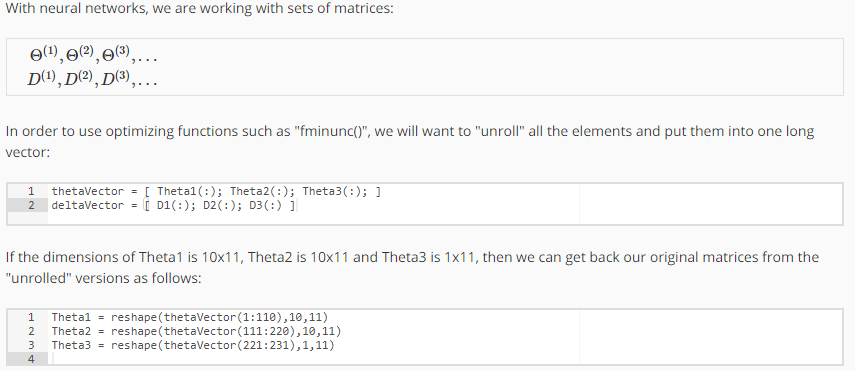
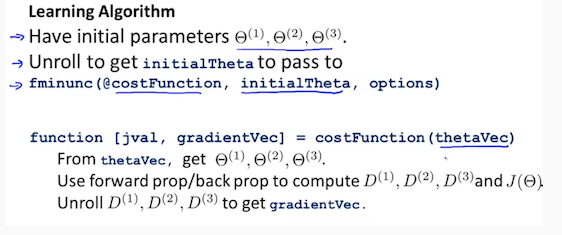
# Backpropagation in practice

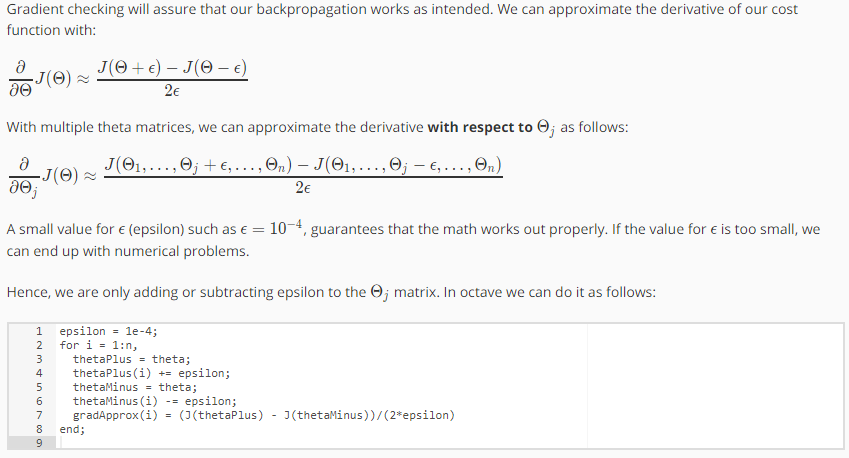
## Unrolling Parameters



To summarize:



## Gradient Checking

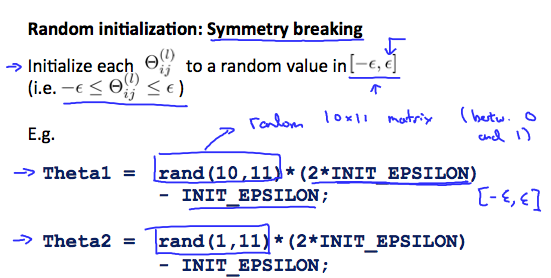


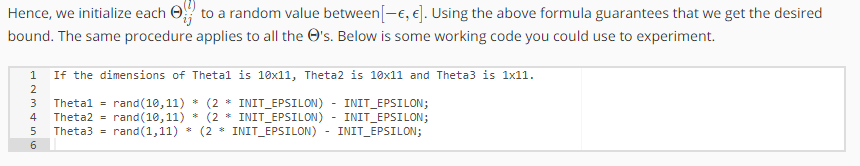
We previously saw how to calculate the deltaVector. So once we compute our gradApprox vector, we can check that gradApprox ≈ deltaVector.

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

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:





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

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.

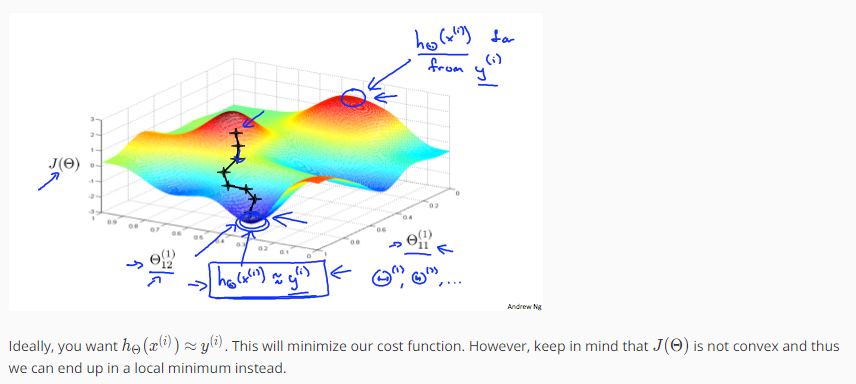
* Number of input units = dimension of features x(i)
* Number of output units = number of classes
* Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
* Defaults: 1 hidden layer. If you have more than 1 hidden layer, 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 hᶿ( 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:

The following image gives us an intuition of what is happening as we are implementing our neural network:



# Evaluating a learning algorithm

## Deciding what to try next

Suppose we are using a regularised linear regression to predict housing prices and you notice that algorithm is making unpredictable large errors in its predictions. **Now arises the question of what to try next?**

We can try below options such as

* Get more training examples
* Try smaller set of features
* Try getting additional features
* Try adding polynomial features
* Try decreasing lambda
* Try increasing lambda

But what should be the most promising avenue to choose? People often spend lot of time in getting more examples and still the algorithm doesn’t give necessary results.

First we will look into how to evaluate a hypothesis and then we will study about machine learning diagnostic.

**Machine Learning Diagnostic:** A test that you can run to gain insight what is/isn’t working with a learning algorithm and gain guidance as to how best to improve its performance.

Diagnostics can take time to implement, but doing so can be a very good use of your time.

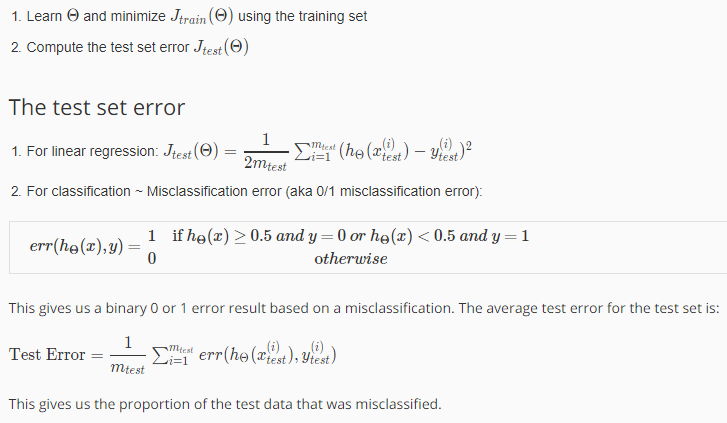
## Evaluating a hypothesis

Once we have done some trouble shooting for errors in our predictions by:

* Getting more training examples
* Trying smaller sets of features
* Trying additional features
* Trying polynomial features
* Increasing or decreasing λ

We can move on to evaluate our new hypothesis. A hypothesis may have a low error for the training examples but still be inaccurate (because of overfitting). Thus, to evaluate a hypothesis, given a dataset of training examples, we can split up the data into two sets: a training set and a test set. Typically, the training set consists of 70 % of your data and the test set is the remaining 30 %.

The new procedure using these two sets is then:



## Model Selection and Train/Validation/Test Sets

Just because a learning algorithm fits a training set well, that does not mean it is a good hypothesis. It could over fit and as a result your predictions on the test set would be poor. The error of your hypothesis as measured on the data set with which you trained the parameters will be lower than the error on any other data set.

Given many models with different polynomial degrees, we can use a systematic approach to identify the 'best' function. In order to choose the model of your hypothesis, you can test each degree of polynomial and look at the error result.

One way to break down our dataset into the three sets is:

* Training set: 60%
* Cross validation set: 20%
* Test set: 20%

We can now calculate three separate error values for the three different sets using the following method:

Optimize the parameters in Θ using the training set for each polynomial degree.

Find the polynomial degree d with the least error using the cross validation set.

Estimate the generalization error using the test set with Jtest ( theta(d) ) (d = theta from polynomial with lower error);

This way, the degree of the polynomial d has not been trained using the test set.