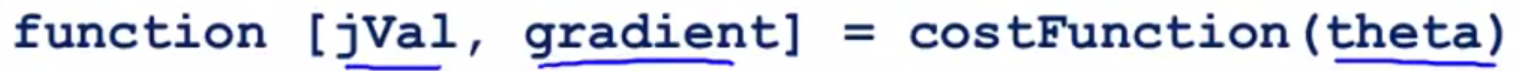
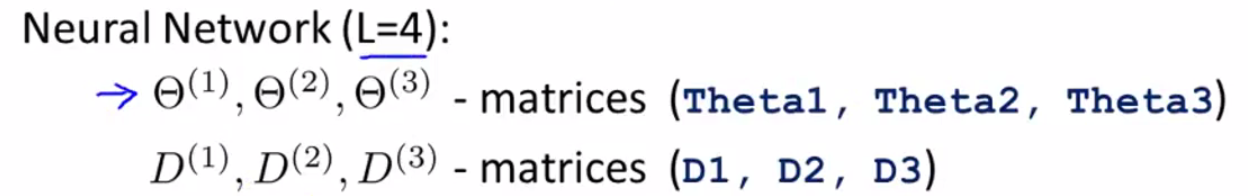
***NNs – Backpropagation in Practice***

**I. IMPLENTATION NOTE: UNROLLING PARAMETERS**

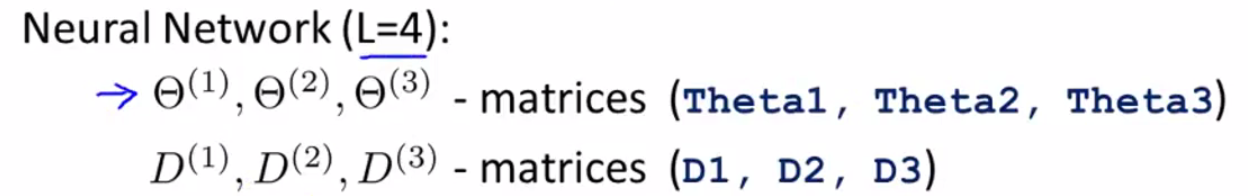
* Let's say you've implemented a cost function that takes as input parameters ϴ + returns the cost function + derivatives.
* 
* Then we can pass this to an advanced authorization algorithm (like fminunc) which takes as input the cost function + some initial value of ϴ.



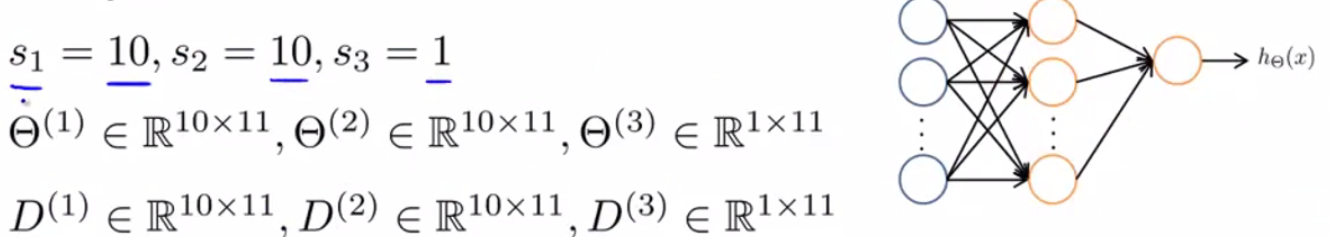
* Both of these routines assume that ϴ + the initial value of ϴ are parameter vectors, maybe Ɍn or Ɍ(n + 1) + that the cost function returns the gradient as a second value, which is also an Ɍn or Ɍ(n + 1) vector
* This worked fine when using logistic regression but in a NN, our parameters are no longer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



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



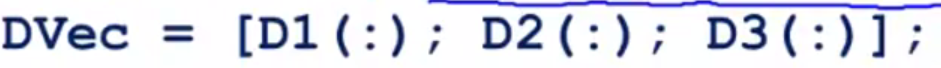
* We want to take these matrices and **unroll** them *into vectors* so they end up being in a format suitable for passing in as ϴ + getting out as a gradient.
* Concretely, let's say we have a NN with 1 input layer with 10 units, 1 hidden layer with 10 units + 1 output layer with just 1 unit, where s1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



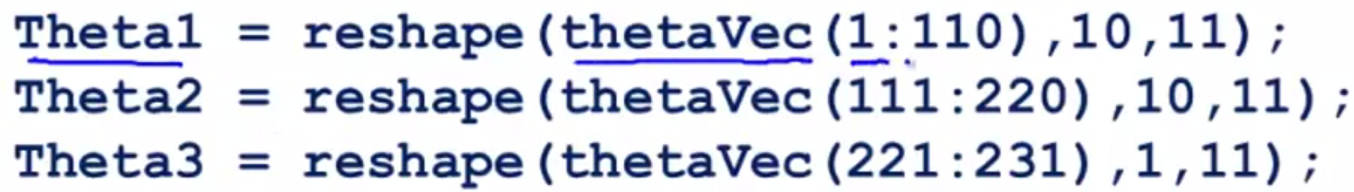
* In this case, the dimension of the matrices ϴ and D are going to be given by the above expressions.
* So in if you want to convert these matrices to vectors in Octave, you can take your ϴ’s + write this piece of code to take all the elements of the 3 ϴ matrices, unroll them, + put them into a big vector.



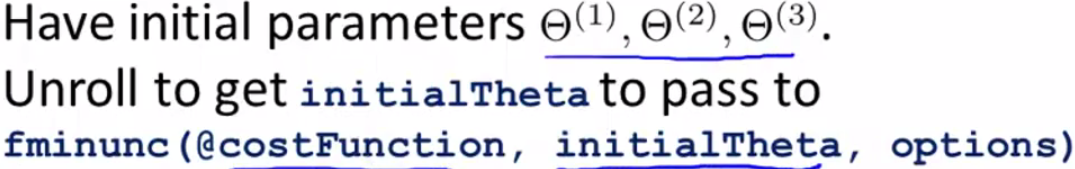
* Similarly we can take all D matrices + unroll them into a big long vector and call them DVec.



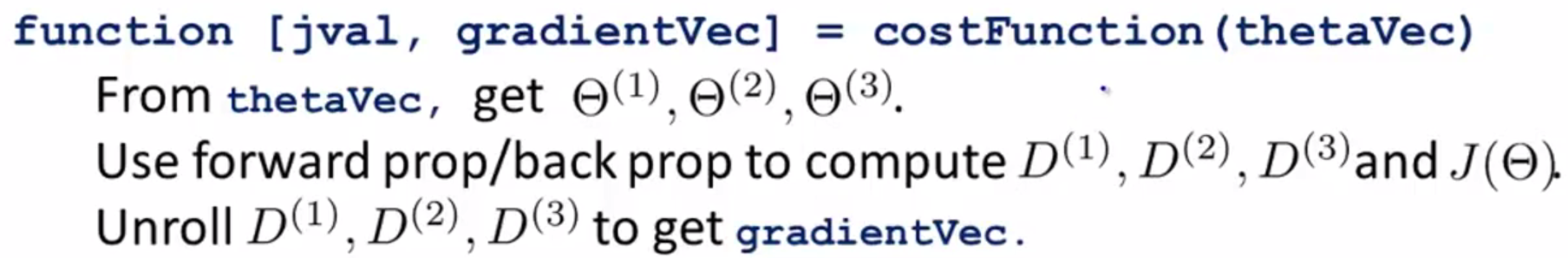
* If you want to go back from the vector representations to the matrix representations, do the following:



* 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.



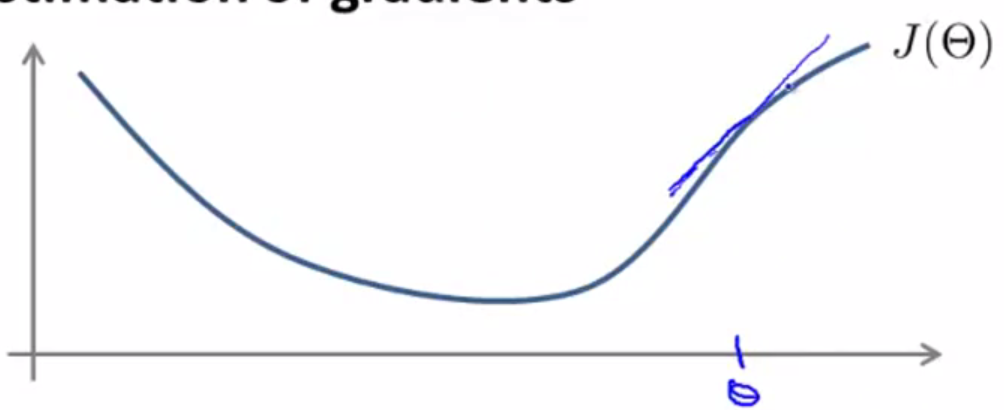
* The other thing we need to do is implement the cost function.



* The cost function gets as input ϴVec = all of parameters vectors that were unrolled into a vector
* 1st, use ϴVec + use the **reshape** functions to pull out elements from ϴVec 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**

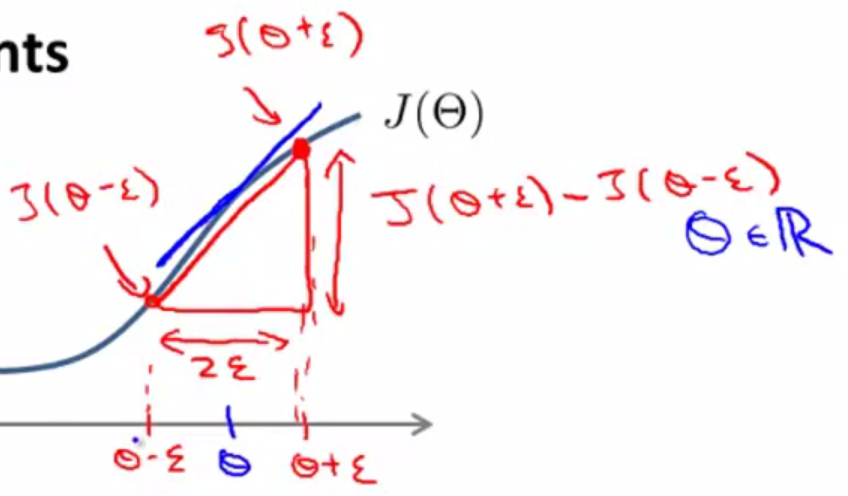
* So we went over how to do forward + backpropagation in a NN to compute derivatives.
* But backpropagation as an algorithm has a lot of details + can be a little bit tricky to implement
* 1 unfortunate property is there are many ways to have subtle bugs in BP, so that if you run it w/ gradient descent 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



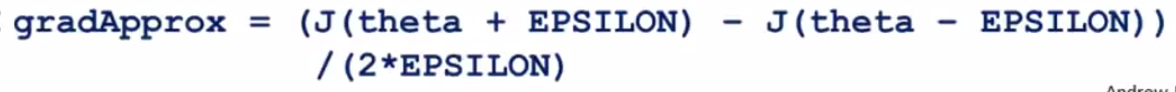
* So my approximation is going to be that the derivative of J(ϴ) w/ respect to ϴ of is approximately **[ J(ϴ + ε) - J(ϴ - ε) ]/ 2ε**



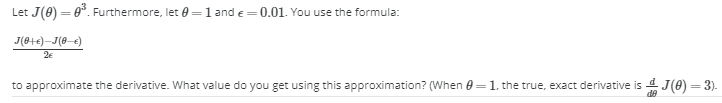
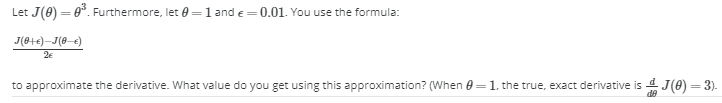
* Usually, use a pretty small value for ε, maybe on the order of 10^-4, but there’s a large range of different values for ε that work just fine
* If you let ε become really small, then mathematically we actually get the true derivative, as it becomes exactly the slope of the function at this point.
* 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

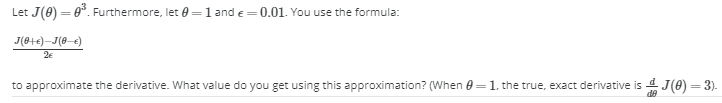


* When you implement in Octave is:



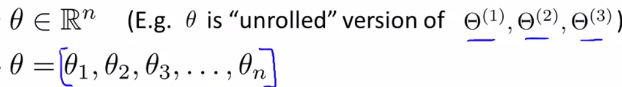
* This will give you a numerical estimate of the gradient at that point







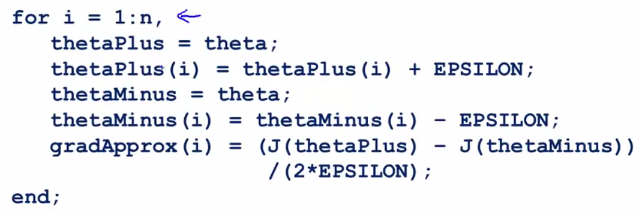
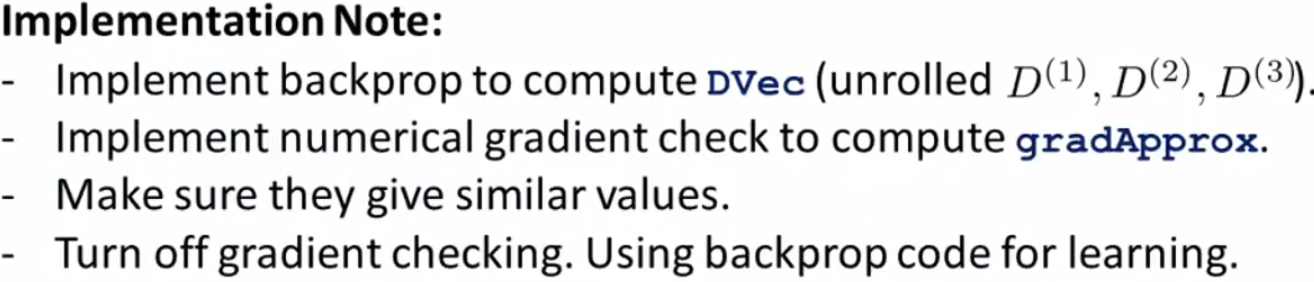
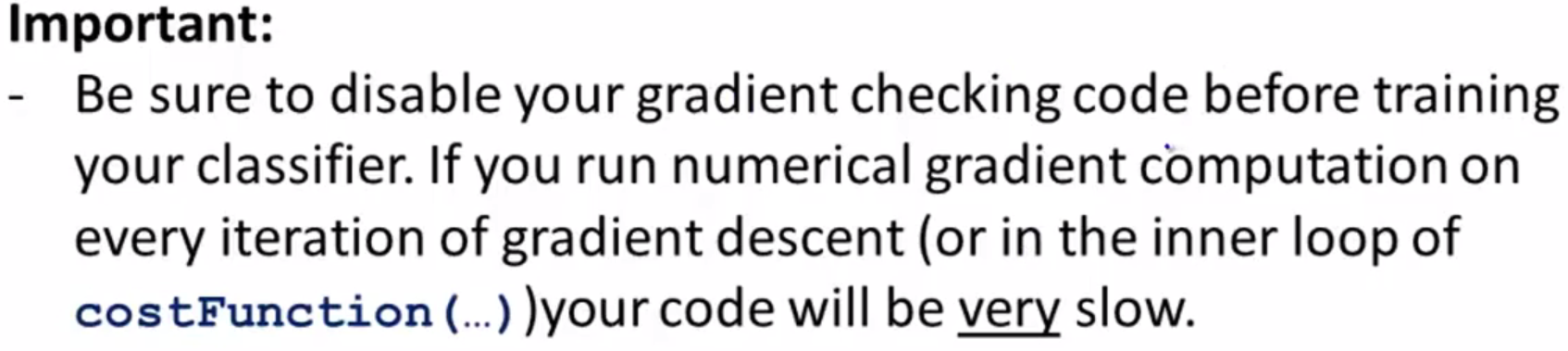
* Now let's look at a more general case of when ϴ is an Rn parameter vector, ϴ1-ϴn.



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

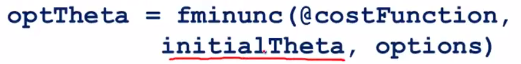




* Concretely the partial derivative of a cost function w/ respect to the 1st parameter, ϴ1 can be obtained by taking J of ϴ1 + ε and so on + then subtract J of ϴ1 - ε + then divide that result by 2 ε
* These equations give you a way to numerically approximate the partial derivative of J w/ respect to 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**

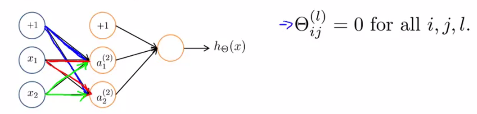
* When running an algorithm like gradient descent or also an advanced optimization algorithm, we need to pick some initial value for the parameters ϴ.



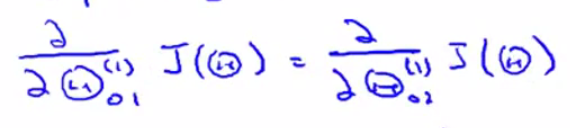
* Consider gradient descent where we also need to initialize a ϴ + then can slowly take steps to go downhill to minimize J(ϴ)
* So what can we set the initial value of ϴ to be? Is it possible to set it to be a vector of all zeros?



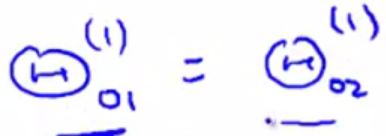
* Whereas this worked okay when using logistic regression, initializing all parameters to 0 actually does NOT work when training a NN.
* Consider the following NN w/ all parameters of the network initialized to 0.



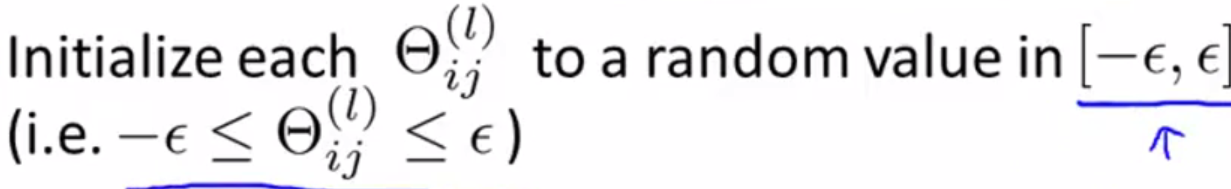
* If so, both blue weights are the same, as are both red and green weights
* This means that both hidden units, a1 + a2, are going to be computing the same function of your inputs + thus you end up w/ (for every one of your training examples) a2(1) = a2(2)
* B/c these outgoing weights are the same, you can also show that the δ values are also going to be the same.
* Concretely, you’d end up with δ2(1) = δ2(2
* If you work through the NN further, you can show that the partial derivatives w/ respect to your parameters will satisfy the following:

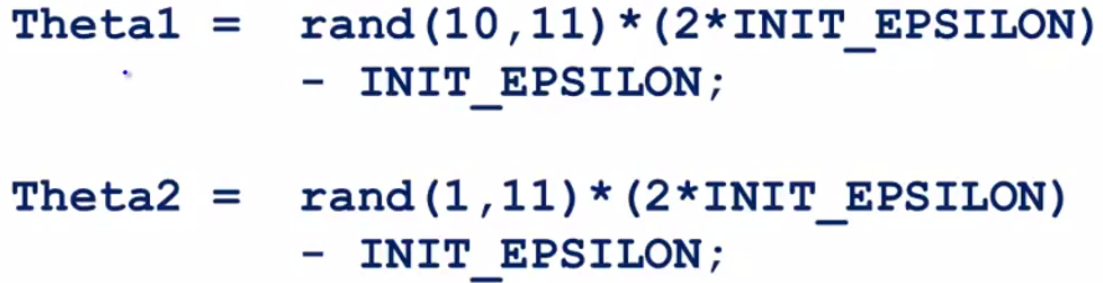


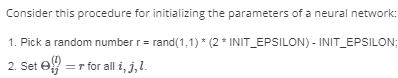
* That the partial derivative of the cost function w/ respected to the 2 blue weights are going to be equal to each other.
* This means that after 1 gradient descent update, we update the 1st blue weight w/ **α** multiplied by its partial derivative + update the 2nd blue rate w/ **α** multiplied by it’s partial derivative, which is the same as the 1st blue weight’s partial derivative
* So after 1 gradient descent update, those 2 blue weights end up the same as each other.



* The same goes for the red and green weights
* So *after each update, the parameters corresponding to the inputs that are going into each of the 2 hidden units are identical.*
* i.e. the 2 green weights are the same, the 2 red weights are the same, the 2 blue weights are the same
* This means is that even after 1 iteration of gradient descent, your 2 hidden units are still computing exactly the same functions of the input (still have the a1(2) = a2(2))
* As you keep running gradient descent, the 2 blue, 2 red, + 2 green weights will stay equal to each other
* *This means your NN really cannot compute very interesting functions*
* Imagine you had not only 2 hidden units, but many, many hidden units.
* Then all your hidden units are computing the *exact same feature,* or the *exact same function of the input*
* This is a highly redundant representation b/c then your final logistic progression unit really has to see only 1 feature, b/c all the “a” values are the same
* This prevents you + your NN from doing something interesting.
* In order to get around this problem, the way we initialize the parameters of a NN is w/ **random initialization**.
* What we just described is something called the **problem of symmetric weights**
* Random initialization is how we perform **symmetry breaking**
* What we do is we initialize each value of ϴ/weight for the parameters to a random number between –**ε** and +**ε**.





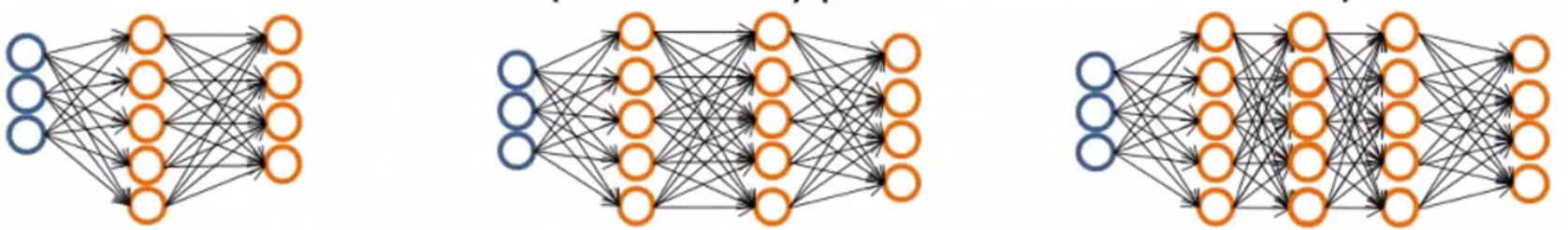
* In Octave set ϴ1 to a random 10 by 11 dimensional matrix where all values are between 0 and 1, take those values + multiply it by 2**ε** and - **ε** to end up w/ a number between - **ε** and + **ε**.
* NOTE: these epsilons have nothing to do with the **ε** we were using when in gradient checking.
* When numerical gradient checking, we were adding some values of **ε** and ϴ, + this is an unrelated value of **ε**.
* 

 🡪 need to set each ϴ to its own random r

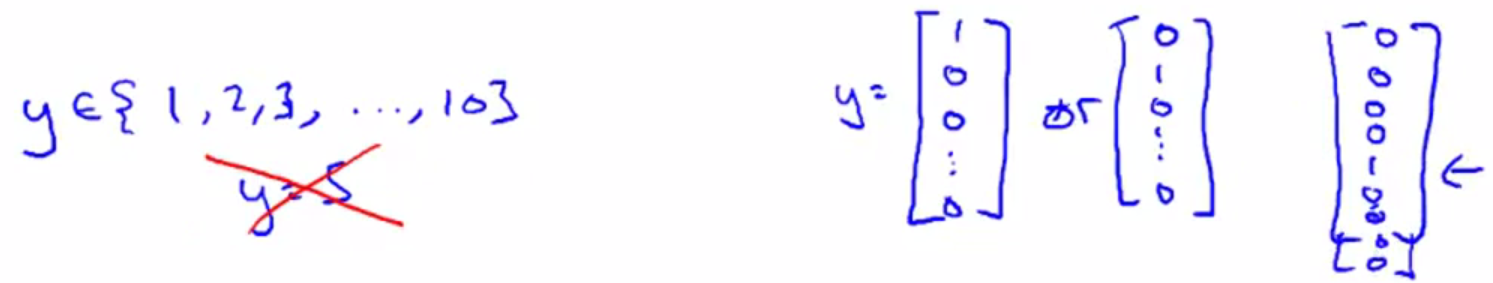
* So to summarize, to create a NN, what you should do is **randomly initialize** the weights to small values close to 0, between - **ε** and + **ε**
* Then implement **back propagation**, do **gradient checking**, and use either great gradient descent or one of the advanced optimization algorithms to try to minimize J(ϴ) as a function of the parameters ϴ starting from a randomly chosen initial value for the parameters ϴ.
* By doing **symmetry breaking**, hopefully great gradient descent or the advanced optimization algorithms will be able to find a good value of ϴ.

**IV. PUTTING IT TOGETHER**

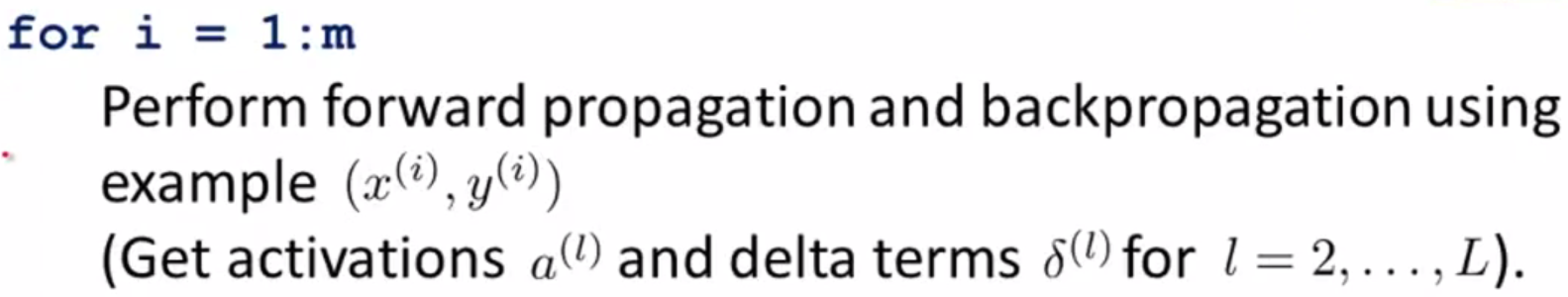
* So, it's taken us a lot of work to get through the NN learning algorithm.
* Now to put all the pieces together to get an overall summary of how all the pieces fit together + of the overall process of how to implement a NN learning algorithm.
* When training a NN, the 1st thing to do is pick some **network architecture** (connectivity pattern between the neurons/units)



* i.e. 3 input units, 5 hidden units, 4 four output units vs 3 input, 5 hidden, 5 hidden, 4 output
* So, how do you make these choices?
* Well 1st, the number of input units is pretty well-defined as the dimensions of features x(i)
* Once you decides on a fixed set of features **x**, the number of input units will just be the dimension of the features x(i)
* If doing multiclass classification, the number of *output* will be determined by the number of classes in the classification problem.
* Ex: y takes on say values between 1 and 10, you have 10 possible classes, + we write/recode our outputs y as vectors (for each class, put 1 in that space, i.e. y = 5 has a “1” in the 5th spot and 0s elsewhere



* So the choice of number of input + output units is reasonably straightforward.
* As for the number of hidden units + number of hidden layers, a reasonable default (and probably most common choice) is to use a *single* hidden layer
* If you use more than 1 hidden layer, a reasonable default will be to have the same number of hidden units in every single layer.
* As for the number of hidden *units*, usually, the more hidden units the better
* NOTE: If you have a lot of hidden units, it can become very computationally expensive, but very often, having more hidden units is a good thing.
* Usually the number of hidden units in each layer will be comparable to the dimensions of x/to the number of features
* Or it could be anywhere from same number of hidden units as input features to maybe 2, 3, or 4 times as much
* So having the number of hidden units several times/somewhat bigger than the number of input features is often a useful thing to do
* So, hopefully this gives you 1 reasonable set of default choices for neural architecture + if you follow these guidelines, you will probably get something that works well
* Next, here's what we need to implement in order train a NN in 6 six steps:
* 1) Set up the NN to **randomly initialize the values of the weights**.
* We usually initialize the weights to small values near 0.
* 2) **Implement** **forward propagation** **algorithm**
* So that we can input any input x into the NN + compute hϴ(x), the output vector of the y values.
* 3) **Implement code to compute a cost function J(ϴ)**.
* 4) **Implement back-propagation algorithm** to compute the partial derivatives terms, partial derivatives of J(ϴ) w/ respect to each parameter ϴ.
* Concretely, to implement FP + BP, usually we use a FOR loop over the m training examples, {(x(i),y(i)}

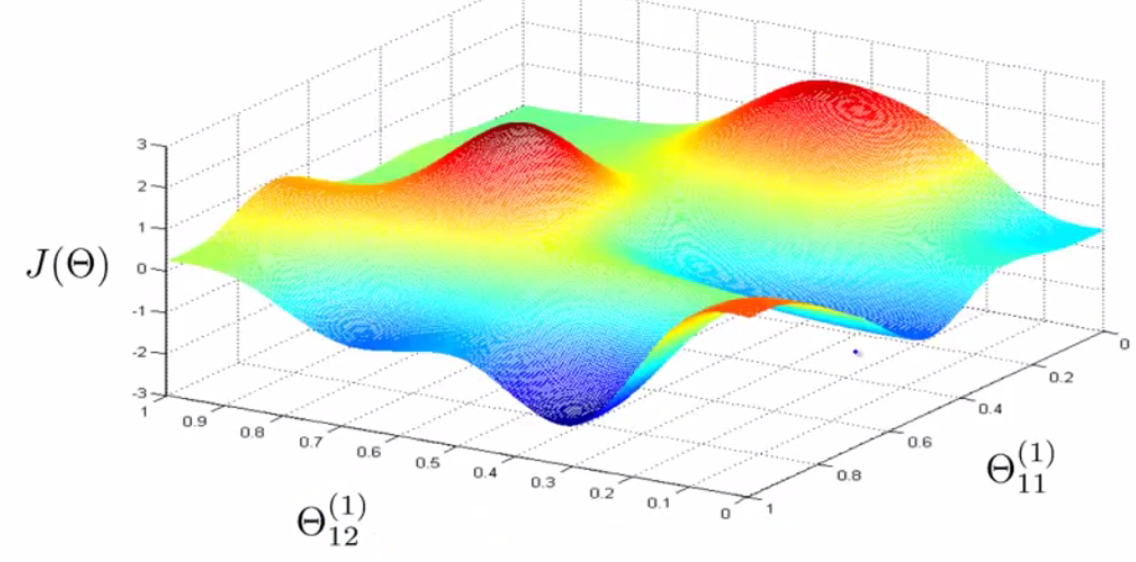


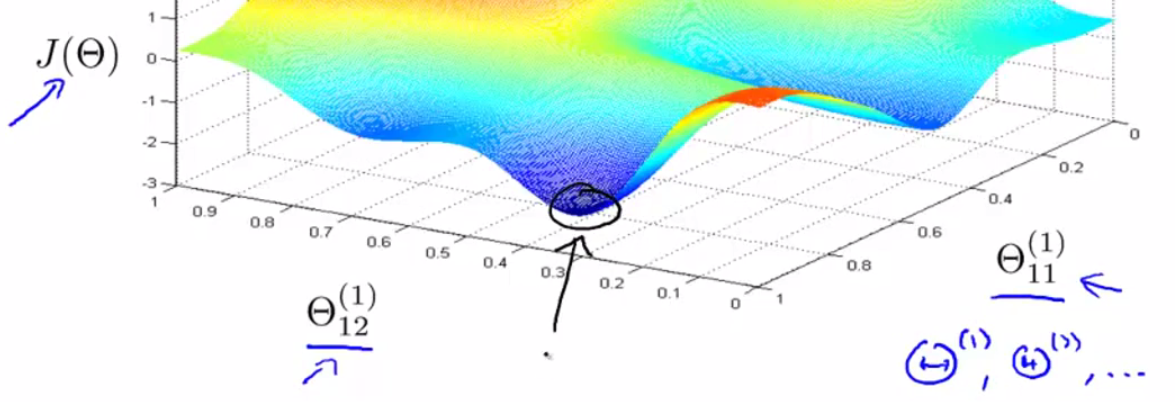
* The 1st time you're implementing BP, there should almost certainly be a FOR loop in your code, where you're iterating over the examples, starting at {(x(1),y(1)}, + do performing FP + BP
* Then in the second iteration of the FOR-loop, do FP + BP on the second example, {(x(2),y(2)}, and so on until you get through the final example, {(x(m),y(m)},
* There should be a FOR loop in your implementation of BP, at least the 1st time implementing it
* There are somewhat more complicated ways to do this w/out a FOR-loop, but I definitely do not recommend trying to do that the 1st time you try to implement BP.
* Concretely, we have a FOR-loop over m training examples + inside the FOR-loop we perform FP + BP using just this 1 example = take x(i), feed it to the input layer, perform FP, perform BP, + this gives all the activation + delta terms for all layers of all my units in the NN

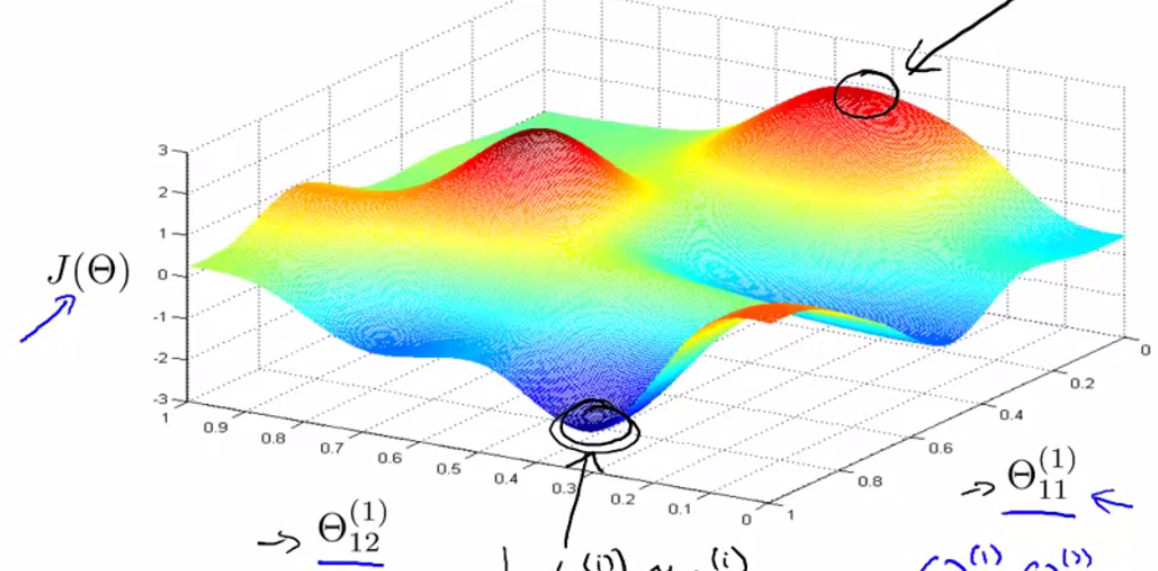


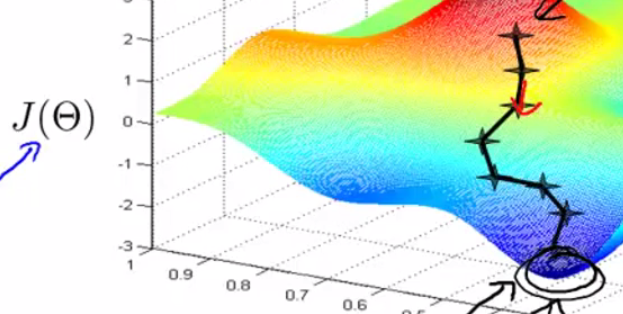
* Then, still inside the FOR-loop, we compute those other delta (Ϫ) terms



* Then finally, outside the FOR loop, having computed these **delta/accumulation terms Ϫ**, we’d have some other code to compute the partial derivative terms, which must take into account the regularization term λ as well.
* 
* 5) **Use gradient checking to compare the partial derivative terms**
* Comparing the ones just computed using BP vs. the ones computed using the numerical estimates of gradients of J(ϴ)/numerical estimates of the derivatives.
* Make sure that both of these give you very similar values.
* Doing gradient checking reassures us that our implementation of BP is correct
* *It’s then very important to disable gradient checking, b/c the gradient checking code is computationally very slow.*
* 6) **Use gradient descent or an advanced optimization algorithm such as gradient descent, or one of the advanced optimization methods together w/ BP (which computed the partial derivatives for us) to try to minimize J(ϴ) as a function of the parameters ϴ.**
* By the way, for NNs, the cost function J(ϴ) is *non-convex*, so it can theoretically be susceptible to local minima
* In fact, algorithms like gradient descent + the advanced optimization methods can, in theory, get stuck in local optima
* But it turns out that in practice, this is not usually a huge problem
* 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 J(ϴ) + getting to a very good local minimum, even if it doesn't get to the global optimum.
* Gradient descent for a NN might still seem a little bit magical, so let’s go over 1 more figure to try to get that intuition about what gradient descent for a NN is doing.
* 
* So, we have some cost function J(ϴ) + a number of parameters ϴ in our NN
* In reality, of course, in the NN, we can have lots of parameters, not just 2, + all of these are matrices so we can have very high-dimensional parameters
* But b/c of the limitations of what we can draw, pretend we have only 2 parameters in this NN.
* This J(ϴ) measures how well the NN fits the training data.



* This point is where J(ϴ) is pretty low, + this corresponds to a setting of the parameters ϴ where the output of my hypothesis hϴ(x(i)) be pretty close to y(i)
* If this is true, than that's what causes my cost function to be pretty low
* 
* In contrast, this point corresponds to where, for many training examples, the output of my NN, hϴ(x(i)), is far from the actual value y(i) that was observed in the training set.
* It’s not fitting the training set well, whereas points like this with low values of the cost function corresponds to where J(ϴ) is low
* What gradient descent does is start from some random initial point + will repeatedly go downhill



* What BP is doing is computing the *direction* of the gradient + gradient descent will take little steps downhill until hopefully it gets to a pretty good local optimum.
* So, when you implement BP + use gradient descent or 1 of the advanced optimization methods, these figures help explain what the algorithm is doing.
* i.e. They’re trying to find a value of the parameters where the output values in the NN closely matches the values of the actual y(i)'s observed in the training set.







* NN learning + back propagation are complicated algorithms, + it’s actually perfectly okay if we implement all these algorithms + optimization methods to fit very complex, powerful, non-linear functions to data w/out completely understanding them.
* NN learning is one of the most effective learning algorithms we have today.