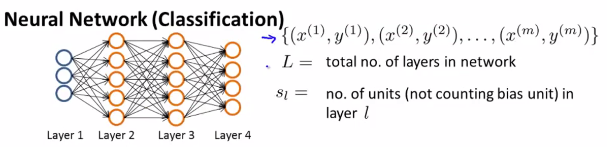
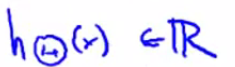
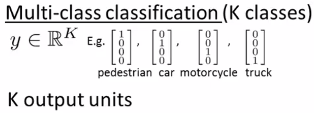
***Neural Networks – Learning: Cost Function and Backpropagation***

**I. COST FUNCTION**

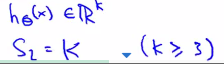
* NN are 1 of the most powerful learning algorithms we have today.
* We need a learning algorithm for fitting the parameters of a NN given a training set 🡪 cost function
* Let’s focus on the application of NNs to classification problems.
* 
* Suppose we have a network like this w/ a training set from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

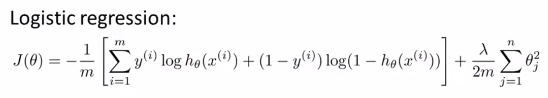


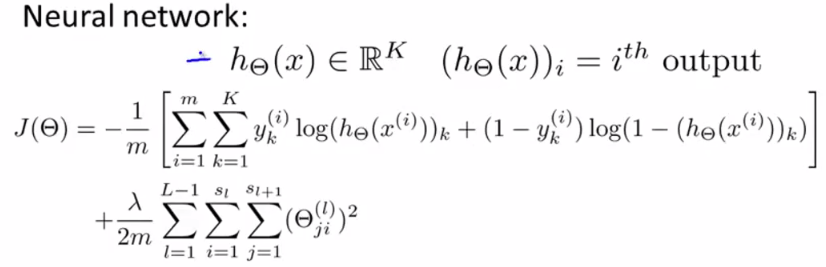
* Also, the number of output units, S(L), where L is the index of the final layer, is = 1
* To simplify notation later, set K = 1 + think of it as also denoting # of units in the output layer
* The second type of classification problem we'll consider will be **multi-class classification problem** w/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



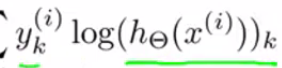
* See the representation for y if we have 4 classes, that we will have K output units, that our h(x) outputs vectors that are K-dimensional, + that the number of output units = K.



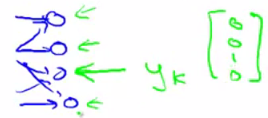
* We have a “K >= 3” constraint b/c if we had 2 classes, we don't need to use **the 1-vs-all method.**
* Use 1-vs-all method only if we have K < 3 classes b/c having only 2 classes requires the need of only 1 output unit.
* Now let's define the cost function for our NN, which is going to be a generalization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



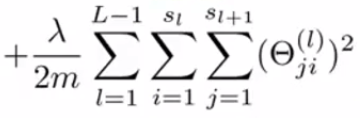
* Our NN outputs vectors in R(K) where R might be = 1 if we have a binary classification problem (i.e. K-dimensional)
* h(x)(i) denotes the ith output 🡪 h(x) is a K-dimensional vector so this subscript i selects out the ith element of the vector output by the NN
* The cost function J(ϴ) is then -1/m 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.

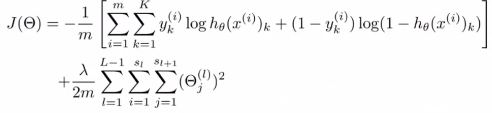


* So this is like a bias unit, and we won't sum over those terms in our regularization term b/c we don't want to regularize them + string their values as 0.
* But this is just *one possible convention*
* Even if you were to sum over i = 0 up to S(L), it would work about the same + doesn't make a big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

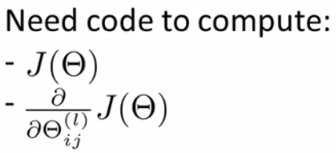
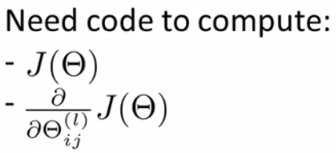
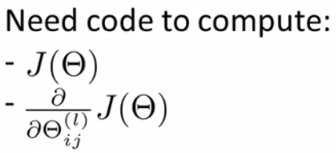


**II. BACKPROPOGATION ALGORITHM**

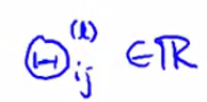
* The **backpropagation algorithm** is an algorithm for trying to minimize the NN cost function



* 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.

 + 

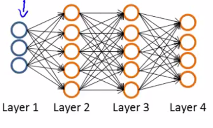
* Remember, the parameters ϴ in a NN is = ϴ(ij)l, which is real number, + can be seen in the partial derivative terms we need to compute.



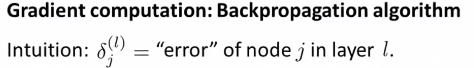
* Ex: Only 1 one training example: = a pair (x1,y1).
* The 1st thing we do is we apply **forward propagation** in order to compute whether a hypotheses actually outputs given the input x.

* 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**.



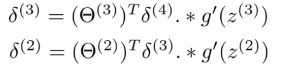
* The intuition of the back propagation algorithm is that for each node, we're going to compute the term **δj(l)** which represent the error of node j in layer l.
* Recall aj(l) is the activation of the jth unit in layer l and so this *delta term is in some sense going to capture our error in the activation 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:



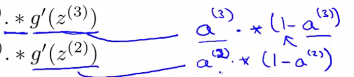
* i.e. the activation of that unit minus the actual value observed in our training example.
* Aj(4) can also be written as hϴ(x)(j), so delta is the difference between our hypothesis (prediction) and our actual
* If you think of delta, a, and y as vectors, you can also take the above + come up w/ a vectorized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



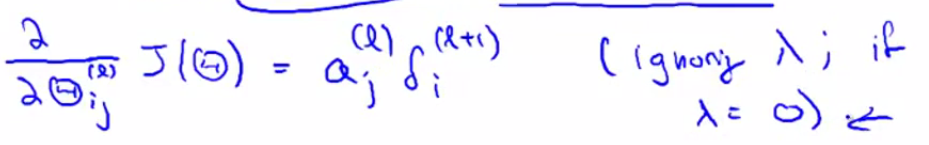
* delta4, a4, + y are vectors whose dimensions are = the number of output units in the network.
* What we do *next* is compute the delta terms for the *earlier* layers in our network.



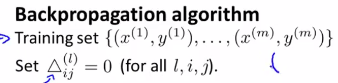
* The “ .\* ” or “dot times” is the element-wise multiplication operation for the 2 vectors
* gprime(z(3)) = derivative of the activation function g evaluated at the input values given by z(3).
* What you do to compute these derivative terms is just a(3) .\* (1 – a(3)) where a(3) = the vector of activation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



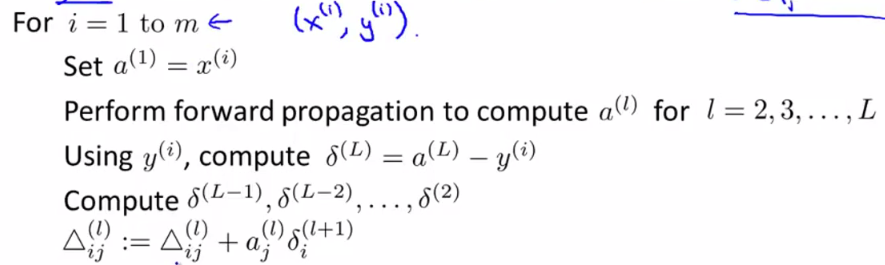
* Next you apply a similar formula to compute delta2
* There is no delta1 term, b/c the 1st layer corresponds to the input layer which are just the features we observed in our training sets, so there’s no error associated w/ them
* The name **back propagation** comes from the fact that we start by computing 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.



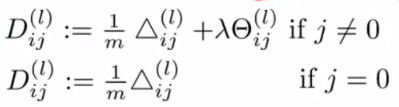
* The 1st thing we're going to do is set a(1) (activation values of input layer) to be = x(i) (inputs for our ith training example)
* Then **forward propagate** to compute the activation values for layer 2 up to the final layer, L
* Next, use the output label y(i) from this specific example to compute 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.
* Here's a NN w/ 2 input units (not counting the bias unit) + 2 hidden units each hidden layer, + 1 output unit.
* In order to illustrate forward propagation, I'm going to draw this network a little bit differently.
* 2:08
* And in particular I'm going to draw this neuro-network with the nodes drawn as these very fat ellipsis, so that I can write text in them. When performing forward propagation, we might have some particular example. Say some example x i comma y i. And it'll be this x i that we feed into the input layer. So this maybe x i 2 and x i 2 are the values we set the input layer to. And when we forward propagated to the first hidden layer here, what we do is compute z (2) 1 and z (2) 2. So these are the weighted sum of inputs of the input units. And then we apply the sigmoid of the logistic function, and the sigmoid activation function applied to the z value. Here's are the activation values. So that gives us a (2) 1 and a (2) 2. And then we forward propagate again to get here z (3) 1. Apply the sigmoid of the logistic function, the activation function to that to get a (3) 1. And similarly, like so until we get z (4) 1. Apply the activation function. This gives us a (4)1, which is the final output value of the neural network.
* 3:24
* Let's erase this arrow to give myself some more space. And if you look at what this computation really is doing, focusing on this hidden unit, let's say. We have to add this weight. Shown in magenta there is my weight theta (2) 1 0, the indexing is not important. And this way here, which I'm highlighting in red, that is theta (2) 1 1 and this weight here, which I'm drawing in cyan, is theta (2) 1 2. So the way we compute this value, z(3)1 is, z(3)1 is as equal to this magenta weight times this value. So that's theta (2) 10 x 1. And then plus this red weight times this value, so that's theta(2) 11 times a(2)1. And finally this cyan weight times this value, which is therefore plus theta(2)12 times a(2)1. And so that's forward propagation. And it turns out that as we'll see later in this video, what backpropagation is doing is doing a process very similar to this. Except that instead of the computations flowing from the left to the right of this network, the computations since their flow from the right to the left of the network. And using a very similar computation as this. And I'll say in two slides exactly what I mean by that. To better understand what backpropagation is doing, let's look at the cost function. It's just the cost function that we had for when we have only one output unit. If we have more than one output unit, we just have a summation you know over the output units indexed by k there. If you have only one output unit then this is a cost function. And we do forward propagation and backpropagation on one example at a time. So let's just focus on the single example, x (i) y (i) and focus on the case of having one output unit. So y (i) here is just a real number. And let's ignore regularization, so lambda equals 0. And this final term, that regularization term, goes away. Now if you look inside the summation, you find that the cost term associated with the training example, that is the cost associated with the training example x(i), y(i). That's going to be given by this expression. So, the cost to live off examplie i is written as follows. And what this cost function does is it plays a role similar to the squared arrow. So, rather than looking at this complicated expression, if you want you can think of cost of i being approximately the square difference between what the neural network outputs, versus what is the actual value. Just as in logistic repression, we actually prefer to use the slightly more complicated cost function using the log. But for the purpose of intuition, feel free to think of the cost function as being the sort of the squared error cost function. And so this cost(i) measures how well is the network doing on correctly predicting example i. How close is the output to the actual observed label y(i)? Now let's look at what backpropagation is doing. One useful intuition is that backpropagation is computing these delta superscript l subscript j terms. And we can think of these as the quote error of the activation value that we got for unit j in the layer, in the lth layer.
* 7:07
* More formally, for, and this is maybe only for those of you who are familiar with calculus. More formally, what the delta terms actually are is this, they're the partial derivative with respect to z,l,j, that is this weighted sum of inputs that were confusing these z terms. Partial derivatives with respect to these things of the cost function.
* 7:27
* So concretely, the cost function is a function of the label y and of the value, this h of x output value neural network. And if we could go inside the neural network and just change those z l j values a little bit, then that will affect these values that the neural network is outputting. And that will end up changing the cost function. And again really, this is only for those of you who are expert in Calculus. If you're comfortable with partial derivatives, what these delta terms are is they turn out to be the partial derivative of the cost function, with respect to these intermediate terms that were confusing.
* 8:06
* And so they're a measure of how much would we like to change the neural network's weights, in order to affect these intermediate values of the computation. So as to affect the final output of the neural network h(x) and therefore affect the overall cost. In case this lost part of this partial derivative intuition, in case that doesn't make sense. Don't worry about the rest of this, we can do without really talking about partial derivatives. But let's look in more detail about what backpropagation is doing. For the output layer, the first set's this delta term, delta (4) 1, as y (i) if we're doing forward propagation and back propagation on this training example i. That says y(i) minus a(4)1. So this is really the error, right? It's the difference between the actual value of y minus what was the value predicted, and so we're gonna compute delta(4)1 like so. Next we're gonna do, propagate these values backwards. I'll explain this in a second, and end up computing the delta terms for the previous layer. We're gonna end up with delta(3)1. Delta(3)2. And then we're gonna propagate this further backward, and end up computing delta(2)1 and delta(2)2. Now the backpropagation calculation is a lot like running the forward propagation algorithm, but doing it backwards. So here's what I mean. Let's look at how we end up with this value of delta(2)2. So we have delta(2)2. And similar to forward propagation, let me label a couple of the weights. So this weight, which I'm going to draw in cyan. Let's say that weight is theta(2)1 2, and this one down here when we highlight this in red. That is going to be let's say theta(2) of 2 2. So if we look at how delta(2)2, is computed, how it's computed with this note. It turns out that what we're going to do, is gonna take this value and multiply it by this weight, and add it to this value multiplied by that weight. So it's really a weighted sum of these delta values, weighted by the corresponding edge strength. So completely, let me fill this in, this delta(2)2 is going to be equal to, Theta(2)1 2 is that magenta lay times delta(3)1. Plus, and the thing I had in red, that's theta (2)2 times delta (3)2. So it's really literally this red wave times this value, plus this magenta weight times this value. And that's how we wind up with that value of delta. And just as another example, let's look at this value. How do we get that value? Well it's a similar process. If this weight, which I'm gonna highlight in green, if this weight is equal to, say, delta (3) 1 2. Then we have that delta (3) 2 is going to be equal to that green weight, theta (3) 12 times delta (4) 1. And by the way, so far I've been writing the delta values only for the hidden units, but excluding the bias units. Depending on how you define the backpropagation algorithm, or depending on how you implement it, you know, you may end up implementing something that computes delta values for these bias units as well. The bias units always output the value of plus one, and they are just what they are, and there's no way for us to change the value. And so, depending on your implementation of back prop, the way I usually implement it. I do end up computing these delta values, but we just discard them, we don't use them. Because they don't end up being part of the calculation needed to compute a derivative. So hopefully that gives you a little better intuition about what back propegation is doing. In case of all of this still seems sort of magical, sort of black box, in a later video, in the putting it together video, I'll try to get a little bit more intuition about what backpropagation is doing. But unfortunately this is a difficult algorithm to try to visualize and understand what it is really doing. But fortunately I've been, I guess many people have been using very successfully for many years. And if you implement the algorithm you can have a very effective learning algorithm. Even though the inner workings of exactly how it works can be harder to visualize.