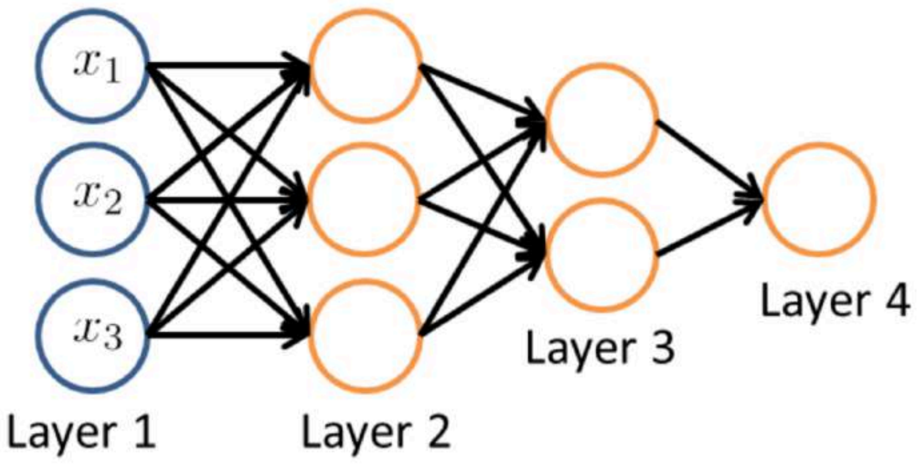
MACHINE LEARNING-5

Question

Consider the network:



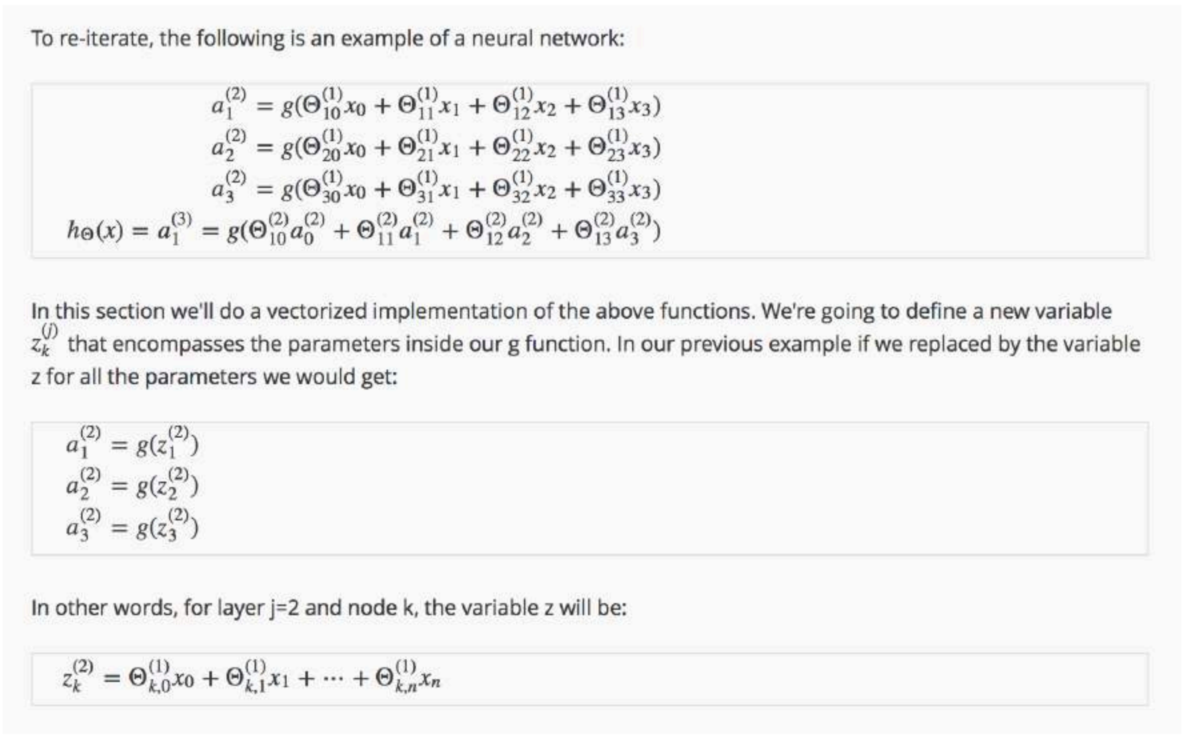
Let $a^{(1)} = x \in \mathbb{R}^{n+1}$ denote the input (with $a_0^{(1)} = 1$).

How would you compute $a^{(2)}$?

- $a^{(2)} = \Theta^{(1)}a^{(1)}$
- $z^{(2)} = \Theta^{(2)}a^{(1)};\ a^{(2)} = g(z^{(2)})$
- [✓ $z^{(2)} = \Theta^{(1)}a^{(1)};\ a^{(2)} = g(z^{(2)})$]
- $z^{(2)} = \Theta^{(2)}g(a^{(1)});\ a^{(2)} = g(z^{(2)})$

So, hopefully from this video you've gotten a sense of how the feed forward propagation step in a neural network works where you start from the activations of the input layer and forward propagate that to the first hidden layer, then the second hidden layer, and then finally the output layer. And you also saw how we can vectorize that computation.In the next, I realized that some of the intuitions in this video of how, you know, other certain layers are computing complex features of the early layers. I realized some of that intuition may be still slightly abstract and kind of a high level. And so what I would like to do in the two videos is work through a detailed example of how a neural network can be used to compute nonlinear functions of the input and hope that will give you a good sense of the sorts of complex nonlinear hypotheses we can get out of Neural Networks.

Model Representation 2 (Transcript)



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

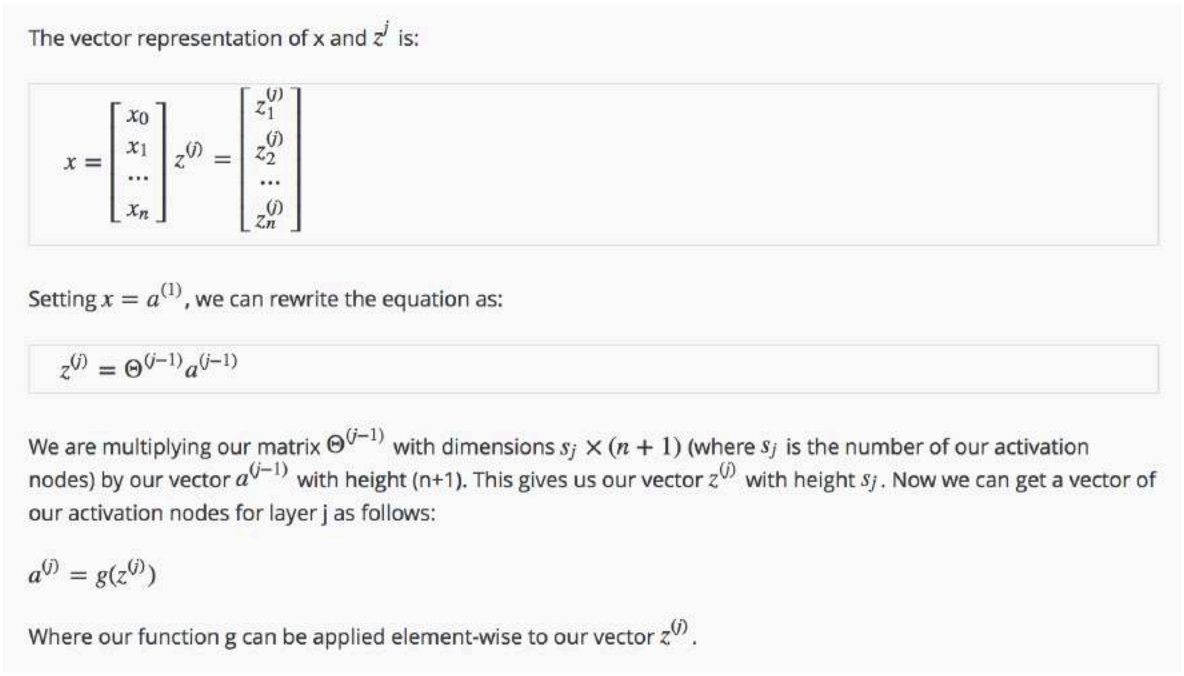
$$\begin{aligned}
a_1^{(2)} &= g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3) \\
a_2^{(2)} &= g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3) \\
a_3^{(2)} &= g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2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

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:

$$\begin{aligned}
a_1^{(2)} &= g(z_1^{(2)}) \\
a_2^{(2)} &= g(z_2^{(2)}) \\
a_3^{(2)} &= g(z_3^{(2)})
\end{aligned}$$

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:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_n \end{bmatrix} z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ z_2^{(j)} \\ \cdots \\ z_n^{(j)} \end{bmatrix}$$

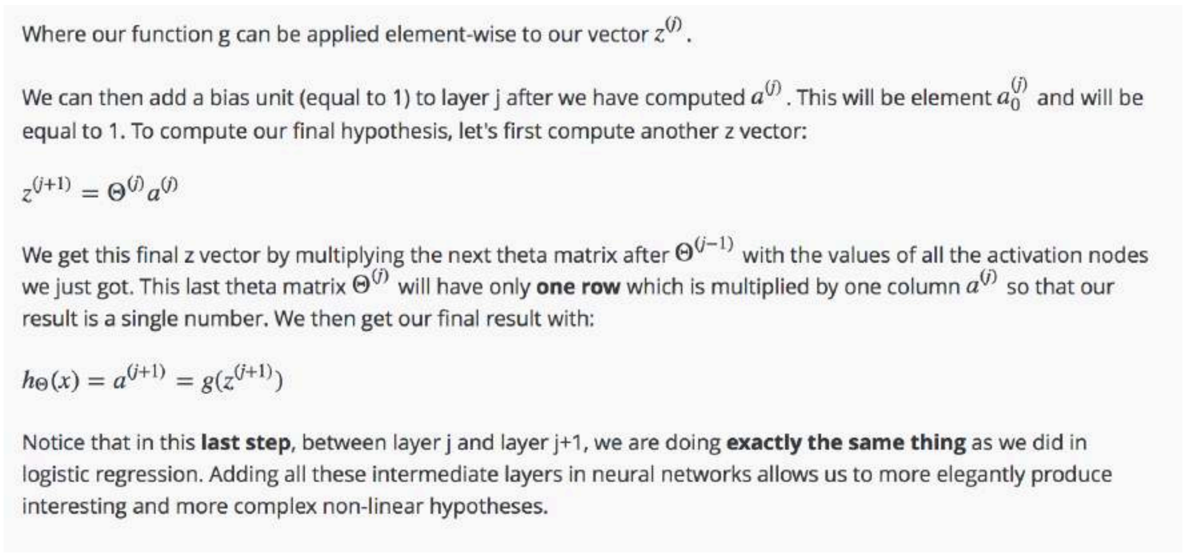
Setting $x = a^{(1)}$, we can rewrite the equation as:

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

We are multiplying our matrix $\Theta^{(j-1)}$ with dimensions $s_j \times (n+1)$ (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 $z^{(j)}$.



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:

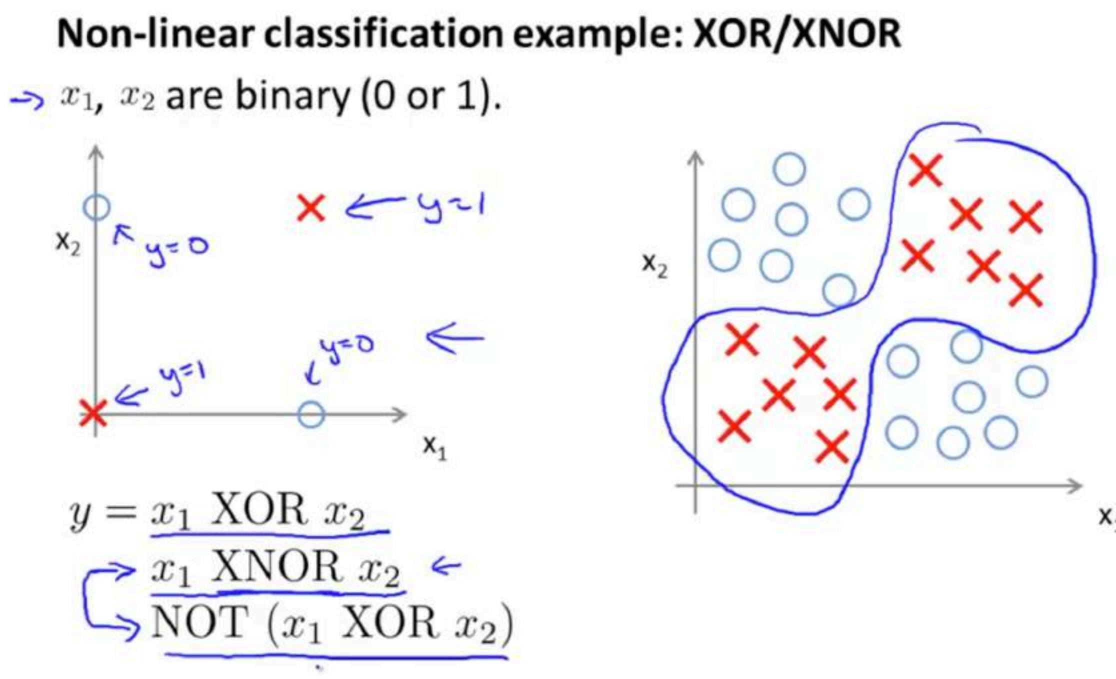
$$h_\Theta(x) = a^{(j+1)} = g(z^{(j+1)})$$

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.

Applications

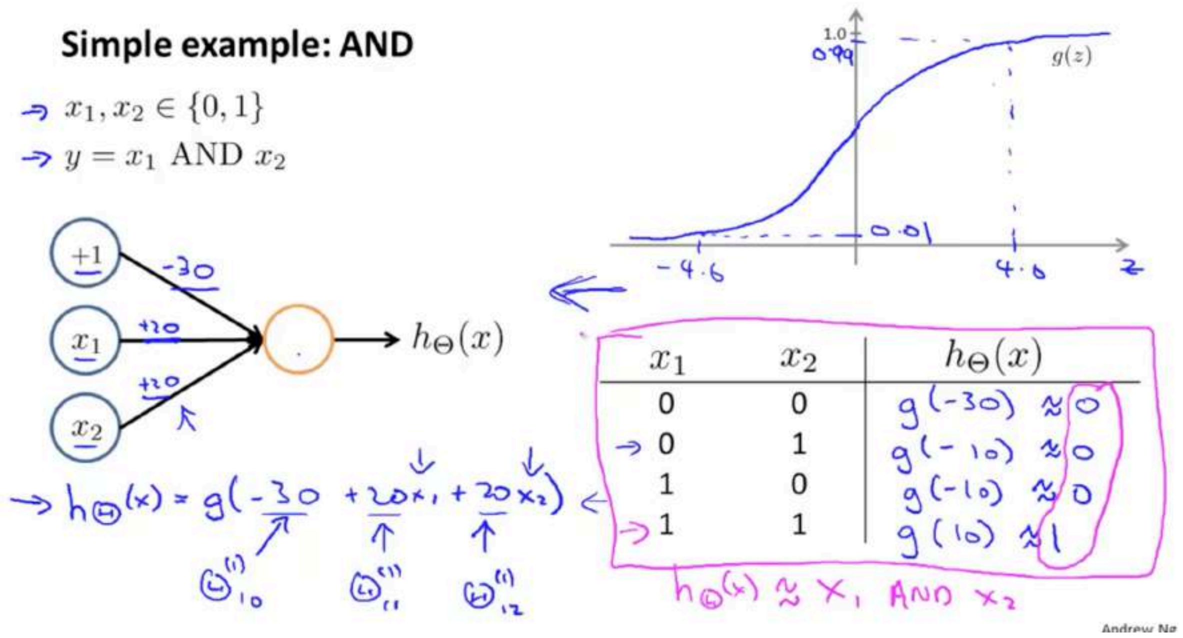
Examples and Intuitions 1 (Video)

In this and the next video I want to work through a detailed example showing how a neural network can compute a complex non linear function of the input. And hopefully this will give you a good sense of why neural networks can be used to learn complex non linear hypotheses. Consider the following problem where we have features X1 and X2 that are binary values. So, either 0 or 1. So, X1 and X2 can each take on only one of two possible values. In this example, I've drawn only two positive examples and two negative examples. That you can think of this as a simplified version of a more complex learning problem where we may have a bunch of positive examples in the upper right and lower left and a bunch of negative examples denoted by the circles. And what we'd like to do is learn a non-linear division of boundary that may need to separate the positive and negative examples. So, how can a neural network do this and rather than using the example and the variable to use this maybe easier to examine example on the left. Concretely what this is, is really computing the type of label y equals x 1 x or x 2. Or actually this is actually the x 1 x nor x 2 function where x nor is the alternative notation for not x 1 or x 2. So, x 1 x or x 2 that's true only if exactly 1 of x 1 or x 2 is equal to 1. It turns out that these specific examples in the works out a little bit better if we use the XNOR example instead. These two are the same of course. This means not x1 or x2 and so, we're going to have positive examples of either both are true or both are false and what have as y equals 1, y equals 1. And we're going to have y equals 0 if only one of them is true and we're going to figure out if we can get a neural network to fit to this sort of training set.



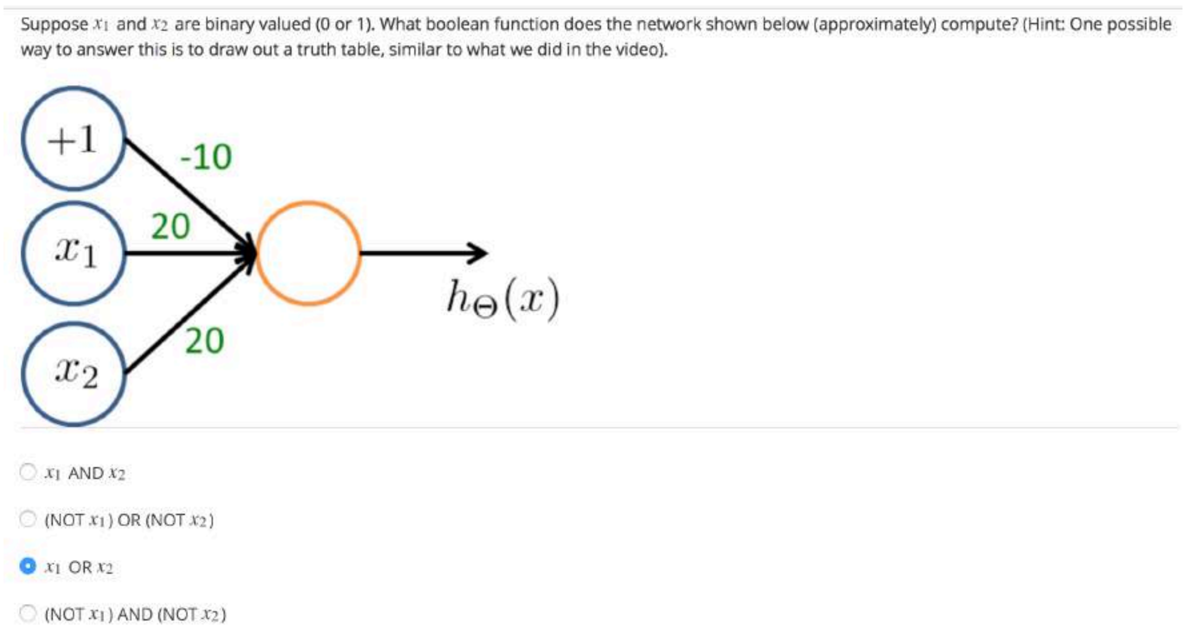
Non-linear classification example: XOR/XNOR
x_1, x_2 are binary (0 or 1).
x_2
x_1
y=0
y=1
y=1
y=0
$y = x_1 \text{ XOR } x_2$
$x_1 \text{ XNOR } x_2$
$\text{NOT } (x_1 \text{ XOR } x_2)$
x_2
x_1

In order to build up to a network that fits the XNOR example we're going to start with a slightly simpler one and show a network that fits the AND function. Concretely, let's say we have input x1 and x2 that are again binaries so, it's either 0 or 1 and let's say our target labels y = x1 AND x2. This is a logical AND. So, can we get a one-unit network to compute this logical AND function? In order to do so, I'm going to actually draw in the bias unit as well the plus one unit. Now let me just assign some values to the weights or parameters of this network. I'm gonna write down the parameters on this diagram here, -30 here. +20 and + 20. And what this mean is just that I'm assigning a value of -30 to the value associated with X0 this +1 going into this unit and a parameter value of +20 that multiplies to X1 a value of +20 for the parameter that multiplies into x 2. So, concretely it's the same that the hypothesis h(x)=g(-30+20 X1 plus 20 X2. So, sometimes it's just convenient to draw these weights. Draw these parameters up here in the diagram within and of course this- 30. This is actually theta 1 of 1 0. This is theta 1 of 1 1 and that's theta 1 of 1 2 but it's just easier to think about it as associating these parameters with the edges of the network. Let's look at what this little single neuron network will compute. Just to remind you the sigmoid activation function g(z) looks like this. It starts from 0 rises smoothly crosses 0.5 and then it asymptotic as 1 and to give you some landmarks, if the horizontal axis value z is equal to 4.6 then the sigmoid function is equal to 0.99. This is very close to 1 and kind of symmetrically, if it's -4.6 then the sigmoid function there is 0.01 which is very close to 0. Let's look at the four possible input values for x1 and x2 and look at what the hypotheses will output in that case. If x1 and x2 are both equal to 0. If you look at this, if x1 x2 are both equal to 0 then the hypothesis of g of -30. So, this is a very far to the left of this diagram so it will be very close to 0. If x 1 equals 0 and x equals 1, then this formula here evaluates the g that is the sigma function applied to -10, and again that's you know to the far left of this plot and so, that's again very close to 0. This is also g of minus 10 that is f x 1 is equal to 1 and x 2 0, this minus 30 plus 20 which is minus 10 and finally if x 1 equals 1 x 2 equals 1 then you have g of minus 30 plus 20 plus 20. So, that's g of positive 10 which is there for very close to 1.And if you look in this column this is exactly the logical and function. So, this is computing h of x is approximately x 1 and x 2. In other words it outputs one If and only if x2, x1 and x2, are both equal to 1. So, by writing out our little truth table like this we manage to figure what's the logical function that our neural network computes.



x_1	x_2	$h_\Theta(x)$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

Question

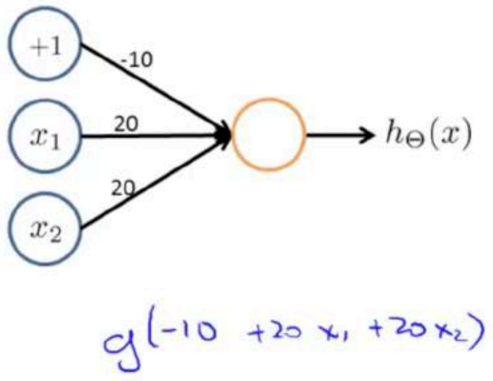


Suppose x_1 and x_2 are binary valued (0 or 1). What boolean function does the network shown below (approximately) compute? (Hint: One possible way to answer this is to draw out a truth table, similar to what we did in the video).

- x_1 AND x_2
- (NOT x_1) OR (NOT x_2)
- [✓ x_1 OR x_2]
- (NOT x_1) AND (NOT x_2)

This network showed here computes the OR function. Just to show you how I worked out that. If you are write out the hypothesis that this confusing g of -10 + 20 x 1 + 20 x 2 and so you fill in these values. You find that's g of minus 10 which is approximately 0. g of 10 which is approximately 1 and so on and these are approximately 1 and approximately 1 and these numbers are essentially the logical OR function.

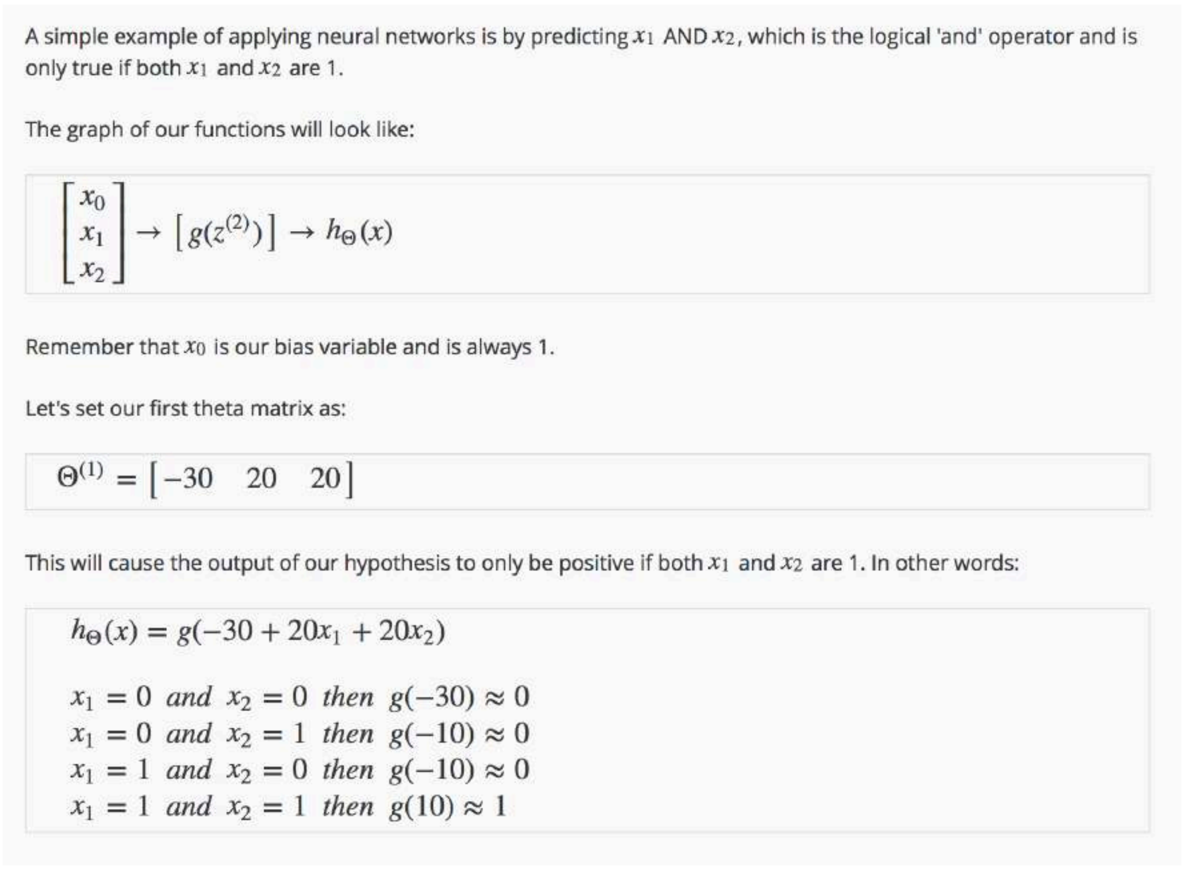
Example: OR function



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

So, hopefully with this you now understand how single neurons in a neural network can be used to compute logical functions like AND and OR and so on. In the next video we'll continue building on these examples and work through a more complex example. We'll get to show you how a neural network now with multiple layers of units can be used to compute more complex functions like the XOR function or the XNOR function.

Examples and Intuitions 1 (Transcript)



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:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \left[g(z^{(2)}) \right] \rightarrow h_\Theta(x)$$

Remember that x_0 is our bias variable and is always 1.

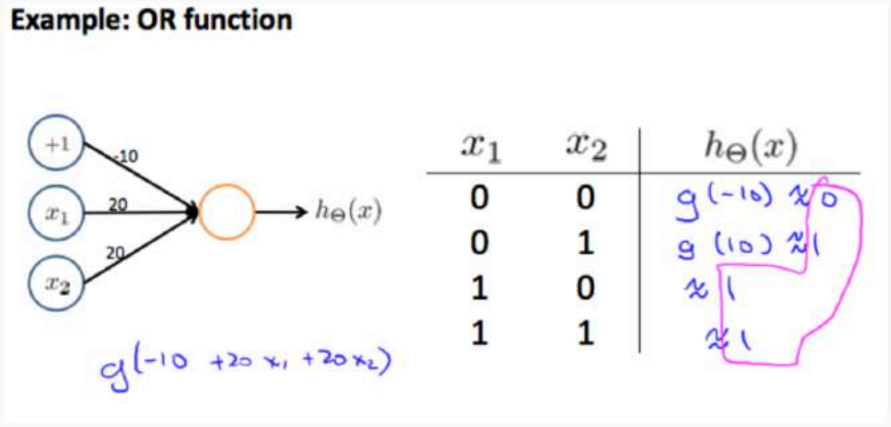
Let's set our first theta matrix as:

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

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

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

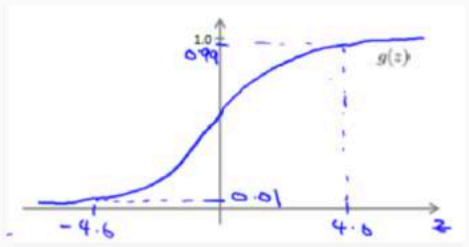
So we have constructed one of the fundamental operations in computers by using a small neural network rather than using an actual AND gate. Neural networks can also be used to simulate all the other logical gates. The following is an example of the logical operator 'OR', meaning either x_1 is true or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



Example: OR function

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

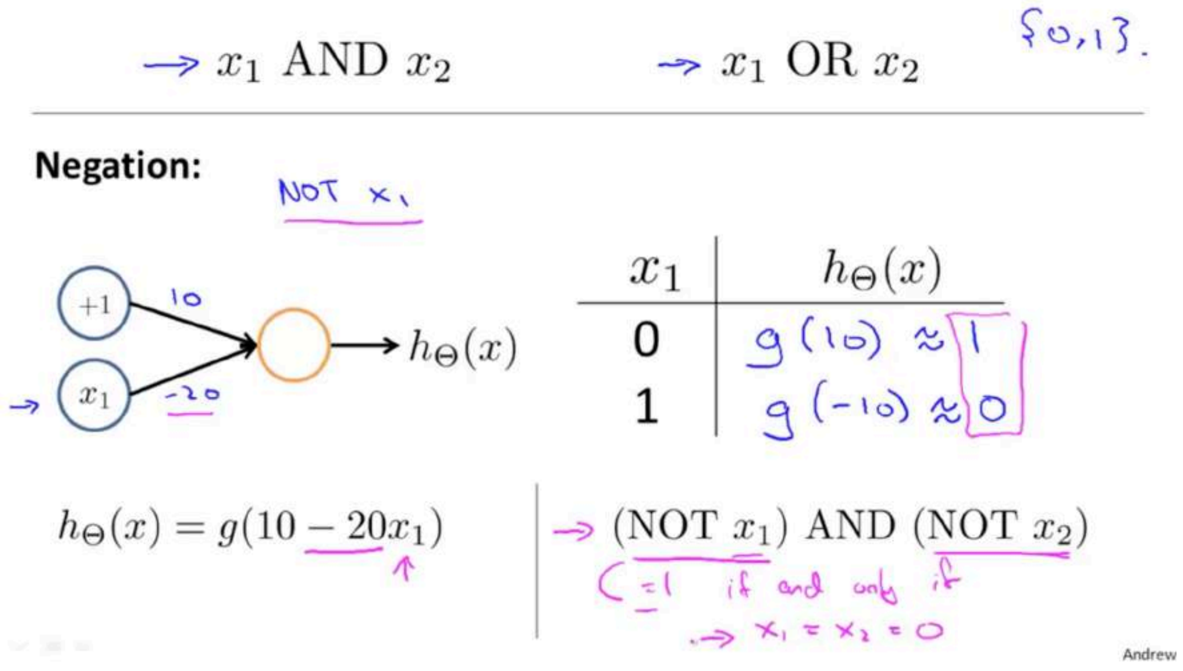
Where g(z) is the following:



Examples and Intuitions 2 (Video)

In this video I'd like to keep working through our example to show how a Neural Network can compute complex non linear hypothesis. In the last video we saw how a Neural Network can be used to compute the functions x1 AND x2, and the function x1 OR x2 when x1 and x2 are binary, that is when they take on values 0,1. We can also have a network to compute negation, that is to compute the function not x1. Let me just write down the ways associated with this network. We have only one input feature x1 in this case and the bias unit +1. And if I associate this with the weights plus 10 and -20, then my hypothesis is computing this h(x) equals sigmoid (10- 20 x1). So when x1 is equal to 0, my hypothesis would be computing g(10- 20 x 0) is just 10. And so that's approximately 1, and when x is equal to 1, this will be g(-10) which is approximately equal to 0. And if you look at what these values are, that's essentially the not x1 function. Cells include negations, the general idea is to put that large negative weight in front of the variable you want to negate.

Minus 20 multiplied by x1 and that's the general idea of how you end up negating x1. And so in an example that I hope that you can figure out yourself. If you want to compute a function like this NOT x1 AND NOT x2, part of that will probably be putting large negative weights in front of x1 and x2, but it should be feasible. So you get a neural network with just one output unit to compute this as well. All right, so this logical function, NOT x1 AND NOT x2, is going to be equal to 1 if and only if x1 equals x2 equals 0. All right since this is a logical function, this says NOT x1 means x1 must be 0 and NOT x2, that means x2 must be equal to 0 as well. So this logical function is equal to 1 if and only if both x1 and x2 are equal to 0 and hopefully you should be able to figure out how to make a small neural network to compute this logical function as well.

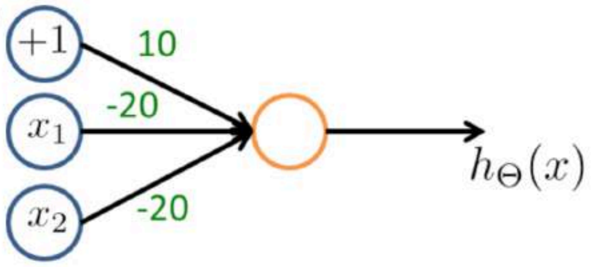


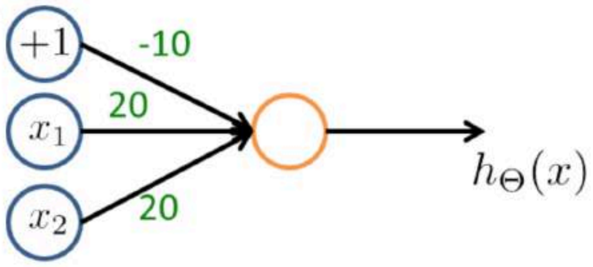
$\rightarrow x_1$ AND x_2
$\rightarrow x_1$ OR x_2
$\{0,1\}$.
Negation:
NOT x_1
+1
x_1
10
-20
$h_\Theta(x)$
x_1 | $h_\Theta(x)$
0 | $g(10) \approx 1$
1 | $g(-10) \approx 0$
$h_\Theta(x) = g(10 - 20x_1)$
$\rightarrow$ (NOT x_1) AND (NOT x_2)
$= 1$ if and only if
$\rightarrow x_1 = x_2 = 0$
Andrew

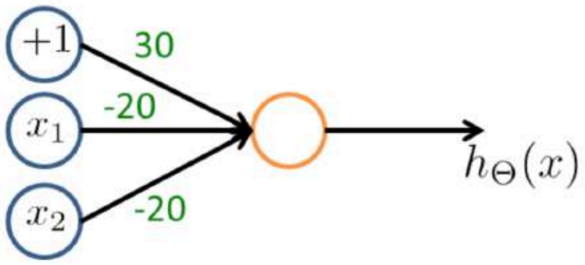
Question

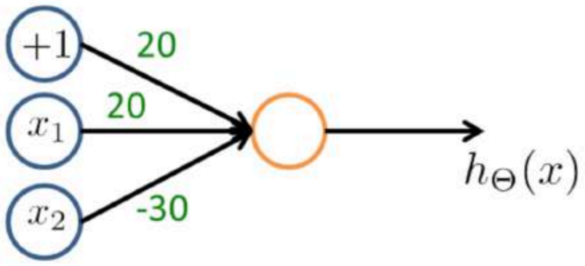
Answer: A

Suppose that x_1 and x_2 are binary valued (0 or 1). Which of the following networks (approximately) computes the boolean function (NOT x_1) AND (NOT x_2)?

○ 

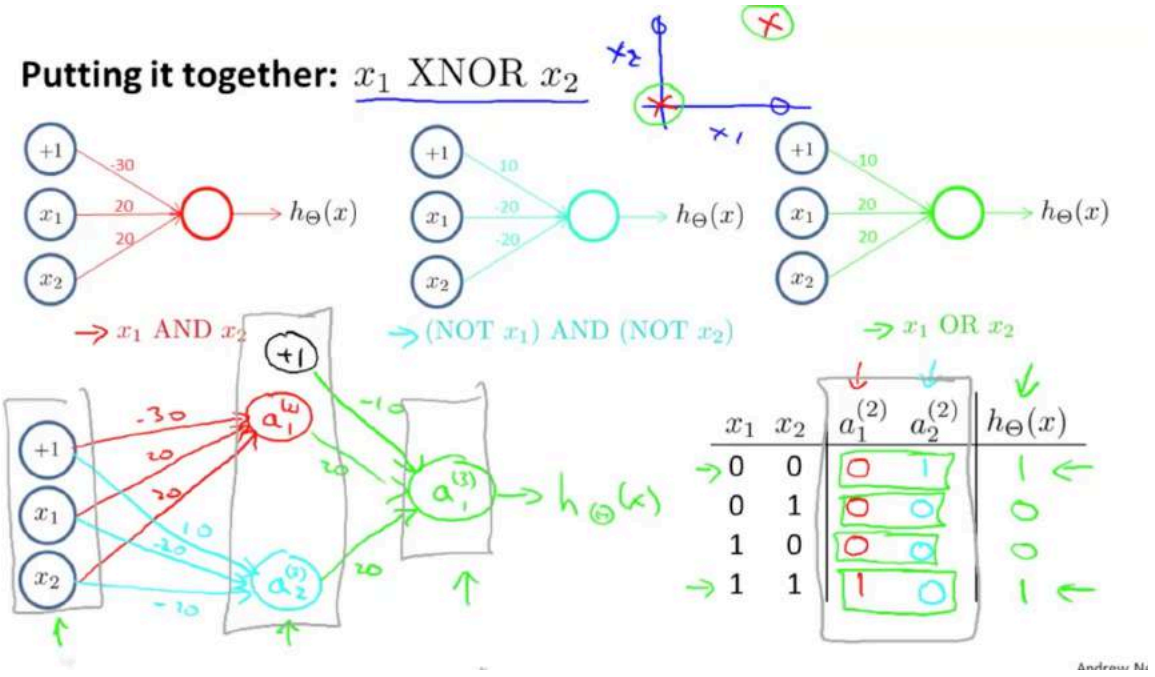
○ 

○ 

○ 

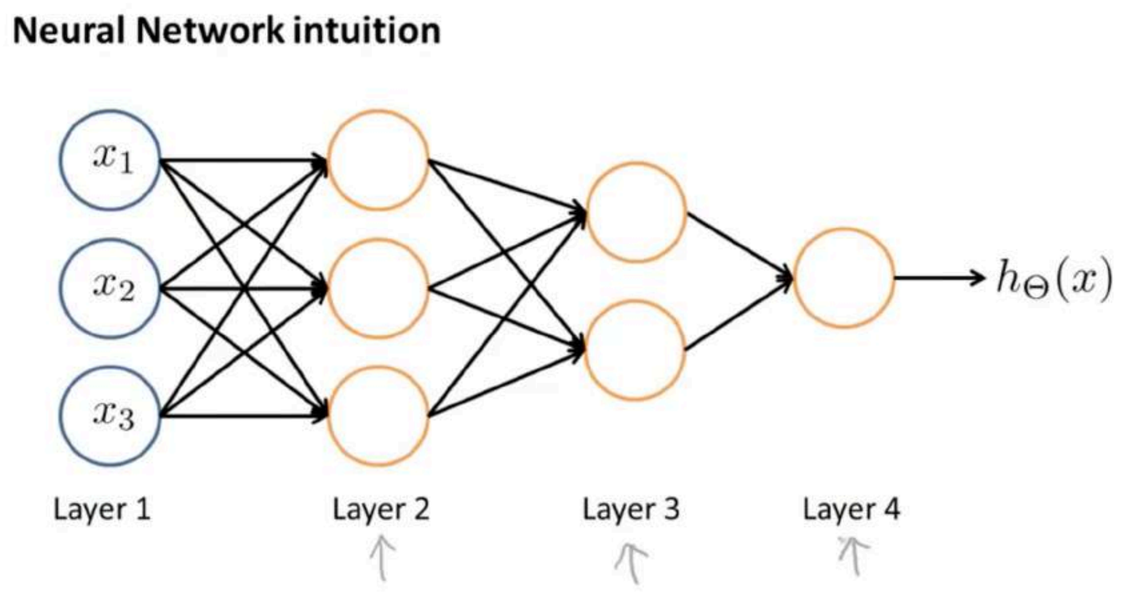
Now, taking the three pieces that we have put together as the network for computing x1 AND x2, and the network computing for computing NOT x1 AND NOT x2. And one last network computing for computing x1 OR x2, we should be able to put these three pieces together to compute this x1 XNOR x2 function. And just to remind you if this is x1, x2, this function that we want to compute would have negative examples here and here, and we'd have positive examples there and there. And so clearly this will need a non linear decision boundary in order to separate the positive and negative examples. Let's draw the network. I'm going to take my input +1, x1, x2 and create my first hidden unit here. I'm gonna call this a 21 cuz that's my first hidden unit. And I'm gonna copy the weight over from the red network, the x1 and x2. As well so then -30, 20, 20. Next let me create a second hidden unit which I'm going to call a 2

2. That is the second hidden unit of layer two. I'm going to copy over the cyan that's work in the middle, so I'm gonna have the weights 10 -20 -20. And so, let's pull some of the truth table values. For the red network, we know that was computing the x1 and x2, and so this will be approximately 0 0 0 1, depending on the values of x1 and x2, and for a 2 2, the cyan network. What do we know? The function NOT x1 AND NOT x2, that outputs 1 0 0 0, for the 4 values of x1 and x2. Finally, I'm going to create my output node, my output unit that is a 3 1. This is one more output h(x) and I'm going to copy over the old network for that. And I'm going to need a +1 bias unit here, so you draw that in, And I'm going to copy over the weights from the green networks. So that's -10, 20, 20 and we know earlier that this computes the OR function. So let's fill in the truth table entries. So the first entry is 0 OR 1 which can be 1 that makes 0 OR 0 which is 0, 0 OR 0 which is 0, 1 OR 0 and that falls to 1. And thus h(x) is equal to 1 when either both x1 and x2 are zero or when x1 and x2 are both 1 and concretely h(x) outputs 1 exactly at these two locations and then outputs 0 otherwise. And thus will this neural network, which has a input layer, one hidden layer, and one output layer, we end up with a nonlinear decision boundary that computes this XNOR function. And the more general intuition is that in the input layer, we just have our four inputs. Then we have a hidden layer, which computed some slightly more complex functions of the inputs that its shown here this is slightly more complex functions. And then by adding yet another layer we end up with an even more complex non linear function.



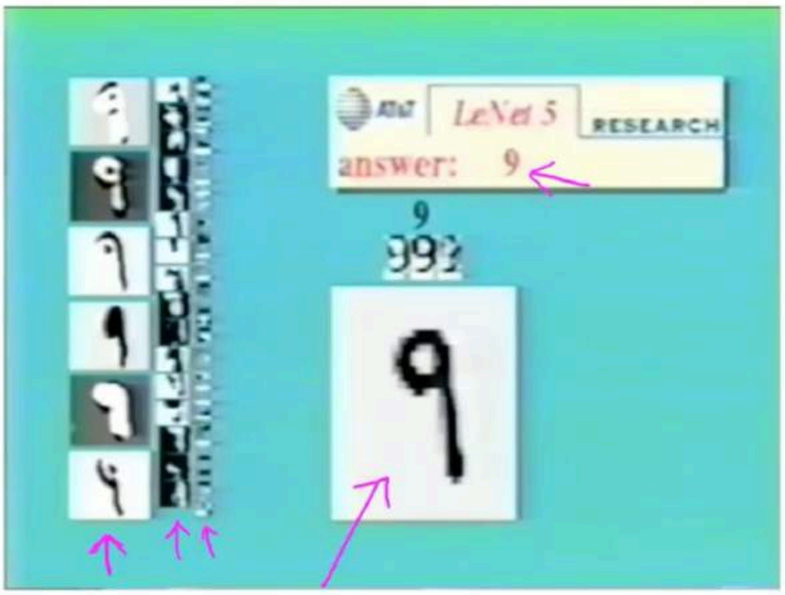
x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_\Theta(x)$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

And this is a sort of intuition about why neural networks can compute pretty complicated functions. That when you have multiple layers you have relatively simple function of the inputs of the second layer. But the third layer I can build on that to complete even more complex functions, and then the layer after that can compute even more complex functions.



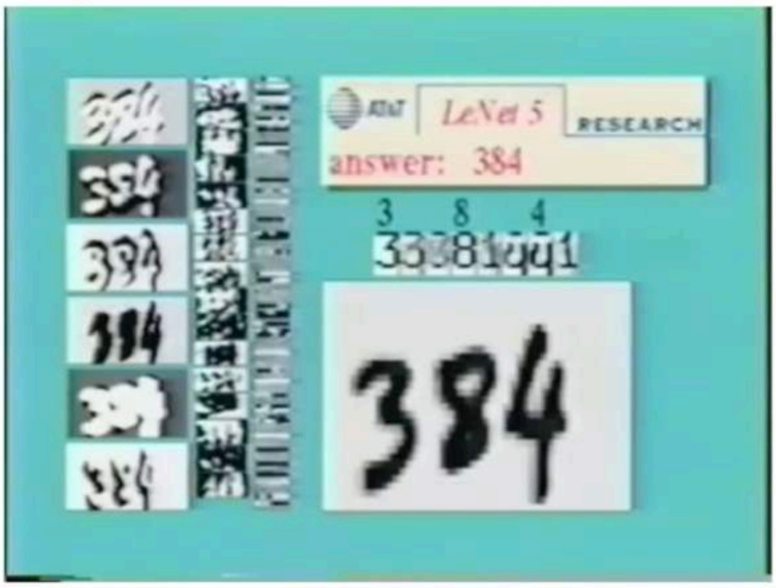
To wrap up this video, I want to show you a fun example of an application of a the Neural Network that captures this intuition of the deeper layers computing more complex features. I want to show you a video of that customer a good friend of mine Yann LeCunj. Yann is a professor at New York University, NYU and he was one of the early pioneers of Neural Network research and is sort of a legend in the field now and his ideas are used in all sorts of products and applications throughout the world now. So I wanna show you a video from some of his early work in which he was using a neural network to recognize handwriting, to do handwritten digit recognition. You might remember early in this class, at the start of this class I said that one of the earliest successes of neural networks was trying to use it to read zip codes to help USPS Laws and read postal codes. So this is one of the attempts, this is one of the algorithms used to try to address that problem. In the video that I'll show you this area here is the input area that shows a canvasing character shown to the network. This column here shows a visualization of the features computed by sort of the first hidden layer of the network. So that the first hidden layer of the network and so the first hidden layer, this visualization shows different features. Different edges and lines and so on detected. This is a visualization of the next hidden layer. It's kinda harder to see, harder to understand the deeper, hidden layers, and that's a visualization of why the next hidden layer is confusing. You probably have a hard time seeing what's going on much beyond the first hidden layer, but then finally, all of these learned features get fed to the upper layer. And shown over here is the final answer, it's the final predictive value for what handwritten digit the neural network thinks it is being shown. So let's take a look at the video.

Handwritten digit classification

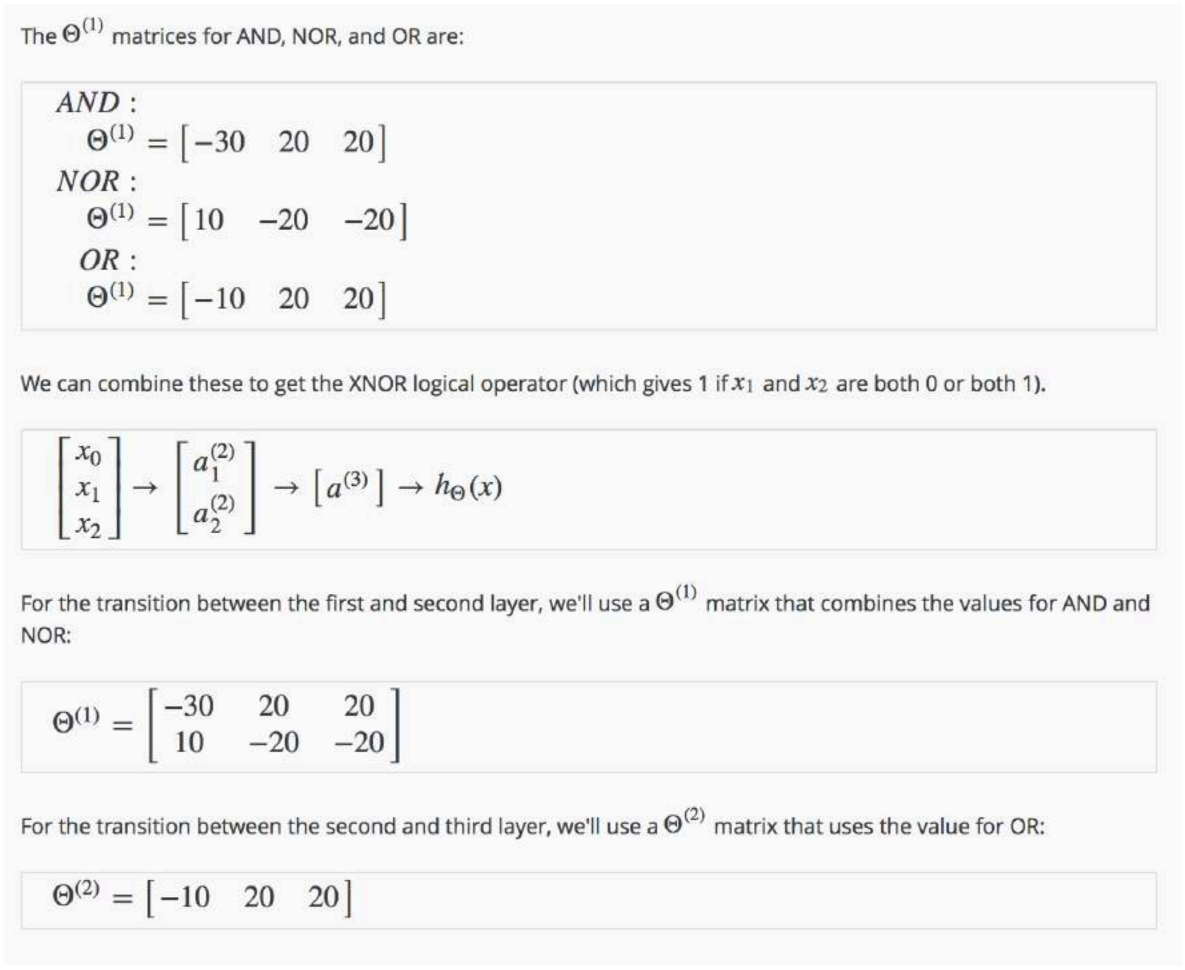


So I hope you enjoyed the video and that this hopefully gave you some intuition about the source of pretty complicated functions neural networks can learn. In which it takes its input this image, just takes this input, the raw pixels and the first hidden layer computes some set of features. The next hidden layer computes even more complex features and even more complex features. And these features can then be used by essentially the final layer of the logistic classifiers to make accurate predictions without the numbers that the network sees.

Handwritten digit classification



Examples and Intuitions 2 (Transcript)



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

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

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:

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

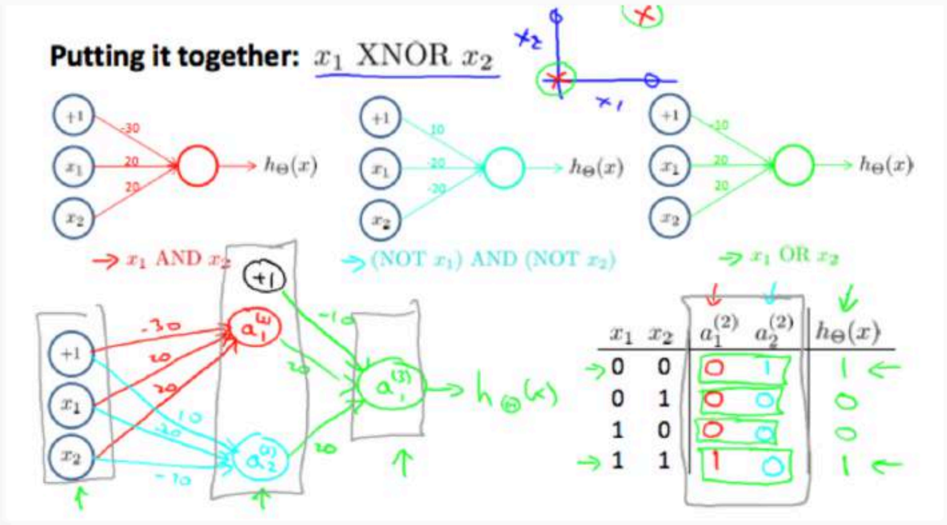
For the transition between the second and third layer, we'll use a $\Theta^{(2)}$ matrix that uses the value for OR:

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

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

$$a^{(2)} = g(\Theta^{(1)} \cdot x)$$
$$a^{(3)} = g(\Theta^{(2)} \cdot a^{(2)})$$
$$h_\Theta(x) = a^{(3)}$$

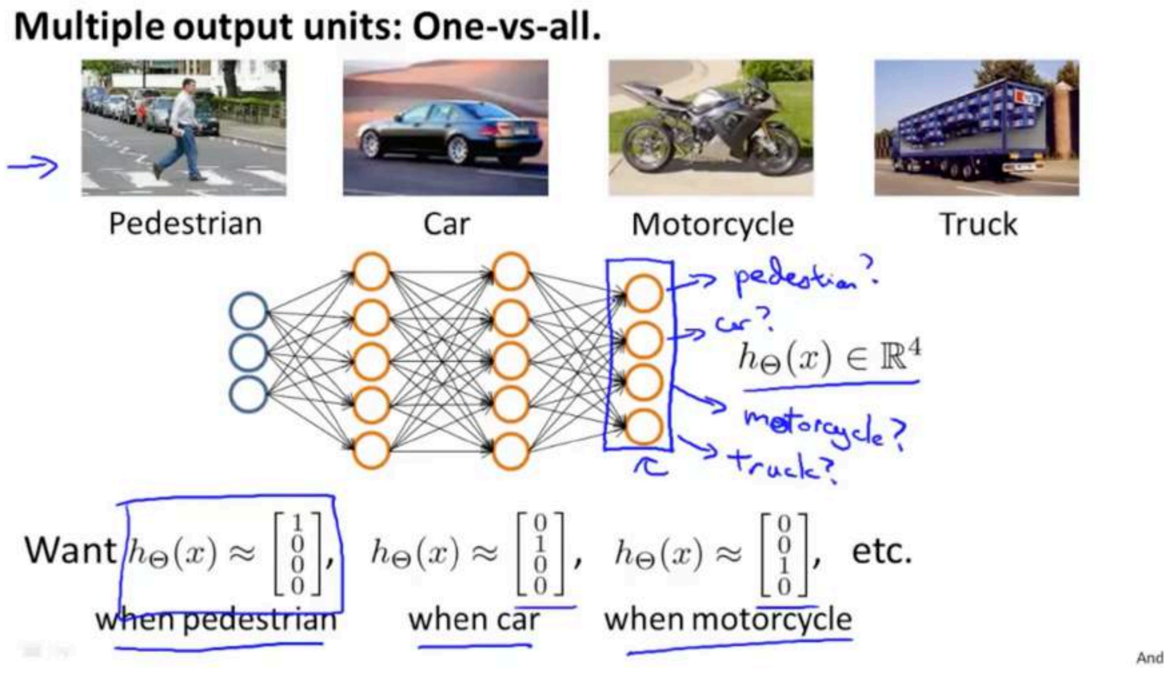
And there we have the XNOR operator using a hidden layer with two nodes! The following summarizes the above algorithm:



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
$h_\Theta(x)$
-30
20
20
10
-20
-20
-10
20
20
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

Multiclass Classification (Video)

In this video, I want to tell you about how to use neural networks to do multiclass classification where we may have more than one category that we're trying to distinguish amongst. In the last part of the last video, where we had the handwritten digit recognition problem, that was actually a multiclass classification problem because there were ten possible categories for recognizing the digits from 0 through 9 and so, if you want us to fill you in on the details of how to do that. The way we do multiclass classification in a neural network is essentially an extension of the one versus all method. So, let's say that we have a computer vision example, where instead of just trying to recognize cars as in the original example that I started off with, but let's say that we're trying to recognize, you know, four categories of objects and given an image we want to decide if it is a pedestrian, a car, a motorcycle or a truck. If that's the case, what we would do is we would build a neural network with four output units so that our neural network now outputs a vector of four numbers. So, the output now is actually needing to be a vector of four numbers and what we're going to try to do is get the first output unit to classify: is the image a pedestrian, yes or no. The second unit to classify: is the image a car, yes or no. This unit to classify: is the image a motorcycle, yes or no, and this would classify: is the image a truck, yes or no. And thus, when the image is of a pedestrian, we would ideally want the network to output 1, 0, 0, 0, when it is a car we want it to output 0, 1, 0, 0, when this is a motorcycle, we get it to or rather, we want it to output 0, 0, 1, 0 and so on.So this is just like the "one versus all" method that we talked about when we were describing logistic regression, and here we have essentially four logistic regression classifiers, each of which is trying to recognize one of the four classes that we want to distinguish amongst.



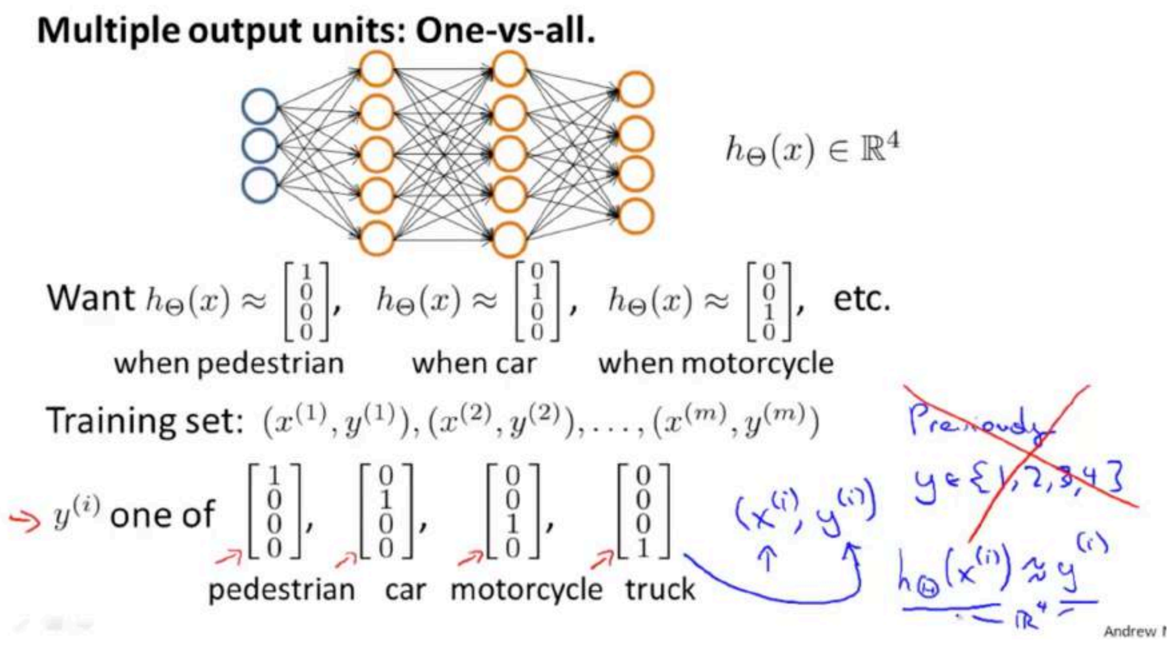
Multiple output units: One-vs-all.

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

And

So, rearranging the slide of it, here's our neural network with four output units and those are what we want h of x to be when we have the different images, and the way we're going to represent the training set in these settings is as follows. So, when we have a training set with different images of pedestrians, cars, motorcycles and trucks, what we're going to do in this example is that whereas previously we had written out the labels as y being an integer from 1, 2, 3 or 4. Instead of representing y this way, we're going to instead represent y as follows: namely Yi will be either 1, 0, 0, 0 or 0, 1, 0, 0 or 0, 0, 1, 0 or 0, 0, 0, 1 depending on what the corresponding image Xi is. And so one training example will be one pair Xi colon Yi where Xi is an image with, you know one of the four objects and Yi will be one of these vectors. And hopefully, we can find a way to get our Neural Networks to output some value. So, the h of x is approximately y and both h of x and Yi, both of these are going to be in our example, four dimensional vectors when we have four classes.



Multiple output units: One-vs-all.

$h_\Theta(x) \in \mathbb{R}^4$

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

Training set: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})$

~~Previously $y \in \{1, 2, 3, 4\}$~~

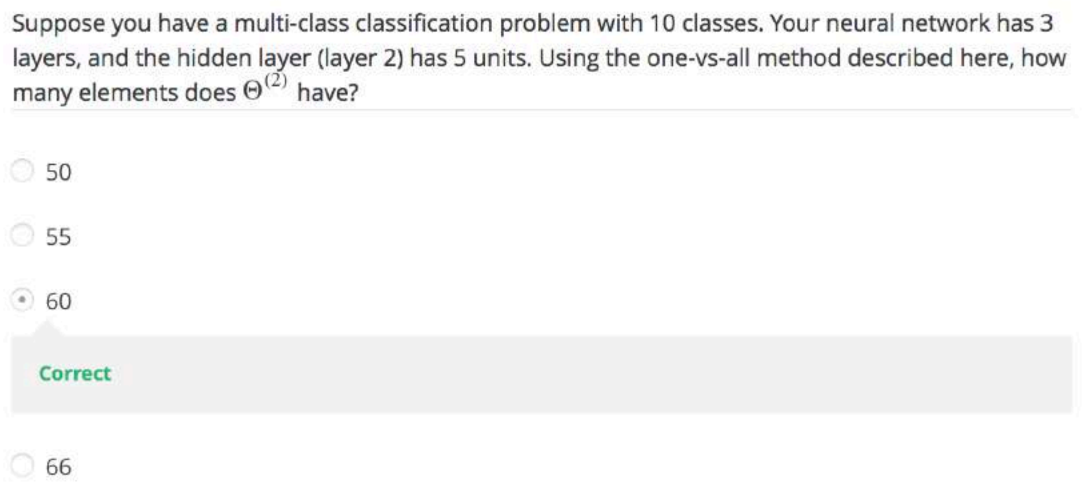
$\rightarrow y^{(i)}$ one of $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$ $(x^{(i)}, y^{(i)})$

pedestrian car motorcycle truck

$\underbrace{h_\Theta(x^{(i)}) \approx y^{(i)}}_{\mathbb{R}^4}$

Andrew N

Question



Suppose you have a multi-class classification problem with 10 classes. Your neural network has 3 layers, and the hidden layer (layer 2) has 5 units. Using the one-vs-all method described here, how many elements does $\Theta^{(2)}$ have?

50

55

[✓ 60]

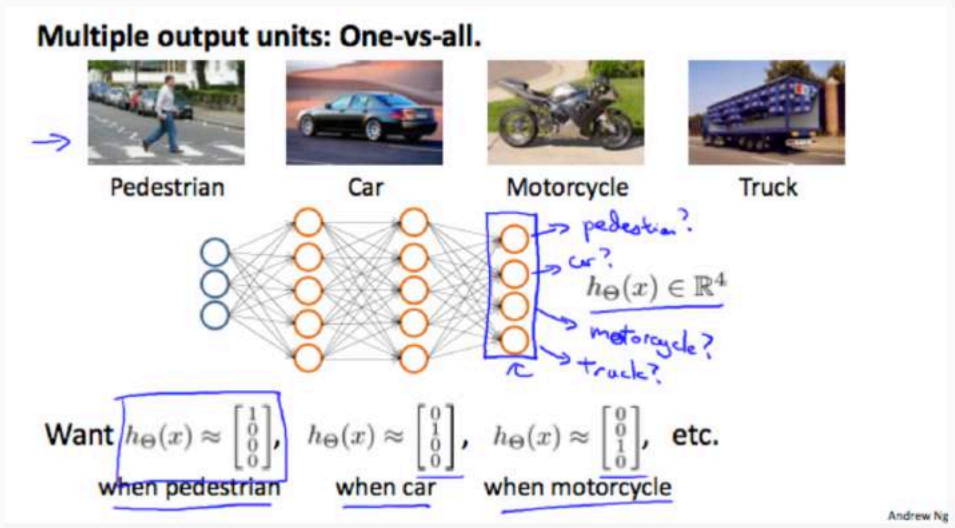
Correct

66

So, that's how you get neural network to do multiclass classification. This wraps up our discussion on how to represent Neural Networks that is on our hypotheses representation. In the next set of videos, let's start to talk about how take a training set and how to automatically learn the parameters of the neural network.

Multiclass Classification (Transcript)

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
Andrew Ng

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:

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

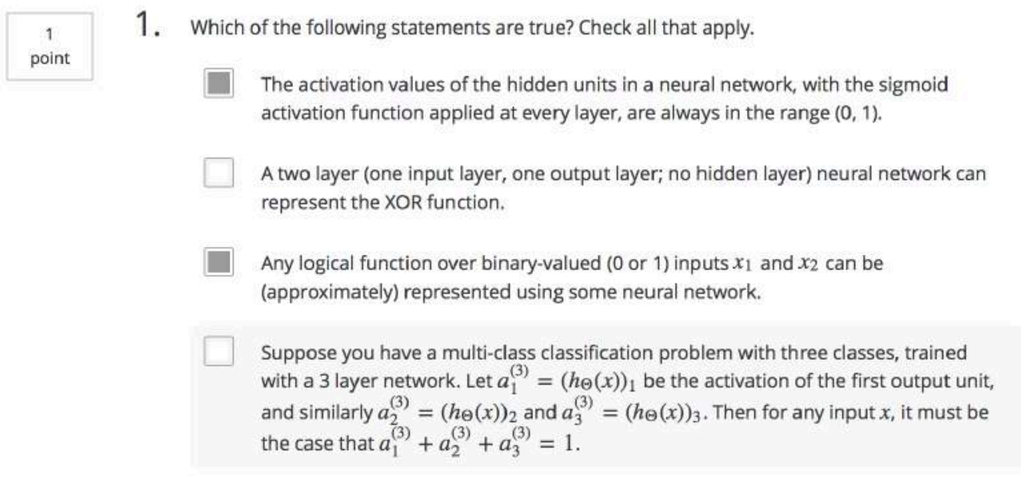
Our resulting hypothesis for one set of inputs may look like:

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

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

Review

Quiz



1 point

1. Which of the following statements are true? Check all that apply.

[✓] The activation values of the hidden units in a neural network, with the sigmoid activation function applied at every layer, are always in the range (0, 1).

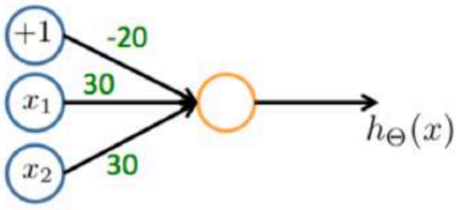
☐ A two layer (one input layer, one output layer; no hidden layer) neural network can represent the XOR function.

[✓] Any logical function over binary-valued (0 or 1) inputs x_1 and x_2 can be (approximately) represented using some neural network.

☐ Suppose you have a multi-class classification problem with three classes, trained with a 3 layer network. Let $a_1^{(3)} = (h_\Theta(x))_1$ be the activation of the first output unit, and similarly $a_2^{(3)} = (h_\Theta(x))_2$ and $a_3^{(3)} = (h_\Theta(x))_3$. Then for any input x, it must be the case that $a_1^{(3)} + a_2^{(3)} + a_3^{(3)} = 1$.

1 point

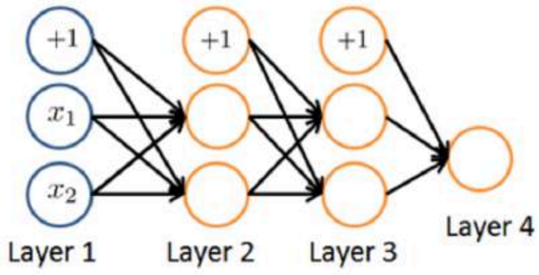
2. Consider the following neural network which takes two binary-valued inputs $x_1, x_2 \in \{0, 1\}$ and outputs $h_\Theta(x)$. Which of the following logical functions does it (approximately) compute?



- [✓ OR]
- AND
- NAND (meaning "NOT AND")
- XOR (exclusive OR)

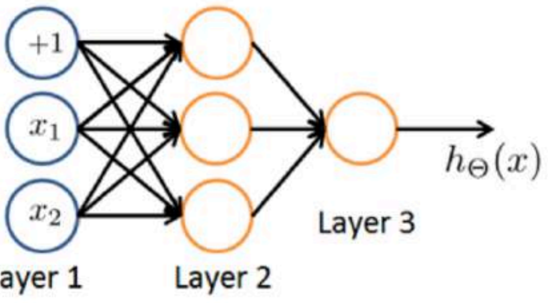
1 point

3. Consider the neural network given below. Which of the following equations correctly computes the activation $a_1^{(3)}$? Note: $g(z)$ is the sigmoid activation function.



- [✓ $a_1^{(3)} = g(\Theta_{1,0}^{(2)} a_0^{(2)} + \Theta_{1,1}^{(2)} a_1^{(2)} + \Theta_{1,2}^{(2)} a_2^{(2)})$]
- $a_1^{(3)} = g(\Theta_{1,0}^{(1)} a_0^{(1)} + \Theta_{1,1}^{(1)} a_1^{(1)} + \Theta_{1,2}^{(1)} a_2^{(1)})$
- $a_1^{(3)} = g(\Theta_{1,0}^{(1)} a_0^{(2)} + \Theta_{1,1}^{(1)} a_1^{(2)} + \Theta_{1,2}^{(1)} a_2^{(2)})$
- The activation $a_1^{(3)}$ is not present in this network.

4. You have the following neural network:



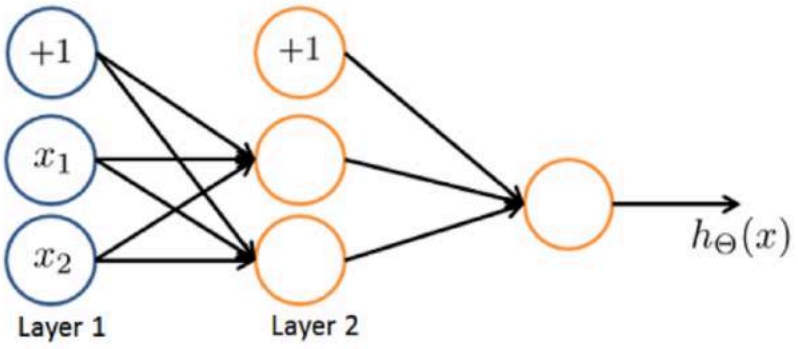
You'd like to compute the activations of the hidden layer $a^{(2)} \in \mathbb{R}^3$. One way to do so is the following Octave code:

```
% Theta1 is Theta with superscript "(1)" from lecture
% ie, the matrix of parameters for the mapping from layer 1 (input) to layer 2
% Theta1 has size 3x3
% Assume 'sigmoid' is a built-in function to compute 1 / (1 + exp(-z))

a2 = zeros (3, 1);
for i = 1:3
  for j = 1:3
    a2(i) = a2(i) + x(j) * Theta1(i, j);
  end
  a2(i) = sigmoid (a2(i));
end
```

You want to have a vectorized implementation of this (i.e., one that does not use for loops). Which of the following implementations correctly compute $a^{(2)}$? Check all that apply.

- [✓ z = Theta1 * x; a2 = sigmoid (z);]
- a2 = sigmoid (x * Theta1);
- a2 = sigmoid (Theta2 * x);
- z = sigmoid(x); a2 = sigmoid (Theta1 * z);

5. You are using the neural network pictured below and have learned the parameters $\Theta^{(1)} = \begin{bmatrix} 1 & -1.5 & 3.7 \\ 1 & 5.1 & 2.3 \end{bmatrix}$ (used to compute $a^{(2)}$) and $\Theta^{(2)} = \begin{bmatrix} 1 & 0.6 & -0.8 \end{bmatrix}$ (used to compute $a^{(3)}$} as a function of $a^{(2)}$). Suppose you swap the parameters for the first hidden layer between its two units so $\Theta^{(1)} = \begin{bmatrix} 1 & 5.1 & 2.3 \\ 1 & -1.5 & 3.7 \end{bmatrix}$ and also swap the output layer so $\Theta^{(2)} = \begin{bmatrix} 1 & -0.8 & 0.6 \end{bmatrix}$. How will this change the value of the output $h_\Theta(x)$?



[✓ It will stay the same.]

It will increase.

It will decrease

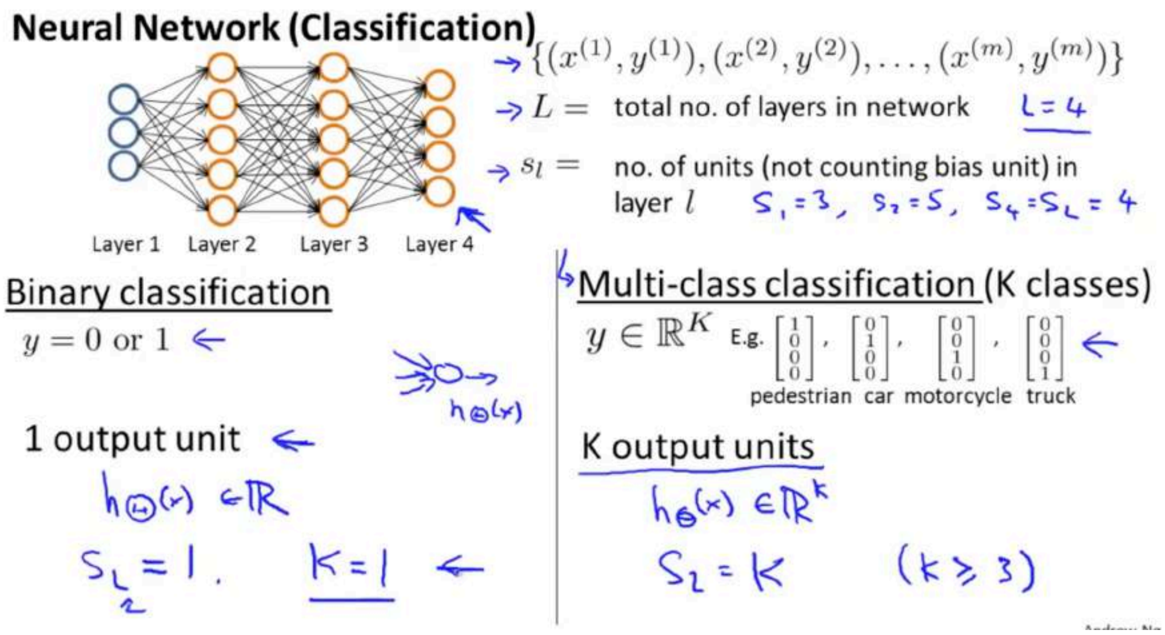
Insufficient information to tell: it may increase or decrease.

WEEK 5

Cost Function and Back Propagation

Cost Function (Video)

Neural networks are one of the most powerful learning algorithms that we have today. In this and in the next few videos, I'd like to start talking about a learning algorithm for fitting the parameters of a neural network given a training set. As with the discussion of most of our learning algorithms, we're going to begin by talking about the cost function for fitting the parameters of the network.I'm going to focus on the application of neural networks to classification problems. So suppose we have a network like that shown on the left. And suppose we have a training set like this is x I, y I pairs of M training example. I'm going to use upper case L to denote the total number of layers in this network. So for the network shown on the left we would have capital L equals 4. I'm going to use S subscript L to denote the number of units, that is the number of neurons. Not counting the bias unit in their L of the network. So for example, we would have a S one, which is equal there, equals S three unit, S two in my example is five units. And the output layer S four, which is also equal to S L because capital L is equal to four. The output layer in my example under that has four units. We're going to consider two types of classification problems. The first is Binary classification, where the labels y are either 0 or 1. In this case, we will have 1 output unit, so this Neural Network unit on top has 4 output units, but if we had binary classification we would have only one output unit that computes h(x). And the output of the neural network would be h(x) is going to be a real number. And in this case the number of output units, S L, where L is again the index of the final layer. Cuz that's the number of layers we have in the network so the number of units we have in the output layer is going to be equal to 1. In this case to simplify notation later, I'm also going to set K=1 so you can think of K as also denoting the number of units in the output layer. The second type of classification problem we'll consider will be multi-class classification problem where we may have K distinct classes. So our early example had this representation for y if we have 4 classes, and in this case we will have capital K output units and our hypothesis or output vectors that are K dimensional. And the number of output units will be equal to K. And usually we would have K greater than or equal to 3 in this case, because if we had two causes, then we don't need to use the one verses all method. We use the one verses all method only if we have K greater than or equals V classes, so having only two classes we will need to use only one upper unit.



Neural Network (Classification)

$\{(x^{(1)},y^{(1)}),(x^{(2)},y^{(2)}),\ldots,(x^{(m)},y^{(m)})\}$

$L =$ total no. of layers in network $L=4$

$s_l =$ no. of units (not counting bias unit) in layer l $s_1=3,\ s_2=5,\ s_4=s_L=4$

Binary classification

$y = 0 \text{ or } 1$

1 output unit

$h_\Theta(x) \in \mathbb{R}$

$s_L = 1$. $K=1$

Multi-class classification (K classes)

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

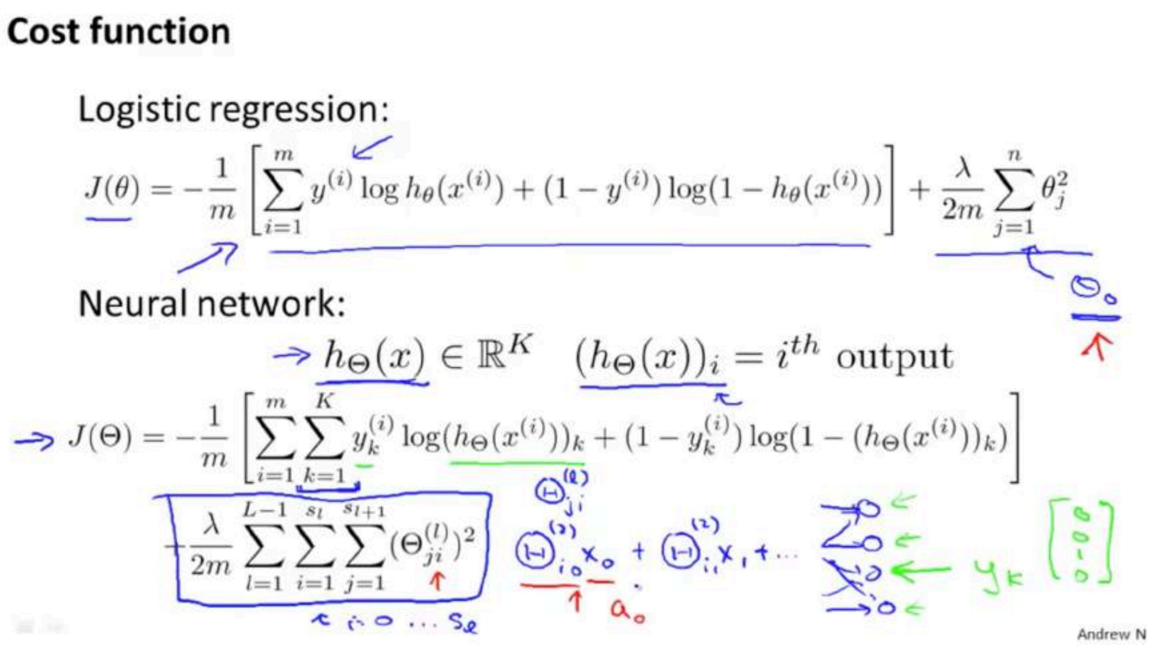
pedestrian car motorcycle truck

K output units

$h_\Theta(x) \in \mathbb{R}^K$

$s_L = K$ $(K \geq 3)$

Now let's define the cost function for our neural network. The cost function we use for the neural network is going to be a generalization of the one that we use for logistic regression. For logistic regression we used to minimize the cost function J(theta) that was minus 1/m of this cost function and then plus this extra regularization term here, where this was a sum from J=1 through n, because we did not regularize the bias term theta0. For a neural network, our cost function is going to be a generalization of this. Where instead of having basically just one, which is the compression output unit, we may instead have K of them. So here's our cost function. Our new network now outputs vectors in R K where R might be equal to 1 if we have a binary classification problem. I'm going to use this notation h(x) subscript i to denote the ith output. That is, h(x) is a k-dimensional vector and so this subscript i just selects out the ith element of the vector that is output by my neural network. My cost function J(theta) is now going to be the following. Is - 1 over M of a sum of a similar term to what we have for logistic regression, except that we have the sum from K equals 1 through K. This summation is basically a sum over my K output. A unit. So if I have four output units, that is if the final layer of my neural network has four output units, then this is a sum from k equals one through four of basically the logistic regression algorithm's cost function but summing that cost function over each of my four output units in turn. And so you notice in particular that this applies to Yk Hk, because we're basically taking the K upper units, and comparing that to the value of Yk which is that one of those vectors saying what cost it should be. And finally, the second term here is the regularization term, similar to what we had for the logistic regression. This summation term looks really complicated, but all it's doing is it's summing over these terms theta j i l for all values of i j and l. Except that we don't sum over the terms corresponding to these bias values like we have for logistic progression. Completely, we don't sum over the terms responding to where i is equal to 0. So that is because when we're computing the activation of a neuron, we have terms like these. Theta i 0. Plus theta i1, x1 plus and so on. Where I guess put in a two there, this is the first hit in there. And so the values with a zero there, that corresponds to something that multiplies into an x0 or an a0. And so this is kinda like a bias unit and by analogy to what we were doing for logistic progression, we won't sum over those terms in our regularization term because we don't want to regularize them and string their values as zero. But this is just one possible convention, and even if you were to sum over i equals 0 up to Sl, it would work about the same and doesn't make a big difference. But maybe this convention of not regularizing the bias term is just slightly more common.



Cost function

Logistic regression:

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

Θ_0

Neural network:

$h_\Theta(x) \in \mathbb{R}^K \quad (h_\Theta(x))_i = i^{th}$ output

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)}\log(h_\Theta(x^{(i)}))_k + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right]$$

$$+\boxed{\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2}$$

$i = 0 \ldots s_l$

$\Theta_{ji}^{(l)}$

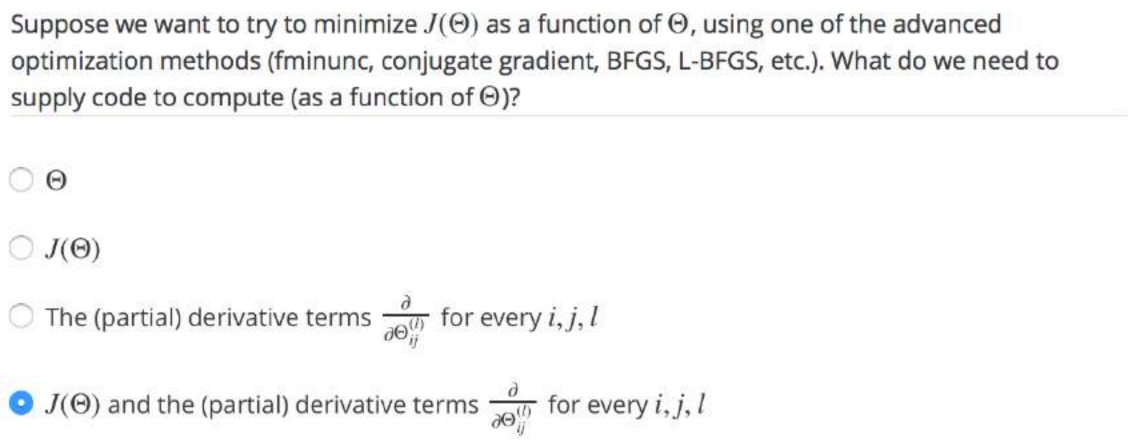
$\Theta_{i0}^{(2)} x_0 + \Theta_{i1}^{(2)} x_1 + \ldots$

a_0

y_k $\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$

Andrew N

Question



Suppose we want to try to minimize $J(\Theta)$ as a function of Θ, using one of the advanced optimization methods (fminunc, conjugate gradient, BFGS, L-BFGS, etc.). What do we need to supply code to compute (as a function of Θ)?

○ Θ

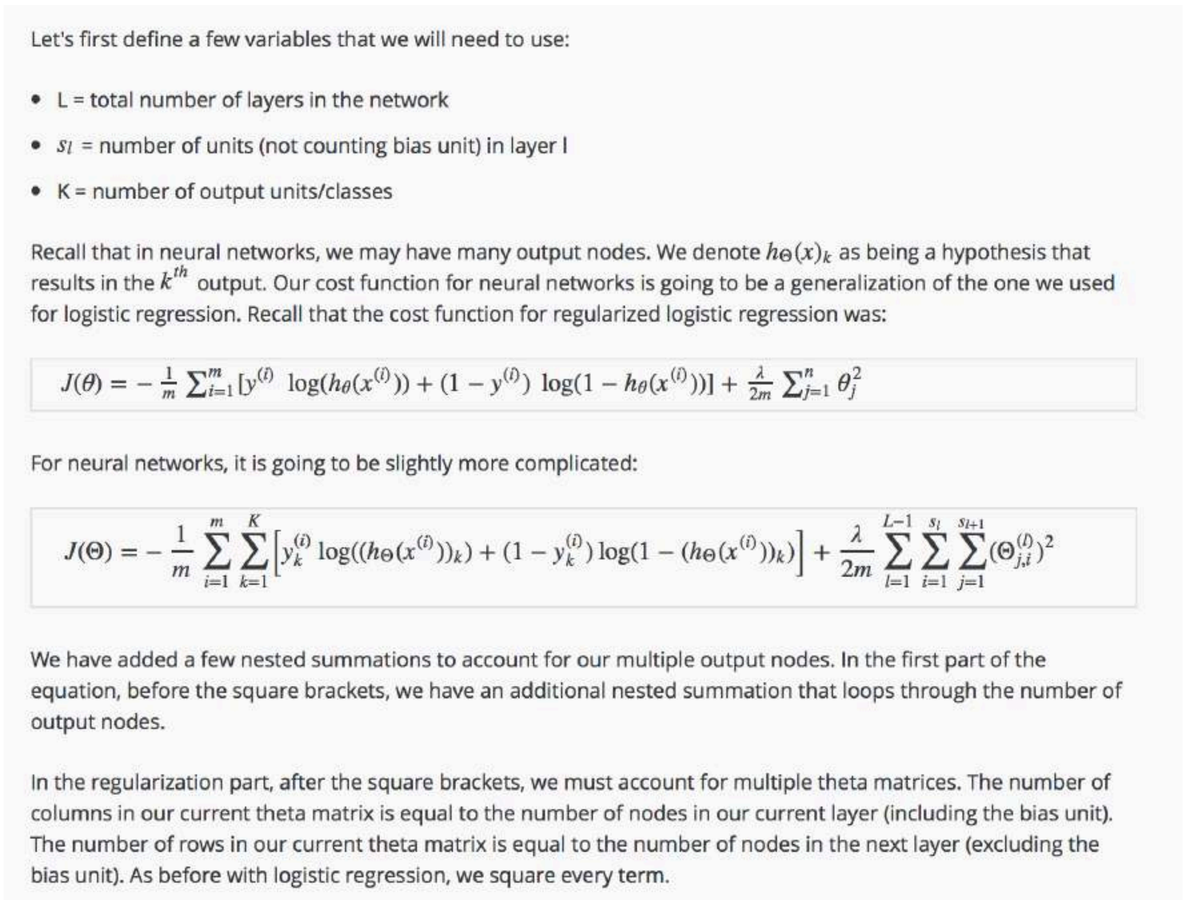
○ $J(\Theta)$

○ The (partial) derivative terms $\frac{\partial}{\partial \Theta_{ij}^{(l)}}$ for every i, j, l

● $J(\Theta)$ and the (partial) derivative terms $\frac{\partial}{\partial \Theta_{ij}^{(l)}}$ for every i, j, l

So that's the cost function we're going to use for our neural network. In the next video we'll start to talk about an algorithm for trying to optimize the cost function.

Cost Function (Transcript)



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

- L = total number of layers in the network
- s_l = 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. Recall that the cost function for regularized logistic regression was:

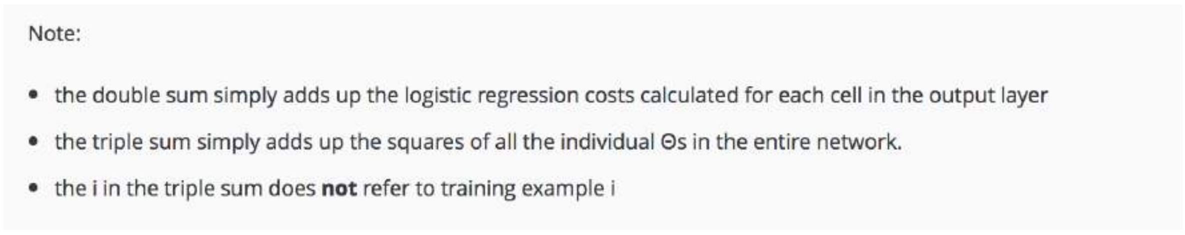
$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\ \log(h_\theta(x^{(i)})) + (1-y^{(i)})\ \log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

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

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

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.

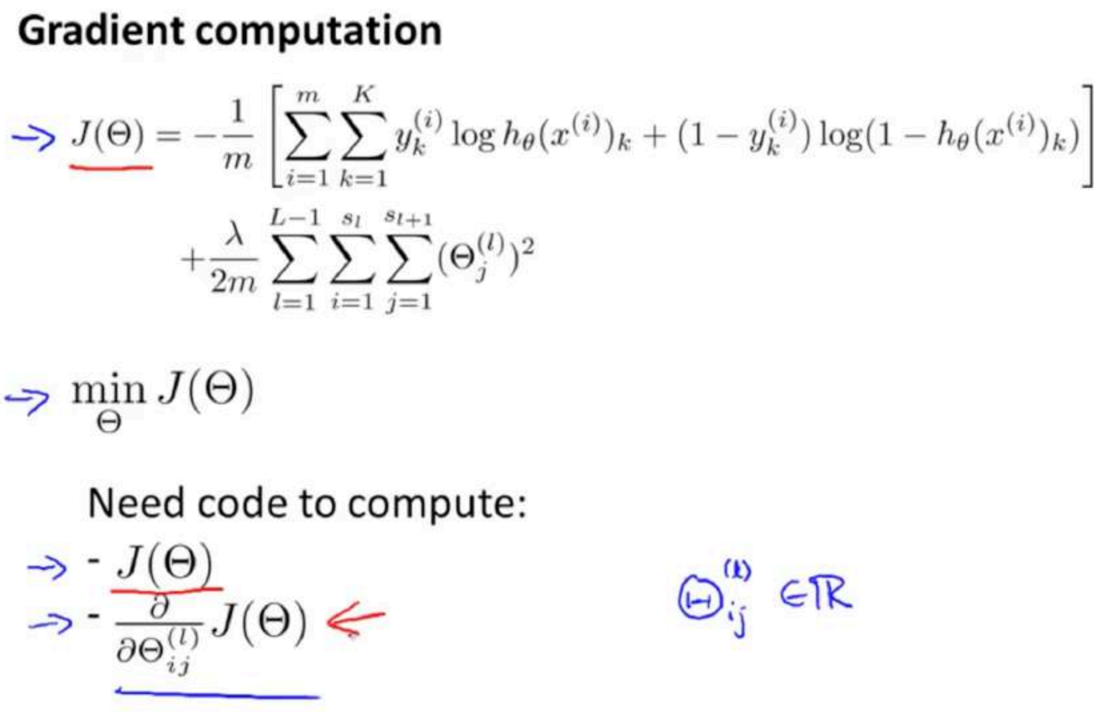


Note:

- 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 (Video)

In the previous video, we talked about a cost function for the neural network. In this video, let's start to talk about an algorithm, for trying to minimize the cost function. In particular, we'll talk about the back propagation algorithm. 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.



Gradient computation

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

$$\rightarrow \min_\Theta J(\Theta)$$

Need code to compute:

$\rightarrow$ - $\underline{J(\Theta)}$

$\rightarrow$ - $\underline{\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta)} \leftarrow$

$\Theta_{ij}^{(l)} \in \mathbb{R}$

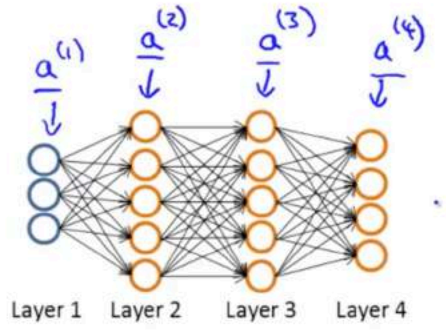
Let's start by talking about the case of when we have only one training example, so imagine, if you will that our entire training set comprises only one training example which is a pair xy. I'm not going to write x1y1 just write this. Write a one training example as xy and let's tap through the sequence of calculations we would do with this one training example. 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.

Gradient computation

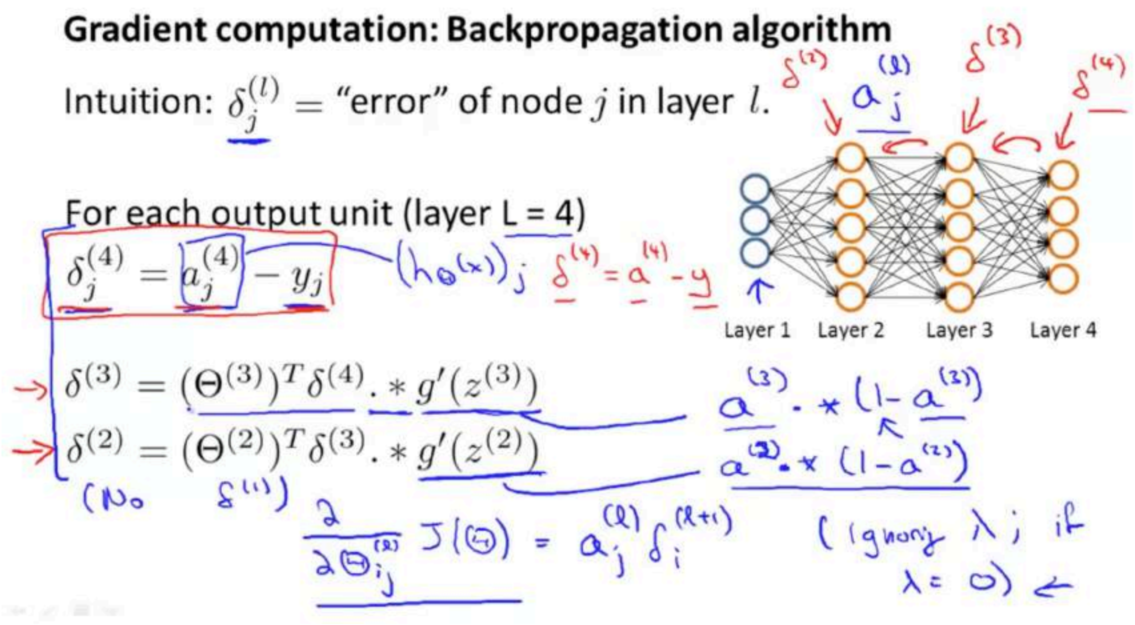
Given one training example ($\underline{x, y}$):

Forward propagation:

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



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 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 just the activation of that unit minus what was the actual value of 0 in our training example. 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. 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. So we've now computed the era term's delta 4 for our network. 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 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. 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. 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. 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. 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 are exactly given by the activations and these delta terms. This is ignoring lambda or alternatively the regularization 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.



Gradient computation: Backpropagation algorithm

Intuition: $\delta_j^{(l)}$ = "error" of node j in layer l.

For each output unit (layer L = 4)

$$\boxed{\delta_j^{(4)} = \boxed{a_j^{(4)}} - y_j} \quad (h_\Theta(x))_j \quad \delta^{(4)} = a^{(4)} - y$$

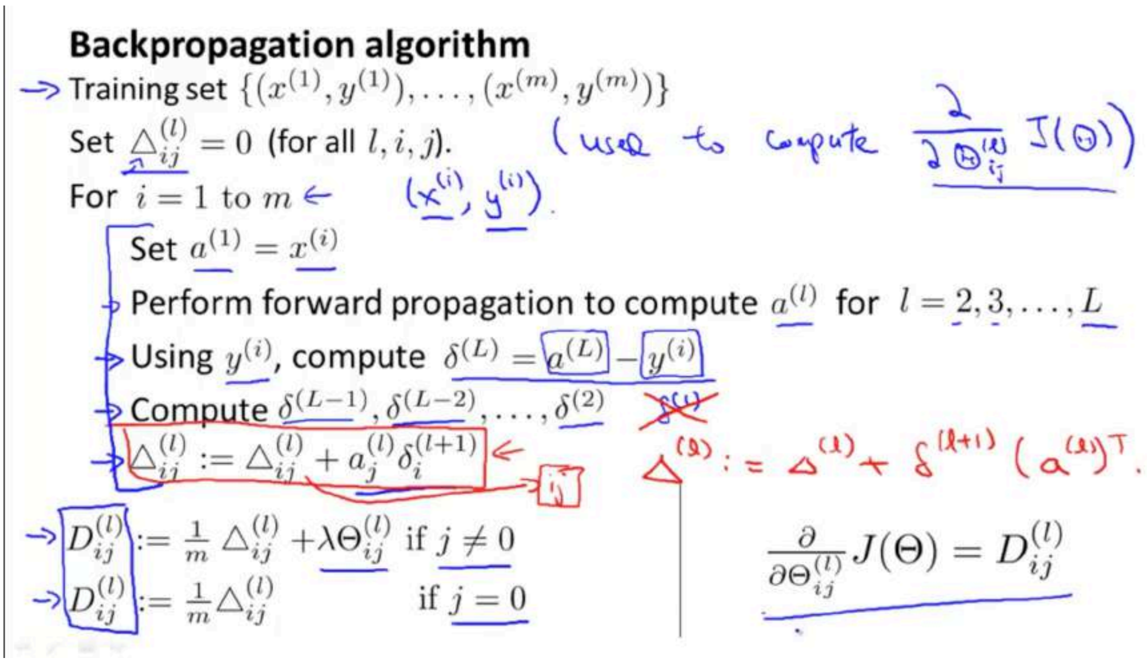
$$\delta^{(3)} = (\Theta^{(3)})^T\delta^{(4)} .* g'(z^{(3)}) \qquad a^{(3)} .* (1-a^{(3)})$$

$$\delta^{(2)} = (\Theta^{(2)})^T\delta^{(3)} .* g'(z^{(2)}) \qquad a^{(2)} .* (1-a^{(2)})$$

(No $\delta^{(1)}$)

$$\frac{\partial}{\partial\Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)}\delta_i^{(l+1)} \qquad (\text{ignoring } \lambda;\ \text{if } \lambda = 0)$$

So this is a lot of detail. Let's take everything and put it all together to talk about how to implement back propagation to compute derivatives with respect to your parameters. 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 to compute the partial derivative term, partial derivative respect to theta l i j of J of theta. 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. 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. 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? 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. 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. 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. Then, if delta L is a matrix we can rewrite this as delta L, gets updated as delta L plus 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. 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. 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 algorithms.



Backpropagation algorithm

$\rightarrow$ Training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$

Set $\triangle_{ij}^{(l)} = 0$ (for all l, i, j). $\left(\text{used to compute } \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)\right)$

For $i = 1$ to $m \leftarrow$ $(x^{(i)}, y^{(i)})$.

```
    Set $a^{(1)} = x^{(i)}$
    Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \ldots, L$
    Using $y^{(i)}$, compute $\delta^{(L)} = \boxed{a^{(L)}} - \boxed{y^{(i)}}$
    Compute $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ $\cancel{\delta^{(1)}}$
    $\boxed{\triangle_{ij}^{(l)} := \triangle_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}} \leftarrow$ $\boxed{ij}$
```

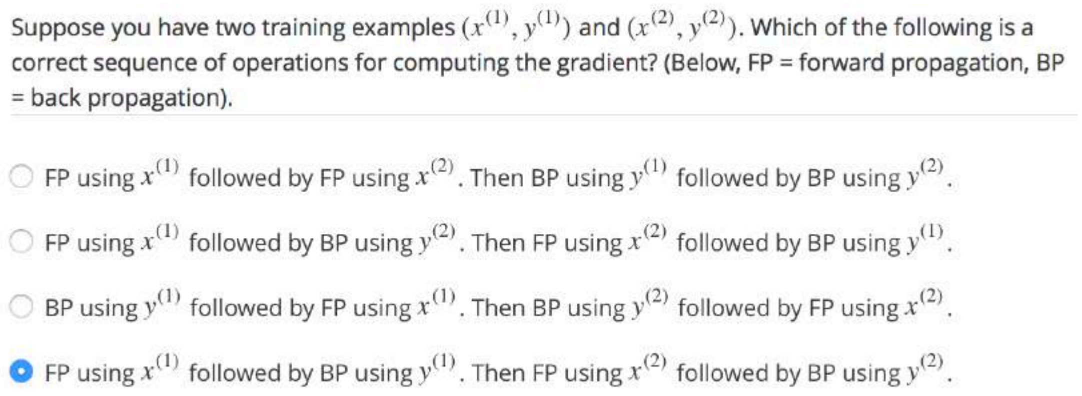
$\triangle^{(l)} := \triangle^{(l)} + \delta^{(l+1)} (a^{(l)})^T$.

$\rightarrow \boxed{D_{ij}^{(l)}} := \frac{1}{m} \triangle_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)}$ if $j \neq 0$

$\rightarrow \boxed{D_{ij}^{(l)}} := \frac{1}{m} \triangle_{ij}^{(l)}$ if $j = 0$

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

Question



Suppose you have two training examples $(x^{(1)}, y^{(1)})$ and $(x^{(2)}, y^{(2)})$. Which of the following is a correct sequence of operations for computing the gradient? (Below, FP = forward propagation, BP = back propagation).

- FP using $x^{(1)}$ followed by FP using $x^{(2)}$. Then BP using $y^{(1)}$ followed by BP using $y^{(2)}$.
- FP using $x^{(1)}$ followed by BP using $y^{(2)}$. Then FP using $x^{(2)}$ followed by BP using $y^{(1)}$.
- BP using $y^{(1)}$ followed by FP using $x^{(1)}$. Then BP using $y^{(2)}$ followed by FP using $x^{(2)}$.
- [✓ FP using $x^{(1)}$ followed by BP using $y^{(1)}$. Then FP using $x^{(2)}$ followed by BP using $y^{(2)}$.]

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.

Backpropagation Algorithm (Transcript)

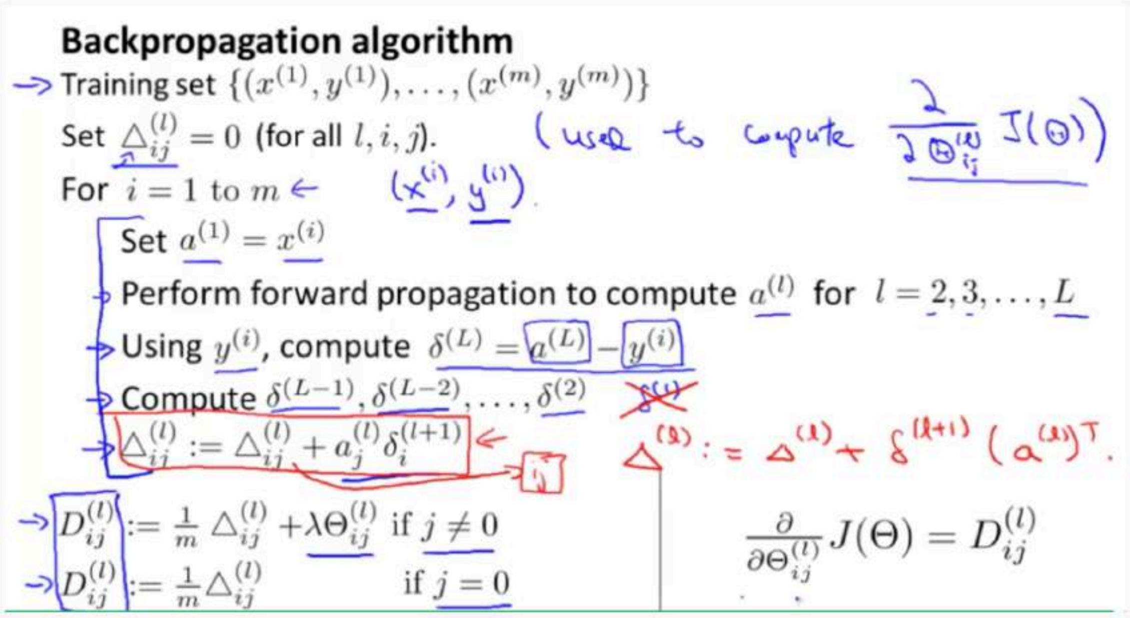
"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute:

$\min_\Theta J(\Theta)$

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(Θ):

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

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

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

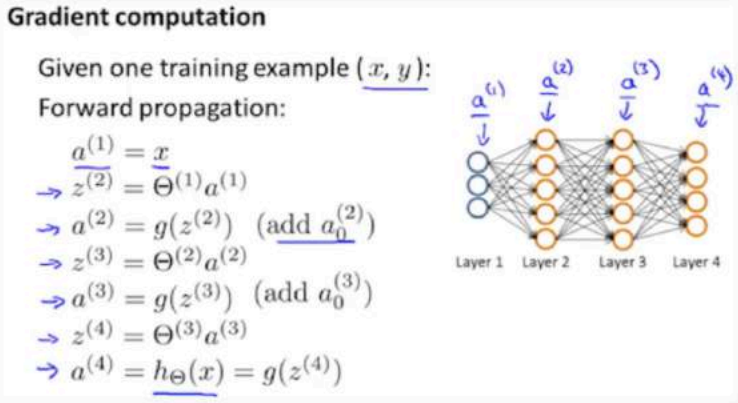
Back propagation Algorithm

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

- 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:

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

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 differences 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 $\delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}$ 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.

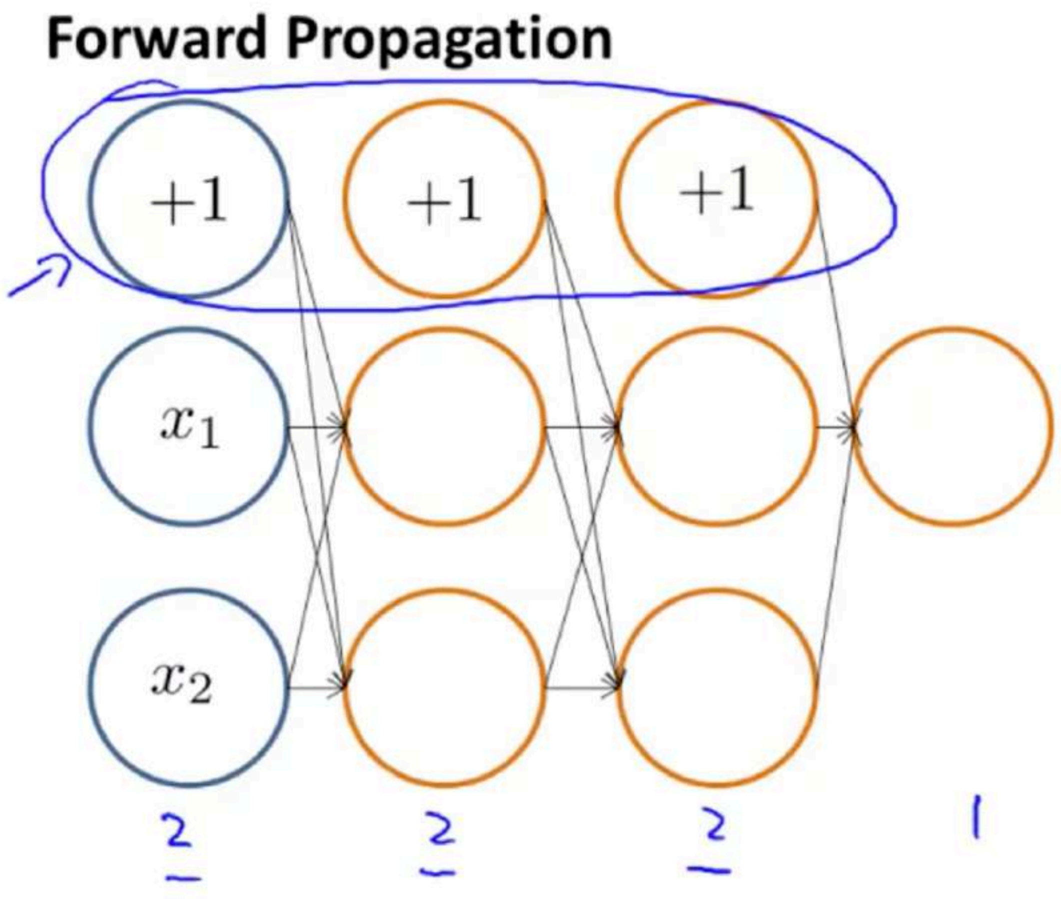
- $D^{(l)}_{i,j} := \dfrac{1}{m}\left(\Delta^{(l)}_{i,j} + \lambda\Theta^{(l)}_{i,j}\right)$, if j≠0.
- $D^{(l)}_{i,j} := \dfrac{1}{m}\Delta^{(l)}_{i,j}$ If j=0

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 $\frac{\partial}{\partial\Theta^{(l)}_{ij}} J(\Theta)$= $D^{(l)}_{ij}$

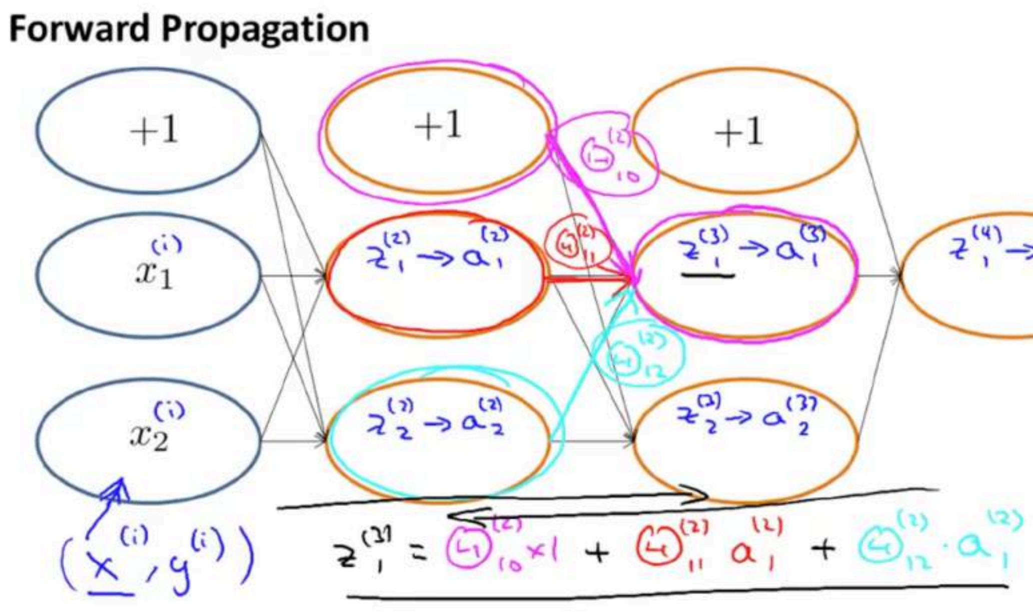
Backpropagation Intuition (Video)

In the previous video, we talked about the backpropagation algorithm. To a lot of people seeing it for the first time, their first impression is often that wow this is a really complicated algorithm, and there are all these different steps, and I'm not sure how they fit together. And it's kinda this black box of all these complicated steps. In case that's how you're feeling about backpropagation, that's actually okay. Backpropagation maybe unfortunately is a less mathematically clean, or less mathematically simple algorithm, compared to linear regression or logistic regression. And I've actually used backpropagation, you know, pretty successfully for many years. And even today I still don't sometimes feel like I have a very good sense of just what it's doing, or intuition about what back propagation is doing. If, for those of you that are doing the programming exercises, that will at least mechanically step you through the different steps of how to implement back prop. So you'll be able to get it to work for yourself. And what I want to do in this video is look a little bit more at the mechanical steps of backpropagation, and try to give you a little more intuition about what the mechanical steps the back prop is doing to hopefully convince you that, you know, it's at least a reasonable algorithm. In case even after this video in case back propagation still seems very black box and kind of like a, too many complicated steps and a little bit magical to you, that's actually okay. And Even though I've used back prop for many years, sometimes this is a difficult algorithm to understand, but hopefully this video will help a little bit.

In order to better understand backpropagation, let's take another closer look at what forward propagation is doing. Here's a neural network with two input units that is not counting the bias unit, and two hidden units in this layer, and two hidden units in the next layer. And then, finally, one output unit. Again, these counts two, two, two, are not counting these bias units on top. In order to illustrate forward propagation, I'm going to draw this network a little bit differently.

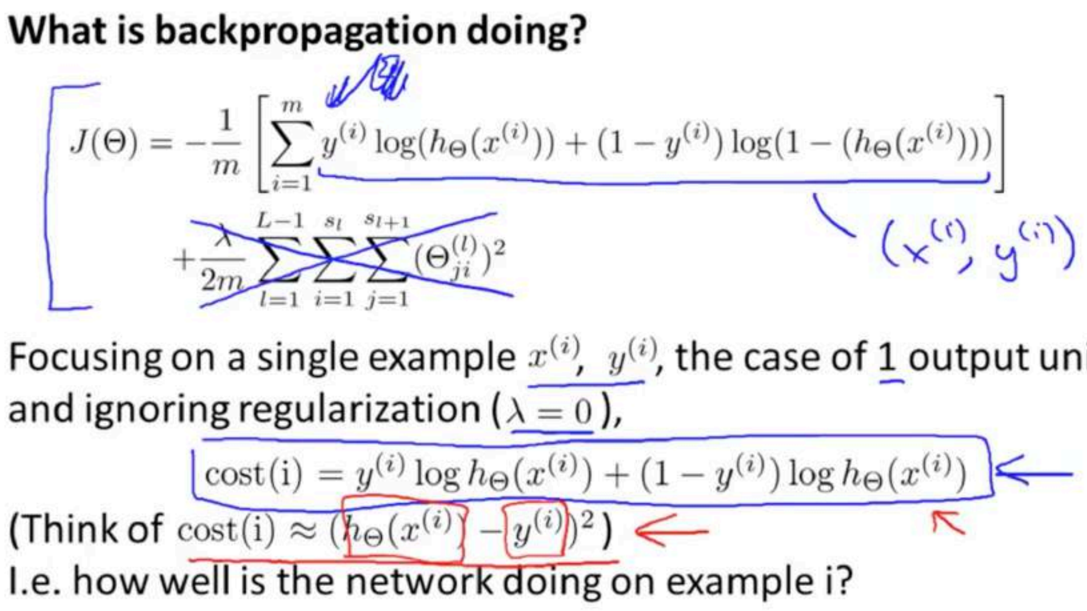


And in particular I'm going to draw this neuro network with the nodes drawn as these very fat ellipsis, so that I can write text in them. When performing forward propagation, we might have some particular example. Say some example x i comma y i. And it'll be this x i that we feed into the input layer. 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. 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.


Forward Propagation
+1
$x_1^{(i)}$
$x_2^{(i)}$
+1
$z_1^{(2)} \to a_1^{(2)}$
$z_2^{(2)} \to a_2^{(2)}$
+1
$z_1^{(3)} \to a_1^{(3)}$
$z_2^{(3)} \to a_2^{(3)}$
$z_1^{(4)} \to a_1^{(4)}$
$\Theta_{10}^{(2)}$
$\Theta_{11}^{(2)}$
$\Theta_{12}^{(2)}$
$(x^{(i)}, y^{(i)})$
$z_1^{(3)} = \Theta_{10}^{(2)} \times 1 + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} \cdot a_1^{(2)}$

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. 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 example 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)



What is backpropagation doing?

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m} \underbrace{y^{(i)}\log(h_\Theta(x^{(i)})) + (1-y^{(i)})\log(1-(h_\Theta(x^{(i)})))}_{(x^{(i)},\, y^{(i)})}\right] + \cancel{\frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{ji}^{(l)})^2}$$

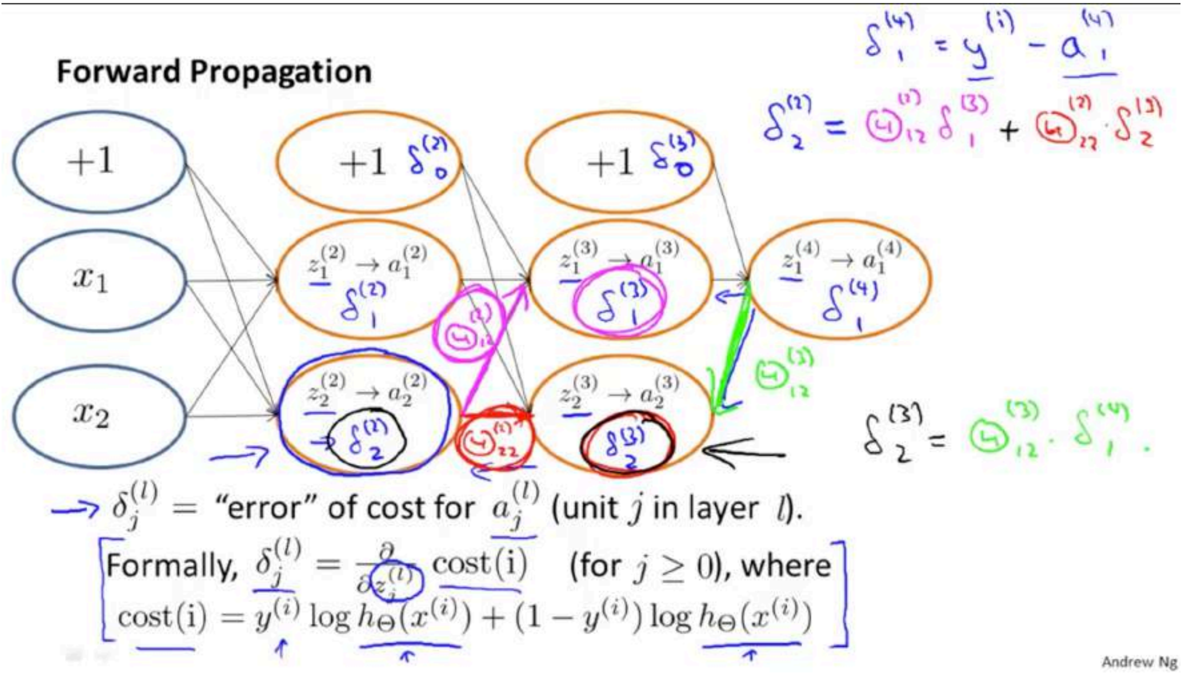
Focusing on a single example $\underline{x^{(i)},\ y^{(i)}}$, the case of $\underline{1}$ output unit, and ignoring regularization ($\underline{\lambda = 0}$),

$$\boxed{\text{cost(i)} = y^{(i)}\log h_\Theta(x^{(i)}) + (1-y^{(i)})\log h_\Theta(x^{(i)})} \leftarrow$$

(Think of $\text{cost(i)} \approx (\boxed{h_\Theta(x^{(i)})} - \boxed{y^{(i)}})^2$) $\leftarrow$

I.e. how well is the network doing on example 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. 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. 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. 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.



Forward Propagation

$\delta_1^{(4)} = y^{(i)} - a_1^{(4)}$

$\delta_2^{(2)} = \Theta_{12}^{(2)} \delta_1^{(3)} + \Theta_{22}^{(2)} \cdot \delta_2^{(3)}$

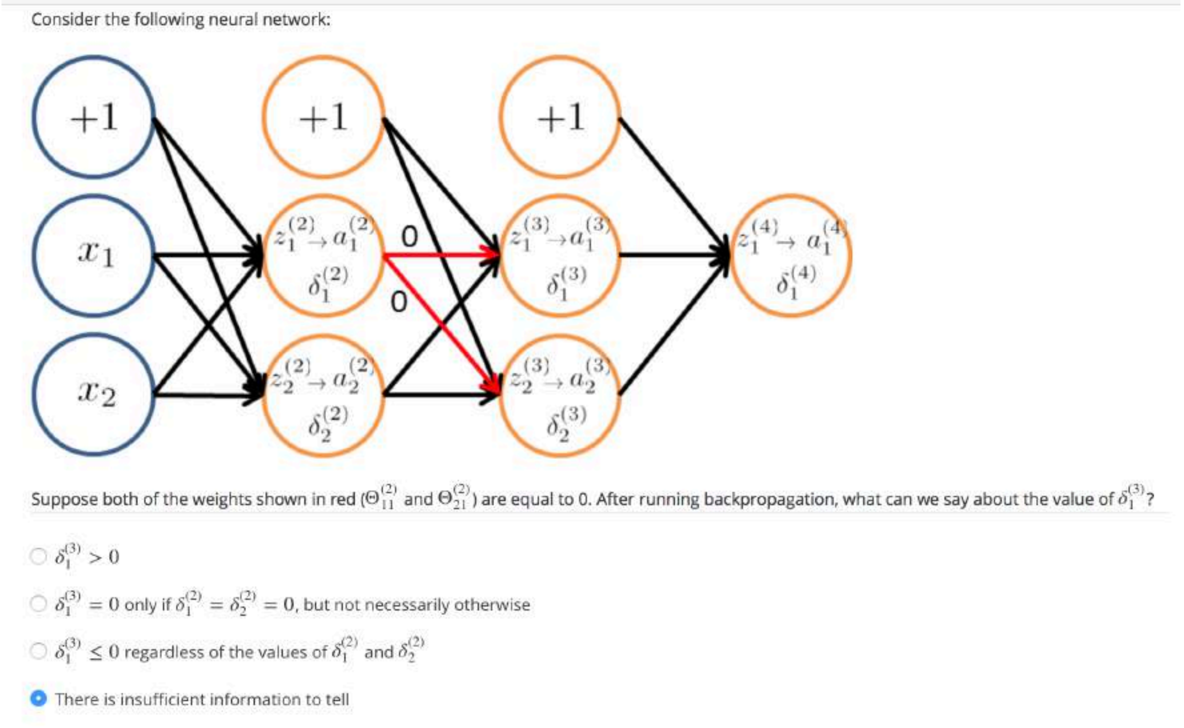
$\delta_2^{(3)} = \Theta_{12}^{(3)} \cdot \delta_1^{(4)}$.

$\delta_j^{(l)} =$ "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
$\text{cost(i)} = y^{(i)} \log h_\Theta(x^{(i)}) + (1 - y^{(i)}) \log h_\Theta(x^{(i)})$

Andrew Ng

Question



Consider the following neural network:

+1
x_1
x_2
+1
$z_1^{(2)} \to a_1^{(2)}$
$\delta_1^{(2)}$
$z_2^{(2)} \to a_2^{(2)}$
$\delta_2^{(2)}$
0
0
+1
$z_1^{(3)} \to a_1^{(3)}$
$\delta_1^{(3)}$
$z_2^{(3)} \to a_2^{(3)}$
$\delta_2^{(3)}$
$z_1^{(4)} \to a_1^{(4)}$
$\delta_1^{(4)}$

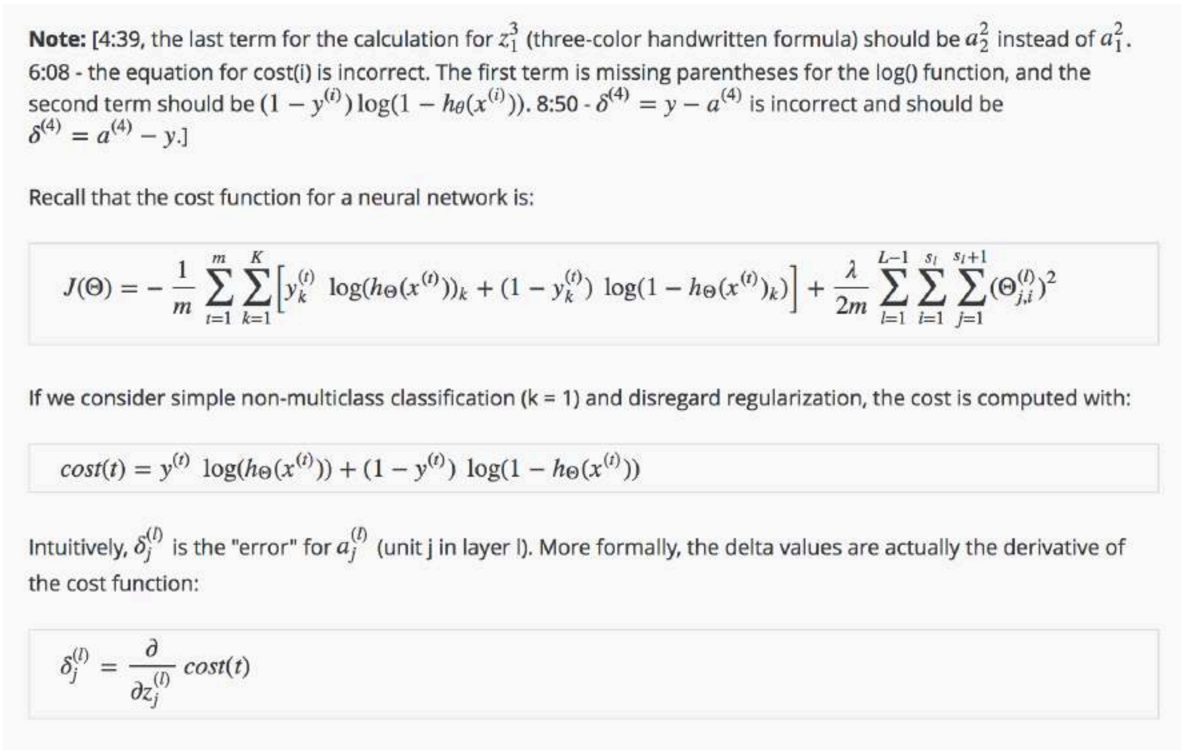
Suppose both of the weights shown in red ($\Theta_{11}^{(2)}$ and $\Theta_{21}^{(2)}$) are equal to 0. After running backpropagation, what can we say about the value of $\delta_1^{(3)}$?

- ○ $\delta_1^{(3)} > 0$
- ○ $\delta_1^{(3)} = 0$ only if $\delta_1^{(2)} = \delta_2^{(2)} = 0$, but not necessarily otherwise
- ○ $\delta_1^{(3)} \leq 0$ regardless of the values of $\delta_1^{(2)}$ and $\delta_2^{(2)}$
- [✓ There is insufficient information to tell]

So hopefully that gives you a little better intuition about what back propagation 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.

Backpropagation Intuition (Transcript)



Note: [4:39, the last term for the calculation for z_1^3 (three-color handwritten formula) should be a_2^2 instead of a_1^2. 6:08 - the equation for cost(i) is incorrect. The first term is missing parentheses for the log() function, and the second term should be $(1 - y^{(i)})\log(1 - h_\theta(x^{(i)}))$. 8:50 - $\delta^{(4)} = y - a^{(4)}$ is incorrect and should be $\delta^{(4)} = a^{(4)} - y$.]

Recall that the cost function for a neural network is:

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

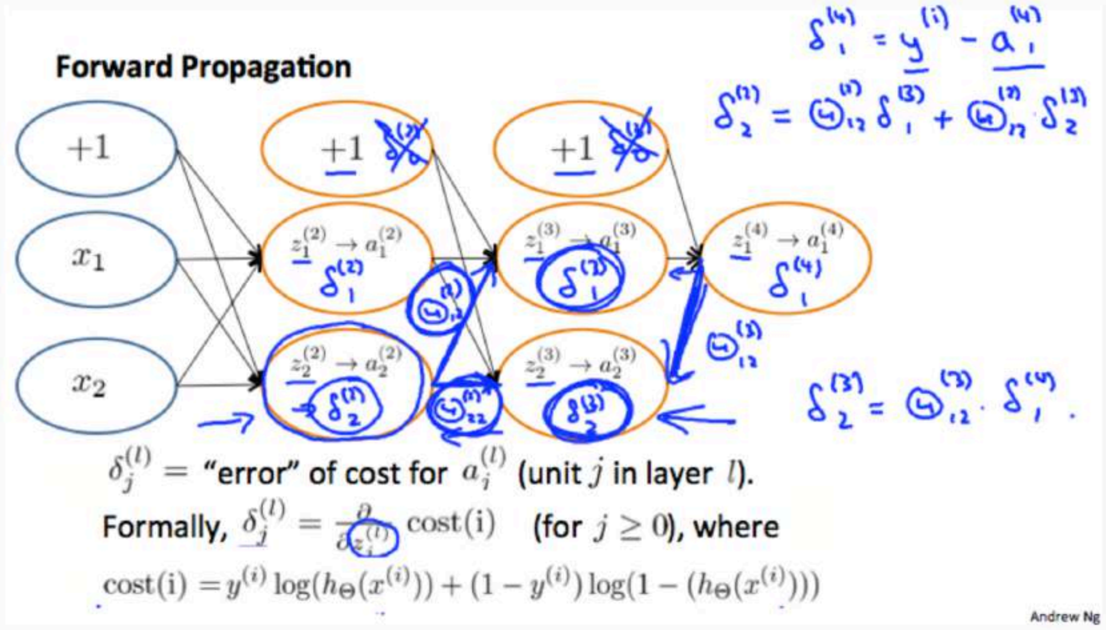
If we consider simple non-multiclass classification (k = 1) and disregard regularization, the cost is computed with:

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

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

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(t)$$

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


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

$\delta_j^{(l)}$ = "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

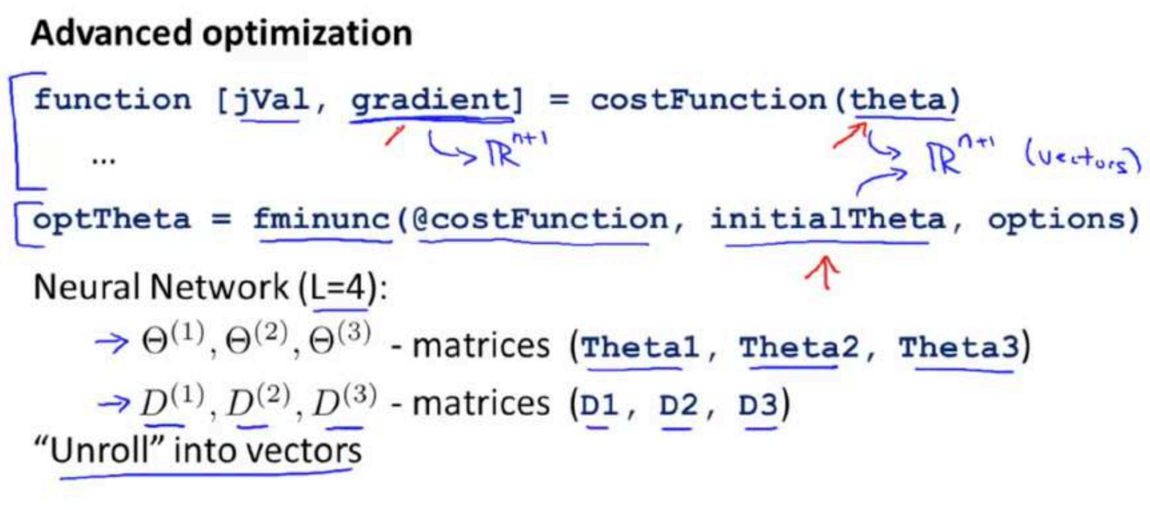
$\text{cost}(i) = y^{(i)} \log(h_\Theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - (h_\Theta(x^{(i)})))$

In the image above, to calculate $\delta_2^{(2)}$, 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 calculate 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 calculate 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 $\delta_2^{(3)} = \Theta_{12}^{(3)} * \delta_1^{(4)}$.

Backpropagation in Practice

Implementation Note: Unrolling Parameters (Video)

In the previous video, we talked about how to use back propagation to compute the derivatives of your cost function. In this video, I want to quickly tell you about one implementational detail of unrolling your parameters from matrices into vectors, which we need in order to use the advanced optimization routines. Concretely, let's say you've implemented a cost function that takes this input, you know, parameters theta and returns the cost function and returns derivatives. Then you can pass this to an advanced authorization algorithm by fminunc and fminunc isn't the only one by the way. There are also other advanced authorization algorithms. But what all of them do is take those input pointedly the cost function, and some initial value of theta. And both, and these routines assume that theta and the initial value of theta, that these are parameter vectors, maybe Rn or Rn plus 1. But these are vectors and it also assumes that, you know, your cost function will return as a second return value this gradient which is also Rn and Rn plus 1. So also a vector. This worked fine when we were using logistic progression but now that we're using a neural network our parameters are no longer vectors, but instead they are these matrices where for a full neural network we would have parameter matrices theta 1, theta 2, theta 3 that we might represent in Octave as these matrices theta 1, theta 2, theta 3. And similarly these gradient terms that were expected to return. Well, in the previous video we showed how to compute these gradient matrices, which was capital D1, capital D2, capital D3, which we might represent an octave as matrices D1, D2, D3. In this video I want to quickly tell you about the idea of how to take these matrices and unroll them into vectors. So that they end up being in a format suitable for passing into as theta here off for getting out for a gradient there.



Advanced optimization

```
function [jVal, gradient] = costFunction(theta)
   ...
optTheta = fminunc(@costFunction, initialTheta, options)
```

$\mathbb{R}^{n+1}$ $\mathbb{R}^{n+1}$ (vectors)

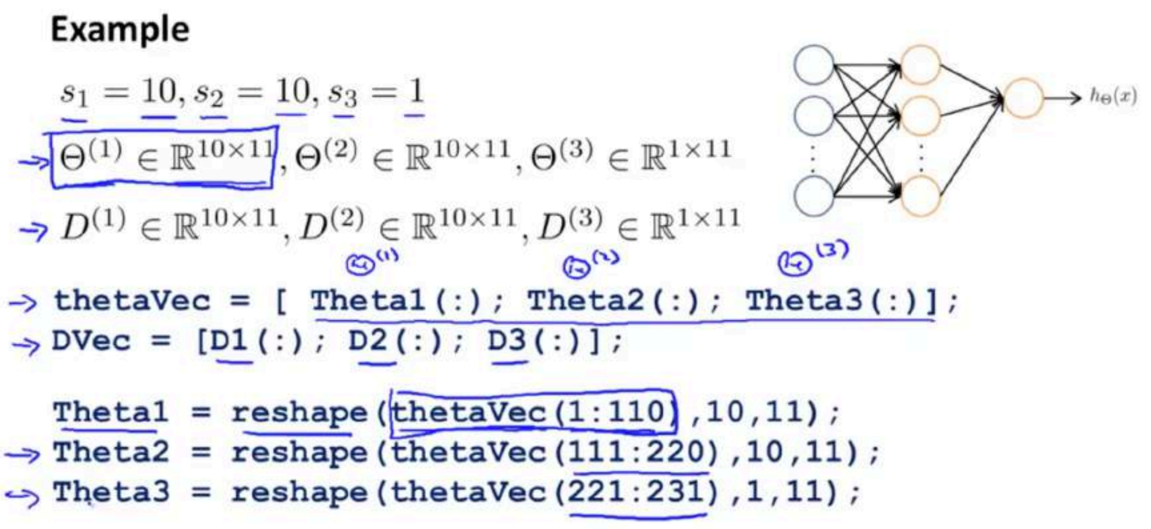
Neural Network (L=4):

$\rightarrow \Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ - matrices (`Theta1, Theta2, Theta3`)

$\rightarrow D^{(1)}, D^{(2)}, D^{(3)}$ - matrices (`D1, D2, D3`)

"Unroll" into vectors

Concretely, let's say we have a neural network with one input layer with ten units, hidden layer with ten units and one output layer with just one unit, so s1 is the number of units in layer one and s2 is the number of units in layer two, and s3 is a number of units in layer three. In this case, the dimension of your matrices theta and D are going to be given by these expressions. For example, theta one is going to a 10 by 11 matrix and so on. So in if you want to convert between these matrices. vectors. What you can do is take your theta 1, theta 2, theta 3, and write this piece of code and this will take all the elements of your three theta matrices and take all the elements of theta one, all the elements of theta 2, all the elements of theta 3, and unroll them and put all the elements into a big long vector. Which is thetaVec and similarly the second command would take all of your D matrices and unroll them into a big long vector and call them DVec. And finally if you want to go back from the vector representations to the matrix representations. What you do to get back to theta one say is take thetaVec and pull out the first 110 elements. So theta 1 has 110 elements because it's a 10 by 11 matrix so that pulls out the first 110 elements and then you can use the reshape command to reshape those back into theta 1. And similarly, to get back theta 2 you pull out the next 110 elements and reshape it. And for theta 3, you pull out the final eleven elements and run reshape to get back the theta 3.



Example

$s_1 = 10, s_2 = 10, s_3 = 1$

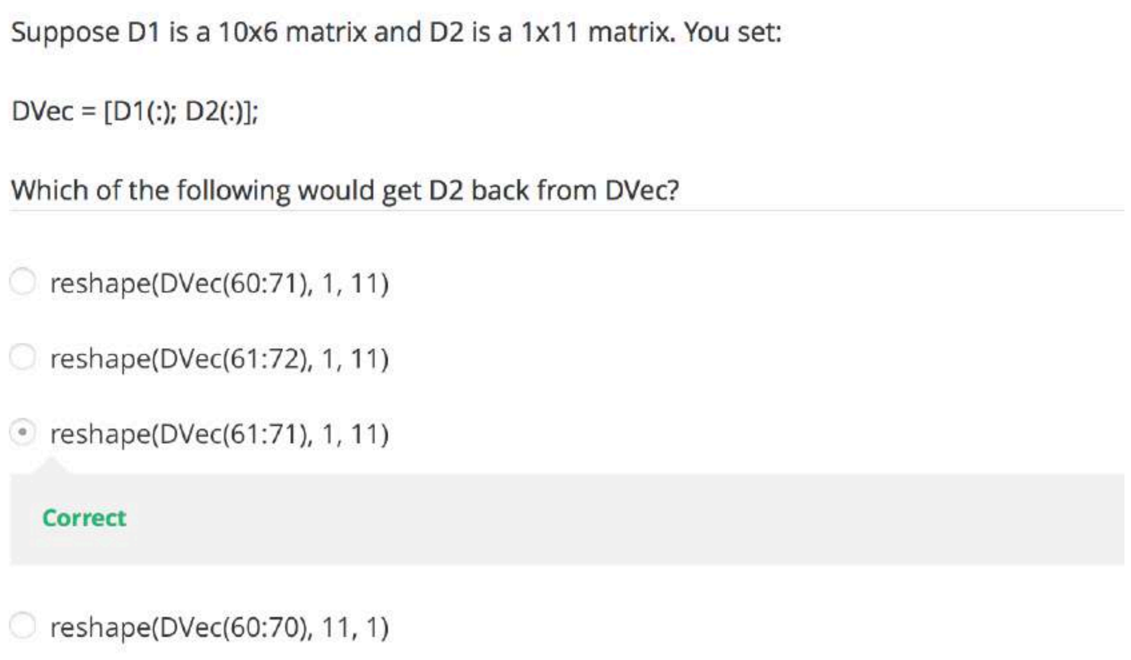
$\rightarrow \boxed{\Theta^{(1)} \in \mathbb{R}^{10\times 11}}, \Theta^{(2)} \in \mathbb{R}^{10\times 11}, \Theta^{(3)} \in \mathbb{R}^{1\times 11}$

$\rightarrow D^{(1)} \in \mathbb{R}^{10\times 11}, D^{(2)} \in \mathbb{R}^{10\times 11}, D^{(3)} \in \mathbb{R}^{1\times 11}$

$\Theta^{(1)}$ $\Theta^{(2)}$ $\Theta^{(3)}$

```
-> thetaVec = [ Theta1(:); Theta2(:); Theta3(:)];
-> DVec = [D1(:); D2(:); D3(:)];

   Theta1 = reshape(thetaVec(1:110),10,11);
-> Theta2 = reshape(thetaVec(111:220),10,11);
-> Theta3 = reshape(thetaVec(221:231),1,11);
```

Question



Suppose D1 is a 10x6 matrix and D2 is a 1x11 matrix. You set:

DVec = [D1(:); D2(:)];

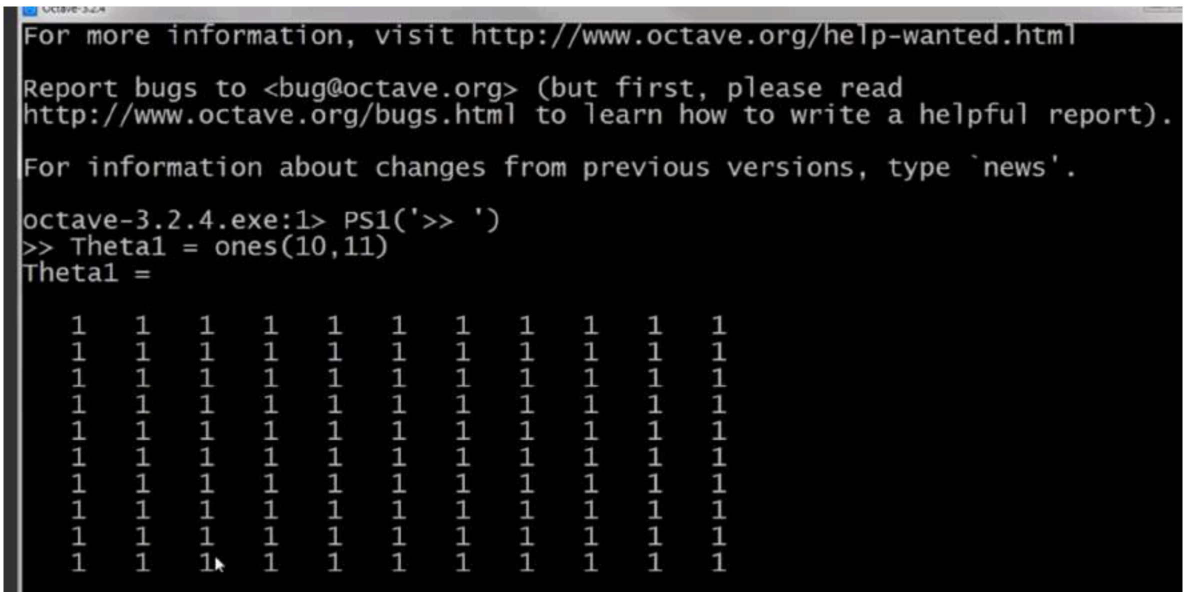
Which of the following would get D2 back from DVec?

- reshape(DVec(60:71), 1, 11)
- reshape(DVec(61:72), 1, 11)
- [✓ reshape(DVec(61:71), 1, 11)]

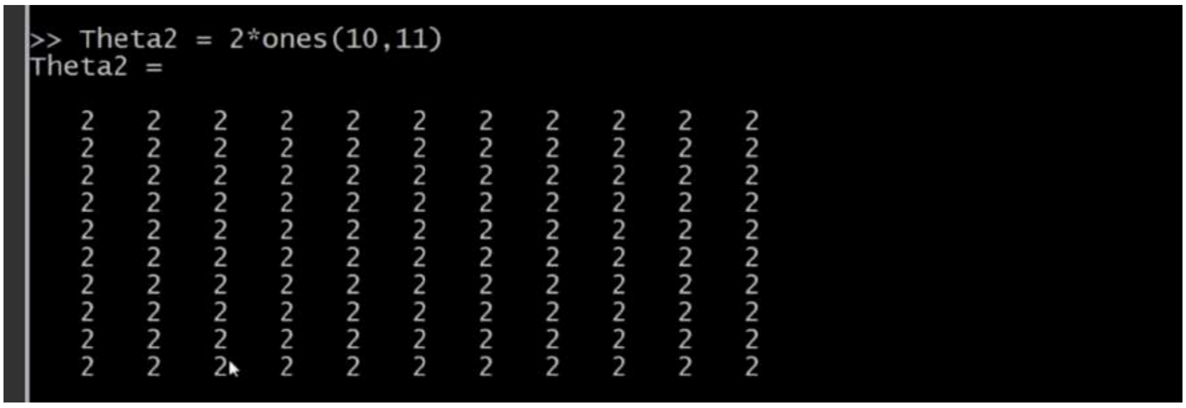
Correct

- reshape(DVec(60:70), 11, 1)

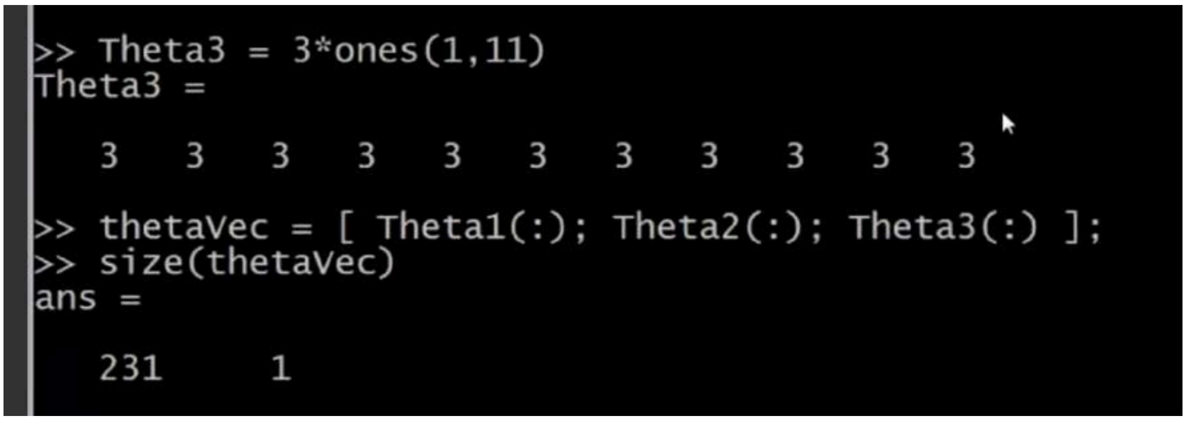
Here's a quick Octave demo of that process. So for this example let's set theta 1 equal to be ones of 10 by 11, so it's a matrix of all ones. And just to make this easier seen, let's set that to be 2 times ones, 10 by 11 and let's set theta 3 equals 3 times 1's of 1 by 11. So this is 3 separate matrices: theta 1, theta 2, theta 3. We want to put all of these as a vector. ThetaVec equals theta 1; theta 2 theta 3. Right, that's a colon in the middle and like so and now thetavec is going to be a very long vector. That's 231 elements. If I display it, I find that this very long vector with all the elements of the first matrix, all the elements of the second matrix, then all the elements of the third matrix.



```
For more information, visit http://www.octave.org/help-wanted.html

Report bugs to <bug@octave.org> (but first, please read
http://www.octave.org/bugs.html to learn how to write a helpful report).

For information about changes from previous versions, type `news'.

octave-3.2.4.exe:1> PS1('>> ')
>> Theta1 = ones(10,11)
Theta1 =

   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
```

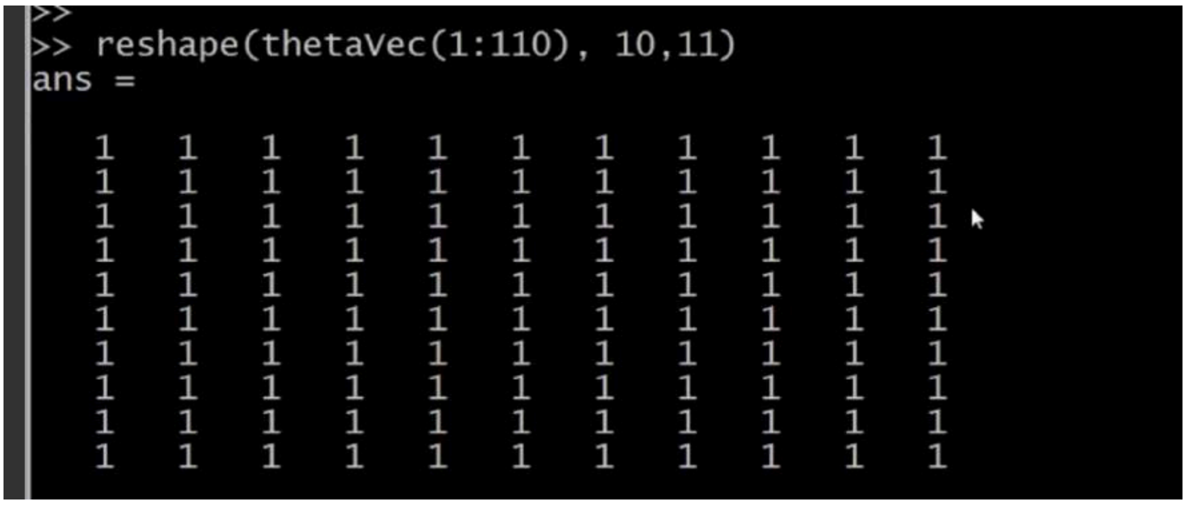


```
>> Theta2 = 2*ones(10,11)
Theta2 =

   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
```

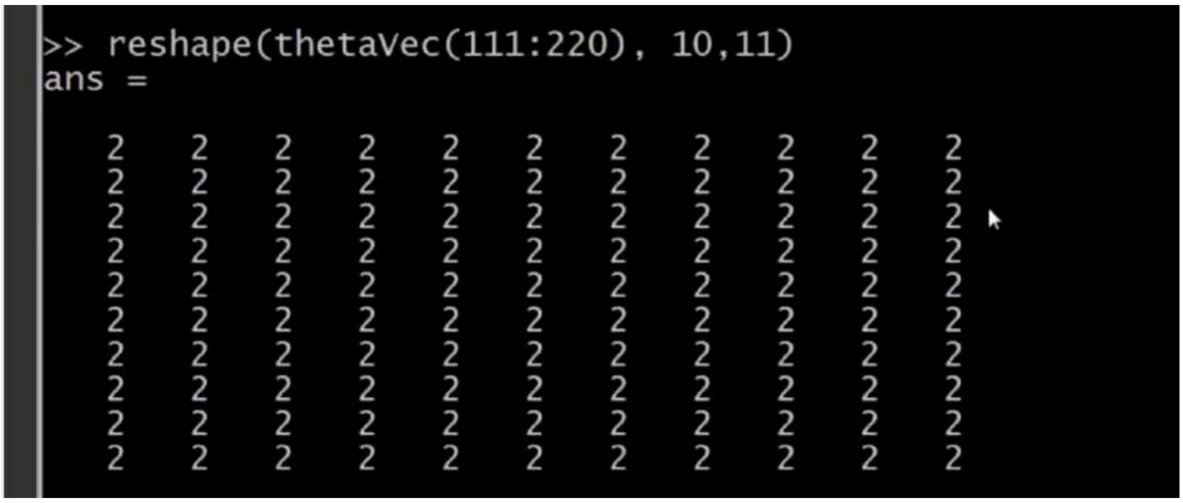


```
>> Theta3 = 3*ones(1,11)
Theta3 =

   3   3   3   3   3   3   3   3   3   3   3

>> thetaVec = [ Theta1(:); Theta2(:); Theta3(:) ];
>> size(thetaVec)
ans =

   231     1
```

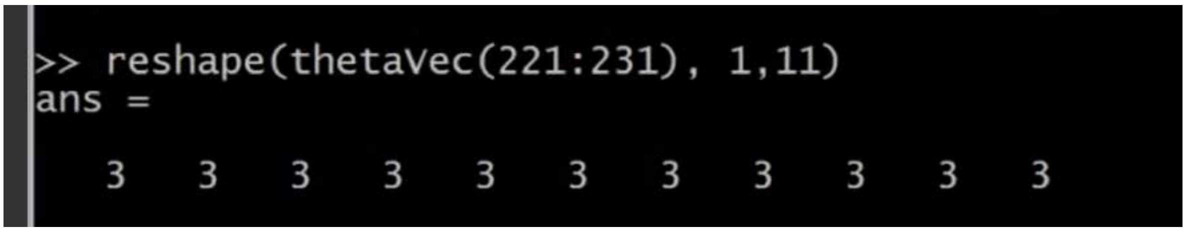
And if I want to get back my original matrices, I can do reshape thetaVec. Let's pull out the first 110 elements and reshape them to a 10 by 11 matrix. This gives me back theta 1. And if I then pull out the next 110 elements. So that's indices 111 to 220. I get back all of my 2's. And if I go from 221 up to the last element, which is element 231, and reshape to 1 by 11, I get back theta 3.



```
>>
>> reshape(thetaVec(1:110), 10,11)
ans =

   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
   1   1   1   1   1   1   1   1   1   1   1
```

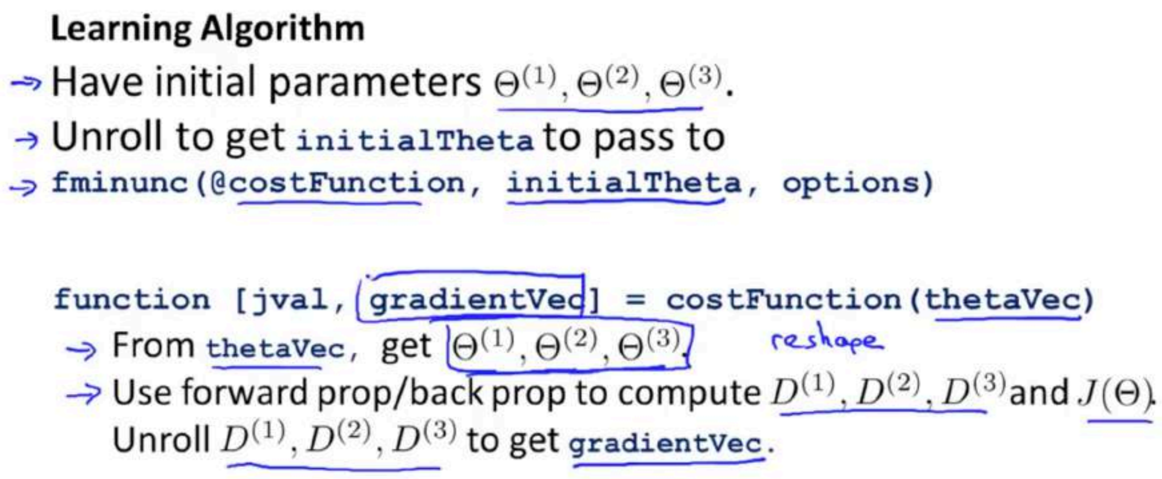


```
>> reshape(thetaVec(111:220), 10,11)
ans =

   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
   2   2   2   2   2   2   2   2   2   2   2
```



```
>> reshape(thetaVec(221:231), 1,11)
ans =

   3   3   3   3   3   3   3   3   3   3   3
```

To make this process really concrete, here's how we use the unrolling idea to implement our learning algorithm. Let's say that you have some initial value of the parameters theta 1, theta 2, theta 3. What we're going to do is take these and unroll them into a long vector we're gonna call initial theta to pass in to fminunc as this initial setting of the parameters theta. The other thing we need to do is implement the cost function. Here's my implementation of the cost function. The cost function is going to give us input, thetaVec, which is going to be all of my parameters vectors that in the form that's been unrolled into a vector. So the first thing I'm going to do is I'm going to use thetaVec and I'm going to use the reshape functions. So I'll pull out elements from thetaVec and use reshape to get back my original parameter matrices, theta 1, theta 2, theta 3. So these are going to be matrices that I'm going to get. So that gives me a more convenient form in which to use these matrices so that I can run forward propagation and back propagation to compute my derivatives, and to compute my cost function j of theta. And finally, I can then take my derivatives and unroll them, to keeping the elements in the same ordering as I did when I unroll my thetas. But I'm gonna unroll D1, D2, D3, to get gradientVec which is now what my cost function can return. It can return a vector of these derivatives.



Learning Algorithm

- Have initial parameters $\underline{\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}}$.
- Unroll to get `initialTheta` to pass to
- `fminunc(@costFunction, initialTheta, options)`

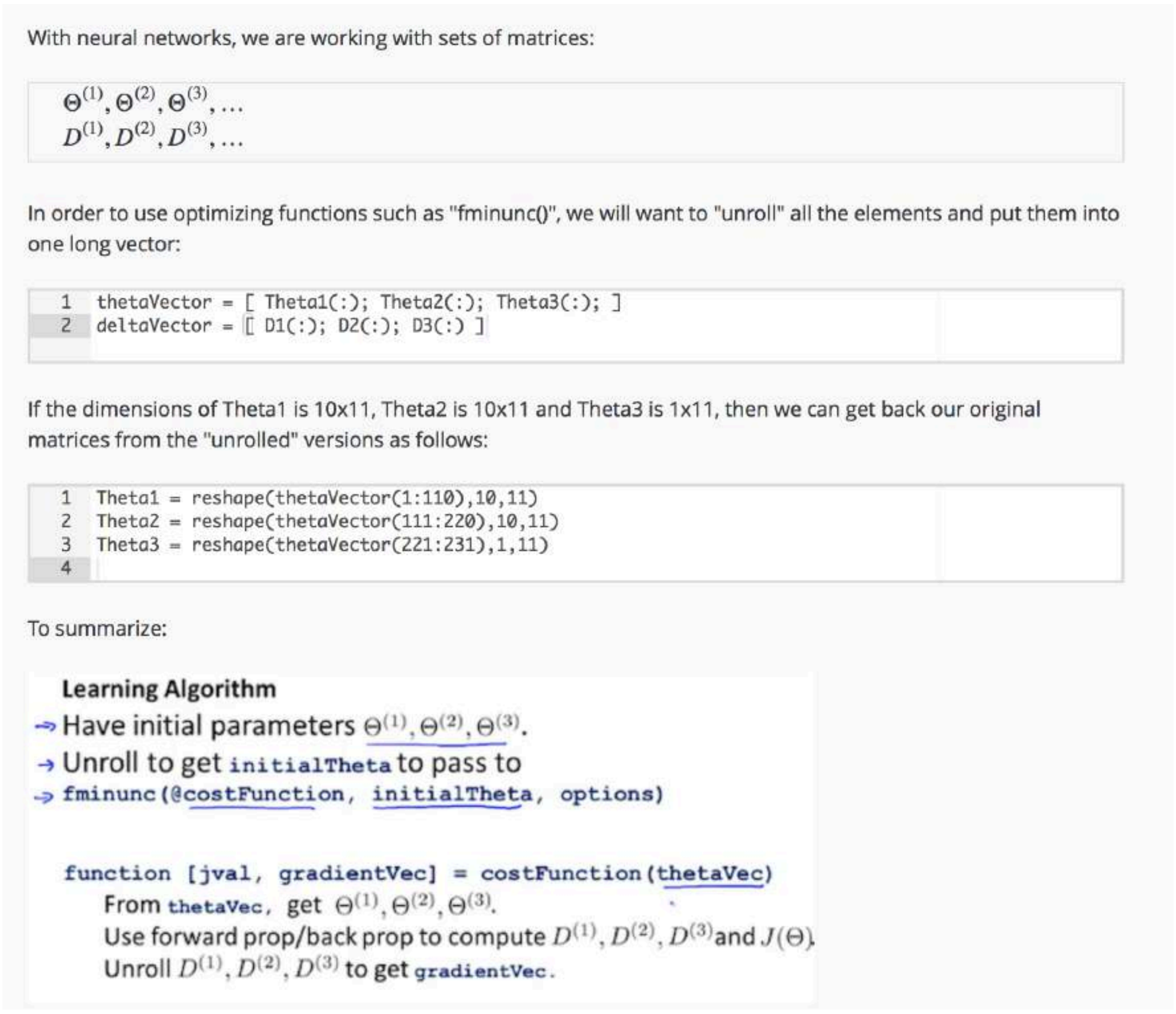
```
function [jval, gradientVec] = costFunction(thetaVec)
```

- From `thetaVec`, get $\boxed{\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}}$ reshape
- Use forward prop/back prop to compute $\underline{D^{(1)}, D^{(2)}, D^{(3)}}$ and $\underline{J(\Theta)}$.

  Unroll $\underline{D^{(1)}, D^{(2)}, D^{(3)}}$ to get `gradientVec`.

So, hopefully, you now have a good sense of how to convert back and forth between the matrix representation of the parameters versus the vector representation of the parameters. The advantage of the matrix representation is that when your parameters are stored as matrices it's more convenient when you're doing forward propagation and back propagation and it's easier when your parameters are stored as matrices to take advantage of the, sort of, vectorized implementations. Whereas in contrast the advantage of the vector representation, when you have like thetaVec or DVec is that when you are using the advanced optimization algorithms. Those algorithms tend to assume that you have all of your parameters unrolled into a big long vector. And so with what we just went through, hopefully you can now quickly convert between the two as needed.

Implementation Note: Unrolling Parameters (Transcript)



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

```
thetaVector = [ Theta1(:); Theta2(:); Theta3(:); ]
deltaVector = [ D1(:); D2(:); D3(:) ]
```

If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11, then we can get back our original matrices from the "unrolled" versions as follows:

```
Theta1 = reshape(thetaVector(1:110),10,11)
Theta2 = reshape(thetaVector(111:220),10,11)
Theta3 = reshape(thetaVector(221:231),1,11)
```

To summarize:

Learning Algorithm

→ Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

→ Unroll to get `initialTheta` to pass to

→ `fminunc(@costFunction, initialTheta, options)`

```
function [jval, gradientVec] = costFunction(thetaVec)
```

From `thetaVec`, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

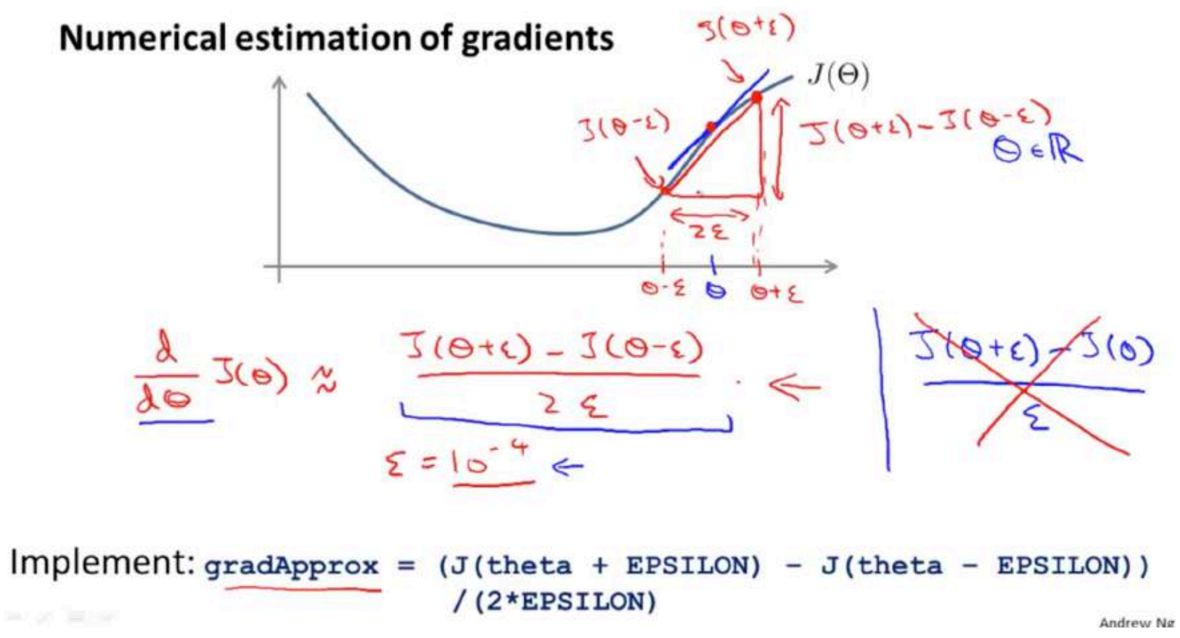
Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$.

Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get `gradientVec`.

Gradient Checking (Video)

In the last few videos we talked about how to do forward propagation and back propagation in a neural network in order to compute derivatives. But back prop as an algorithm has a lot of details and can be a little bit tricky to implement. And one unfortunate property is that there are many ways to have subtle bugs in back prop. So that if you run it with gradient descent or some other optimizational algorithm, it could actually look like it's working. And your cost function, J of theta may end up decreasing on every iteration of gradient descent. But this could prove true even though there might be some bug in your implementation of back prop. So that it looks J of theta is decreasing, but you might just wind up with a neural network that has a higher level of error than you would with a bug free implementation. And you might just not know that there was this subtle bug that was giving you worse performance. So, what can we do about this? There's an idea called gradient checking that eliminates almost all of these problems. So, today every time I implement back propagation or a similar gradient to a [INAUDIBLE] on a neural network or any other reasonably complex model, I always implement gradient checking. And if you do this, it will help you make sure and sort of gain high confidence that your implementation of four prop and back prop or whatever is 100% correct. And from what I've seen this pretty much eliminates all the problems associated with a sort of a buggy implementation as a back prop. And in the previous videos I asked you to take on faith that the formulas I gave for computing the deltas and the vs and so on, I asked you to take on faith that those actually do compute the gradients of the cost function. But once you implement numerical gradient checking, which is the topic of this video, you'll be able to absolute verify for yourself that the code you're writing does indeed, is indeed computing the derivative of the cross function J.

So here's the idea, consider the following example. Suppose that I have the function J of theta and I have some value theta and for this example gonna assume that theta is just a real number. And let's say that I want to estimate the derivative of this function at this point and so the derivative is equal to the slope of that tangent one. Here's how I'm going to numerically approximate the derivative, or rather here's a procedure for numerically approximating the derivative. I'm going to compute theta plus epsilon, so now we move it to the right. And I'm gonna compute theta minus epsilon and I'm going to look at those two points, And connect them by a straight line And I'm gonna connect these two points by a straight line, and I'm gonna use the slope of that little red line as my approximation to the derivative. Which is, the true derivative is the slope of that blue line over there. So, you know it seems like it would be a pretty good approximation. Mathematically, the slope of this red line is this vertical height divided by this horizontal width. So this point on top is the J of (Theta plus Epsilon). This point here is J (Theta minus Epsilon), so this vertical difference is J (Theta plus Epsilon) minus J of theta minus epsilon and this horizontal distance is just 2 epsilon. So my approximation is going to be that the derivative respect of theta of J of theta at this value of theta, that that's approximately J of theta plus epsilon minus J of theta minus epsilon over 2 epsilon. Usually, I use a pretty small value for epsilon, expect epsilon to be maybe on the order of 10 to the minus 4. There's usually a large range of different values for epsilon that work just fine. And in fact, if you let epsilon become really small, then mathematically this term here, actually mathematically, it becomes the derivative. It becomes exactly the slope of the function at this point. It's just that we don't want to use epsilon that's too, too small, because then you might run into numerical problems. So I usually use epsilon around ten to the minus four. And by the way some of you may have seen an alternative formula for s meeting the derivative which is this formula. This one on the right is called a one-sided difference, whereas the formula on the left, that's called a two-sided difference. The two sided difference gives us a slightly more accurate estimate, so I usually use that, rather than this one sided difference estimate. So, concretely, when you implement an octave, is you implemented the following, you implement call to compute gradApprox, which is going to be our approximation derivative as just here this formula, J of theta plus epsilon minus J of theta minus epsilon divided by 2 times epsilon. And this will give you a numerical estimate of the gradient at that point. And in this example it seems like it's a pretty good estimate.



Numerical estimation of gradients

$J(\Theta)$
$J(\theta+\varepsilon)$
$J(\theta-\varepsilon)$
$J(\theta+\varepsilon)-J(\theta-\varepsilon)$
$\theta\in\mathbb{R}$
2ε
$\theta-\varepsilon$ θ $\theta+\varepsilon$

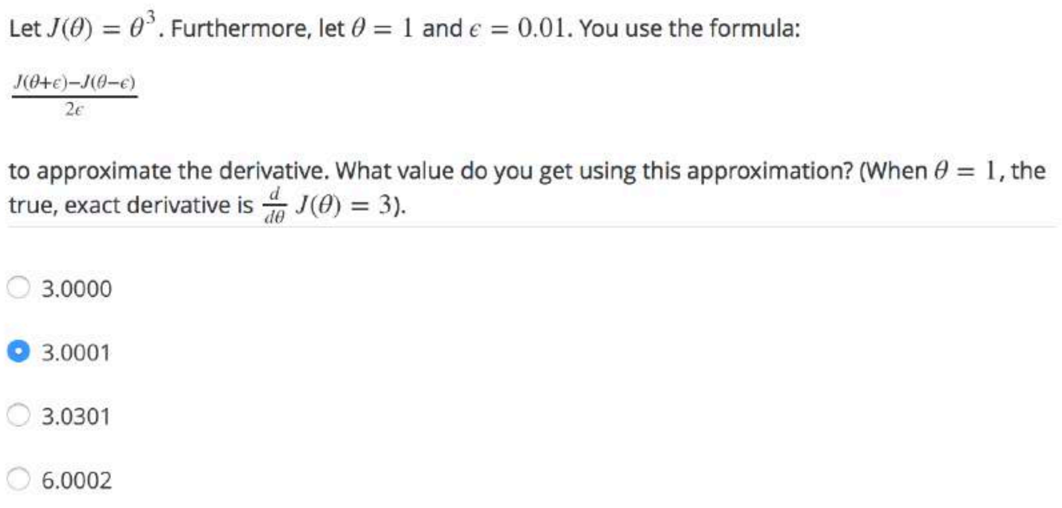
$$\frac{d}{d\theta}J(\theta) \approx \frac{J(\theta+\varepsilon)-J(\theta-\varepsilon)}{2\varepsilon} \leftarrow \qquad \left| \; \cancel{\frac{J(\theta+\varepsilon)-J(\theta)}{\varepsilon}} \right.$$

$\varepsilon = 10^{-4} \leftarrow$

Implement: `gradApprox = (J(theta + EPSILON) - J(theta - EPSILON))`
`/(2*EPSILON)`

Andrew Ng

Question



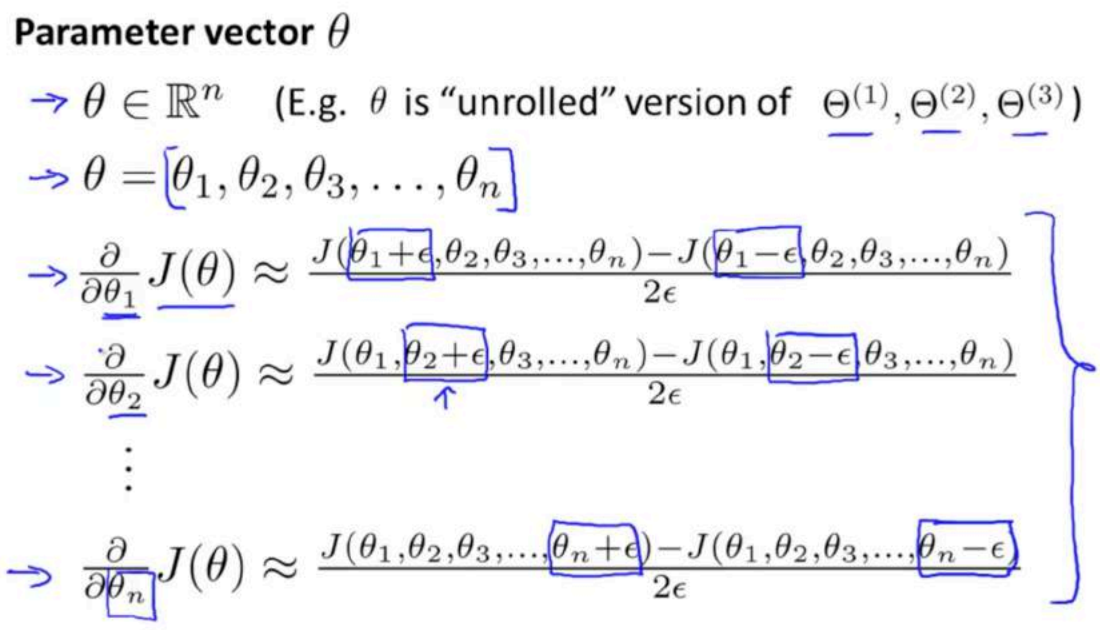
Let $J(\theta)=\theta^3$. Furthermore, let $\theta=1$ and $\epsilon=0.01$. You use the formula:

$$\frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}$$

to approximate the derivative. What value do you get using this approximation? (When $\theta=1$, the true, exact derivative is $\frac{d}{d\theta}J(\theta)=3$).

- ○ 3.0000
- [✓ 3.0001]
- ○ 3.0301
- ○ 6.0002

Now on the previous slide, we considered the case of when theta was a rolled number. Now let's look at a more general case of when theta is a vector parameter, so let's say theta is an R n. And it might be an unrolled version of the parameters of our neural network. So theta is a vector that has n elements, theta 1 up to theta n. We can then use a similar idea to approximate all the partial derivative terms. Concretely the partial derivative of a cost function with respect to the first parameter, theta one, that can be obtained by taking J and increasing theta one. So you have J of theta one plus epsilon and so on. Minus J of this theta one minus epsilon and divide it by two epsilon. The partial derivative respect to the second parameter theta two, is again this thing except that you would take J of here you're increasing theta two by epsilon, and here you're decreasing theta two by epsilon and so on down to the derivative. With respect of theta n would give you increase and decrease theta and by epsilon over there. So, these equations give you a way to numerically approximate the partial derivative of J with respect to any one of your parameters theta i.



Parameter vector θ

$\rightarrow \theta \in \mathbb{R}^n$ (E.g. θ is "unrolled" version of $\underline{\Theta^{(1)}}, \underline{\Theta^{(2)}}, \underline{\Theta^{(3)}}$)

$\rightarrow \theta = [\theta_1, \theta_2, \theta_3, \ldots, \theta_n]$

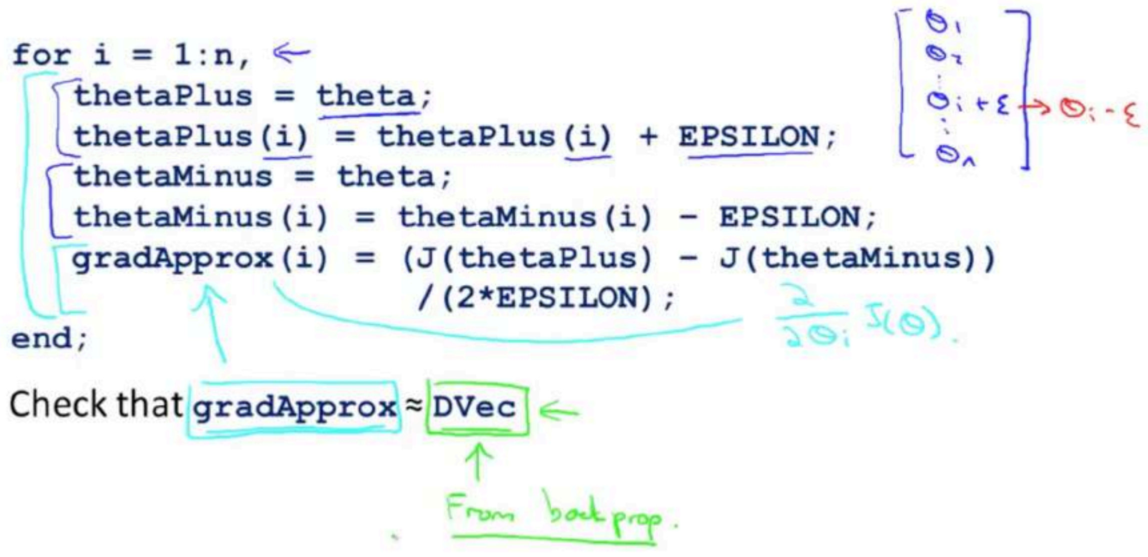
$$\rightarrow \frac{\partial}{\partial \underline{\theta_1}} \underline{J(\theta)} \approx \frac{J(\boxed{\theta_1+\epsilon}, \theta_2, \theta_3, \ldots, \theta_n) - J(\boxed{\theta_1-\epsilon}, \theta_2, \theta_3, \ldots, \theta_n)}{2\epsilon}$$

$$\rightarrow \frac{\partial}{\partial \underline{\theta_2}} J(\theta) \approx \frac{J(\theta_1, \underset{\uparrow}{\boxed{\theta_2+\epsilon}}, \theta_3, \ldots, \theta_n) - J(\theta_1, \boxed{\theta_2-\epsilon}, \theta_3, \ldots, \theta_n)}{2\epsilon}$$

$$\vdots$$

$$\rightarrow \frac{\partial}{\partial \boxed{\theta_n}} J(\theta) \approx \frac{J(\theta_1, \theta_2, \theta_3, \ldots, \boxed{\theta_n+\epsilon}) - J(\theta_1, \theta_2, \theta_3, \ldots, \boxed{\theta_n-\epsilon})}{2\epsilon}$$

Completely, what you implement is therefore the following. We implement the following in octave to numerically compute the derivatives. We say, for i = 1:n, where n is the dimension of our parameter of vector theta. And I usually do this with the unrolled version of the parameter. So theta is just a long list of all of my parameters in my neural network, say. I'm gonna set thetaPlus = theta, then increase thetaPlus of the (i) element by epsilon. And so this is basically thetaPlus is equal to theta except for thetaPlus(i) which is now incremented by epsilon. Epsilon, so theta plus is equal to, write theta 1, theta 2 and so on. Then theta I has epsilon added to it and then we go down to theta N. So this is what theta plus is. And similar these two lines set theta minus to something similar except that this instead of theta I plus Epsilon, this now becomes theta I minus Epsilon. And then finally you implement this gradApprox (i) and this would give you your approximation to the partial derivative respect of theta i of J of theta. And the way we use this in our neural network implementation is, we would implement this four loop to compute the top partial derivative of the cost function for respect to every parameter in that network, and we can then take the gradient that we got from backprop. So DVec was the derivative we got from backprop. All right, so backprop, backpropogation, was a relatively efficient way to compute a derivative or a partial derivative. Of a cost function with respect to all our parameters. And what I usually do is then, take my numerically computed derivative that is this gradApprox that we just had from up here. And make sure that that is equal or approximately equal up to small values of numerical round up, that it's pretty close. So the DVec that I got from backprop. And if these two ways of computing the derivative give me the same answer, or give me any similar answers, up to a few decimal places, then I'm much more confident that my implementation of backprop is correct. And when I plug these DVec vectors into gradient assent or some advanced optimization algorithm, I can then be much more confident that I'm computing the derivatives correctly, and therefore that hopefully my code will run correctly and do a good job optimizing J of theta.

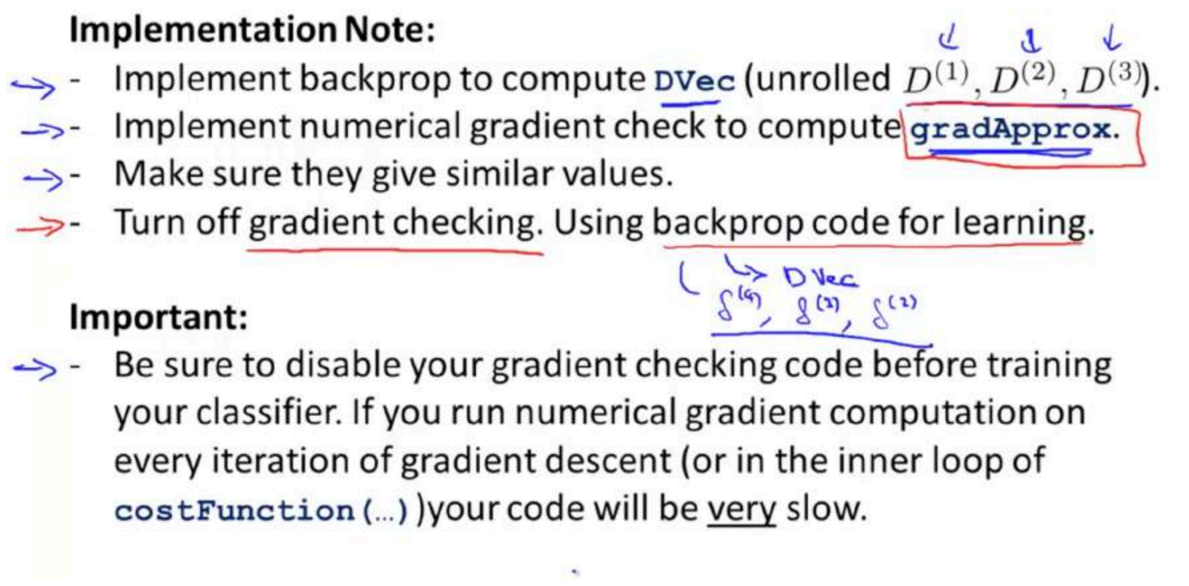


```
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) = thetaPlus(i) + EPSILON;
  thetaMinus = theta;
  thetaMinus(i) = thetaMinus(i) - EPSILON;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))
                  /(2*EPSILON);
end;
```

Check that gradApprox ≈ DVec

Finally, I wanna put everything together and tell you how to implement this numerical gradient checking. Here's what I usually do. First thing I do is implement back propagation to compute DVec. So there's a procedure we talked about in the earlier video to compute DVec which may be our unrolled version of these matrices. So then what I do, is implement a numerical gradient checking to compute gradApprox. So this is what I described earlier in this video and in the previous slide. Then should make sure that DVec and gradApprox give similar values, you know let's say up to a few decimal places.

And finally and this is the important step, before you start to use your code for learning, for seriously training your network, it's important to turn off gradient checking and to no longer compute this gradApprox thing using the numerical derivative formulas that we talked about earlier in this video. And the reason for that is the numeric code gradient checking code, the stuff we talked about in this video, that's a very computationally expensive, that's a very slow way to try to approximate the derivative. Whereas In contrast, the back propagation algorithm that we talked about earlier, that is the thing we talked about earlier for computing. You know, D1, D2, D3 for Dvec. Backprop is much more computationally efficient way of computing for derivatives. So once you've verified that your implementation of back propagation is correct, you should turn off gradient checking and just stop using that. So just to reiterate, you should be sure to disable your gradient checking code before running your algorithm for many iterations of gradient descent or for many iterations of the advanced optimization algorithms, in order to train your classifier. Concretely, if you were to run the numerical gradient checking on every single iteration of gradient descent. Or if you were in the inner loop of your costFunction, then your code would be very slow. Because the numerical gradient checking code is much slower than the backpropagation algorithm, than the backpropagation method where, you remember, we were computing delta(4), delta(3), delta(2), and so on. That was the backpropagation algorithm. That is a much faster way to compute derivates than gradient checking. So when you're ready, once you've verified the implementation of back propagation is correct, make sure you turn off or you disable your gradient checking code while you train your algorithm, or else you code could run very slowly.



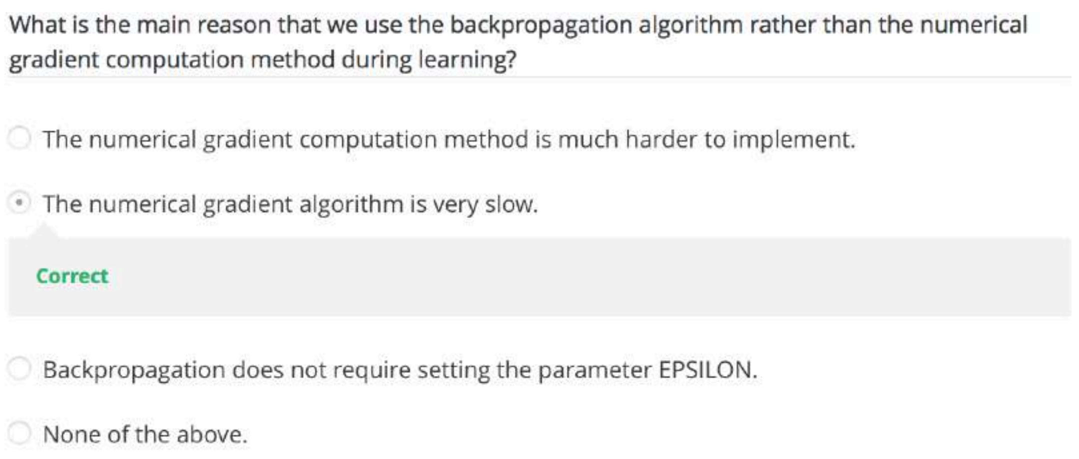
Implementation Note:

- Implement backprop to compute `DVec` (unrolled $D^{(1)}, D^{(2)}, D^{(3)}$).
- Implement numerical gradient check to compute `gradApprox`.
- Make sure they give similar values.
- Turn off gradient checking. Using backprop code for learning.

Important:

- Be sure to disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent (or in the inner loop of `costFunction(…)`)your code will be very slow.

Question


What is the main reason that we use the backpropagation algorithm rather than the numerical gradient computation method during learning?

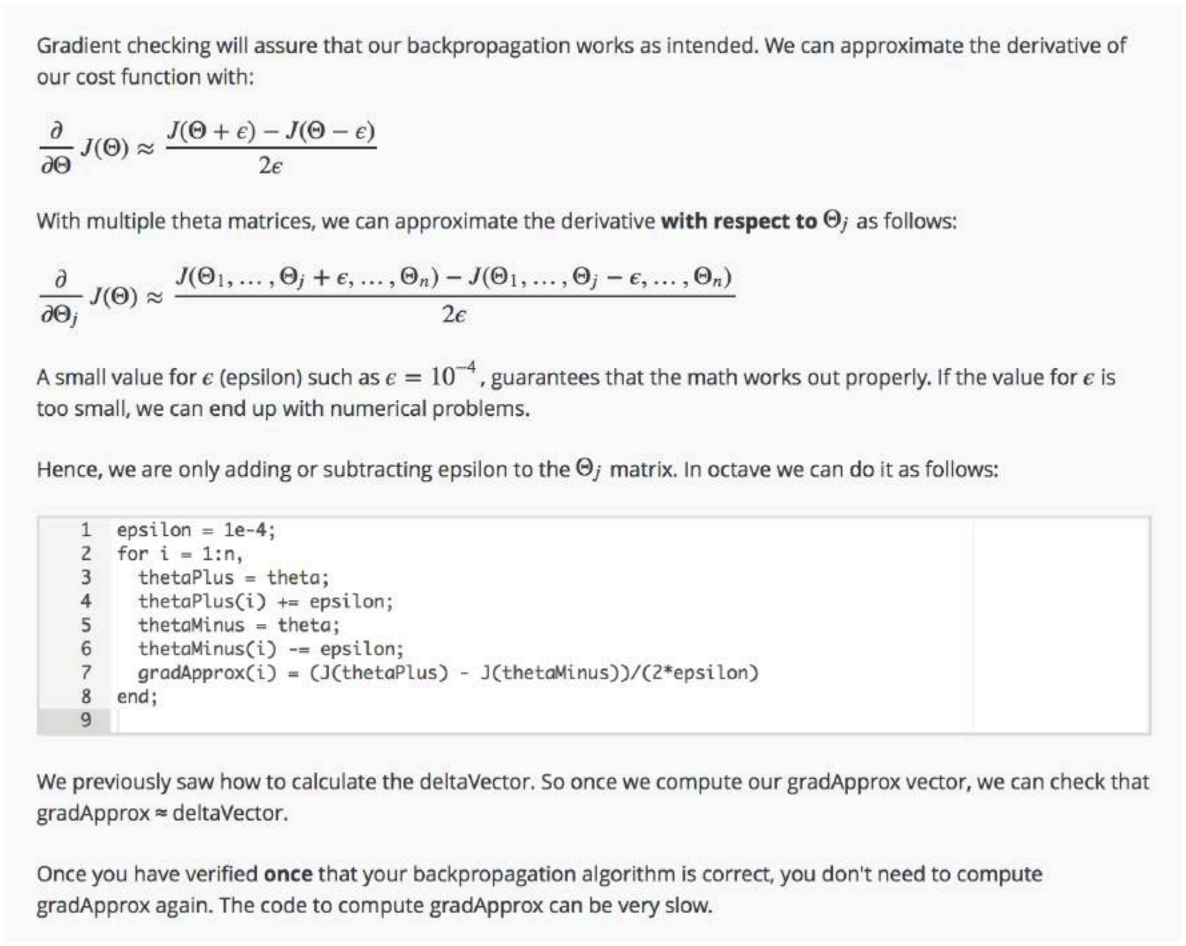
- ○ The numerical gradient computation method is much harder to implement.
- ● The numerical gradient algorithm is very slow.

Correct

- ○ Backpropagation does not require setting the parameter EPSILON.
- ○ None of the above.

So, that's how you take gradients numerically, and that's how you can verify the implementation of back propagation is correct. Whenever I implement back propagation or similar gradient discerning algorithm for a complicated mode, I always use gradient checking and this really helps me make sure that my code is correct.

Gradient Checking (Transcript)



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 octave we can do it as follows:

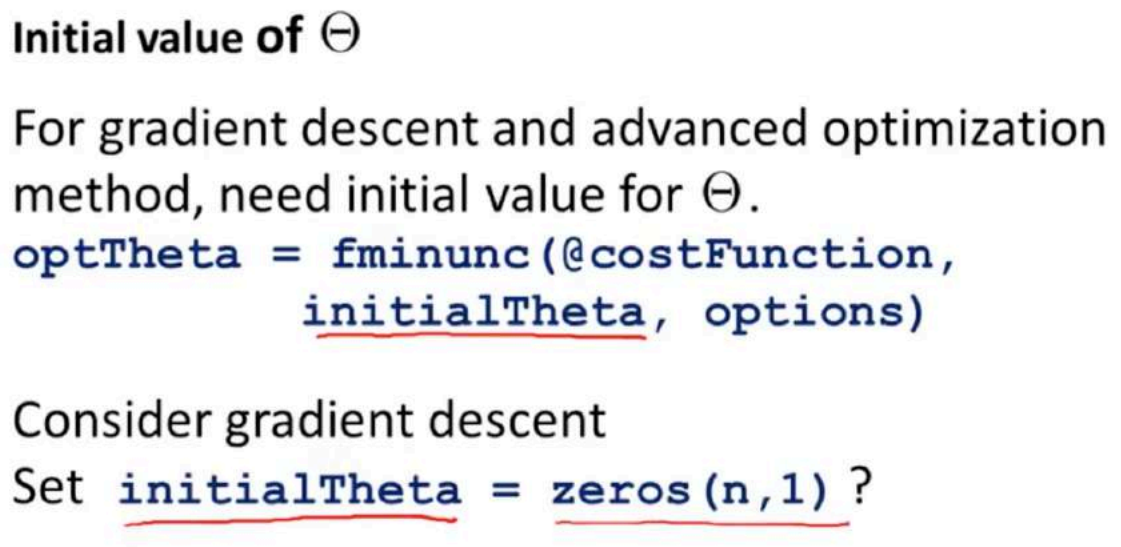
```
epsilon = 1e-4;
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) += epsilon;
  thetaMinus = theta;
  thetaMinus(i) -= epsilon;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
end;

```

We previously saw how to calculate the deltaVector. So once we compute our gradApprox vector, we can check that gradApprox ≈ deltaVector.

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

Random Initialization (Video)

In the previous video, we've put together almost all the pieces you need in order to implement and train in your network. There's just one last idea I need to share with you, which is the idea of random initialization. When you're running an algorithm of gradient descent, or also the advanced optimization algorithms, we need to pick some initial value for the parameters theta. So for the advanced optimization algorithm, it assumes you will pass it some initial value for the parameters theta. Now let's consider a gradient descent. For that, we'll also need to initialize theta to something, and then we can slowly take steps to go downhill using gradient descent. To go downhill, to minimize the function j of theta. So what can we set the initial value of theta to? Is it possible to set the initial value of theta to the vector of all zeros? Whereas this worked okay when we were using logistic regression, initializing all of your parameters to zero actually does not work when you are trading on your own network.



Initial value of Θ

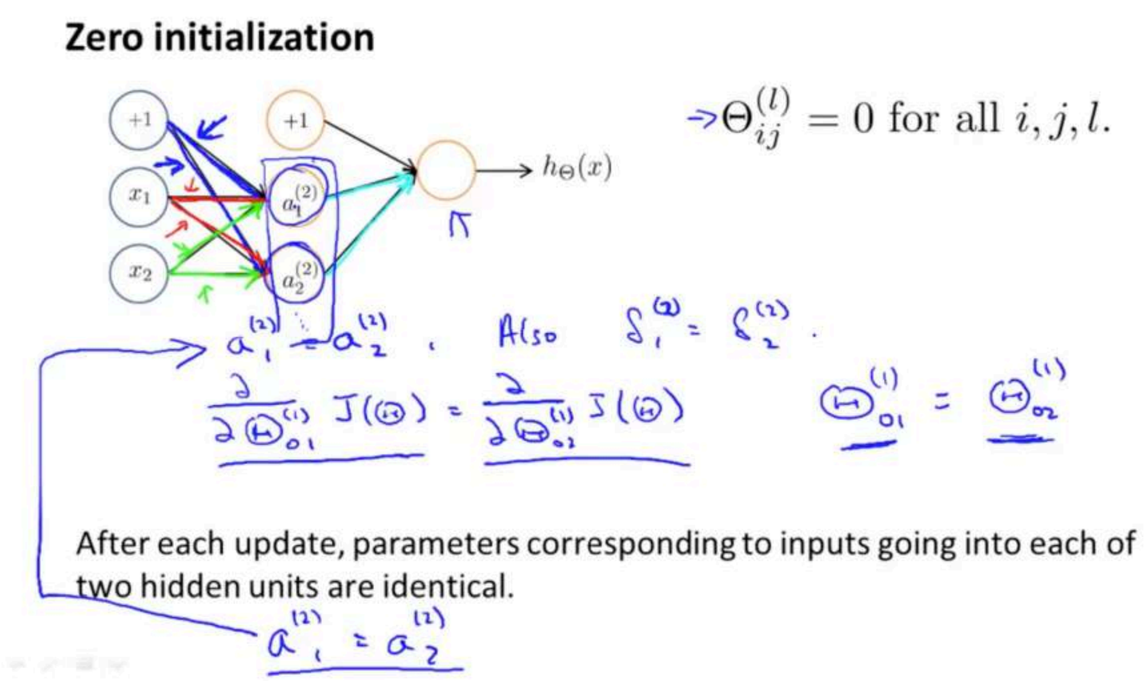
For gradient descent and advanced optimization method, need initial value for Θ.

```
optTheta = fminunc(@costFunction,
           initialTheta, options)
```

Consider gradient descent

Set `initialTheta = zeros(n,1)` ?

Consider trading the follow Neural network, and let's say we initialize all the parameters of the network to 0. And if you do that, then what you, what that means is that at the initialization, this blue weight, colored in blue is gonna equal to that weight, so they're both 0. And this weight that I'm coloring in in red, is equal to that weight, colored in red, and also this weight, which I'm coloring in green is going to equal to the value of that weight. And what that means is that both of your hidden units, A1 and A2, are going to be computing the same function of your inputs. And thus you end up with for every one of your training examples, you end up with A 2 1 equals A 2 2. And moreover because I'm not going to show this in too much detail, but because these outgoing weights are the same you can also show that the delta values are also gonna be the same. So concretely you end up with delta 1 1, delta 2 1 equals delta 2 2, and if you work through the map further, what you can show is that the partial derivatives with respect to your parameters will satisfy the following, that the partial derivative of the cost function with respected to breaking out the derivatives respect to these two blue waves in your network. You find that these two partial derivatives are going to be equal to each other. And so what this means is that even after say one greater descent update, you're going to update, say, this first blue rate was learning rate times this, and you're gonna update the second blue rate with some learning rate times this. And what this means is that even after one created the descent update, those two blue rates, those two blue color parameters will end up the same as each other. So there'll be some nonzero value, but this value would equal to that value. And similarly, even after one gradient descent update, this value would equal to that value. There'll still be some non-zero values, just that the two red values are equal to each other. And similarly, the two green ways. Well, they'll both change values, but they'll both end up with the same value as each other. So after each update, the parameters corresponding to the inputs going into each of the two hidden units are identical. That's just saying that the two green weights are still the same, the two red weights are still the same, the two blue weights are still the same, and what that means is that even after one iteration of say, gradient descent and descent. You find that your two headed units are still computing exactly the same functions of the inputs. You still have the a1(2) = a2(2). And so you're back to this case. And as you keep running greater descent, the blue waves,, the two blue waves, will stay the same as each other. The two red waves will stay the same as each other and the two green waves will stay the same as each other. And what this means is that your neural network really can compute very interesting functions, right? Imagine that you had not only two hidden units, but imagine that you had many, many hidden units. Then what this is saying is that all of your headed units are computing the exact same feature. All of your hidden units are computing the exact same function of the input. And this is a highly redundant representation because you find the logistic progression unit. It really has to see only one feature because all of these are the same. And this prevents you and your network from doing something interesting.



Zero initialization

$\Theta_{ij}^{(l)} = 0$ for all i, j, l.

$a_1^{(2)} = a_2^{(2)}$. Also $\delta_1^{(2)} = \delta_2^{(2)}$.

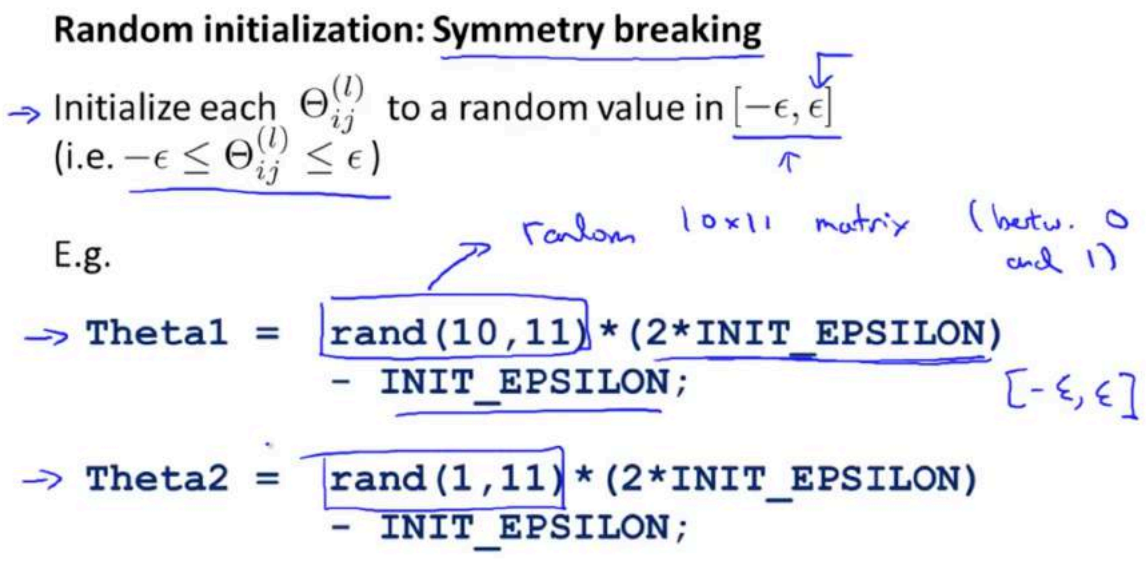
$\frac{\partial}{\partial \Theta_{01}^{(1)}} J(\Theta) = \frac{\partial}{\partial \Theta_{02}^{(1)}} J(\Theta)$ $\quad \Theta_{01}^{(1)} = \Theta_{02}^{(1)}$

After each update, parameters corresponding to inputs going into each of two hidden units are identical.

$a_1^{(2)} = a_2^{(2)}$

In order to get around this problem, the way we initialize the parameters of a neural network therefore is with random initialization. Concretely, the problem was saw on the previous slide is something called the problem of symmetric ways, that's the ways are being the same. So this random initialization is how we perform symmetry breaking. So what we do is we initialize each value of theta to a random number between minus epsilon and epsilon. So this is a notation to b numbers between minus epsilon and plus epsilon. So my weight for my parameters are all going to be randomly initialized between minus epsilon and plus epsilon. The way I write code to do this in octave is I've said

Theta1 should be equal to this. So this rand 10 by 11, that's how you compute a random 10 by 11 dimensional matrix. All the values are between 0 and 1, so these are going to be raw numbers that take on any continuous values between 0 and 1. And so if you take a number between zero and one, multiply it by two times INIT_EPSILON then minus INIT_EPSILON, then you end up with a number that's between minus epsilon and plus epsilon. And the so that leads us, this epsilon here has nothing to do with the epsilon that we were using when we were doing gradient checking. So when numerical gradient checking, there we were adding some values of epsilon and theta. This is your unrelated value of epsilon. We just wanted to notate init epsilon just to distinguish it from the value of epsilon we were using in gradient checking. And similarly if you want to initialize theta2 to a random 1 by 11 matrix you can do so using this piece of code here.



Random initialization: Symmetry breaking

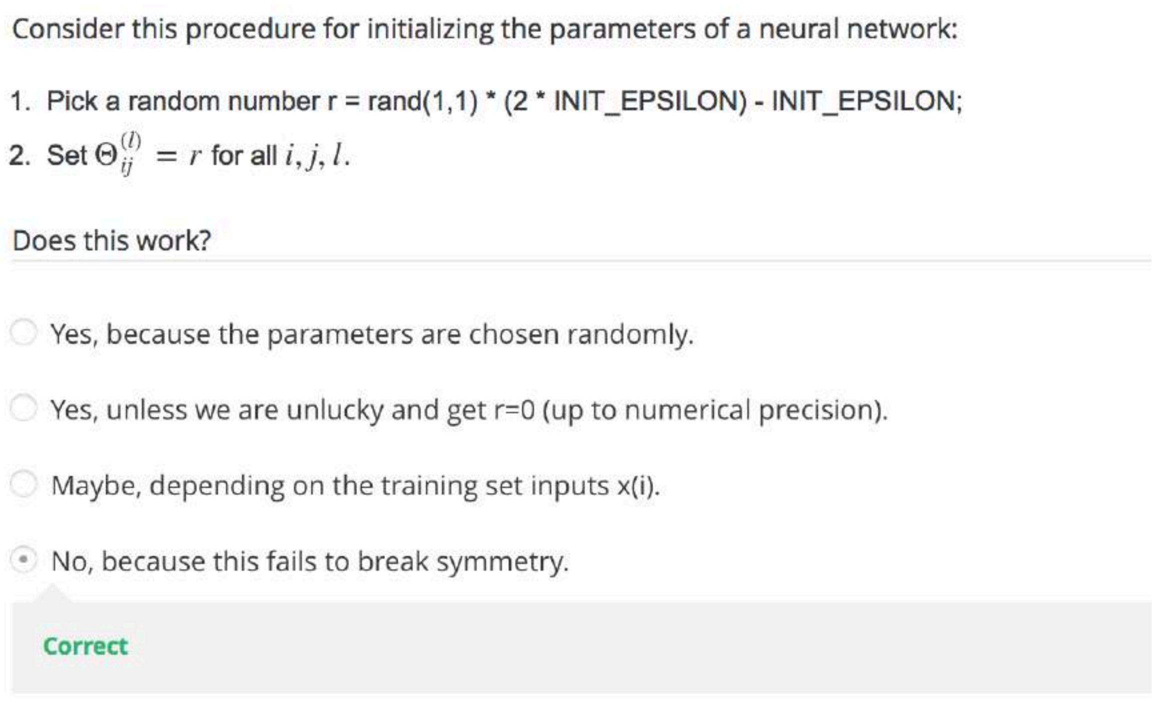
Initialize each $\Theta_{ij}^{(l)}$ to a random value in $[-\epsilon, \epsilon]$
(i.e. $-\epsilon \le \Theta_{ij}^{(l)} \le \epsilon$)

E.g.

```
Theta1 =  rand(10,11)*(2*INIT_EPSILON)
          - INIT_EPSILON;

Theta2 =  rand(1,11)*(2*INIT_EPSILON)
          - INIT_EPSILON;
```

random 10×11 matrix (betw. 0 and 1)
$[-\epsilon, \epsilon]$

Question


Consider this procedure for initializing the parameters of a neural network:

1. Pick a random number r = rand(1,1) * (2 * INIT_EPSILON) - INIT_EPSILON;
2. Set $\Theta_{ij}^{(l)} = r$ for all i, j, l.

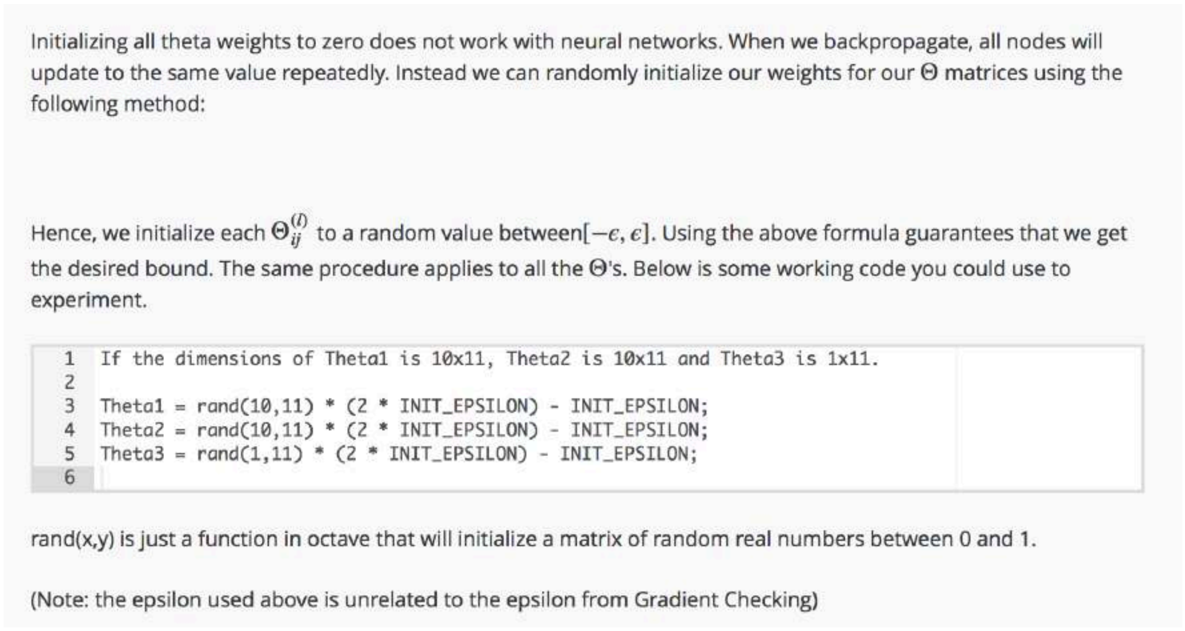
Does this work?

- ○ Yes, because the parameters are chosen randomly.
- ○ Yes, unless we are unlucky and get r=0 (up to numerical precision).
- ○ Maybe, depending on the training set inputs x(i).
- [✓ No, because this fails to break symmetry.]

Correct

So to summarize, to create a neural network what you should do is randomly initialize the waves to small values close to zero, between -epsilon and +epsilon say. And then implement back propagation, do great in checking, and use either great in descent or 1b advanced optimization algorithms to try to minimize j(theta) as a function of the parameters theta starting from just randomly chosen initial value for the parameters. And by doing symmetry breaking, which is this process, hopefully great gradient descent or the advanced optimization algorithms will be able to find a good value of theta.

Random Initialization (Transcript)



Initializing all theta weights to zero does 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:

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. Below is some working code you could use to experiment.

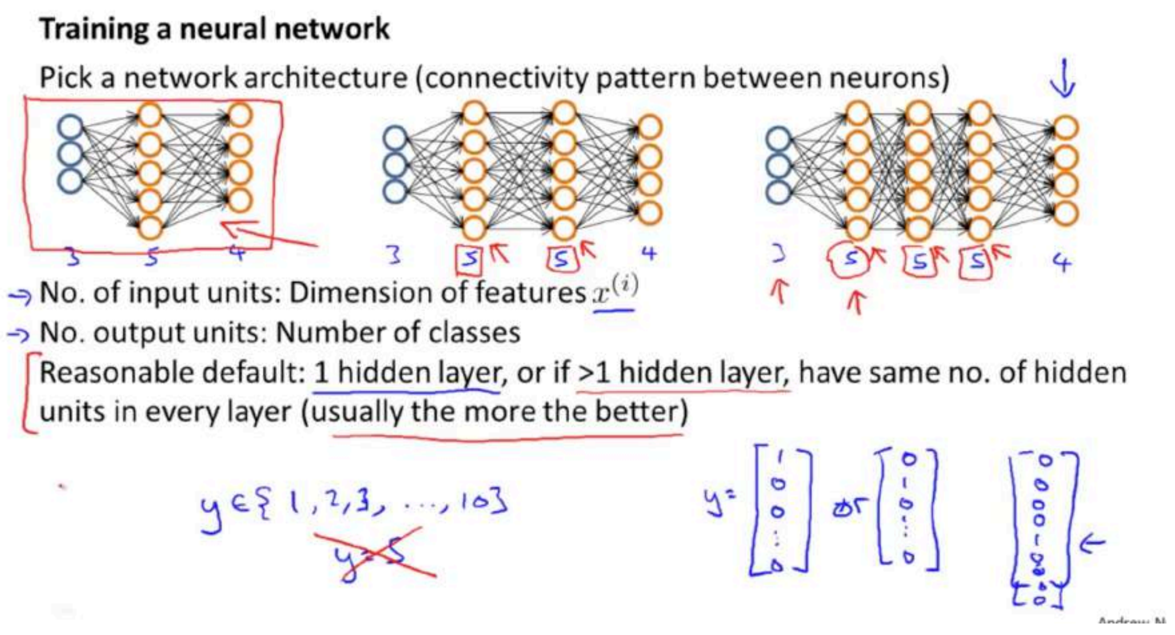
```
If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11.

Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta3 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;

```

rand(x,y) is just a function in octave that will initialize a matrix of random real numbers between 0 and 1.

(Note: the epsilon used above is unrelated to the epsilon from Gradient Checking)

Putting it Together (Video)

So, it's taken us a lot of videos to get through the neural network learning algorithm. In this video, what I'd like to do is try to put all the pieces together, to give a overall summary or a bigger picture view, of how all the pieces fit together and of the overall process of how to implement a neural network learning algorithm. When training a neural network, the first thing you need to do is pick some network architecture and by architecture I just mean connectivity pattern between the neurons. So, you know, we might choose between say, a neural network with three input units and five hidden units and four output units versus one of 3, 5 hidden, 5 hidden, 4 output and here are 3, 5, 5, 5 units in each of three hidden layers and four open units, and so these choices of how many hidden units in each layer and how many hidden layers, those are architecture choices. So, how do you make these choices? Well first, the number of input units well that's pretty well defined. And once you decides on the fix set of features x the number of input units will just be, you know, the dimension of your features x(i) would be determined by that. And if you are doing multiclass classifications the number of output of this will be determined by the number of classes in your classification problem. And just a reminder if you have a multiclass classification where y takes on say values between 1 and 10, so that you have ten possible classes. Then remember to right, your output y as these were the vectors. So instead of clause one, you recode it as a vector like that, or for the second class you recode it as a vector like that. So if one of these apples takes on the fifth class, you know, y equals 5, then what you're showing to your neural network is not actually a value of y equals 5, instead here at the upper layer which would have ten output units, you will instead feed to the vector which you know with one in the fifth position and a bunch of zeros down here. So the choice of number of input units and number of output units is maybe somewhat reasonably straightforward. And as for the number of hidden units and the number of hidden layers, a reasonable default is to use a single hidden layer and so this type of neural network shown on the left with just one hidden layer is probably the most common. Or if you use more than one hidden layer, again the reasonable default will be to have the same number of hidden units in every single layer. So here we have two hidden layers and each of these hidden layers have the same number five of hidden units and here we have, you know, three hidden layers and each of them has the same number, that is five hidden units. Rather than doing this sort of network architecture on the left would be a perfect ably reasonable default. And as for the number of hidden units - usually, the more hidden units the better; it's just that if you have a lot of hidden units, it can become more computationally expensive, but very often, having more hidden units is a good thing. And usually the number of hidden units in each layer will be maybe comparable to the dimension of x, comparable to the number of features, or it could be any where from same number of hidden units of input features to maybe so that three or four times of that. So having the number of hidden units is comparable. You know, several times, or some what bigger than the number of input features is often a useful thing to do So, hopefully this gives you one reasonable set of default choices for neural architecture and and if you follow these guidelines, you will probably get something that works well, but in a later set of videos where I will talk specifically about advice for how to apply algorithms, I will actually say a lot more about how to choose a neural network architecture. Or actually have quite a lot I want to say later to make good choices for the number of hidden units, the number of hidden layers, and so on.



Training a neural network

Pick a network architecture (connectivity pattern between neurons)

→ No. of input units: Dimension of features $x^{(i)}$

→ No. output units: Number of classes

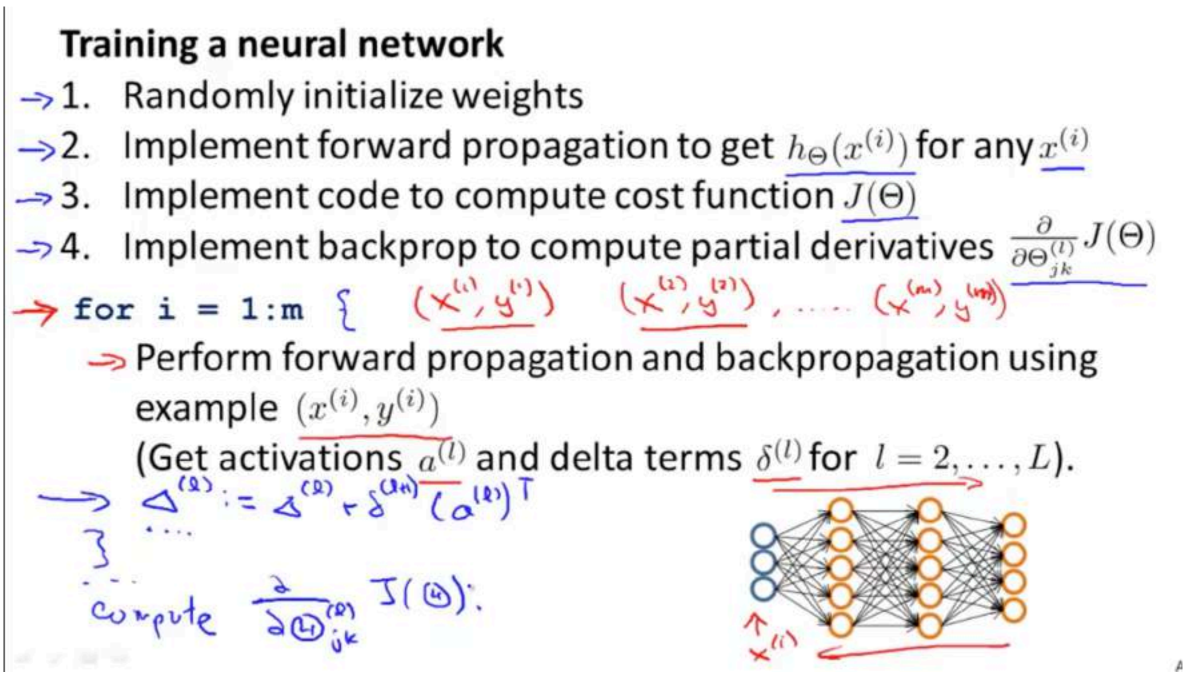
Reasonable default: 1 hidden layer, or if >1 hidden layer, have same no. of hidden units in every layer (usually the more the better)

$y \in \{1, 2, 3, \dots, 10\}$

~~$y = 5$~~

$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ or $\begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ ←

Next, here's what we need to implement in order to trade in neural network, there are actually six steps that I have; I have four on this slide and two more steps on the next slide. First step is to set up the neural network and to randomly initialize the values of the weights. And we usually initialize the weights to small values near zero. Then we implement forward propagation so that we can input any excellent neural network and compute h of x which is this output vector of the y values. We then also implement code to compute this cost function j of theta. And next we implement back-prop, or the back-propagation algorithm, to compute these partial derivatives terms, partial derivatives of j of theta with respect to the parameters. Concretely, to implement back prop. Usually we will do that with a fore loop over the training examples. Some of you may have heard of advanced, and frankly very advanced factorization methods where you don't have a four-loop over the m-training examples, that the first time you're implementing back prop there should almost certainly the four loop in your code, where you're iterating over the examples, you know, x1, y1, then so you do forward prop and back prop on the first example, and then in the second iteration of the four-loop, you do forward propagation and back propagation on the second example, and so on. Until you get through the final example. So there should be a four-loop in your implementation of back prop, at least the first time implementing it. And then there are frankly somewhat complicated ways to do this without a four-loop, but I definitely do not recommend trying to do that much more complicated version the first time you try to implement back prop. So concretely, we have a four-loop over my m-training examples and inside the four-loop we're going to perform fore prop and back prop using just this one example. And what that means is that we're going to take x(i), and feed that to my input layer, perform forward-prop, perform back-prop and that will if all of these activations and all of these delta terms for all of the layers of all my units in the neural network then still inside this four-loop, let me draw some curly braces just to show the scope with the four-loop, this is in octave code of course, but it's more a sequence Java code, and a four-loop encompasses all this. We're going to compute those delta terms, which are is the formula that we gave earlier. Plus, you know, delta l plus one times a, l transpose of the code. And then finally, outside the having computed these delta terms, these accumulation terms, we would then have some other code and then that will allow us to compute these partial derivative terms. Right and these partial derivative terms have to take into account the regularization term lambda as well. And so, those formulas were given in the earlier video. So, how do you done that you now hopefully have code to compute these partial derivative terms.



Training a neural network

→1. Randomly initialize weights

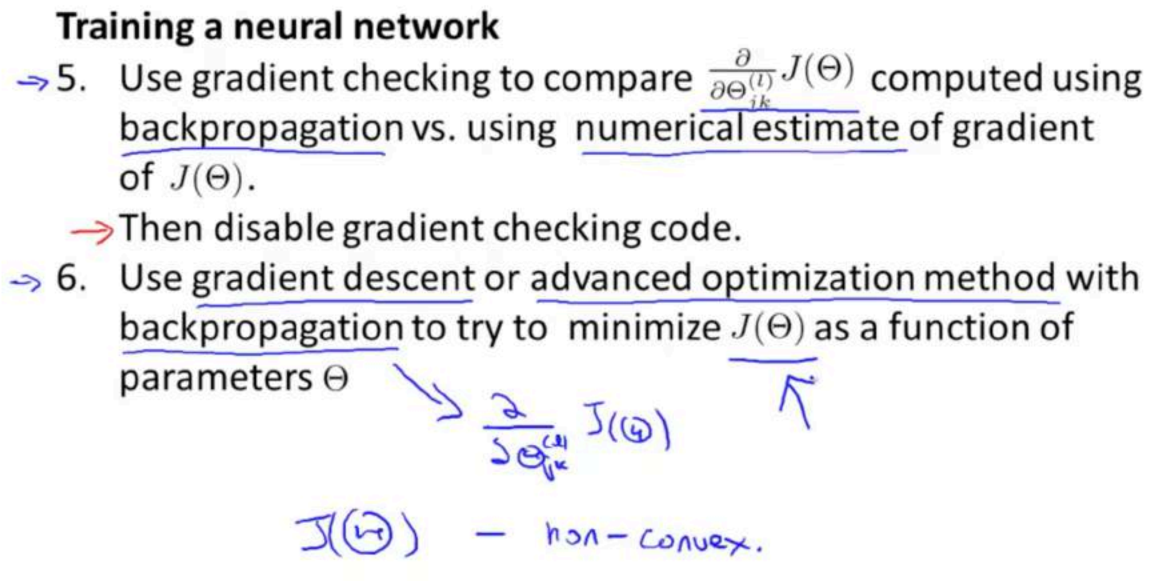
→2. Implement forward propagation to get $h_\Theta(x^{(i)})$ for any $x^{(i)}$

→3. Implement code to compute cost function $J(\Theta)$

→4. Implement backprop to compute partial derivatives $\frac{\partial}{\partial\Theta^{(l)}_{jk}}J(\Theta)$

```
→ for i = 1:m {   $(x^{(1)}, y^{(1)})$   $(x^{(2)}, y^{(2)})$, .....   $(x^{(m)}, y^{(m)})$
      → Perform forward propagation and backpropagation using
        example $(x^{(i)}, y^{(i)})$
        (Get activations $a^{(l)}$ and delta terms $\delta^{(l)}$ for $l = 2, \ldots, L$).
  → $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$
      ....
  }
  ...
  compute $\frac{\partial}{\partial\Theta^{(l)}_{jk}} J(\Theta)$.
```

Next is step five, what I do is then use gradient checking to compare these partial derivative terms that were computed. So, I've compared the versions computed using back propagation versus the partial derivatives computed using the numerical estimates as using numerical estimates of the derivatives. So, I do gradient checking to make sure that both of these give you very similar values. Having done gradient checking just now reassures us that our implementation of back propagation is correct, and is then very important that we disable gradient checking, because the gradient checking code is computationally very slow. And finally, we then use an optimization algorithm such as gradient descent, or one of the advanced optimization methods such as LB of GS, contract gradient has embodied into fminunc or other optimization methods. We use these together with back propagation, so back propagation is the thing that computes these partial derivatives for us. And so, we know how to compute the cost function, we know how to compute the partial derivatives using back propagation, so we can use one of these optimization methods to try to minimize j of theta as a function of the parameters theta. And by the way, for neural networks, this cost function j of theta is non-convex, or is not convex and so it can theoretically be susceptible to local minima, and in fact algorithms like gradient descent and the advance optimization methods can, in theory, get stuck in local optima, but it turns out that in practice this is not usually a huge problem and even though we can't guarantee that these algorithms will find a global optimum, usually algorithms like gradient descent will do a very good job minimizing this cost function j of theta and get a very good local minimum, even if it doesn't get to the global optimum.



Training a neural network

5. Use gradient checking to compare $\frac{\partial}{\partial\Theta_{jk}^{(l)}}J(\Theta)$ computed using backpropagation vs. using numerical estimate of gradient of $J(\Theta)$.

   Then disable gradient checking code.

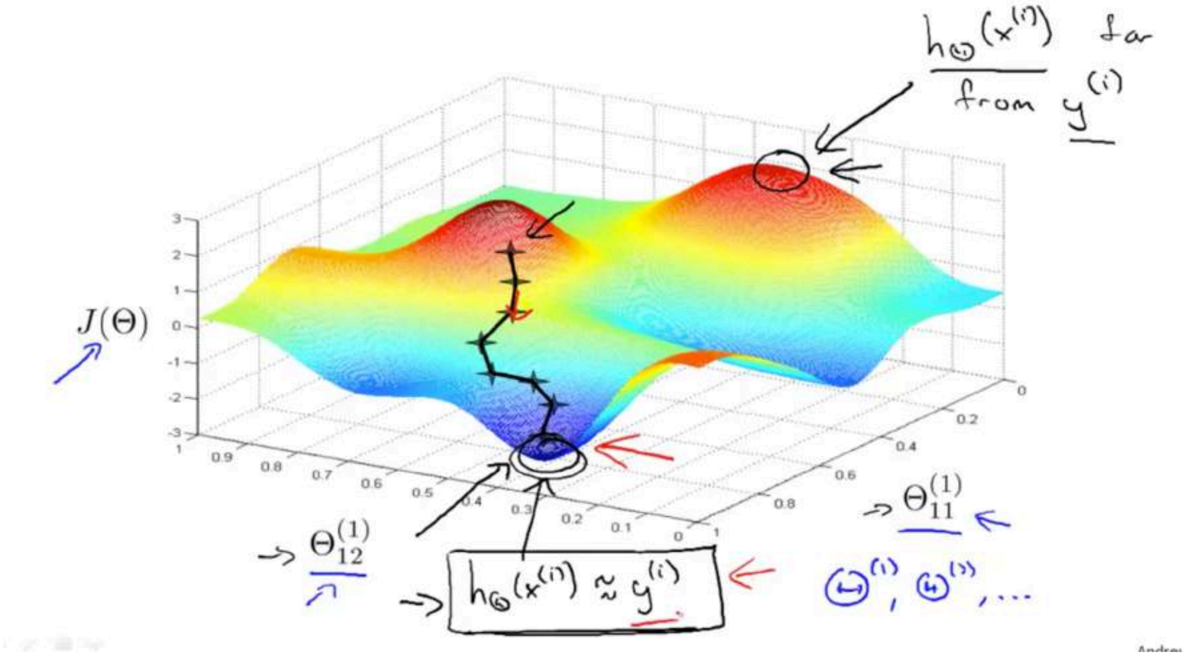
6. Use gradient descent or advanced optimization method with backpropagation to try to minimize $J(\Theta)$ as a function of parameters Θ

$\frac{\partial}{\partial\Theta_{jk}^{(l)}} J(\Theta)$

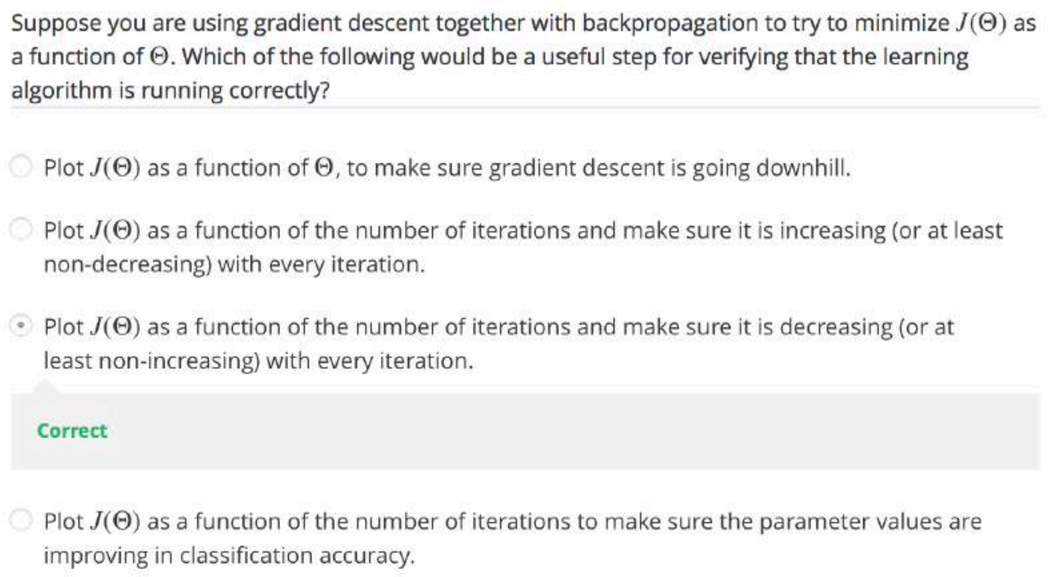
$J(\Theta)$ — non-convex.

Finally, gradient descents for a neural network might still seem a little bit magical. So, let me just show one more figure to try to get that intuition about what gradient descent for a neural network is doing. This was actually similar to the figure that I was using earlier to explain gradient descent. So, we have some cost function, and we have a number of parameters in our neural network. Right here I've just written down two of the parameter values. In reality, of course, in the neural network, we can have lots of parameters with these. Theta one, theta two--all of these are matrices, right? So we can have very high dimensional parameters but because of the limitations the source of parts we can draw. I'm pretending that we have only two parameters in this neural network. Although obviously we have a lot more in practice. Now, this cost function j of theta measures how well the neural network fits the training data. So, if you take a point like this one, down here, that's a point where j of theta is pretty low, and so this corresponds to a setting of the parameters. There's a setting of the parameters theta, where, you know, for most of the training examples, the output of my hypothesis, that may be pretty close to y(i) and if this is true than that's what causes my cost function to be pretty low.

Whereas in contrast, if you were to take a value like that, a point like that corresponds to, where for many training examples, the output of my neural network is far from the actual value y(i) that was observed in the training set. So points like this on the line correspond to where the hypothesis, where the neural network is outputting values on the training set that are far from y(i). So, it's not fitting the training set well, whereas points like this with low values of the cost function corresponds to where j of theta is low, and therefore corresponds to where the neural network happens to be fitting my training set well, because I mean this is what's needed to be true in order for j of theta to be small. So what gradient descent does is we'll start from some random initial point like that one over there, and it will repeatedly go downhill. And so what back propagation is doing is computing the direction of the gradient, and what gradient descent is doing is it's taking little steps downhill until hopefully it gets to, in this case, a pretty good local optimum. So, when you implement back propagation and use gradient descent or one of the advanced optimization methods, this picture sort of explains what the algorithm is doing. It's trying to find a value of the parameters where the output values in the neural network closely matches the values of the y(i)'s observed in your training set.



Question


Suppose you are using gradient descent together with backpropagation to try to minimize $J(\Theta)$ as a function of Θ. Which of the following would be a useful step for verifying that the learning algorithm is running correctly?

- ○ Plot $J(\Theta)$ as a function of Θ, to make sure gradient descent is going downhill.
- ○ Plot $J(\Theta)$ as a function of the number of iterations and make sure it is increasing (or at least non-decreasing) with every iteration.
- [✓ Plot $J(\Theta)$ as a function of the number of iterations and make sure it is decreasing (or at least non-increasing) with every iteration.]

Correct

- ○ Plot $J(\Theta)$ as a function of the number of iterations to make sure the parameter values are improving in classification accuracy.

So, hopefully this gives you a better sense of how the many different pieces of neural network learning fit together.In case even after this video, in case you still feel like there are, like, a lot of different pieces and it's not entirely clear what some of them do or how all of these pieces come together, that's actually okay. Neural network learning and back propagation is a complicated algorithm. And even though I've seen the math behind back propagation for many years and I've used back propagation, I think very successfully, for many years, even today I still feel like I don't always have a great grasp of exactly what back propagation is doing sometimes. And what the optimization process looks like of minimizing j if theta. Much this is a much harder algorithm to feel like I have a much less good handle on exactly what this is doing compared to say, linear regression or logistic regression. Which were mathematically and conceptually much simpler and much cleaner algorithms. But so in case if you feel the same way, you know, that's actually perfectly okay, but if you do implement back propagation, hopefully what you find is that this is one of the most powerful learning algorithms and if you implement this algorithm, implement back propagation, implement one of these optimization methods, you find that back propagation will be able to fit very complex, powerful, non-linear functions to your data, and this is one of the most effective learning algorithms we have today.

Putting it Together (Transcript)

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.

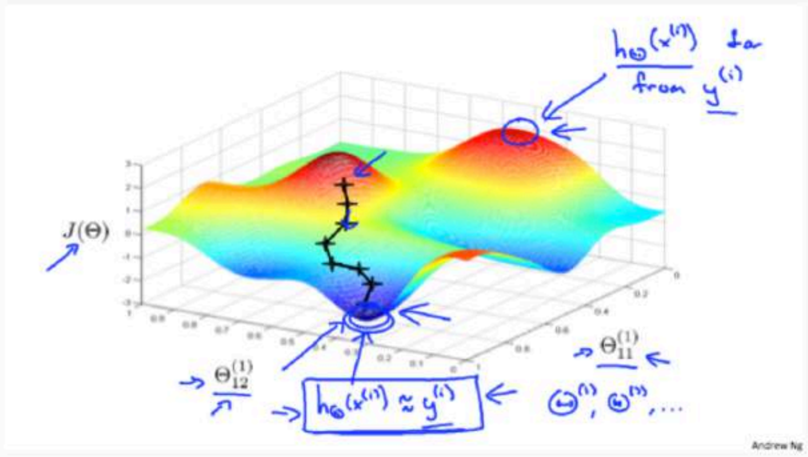
Training a Neural Network

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 built-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 = 1: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:

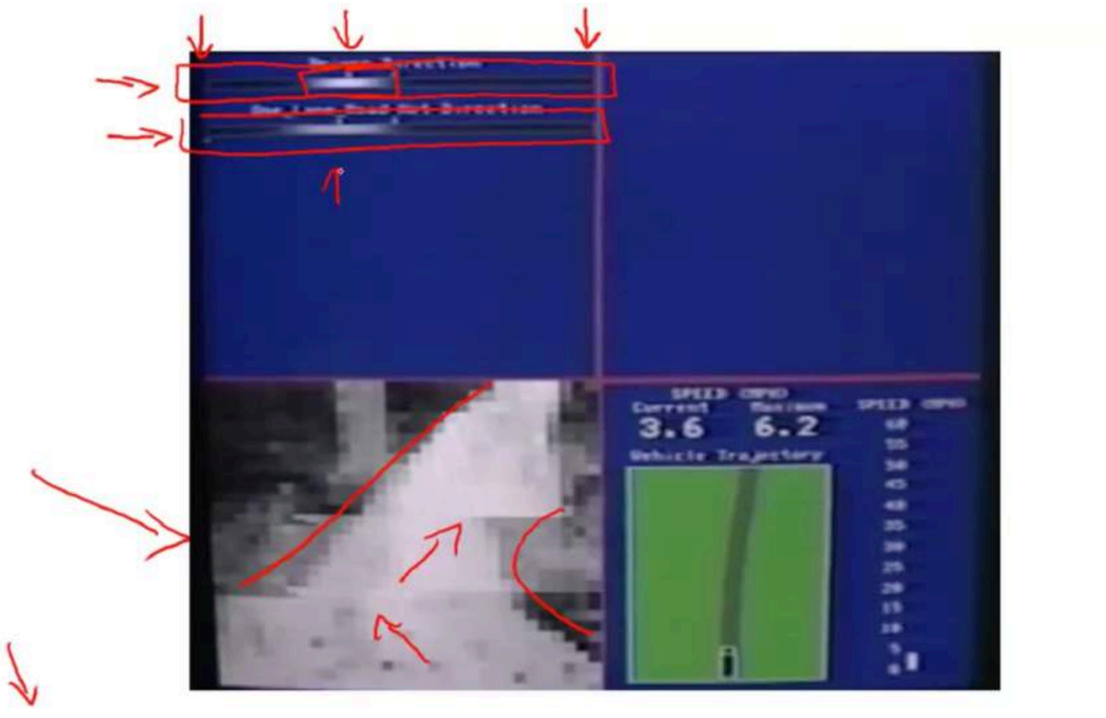


Ideally, you want $h_\Theta(x^{(i)}) \approx y^{(i)}$. This will minimize our cost function. However, keep in mind that $J(\Theta)$ is not convex and thus we can end up in a local minimum instead.

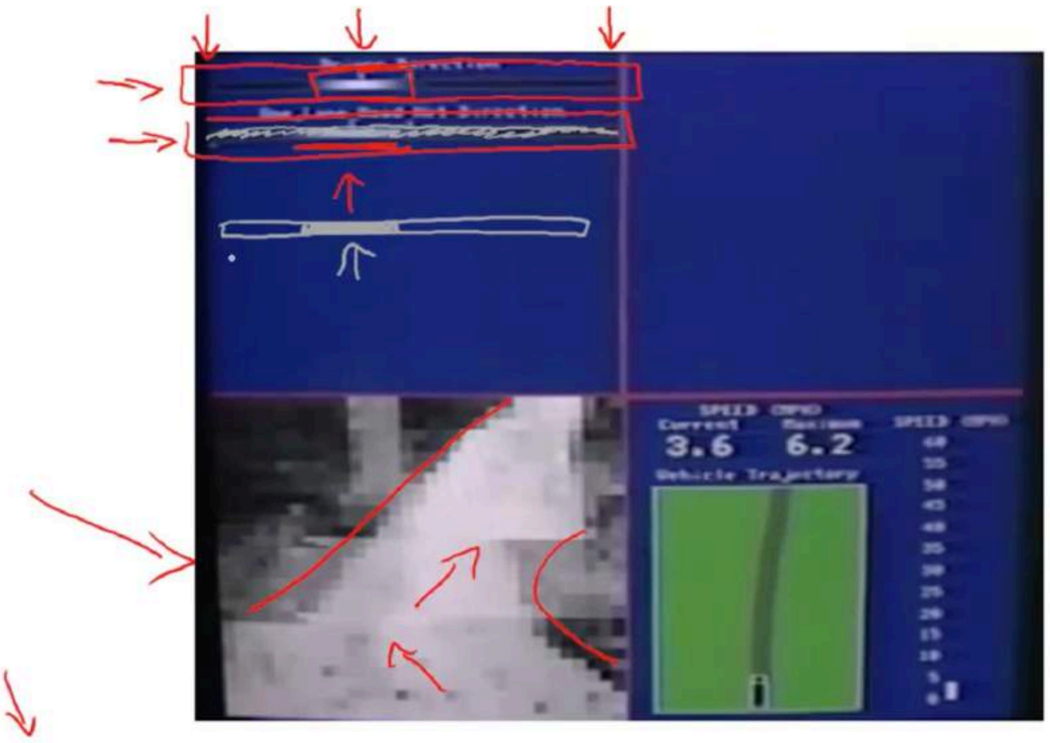
Applications of Neural Networks

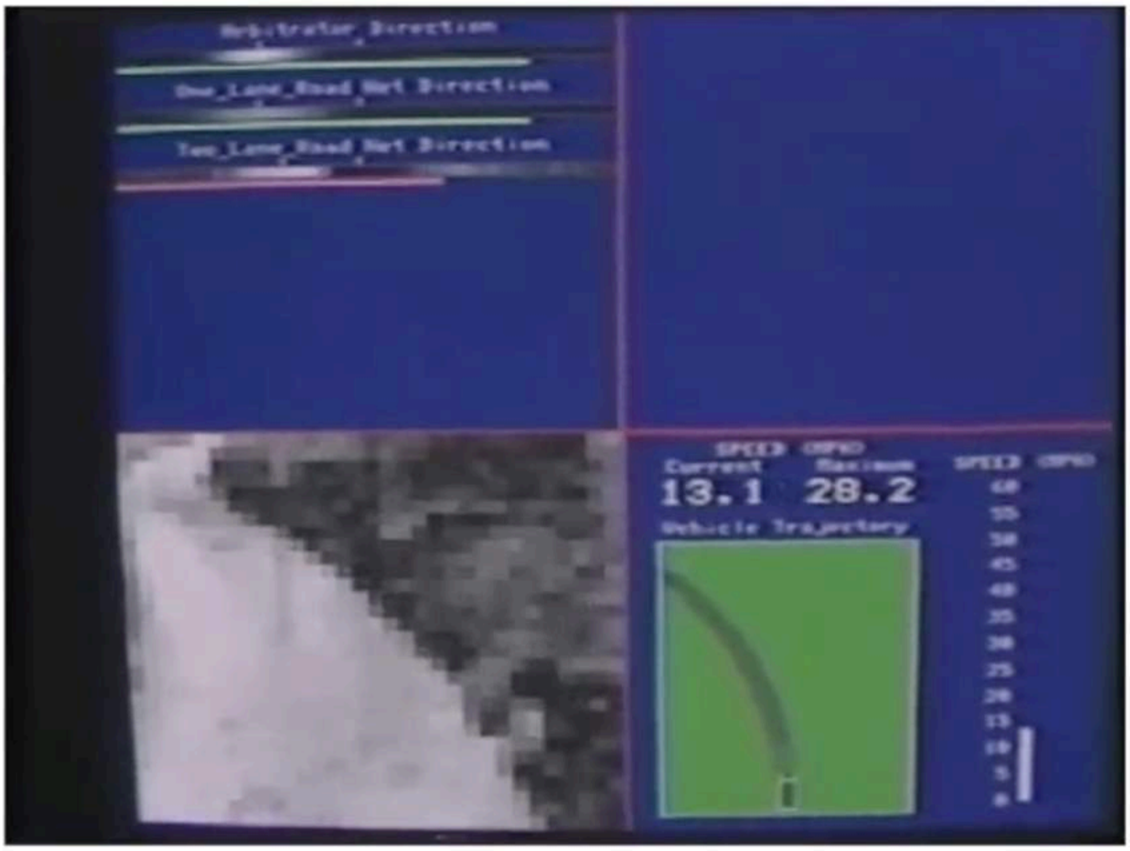
Autonomous Driving

In this video, I'd like to show you a fun and historically important example of neural networks learning of using a neural network for autonomous driving. That is getting a car to learn to drive itself. The video that I'll showed a minute was something that I'd gotten from Dean Pomerleau, who was a colleague who works out in Carnegie Mellon University out on the east coast of the United States. And in part of the video you see visualizations like this. And I want to tell what a visualization looks like before starting the video. Down here on the lower left is the view seen by the car of what's in front of it. And so here you kinda see a road that's maybe going a bit to the left, and then going a little bit to the right. And up here on top, this first horizontal bar shows the direction selected by the human driver. And this location of this bright white band that shows the steering direction selected by the human driver where you know here far to the left corresponds to steering hard left, here corresponds to steering hard to the right. And so this location which is a little bit to the left, a little bit left of center means that the human driver at this point was steering slightly to the left. And this second bot here corresponds to the steering direction selected by the learning algorithm and again the location of this sort of white band means that the neural network was here selecting a steering direction that's slightly to the left.



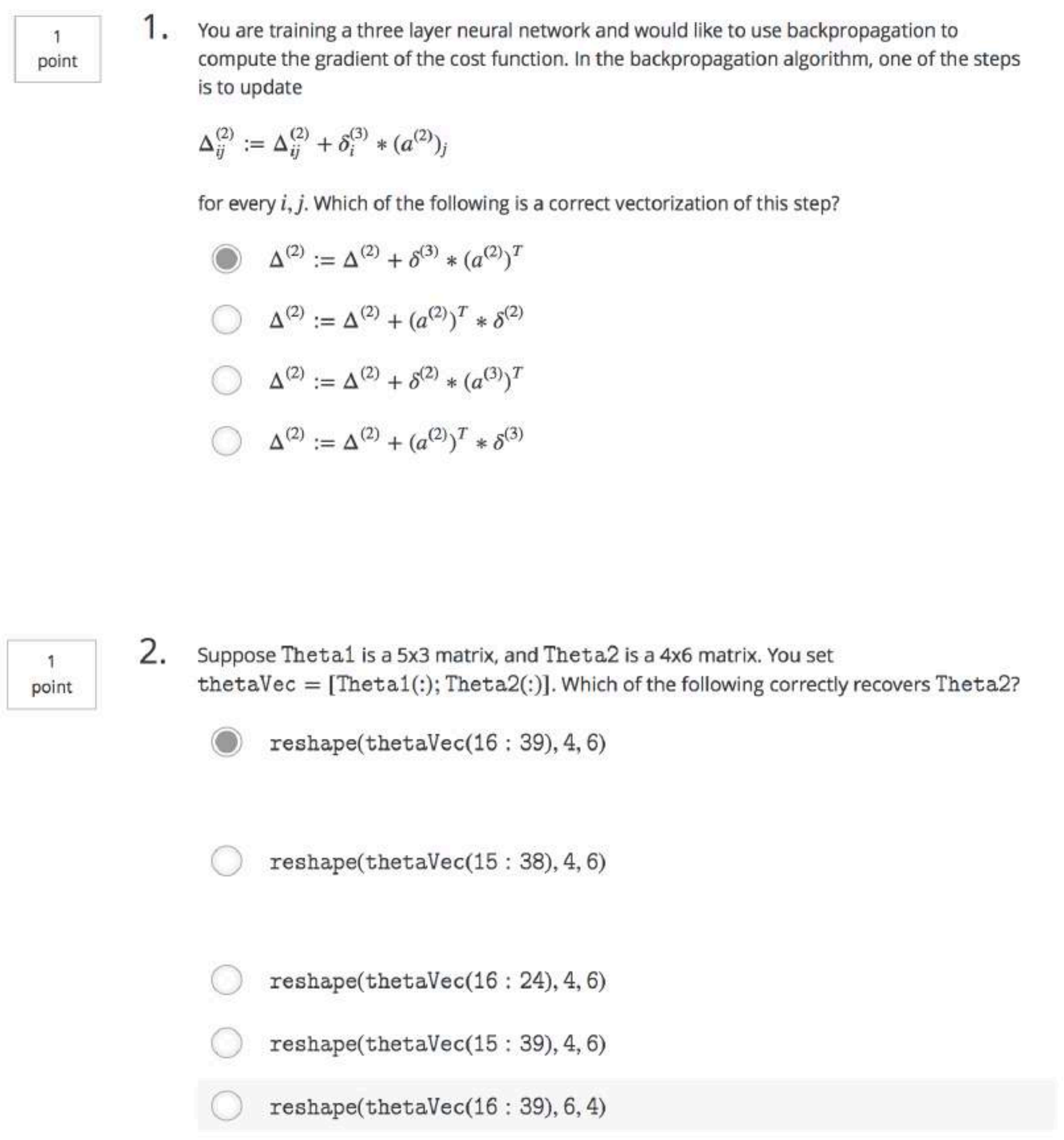
And in fact before the neural network starts leaning initially, you see that the network outputs a grey band, like a grey, like a uniform grey band throughout this region and sort of a uniform gray fuzz corresponds to the neural network having been randomly initialized. And initially having no idea how to drive the car. Or initially having no idea of what direction to steer in. And is only after it has learned for a while, that will then start to output like a solid white band in just a small part of the region corresponding to choosing a particular steering direction. And that corresponds to when the neural network becomes more confident in selecting a band in one particular location, rather than outputting a sort of light gray fuzz, but instead outputting a white band that's more constantly selecting one's steering direction.


Direction
One_Lane_Road_Net Direction
SPEED (MPH)
Current
3.6
Maximum
6.2
Vehicle Trajectory
SPEED (MPH)
60
55
50
45
40
35
30
25
20
15
10
5
0


Arbitrator_Direction
One_Lane_Road_Net Direction
Two_Lane_Road_Net Direction
SPEED (MPH)
Current
13.1
Maximum
28.2
Vehicle Trajectory
SPEED (MPH)
60
55
50
45
40
35
30
25
20
15
10
5
0

Review

Quiz



1 point

1. You are training a three layer neural network and would like to use backpropagation to compute the gradient of the cost function. In the backpropagation algorithm, one of the steps is to update

$\Delta^{(2)}_{ij} := \Delta^{(2)}_{ij} + \delta^{(3)}_i * (a^{(2)})_j$

for every i, j. Which of the following is a correct vectorization of this step?

- [✓ $\Delta^{(2)} := \Delta^{(2)} + \delta^{(3)} * (a^{(2)})^T$]
- $\Delta^{(2)} := \Delta^{(2)} + (a^{(2)})^T * \delta^{(2)}$
- $\Delta^{(2)} := \Delta^{(2)} + \delta^{(2)} * (a^{(3)})^T$
- $\Delta^{(2)} := \Delta^{(2)} + (a^{(2)})^T * \delta^{(3)}$

1 point

2. Suppose Theta1 is a 5x3 matrix, and Theta2 is a 4x6 matrix. You set thetaVec = [Theta1(:); Theta2(:)]. Which of the following correctly recovers Theta2?

- [✓ reshape(thetaVec(16 : 39), 4, 6)]
- reshape(thetaVec(15 : 38), 4, 6)
- reshape(thetaVec(16 : 24), 4, 6)
- reshape(thetaVec(15 : 39), 4, 6)
- reshape(thetaVec(16 : 39), 6, 4)

1
point

3. Let $J(\theta) = 3\theta^3 + 2$. Let $\theta = 1$, and $\epsilon = 0.01$. Use the formula $\frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}$ to numerically compute an approximation to the derivative at $\theta = 1$. What value do you get? (When $\theta = 1$, the true/exact derivative is $\frac{dJ(\theta)}{d\theta} = 9$.)

- ○ 11
- ○ 9
- ○ 8.9997
- [✓ ● 9.0003]

4. Which of the following statements are true? Check all that apply.

- ☐ Computing the gradient of the cost function in a neural network has the same efficiency when we use backpropagation or when we numerically compute it using the method of gradient checking.
- [✓ ■] For computational efficiency, after we have performed gradient checking to

  verify that our backpropagation code is correct, we usually disable gradient checking before using backpropagation to train the network.
- [✓ ■] Using gradient checking can help verify if one's implementation of backpropagation is bug-free.
- ☐ Gradient checking is useful if we are using one of the advanced optimization methods (such as in fminunc) as our optimization algorithm. However, it serves little purpose if we are using gradient descent.

1
point

5. Which of the following statements are true? Check all that apply.

- ☐ Suppose you have a three layer network with parameters $\Theta^{(1)}$ (controlling the function mapping from the inputs to the hidden units) and $\Theta^{(2)}$ (controlling the mapping from the hidden units to the outputs). If we set all the elements of $\Theta^{(1)}$ to be 0, and all the elements of $\Theta^{(2)}$ to be 1, then this suffices for symmetry breaking, since the neurons are no longer all computing the same function of the input.

- ☐ If we initialize all the parameters of a neural network to ones instead of zeros, this will suffice for the purpose of "symmetry breaking" because the parameters are no longer symmetrically equal to zero.

- ■ If we are training a neural network using gradient descent, one reasonable "debugging" step to make sure it is working is to plot $J(\Theta)$ as a function of the number of iterations, and make sure it is decreasing (or at least non-increasing) after each iteration.

- ■ Suppose you are training a neural network using gradient descent. Depending on your random initialization, your algorithm may converge to different local optima (i.e., if you run the algorithm twice with different random initializations, gradient descent may converge to two different solutions).