of the 1st/input layer.



* Set a(1) to x + to compute z(2) = ϴ(1)\*a(1) and then a(2) = g(z(2)), i.e the **sigmoid activation function** applied to z(2)
* This would give us our activations for the 1st middle layer
* Then apply this 2 more times in our forward propagation to compute a(3) and a(4), which is also the output of a hypotheses hϴ(x).
* This is our *vectorized implementation* of forward propagation + it allows us to compute the activation values for all of the neurons in our NN.
* Next, in order to compute the derivatives, we're going to use an algorithm called **back propagation**.



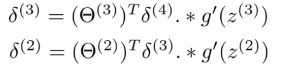
* The intuition of the back propagation algorithm is that for each node, we're going to compute the term **δj(l)** which represent the error of node j in layer l.
* Recall aj(l) is the activation of the jth unit in layer l and so this *delta term is in some sense going to capture our error in the activation of that node*.
* Concretely, taking our example NN w/ 4 layers (L = 4):
* For each output unit, we're going to compute this delta term.
* So, delta for the jth unit in the 4th layer is:



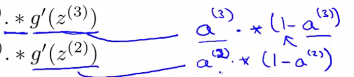
* i.e. the activation of that unit minus the actual value observed in our training example.
* Aj(4) can also be written as hϴ(x)(j), so delta is the difference between our hypothesis (prediction) and our actual
* If you think of delta, a, and y as vectors, you can also take the above + come up w/ a vectorized implementation of it:



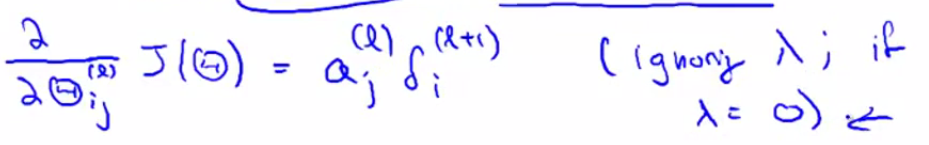
* delta4, a4, + y are vectors whose dimensions are = the number of output units in the network.
* What we do *next* is compute the delta terms for the *earlier* layers in our network.



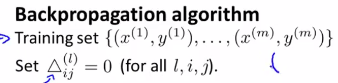
* The “ .\* ” or “dot times” is the element-wise multiplication operation for the 2 vectors
* gprime(z(3)) = derivative of the activation function g evaluated at the input values given by z(3).
* What you do to compute these derivative terms is just a(3) .\* (1 – a(3)) where a(3) = the vector of activation values for that layer and 1 is the vector of ones.



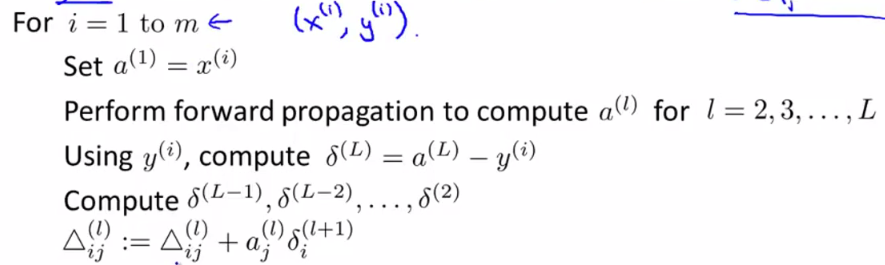
* Next you apply a similar formula to compute delta2
* There is no delta1 term, b/c the 1st layer corresponds to the input layer which are just the features we observed in our training sets, so there’s no error associated w/ them
* The name **back propagation** comes from the fact that we start by computing the delta term for the *output* layer + then *go back* a layer + compute the delta terms for the 3rd hidden layer and so on
* The derivation is surprisingly complicated + involved
* But if you just do these few steps of computation of delta4 to delta2, it’s possible to prove (via, frankly, somewhat complicated mathematical proofs) that if you ignore regularization, the partial derivative terms you want are exactly given by the activations + delta terms (ignoring lambda)



* So this is a lot of detail. Let's take everything + put it all together to talk about how to implement back propagation to compute derivatives w/ respect to your parameters.
* Suppose we have a large training set of m examples, the 1st we do is set these Ϫ(ij)l = 0 for all values of l, I, j.



* Eventually, this Ϫ(i,j)l will be used to compute the partial derivatives of J(ϴ) w/ respect to ϴ(i,j)l.
* These deltas are going to be used as accumulators that will slowly add things in order to compute these partial derivatives.
* Next, we're going to loop through our training set.



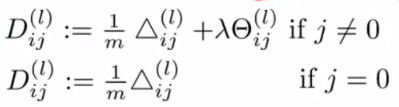
* The 1st thing we're going to do is set a(1) (activation values of input layer) to be = x(i) (inputs for our ith training example)
* Then **forward propagate** to compute the activation values for layer 2 up to the final layer, L
* Next, use the output label y(i) from this specific example to compute the error term **δL** for the output (hypotheses output - the target label)
* Then **back propagate** to compute δ(L – 1), δ(L – 2), down to δ2 (no δ1 b/c we don't associate an error term with the input layer)
* Finally, **accumulate** the partial derivative terms
* And by the way, if you look at the last line, it's possible to vectorize this as well.



* Concretely, if you think of Ϫ(i,j) as a matrix indexed by subscript (i,j), then if ϪL is a matrix, we can rewrite this as ϪL gets updated as ϪL + δ(L + 1)\*a(L)(t)



* This automatically does an update for all values of i and j.
* Finally, after executing the body of FOR loop, we go outside it + compute the following:



* We have 2 separate cases for j = 0 + j != 0 where the case j = 0 corresponds to the bias term, so that's why we're missing an extra regularization term.
* While the formal proof is pretty complicated, it shows that, once you've computed this D term, they are exactly the partial derivative of the cost function w/ respect to each of your parameters

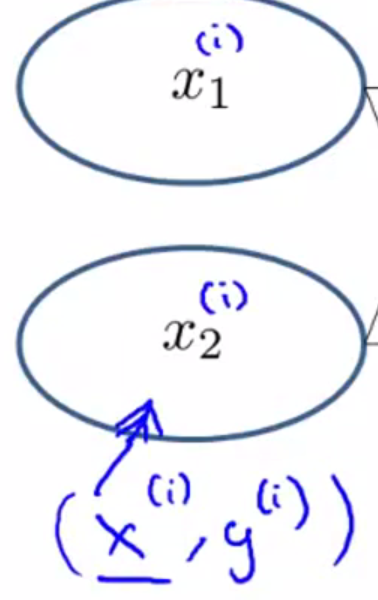


* You can use those in either gradient descent or in one of the advanced authorization algorithms.
* 

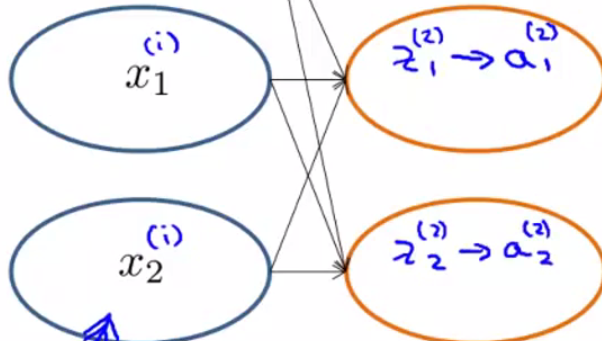


**III. BACKPROPOGATION INTUITION**

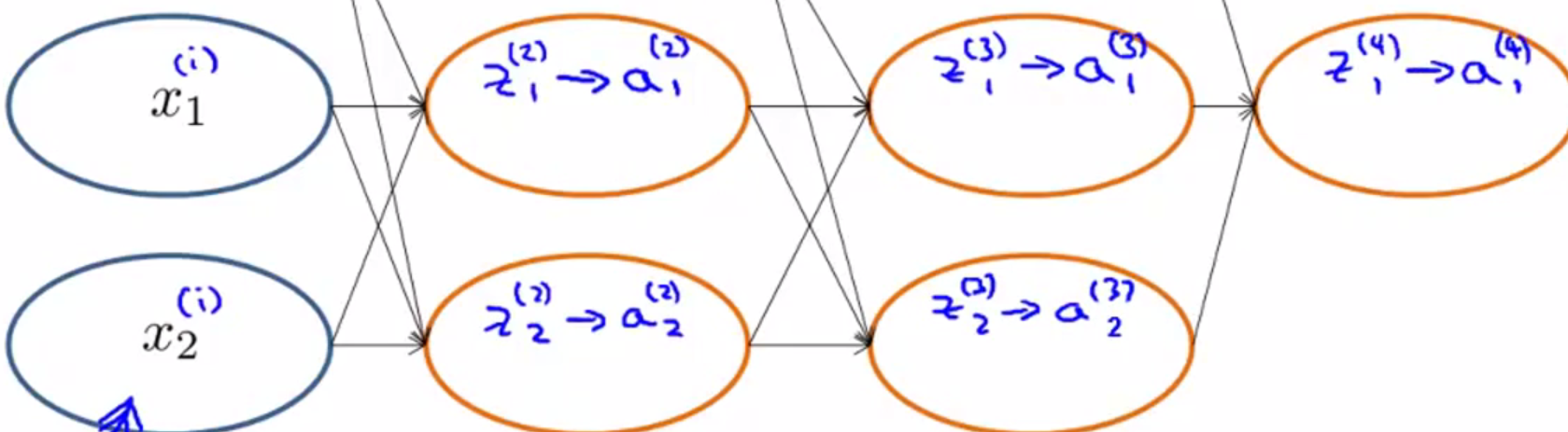
* To a lot of people seeing BP for the 1st time, their 1st impression is often that it’s a really complicated algorithm w/ all these different steps, kind of like a black box
* Backpropagation is, unfortunately, a less-mathematically-clean/simple algorithm, compared to linear or logistic regression.
* Even after using used BP for many years, sometimes it can still be a difficult algorithm to understand
* In order to better understand *back*propagation, let's take another closer look at what *forward* propagation is doing.
* Imagine a NN w/ 2 input units (not counting the bias unit) + 2 hidden units each hidden layer, + 1 output unit.
* When performing forward propagation, we might have some particular example, {x(i), y(i)}, + we feed those x(i) values into the input layer

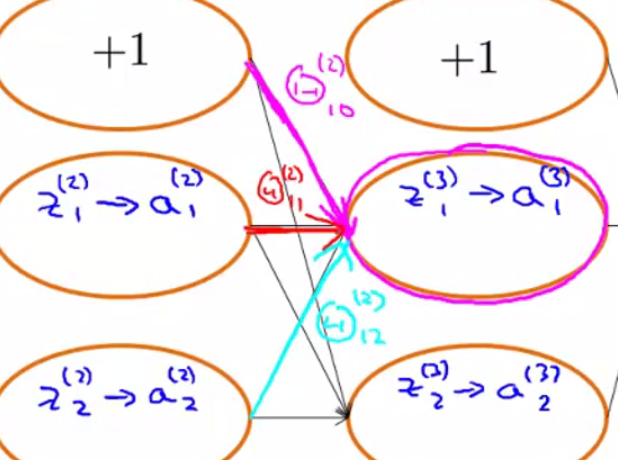


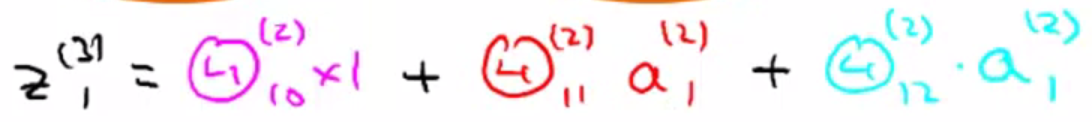
* When we forward propagated to the 1st hidden layer, we compute z2(1) + z2(2) (the weighted sum of inputs of the input units) + apply the sigmoid logistic/activation function to the z value to get the activation values, a2(1) and a2(2)



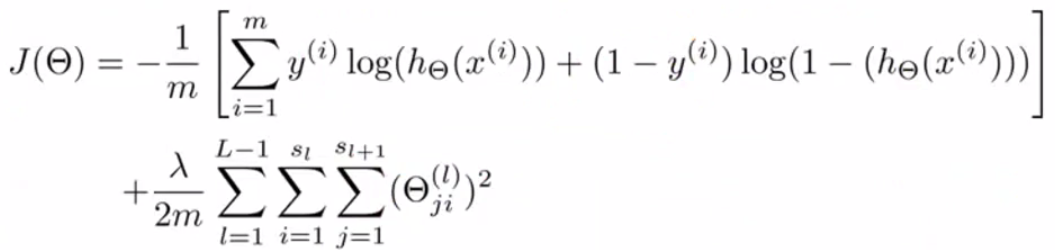
* Then we forward propagate again to get z3(1) + z3(2) + again apply the sigmoid logistic/activation function to get a3(1) + a3(2)
* Then continue on until we get z4(1), apply the activation function, + get a4(1), the final output value of the NN.

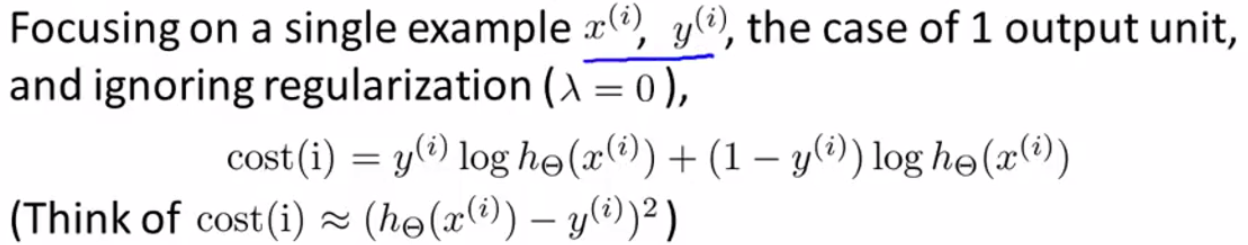


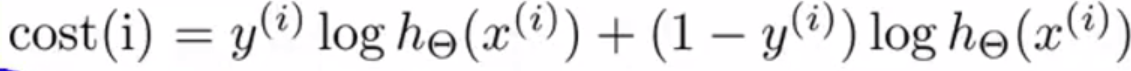
* If we look at what this computation is really doing, focusing on one hidden unit, is adding **weight**.
* 
* The way we compute this z3(1) is the weight multiplied by the prior values of a they are attached to:



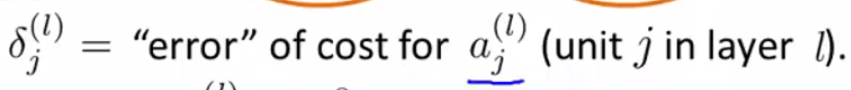
* That's *forward* propagation.
* What *back*propagation is doing is doing a process very similar to this, except instead of computations flowing from the left to the right of this network, they flow from the right to the left
* To better understand what backpropagation is doing, let's look at the cost function.



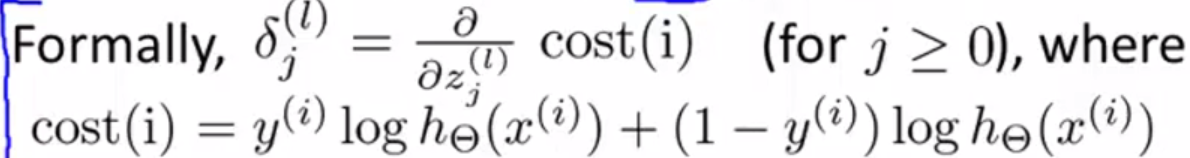
* *\*\*\*\*\*This is the cost function we had for when we have only 1 output unit.*
* *If we have > 1 output unit, we just have a summation over the output units indexed by k*
* We do forward propagation and backpropagation on 1 example {x(i),y(i)} at a time in the case of having 1 output unit, y(i), which is a real number, + ignoring regularization (lambda = 0)
* 
* Now if you look inside the summation in the original cost function, you find that the cost term associated w/ training example {x(i),y(i)} is given by that 2nd, simpler expression.



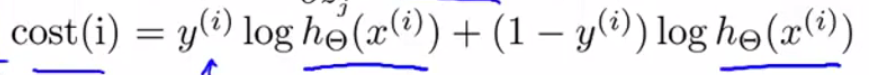
* This this cost function plays a role similar to squared error.
* So, rather than looking at this complicated expression, think of cost(i) being approximately the squared difference between what the NN outputs vs the actual value.
* Just as in logistic regression, we prefer to use the slightly more complicated function using log, but for the purpose of intuition, feel free to think of the cost function as being like the squared error cost function.
* This cost(i) measures how well the network is doing on correctly predicting example I (i.e. How close the output is to the actual observed label y(i))
* Now let's look at what backpropagation is doing.
* 1 useful intuition is that backpropagation is computing δj(l) terms, the “error” of the activation value that we got for unit j in the lth layer.



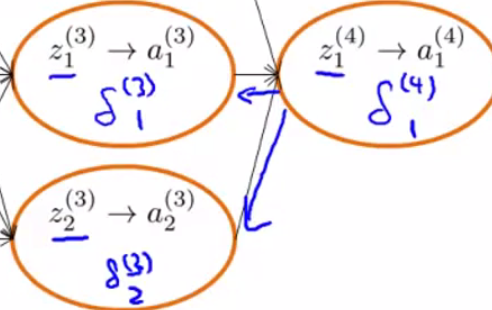
* More formally, what the delta terms actually are is the partial derivatives of cost(i) w/ respect to zj(l)/the weighted sum of inputs computing these z terms.



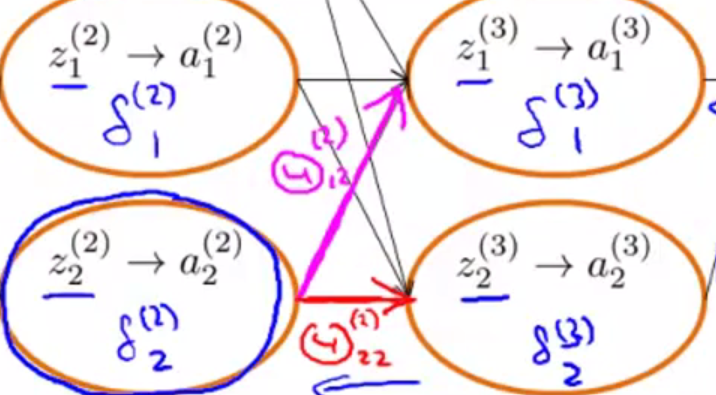
* Concretely, the cost function is a function of the label y and of the h(x) output value



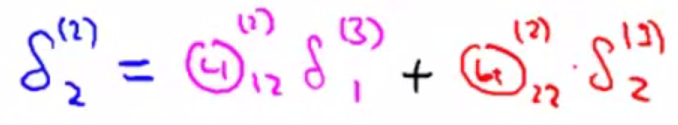
* If we could go inside the NN + change those zj(l) values a little bit, it will affect the h(x) values the NN is outputting + will end up changing the cost function.
* So, again, these delta terms are the partial derivatives of the cost function w/ respect to these intermediate terms were computing.
* They're a measure of how much we’d like to change the NN’s weights in order to affect the intermediate values of the computation so as to affect the final output of the NN, h(x), + therefore affect the overall cost.
* For the output layer, backpropagation first sets δ4(1) = y(i) – a4(1) for the ith training example
* This is really calculating the error, right? It's the difference between the actual value of y minus the value predicted
* Next we propagate these values backwards + end up computing the δ terms for the previous layer



* A backpropagation calculation is a lot like running a forward propagation algorithm, but backwards
* Ex: δ2(2)



* So δ2(2) is computed by taking δ3(1) multiplied by its weight (pink) + δ3(2) multiplied by its weight red) 🡪 *a sum of these delta values weighted by the corresponding edge strength*





* So far we've been writing δ values only for hidden units, excluding the bias units.
* Depending on how you define or implement the backpropagation algorithm, you may end up computing δ values for these bias units as well, which always output +1, + there's no way to change the value.
* \*\*\*They don't end up being part of the calculation needed to compute a derivative.
* If you implement this algorithm you can have a very effective learning algorithm, even though the inner workings of exactly how it works can be harder to visualize.