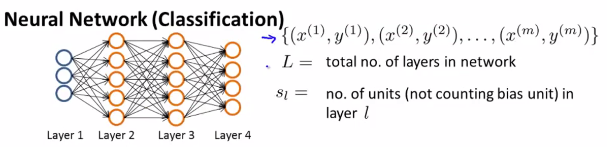
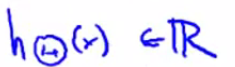
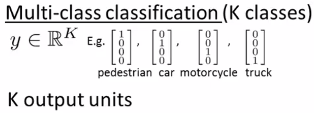
***Neural Networks – Learning: Cost Function and Backpropagation***

**I. COST FUNCTION**

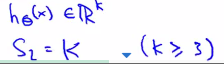
* NN are 1 of the most powerful learning algorithms we have today.
* We need a learning algorithm for fitting the parameters of a NN given a training set 🡪 cost function
* Let’s focus on the application of NNs to classification problems.
* 
* Suppose we have a network like this w/ a training set from 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.

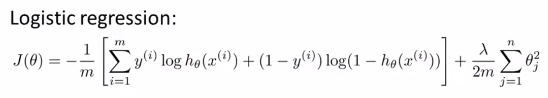


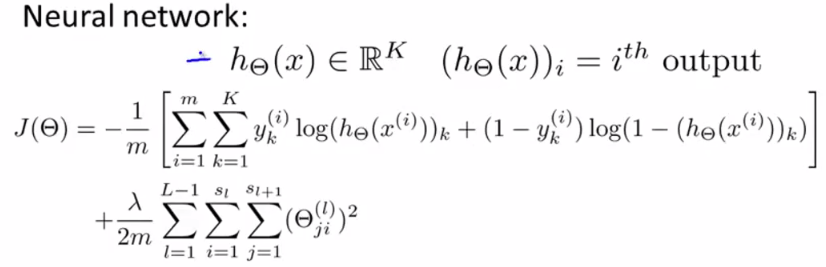
* Also, the number of output units, S(L), where L is the index of the final layer, is = 1
* To simplify notation later, set K = 1 + think of it as also denoting # of units in the output layer
* The second type of classification problem we'll consider will be **multi-class classification problem** w/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



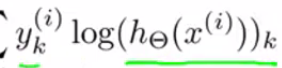
* See the representation for y if we have 4 classes, that we will have K output units, that our h(x) outputs vectors that are K-dimensional, + that the number of output units = K.



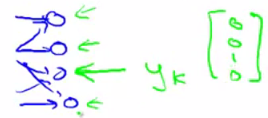
* We have a “K >= 3” constraint b/c if we had 2 classes, we don't need to use **the 1-vs-all method.**
* Use 1-vs-all method only if we have K < 3 classes b/c having only 2 classes requires the need of only 1 output unit.
* Now let's define the cost function for our NN, which is going to be a generalization 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.



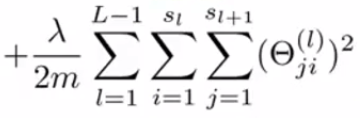
* Our NN outputs vectors in R(K) where R might be = 1 if we have a binary classification problem (i.e. K-dimensional)
* h(x)(i) denotes the ith output 🡪 h(x) is a K-dimensional vector so this subscript i selects out the ith element of the vector output by the NN
* The cost function J(ϴ) is then -1/m 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 my K output units
* So if I have 4 output units (final layer of NN has 4 output units), 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 cost function
* Here's the cost function that we wrote down in the previous video. What we'd like to do is try to find parameters theta to try to minimize j of theta. In order to use either gradient descent or one of the advance optimization algorithms. What we need to do therefore is to write code that takes this input the parameters theta and computes j of theta and these partial derivative terms. Remember, that the parameters in the the neural network of these things, theta superscript l subscript ij, that's the real number and so, these are the partial derivative terms we need to compute. In order to compute the cost function j of theta, we just use this formula up here and so, what I want to do for the most of this video is focus on talking about how we can compute these partial derivative terms. Let's start by talking about the case of when we have only one training example, so imagine, if you will that our entire training set comprises only one training example which is a pair xy. I'm not going to write x1y1 just write this. Write a one training example as xy and let's tap through the sequence of calculations we would do with this one training example.
* 1:25
* The first thing we do is we apply forward propagation in order to compute whether a hypotheses actually outputs given the input. Concretely, the called the a(1) is the activation values of this first layer that was the input there. So, I'm going to set that to x and then we're going to compute z(2) equals theta(1) a(1) and a(2) equals g, the sigmoid activation function applied to z(2) and this would give us our activations for the first middle layer. That is for layer two of the network and we also add those bias terms. Next we apply 2 more steps of this four and propagation to compute a(3) and a(4) which is also the upwards of a hypotheses h of x. So this is our vectorized implementation of forward propagation and it allows us to compute the activation values for all of the neurons in our neural network.
* 2:27
* Next, in order to compute the derivatives, we're going to use an algorithm called back propagation.
* 2:34
* The intuition of the back propagation algorithm is that for each note we're going to compute the term delta superscript l subscript j that's going to somehow represent the error of note j in the layer l. So, recall that a superscript l subscript j that does the activation of the j of unit in layer l and so, this delta term is in some sense going to capture our error in the activation of that neural duo. So, how we might wish the activation of that note is slightly different. Concretely, taking the example neural network that we have on the right which has four layers. And so capital L is equal to 4. For each output unit, we're going to compute this delta term. So, delta for the j of unit in the fourth layer is equal to
* 3:23
* just the activation of that unit minus what was the actual value of 0 in our training example.
* 3:29
* So, this term here can also be written h of x subscript j, right. So this delta term is just the difference between when a hypotheses output and what was the value of y in our training set whereas y subscript j is the j of element of the vector value y in our labeled training set.
* 3:56
* And by the way, if you think of delta a and y as vectors then you can also take those and come up with a vectorized implementation of it, which is just delta 4 gets set as a4 minus y. Where here, each of these delta 4 a4 and y, each of these is a vector whose dimension is equal to the number of output units in our network.
* 4:25
* So we've now computed the era term's delta 4 for our network.
* 4:31
* What we do next is compute the delta terms for the earlier layers in our network. Here's a formula for computing delta 3 is delta 3 is equal to theta 3 transpose times delta 4. And this dot times, this is the element y's multiplication operation
* 4:47
* that we know from MATLAB. So delta 3 transpose delta 4, that's a vector; g prime z3 that's also a vector and so dot times is in element y's multiplication between these two vectors.
* 5:01
* This term g prime of z3, that formally is actually the derivative of the activation function g evaluated at the input values given by z3. If you know calculus, you can try to work it out yourself and see that you can simplify it to the same answer that I get. But I'll just tell you pragmatically what that means. What you do to compute this g prime, these derivative terms is just a3 dot times1 minus A3 where A3 is the vector of activations. 1 is the vector of ones and A3 is again the activation the vector of activation values for that layer. Next you apply a similar formula to compute delta 2 where again that can be computed using a similar formula.
* 5:48
* Only now it is a2 like so and I then prove it here but you can actually, it's possible to prove it if you know calculus that this expression is equal to mathematically, the derivative of the g function of the activation function, which I'm denoting by g prime. And finally, that's it and there is no delta1 term, because the first layer corresponds to the input layer and that's just the feature we observed in our training sets, so that doesn't have any error associated with that. It's not like, you know, we don't really want to try to change those values. And so we have delta terms only for layers 2, 3 and for this example.
* 6:30
* The name back propagation comes from the fact that we start by computing the delta term for the output layer and then we go back a layer and compute the delta terms for the third hidden layer and then we go back another step to compute delta 2 and so, we're sort of back propagating the errors from the output layer to layer 3 to their to hence the name back complication.
* 6:51
* Finally, the derivation is surprisingly complicated, surprisingly involved but if you just do this few steps steps of computation it is possible to prove viral frankly some what complicated mathematical proof. It's possible to prove that if you ignore authorization then the partial derivative terms you want
* 7:12
* are exactly given by the activations and these delta terms. This is ignoring lambda or alternatively the regularization
* 7:23
* term lambda will equal to 0. We'll fix this detail later about the regularization term, but so by performing back propagation and computing these delta terms, you can, you know, pretty quickly compute these partial derivative terms for all of your parameters. So this is a lot of detail. Let's take everything and put it all together to talk about how to implement back propagation
* 7:46
* to compute derivatives with respect to your parameters.
* 7:49
* And for the case of when we have a large training set, not just a training set of one example, here's what we do. Suppose we have a training set of m examples like that shown here. The first thing we're going to do is we're going to set these delta l subscript i j. So this triangular symbol? That's actually the capital Greek alphabet delta . The symbol we had on the previous slide was the lower case delta. So the triangle is capital delta. We're gonna set this equal to zero for all values of l i j. Eventually, this capital delta l i j will be used
* 8:26
* to compute the partial derivative term, partial derivative respect to theta l i j of J of theta.
* 8:39
* So as we'll see in a second, these deltas are going to be used as accumulators that will slowly add things in order to compute these partial derivatives.
* 8:49
* Next, we're going to loop through our training set. So, we'll say for i equals 1 through m and so for the i iteration, we're going to working with the training example xi, yi.
* 9:00
* So the first thing we're going to do is set a1 which is the activations of the input layer, set that to be equal to xi is the inputs for our i training example, and then we're going to perform forward propagation to compute the activations for layer two, layer three and so on up to the final layer, layer capital L. Next, we're going to use the output label yi from this specific example we're looking at to compute the error term for delta L for the output there. So delta L is what a hypotheses output minus what the target label was?
* 9:41
* And then we're going to use the back propagation algorithm to compute delta L minus 1, delta L minus 2, and so on down to delta 2 and once again there is now delta 1 because we don't associate an error term with the input layer.
* 9:57
* And finally, we're going to use these capital delta terms to accumulate these partial derivative terms that we wrote down on the previous line.
* 10:06
* And by the way, if you look at this expression, it's possible to vectorize this too. Concretely, if you think of delta ij as a matrix, indexed by subscript ij.
* 10:19
* Then, if delta L is a matrix we can rewrite this as delta L, gets updated as delta L plus
* 10:27
* lower case delta L plus one times aL transpose. So that's a vectorized implementation of this that automatically does an update for all values of i and j. Finally, after executing the body of the four-loop we then go outside the four-loop and we compute the following. We compute capital D as follows and we have two separate cases for j equals zero and j not equals zero.
* 10:56
* The case of j equals zero corresponds to the bias term so when j equals zero that's why we're missing is an extra regularization term.
* 11:05
* Finally, while the formal proof is pretty complicated what you can show is that once you've computed these D terms, that is exactly the partial derivative of the cost function with respect to each of your perimeters and so you can use those in either gradient descent or in one of the advanced authorization
* 11:25
* algorithms.
* 11:28
* So that's the back propagation algorithm and how you compute derivatives of your cost function for a neural network. I know this looks like this was a lot of details and this was a lot of steps strung together. But both in the programming assignments write out and later in this video, we'll give you a summary of this so we can have all the pieces of the algorithm together so that you know exactly what you need to implement if you want to implement back propagation to compute the derivatives of your neural network's cost function with respect to those parameters.