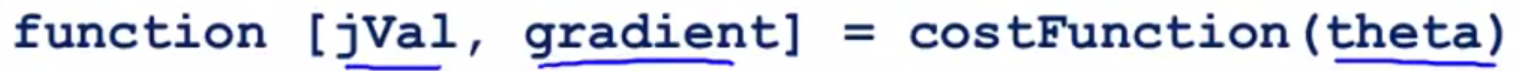
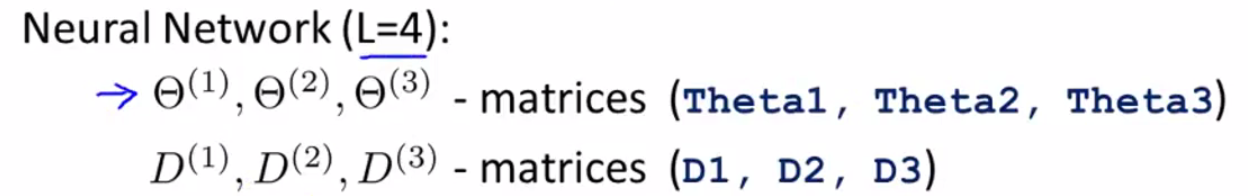
***Neural Networks – Backpropagation in Practice***

**I. IMPLENTATION NOTE: UNROLLING PARAMETERS**

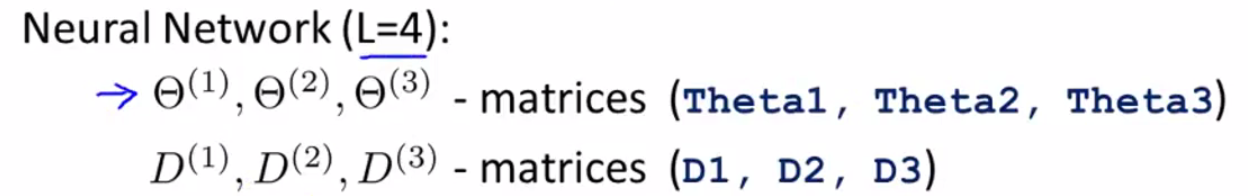
* let's say you've implemented a cost function that takes as input parameters ϴ + returns the cost function + derivatives.
* 
* Then we can pass this to an advanced authorization algorithm (like fminunc) which takes as input the cost function + some initial value of ϴ.



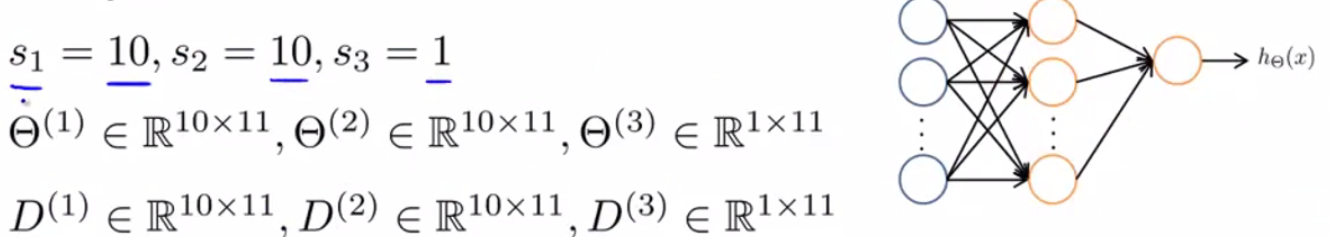
* Both of these routines assume that ϴ + the initial value of ϴ are parameter vectors, maybe Ɍn or Ɍ(n + 1) + that the cost function returns the gradient as a second value, which is also an Ɍn or Ɍ(n + 1) vector
* This worked fine when using logistic regression but in a NN, our parameters are no longer vectors, but instead are matrices



* Similarly the gradient terms we’re expected to return are matrices



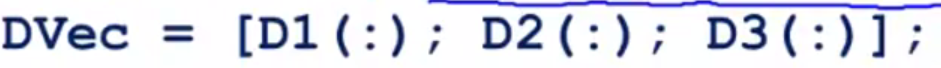
* We want to take these matrices and **unroll** them *into vectors* so they end up being in a format suitable for passing in as ϴ + getting out as a gradient.
* Concretely, let's say we have a NN with 1 input layer with 10 units, 1 hidden layer with 10 units + 1 output layer with just 1 unit, where s1 = number of units in layer 1 + s2 = number of units in layer 2



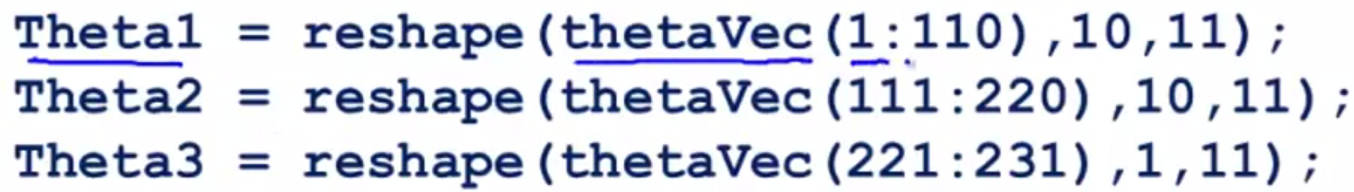
* In this case, the dimension of the matrices ϴ and D are going to be given by the above expressions.
* So in if you want to convert these matrices to vectors in Octave, you can take your ϴ’s + write this piece of code to take all the elements of the 3 ϴ matrices, unroll them, + put them into a big vector.



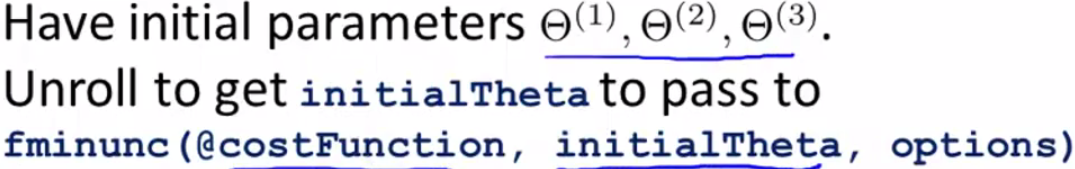
* Similarly we can take all D matrices + unroll them into a big long vector and call them DVec.



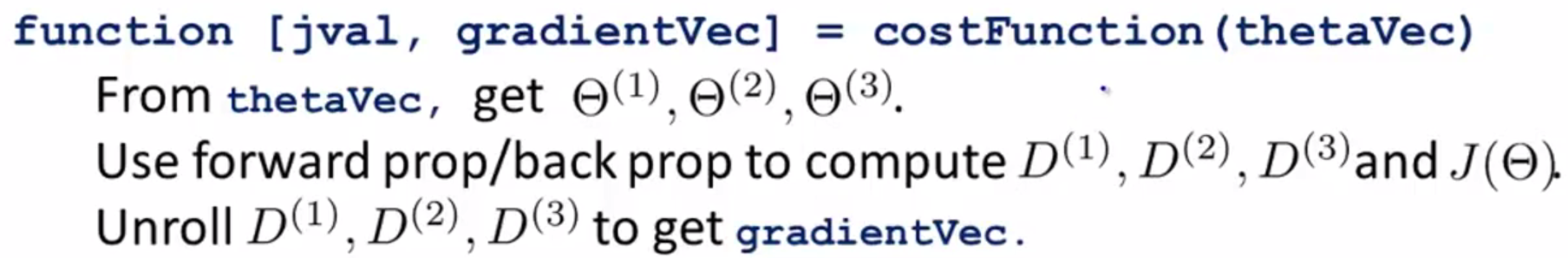
* If you want to go back from the vector representations to the matrix representations, do the following:



* ϴ1 has 110 elements b/c it's a 10x11
* To make this process really concrete, here's how we use the **unrolling** idea to implement our learning algorithm.
* Say we have some initial value of the parameters ϴ1-ϴ3
* We unroll them into a long vector called initialϴ to pass in to fminunc as the initial parameters ϴ.



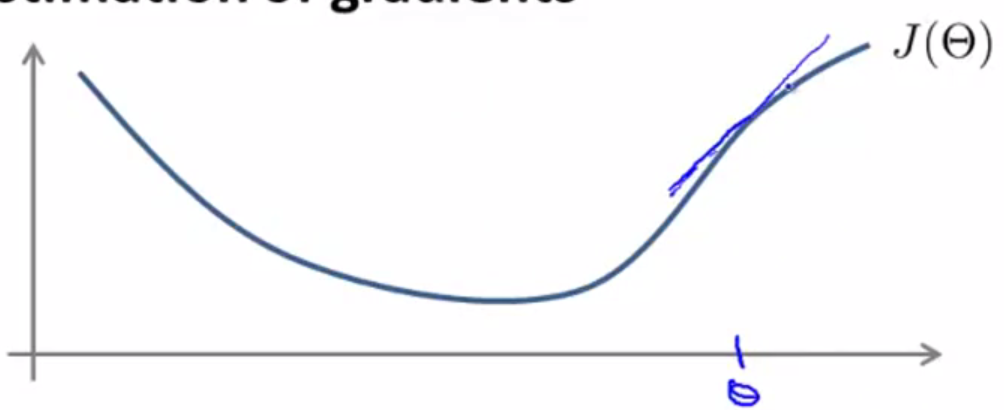
* The other thing we need to do is implement the cost function.



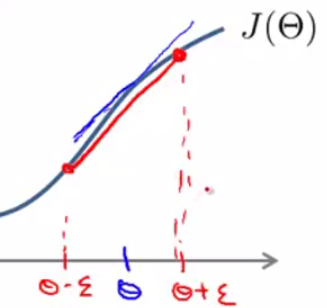
* The cost function gets as input thetaVec = all of parameters vectors that were unrolled into a vector
* 1st, use thetaVec + use the **reshape** functions to pull out elements from thetaVec to get back my original parameter matrices
* These matrices give a more convenient form in which to run forward + back propagation to compute my derivatives + cost function j(ϴ).
* Finally we take the derivatives + unroll them (keeping the elements in the same ordering as I did when I unroll my ϴ’s) into a vector of derivatives to get **gradientVec**, which what my cost function can return.
* The advantage of the matrix representation is that when parameters are stored as matrices, it's more convenient when doing forward + back propagation to take advantage of the vectorized implementations.
* Whereas in contrast, the advantage of the vector representation is that when using advanced optimization algorithms, those algorithms tend to assume you have all parameters unrolled into a big long vector

**II. GRADIENT CHECKING**

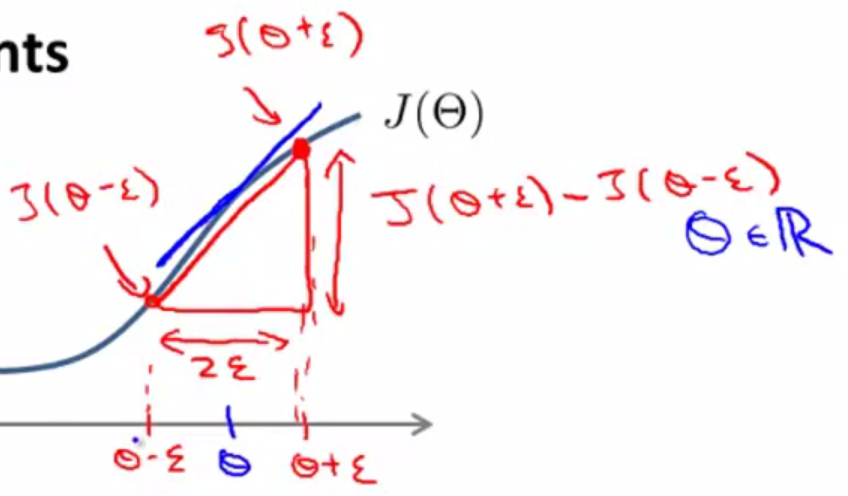
* So we went over how to do forward + backpropagation in a NN to compute derivatives.
* But backpropagation as an algorithm has a lot of details + can be a little bit tricky to implement
* 1 unfortunate property is there are many ways to have subtle bugs in BP, so that if you run it w/ gradient descent or some other optimization algorithm, it could actually look like it's working (the cost function J(ϴ) 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 BP.
* You might just wind up with a NN that has a higher level of error than w/ a bug-free implementation, + you might just not know there was a subtle bug giving you worse performance.
* There's an idea called **gradient checking** that eliminates almost all of these problems.
* This will help make sure + gain high confidence that an implementation of FP/BP is 100% correct.
* Once you implement numerical gradient checking, you'll be able to verify for yourself that code you're writing is indeed computing the derivative of the cost function J.
* Suppose that I have the function J(ϴ) + some value ϴ (assumed to be a real number) + we want to estimate the derivative of J(ϴ) at this point (slope of tangent line at ϴ)



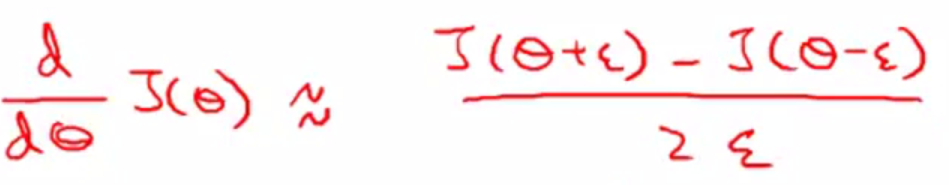
* Here's a procedure for numerically approximating the derivative
* Compute ϴ + ε and ϴ - ε, look at those 2 points, connect them by a straight line, + use the slope of that line as my approximation to the derivative.



* Mathematically, the slope of this red line is the vertical height divided by the horizontal width of the triangle made by the points



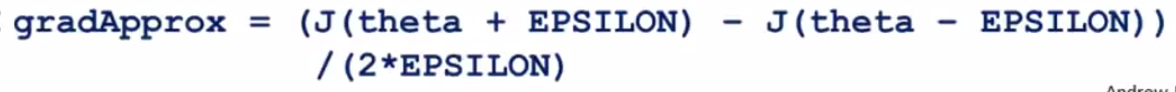
* So my approximation is going to be that the derivative of J(ϴ) w/ respect to ϴ of is approximately **[ J(ϴ + ε) - J(ϴ - ε) ]/ 2ε**



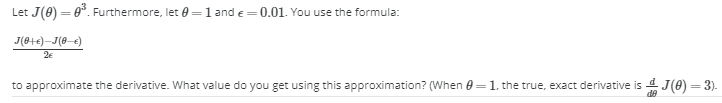
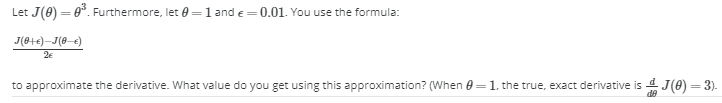
* Usually, use a pretty small value for ε, maybe on the order of 10^-4, but there’s a large range of different values for ε that work just fine
* If you let ε become really small, then mathematically we actually get the true derivative, as it becomes exactly the slope of the function at this point.
* We don't want to use ε that's too, too small, b/c we might run into numerical problems.
* So I usually use epsilon around ten to the minus four.
* \*\*\***NOTE**: 2-sided difference works better than 1-sided difference

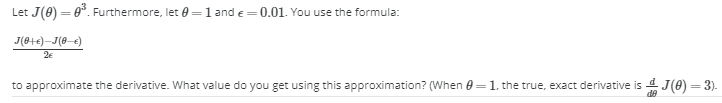


* When you implement in Octave is:



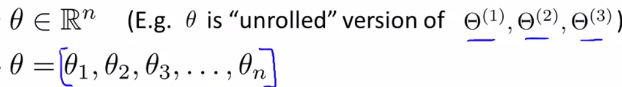
* This will give you a numerical estimate of the gradient at that point







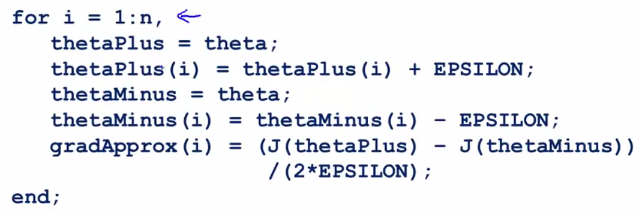
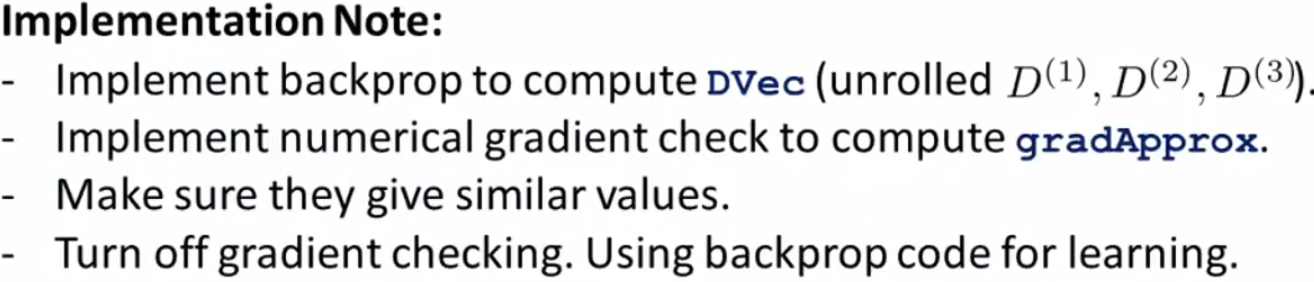
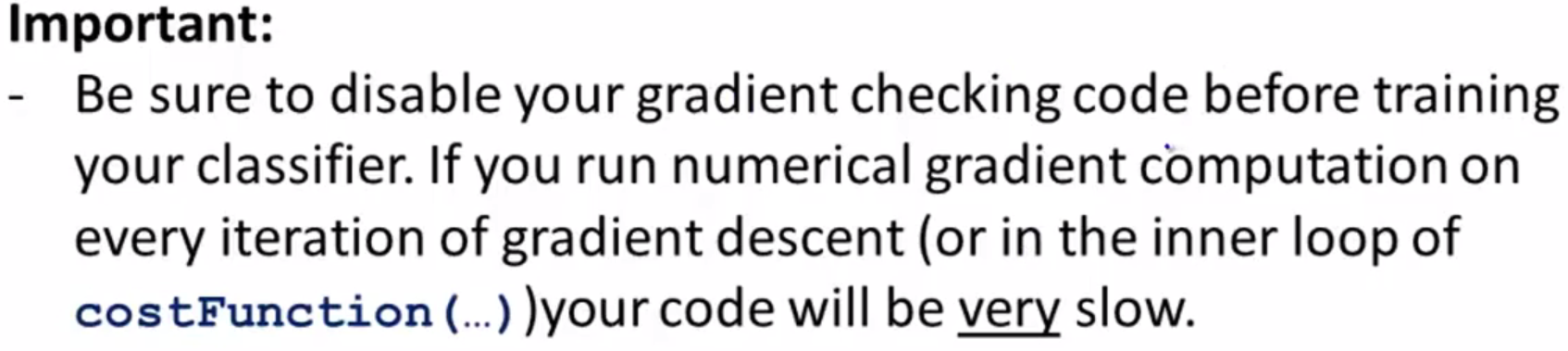
* Now let's look at a more general case of when theta is an Rn parameter vector, ϴ1-ϴn.



* We can then use a similar idea to before to approximate all the partial derivative terms.





* Concretely the partial derivative of a cost function w/ respect to the 1st parameter, ϴ1 can be obtained by taking J of ϴ1 + ε and so on + then subtract J of ϴ1 - ε + then divide that result by 2 ε
* These equations give you a way to numerically approximate the partial derivative of J w/ respect to any one of your parameters ϴi
* Concretely, what you implement in Octave is therefore the following:
* 
* \*\*\*usually do this w/ the unrolled version of the parameter 🡪 ϴ is just a long list of all parameters in the NN
* The way we use this in our NN implementation is using this FOR loop to compute the partial derivative of the cost function for respect to *every* parameter in that network
* We can then take the gradient/derivatives that we got from BP, **DVec** + make sure our **gradApprox** is approximately equal up to the DVec we got from BP
* If these 2 ways of computing the derivative give me the same/very similar answers, I'm much more confident my implementation of BP is correct.
* So when we plug these DVec vectors into gradient descent or some advanced optimization algorithm, we can then be much more confident we’re computing the derivatives correctly + therefore the code will run correctly + do a good job optimizing J(ϴ).
* Finally, put everything together to implement this numerical gradient checking:
* 
* 
* \*\*\*Before seriously training a NN, turn off gradient checking to no longer compute gradApprox using the numerical derivative formulas.
* The numerical gradient checking code is a very computationally expensive/slow way to try to approximate the derivative.
* Whereas the BP algorithm is a much more computationally efficient way of computing for derivatives.
* So once you've verified your implementation of BP is correct, turn off gradient checking before running your algorithm for many iterations of gradient descent/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 it were in the inner loop of costFunction), the your code would be very slow b/c the numerical gradient checking code is much slower than the BP algorithm
* So, the BP algorithm is a much faster way to compute derivatives than gradient checking, but we can find gradients numerically to verify an implementation of BP is correct.

**III. RANDOM INITIALIZATION**

**IV. PUTTING IT TOGETHER**