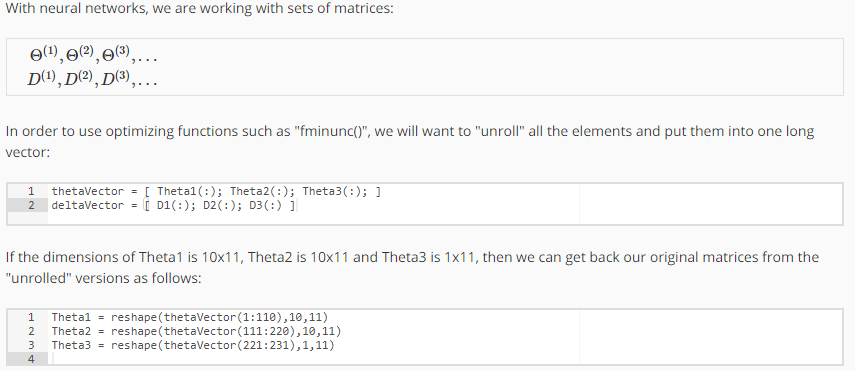
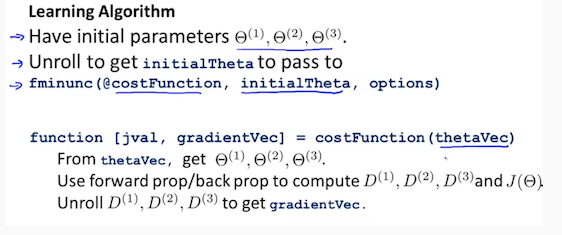
# Backpropagation in practice

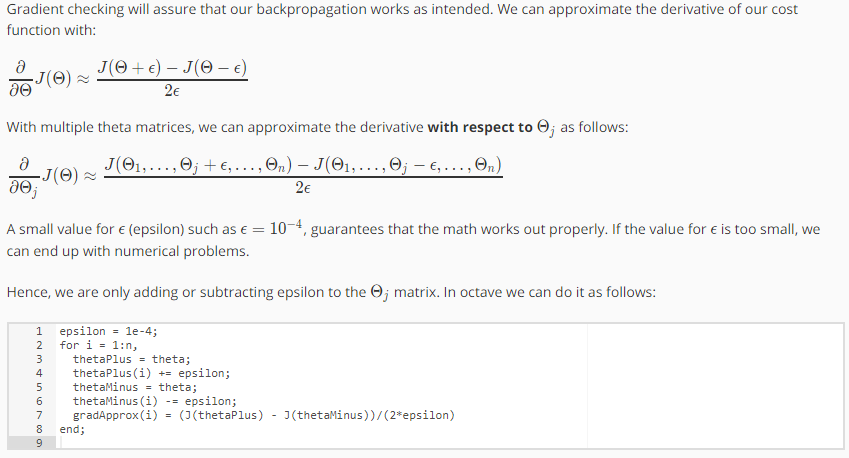
## Unrolling Parameters



To summarize:



## Gradient Checking

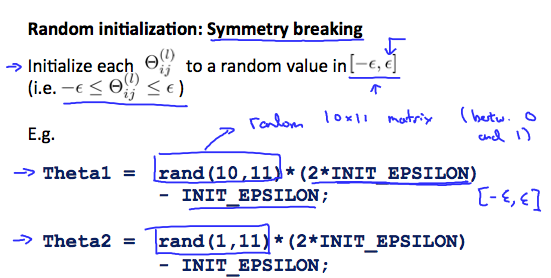


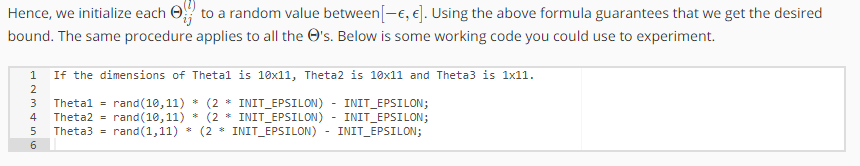
We previously saw how to calculate the deltaVector. So once we compute our gradApprox vector, we can check that gradApprox ≈ deltaVector.

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.

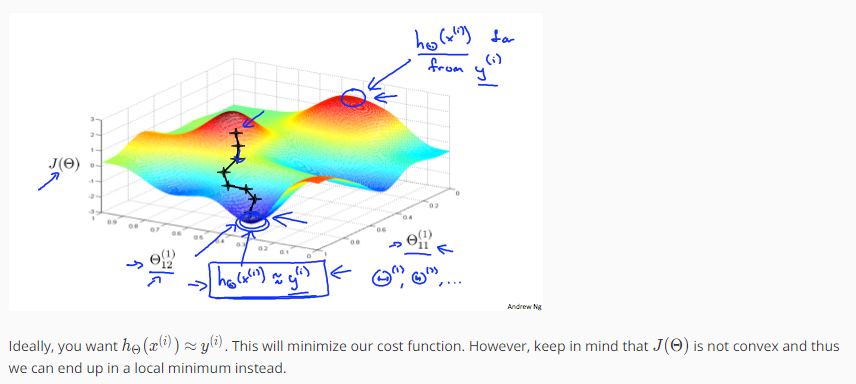
* Number of input units = dimension of features x(i)
* Number of output units = number of classes
* Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
* Defaults: 1 hidden layer. If you have more than 1 hidden layer,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

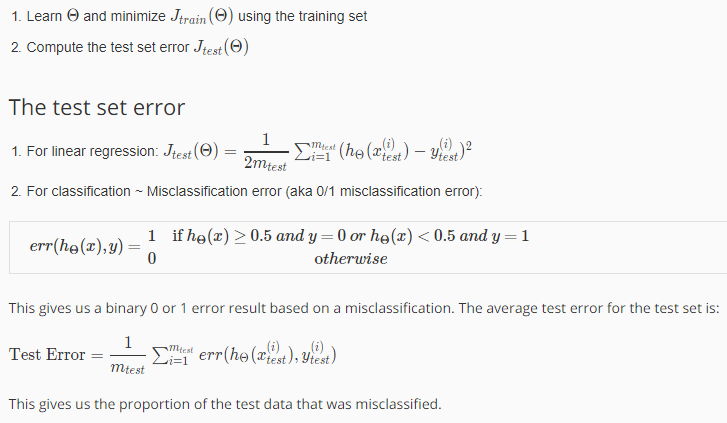
## Evaluating a hypothesis

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.

# Bias vs Variance

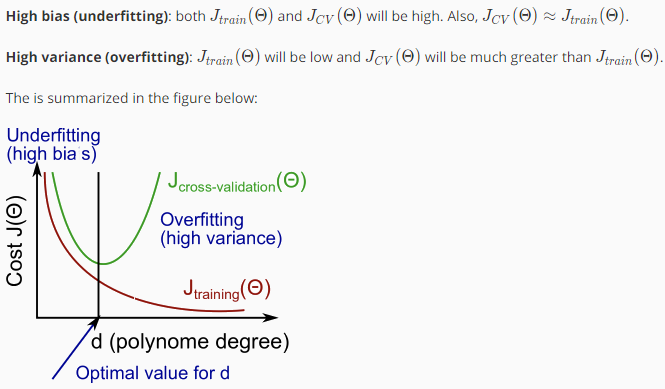
## Diagnosing Bias vs Variance

In this section we examine the relationship between the degree of the polynomial d and the underfitting or overfitting of our hypothesis.

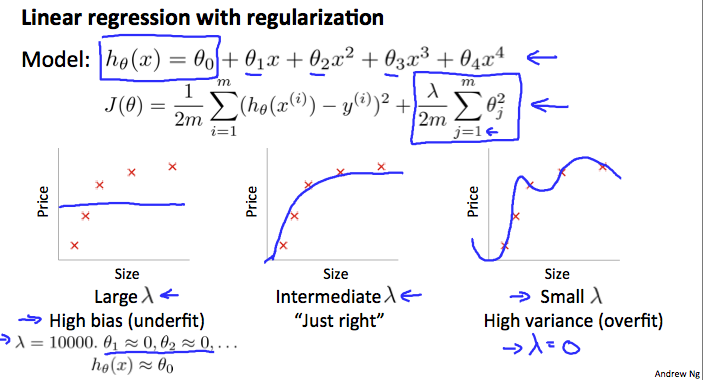
* We need to distinguish whether **bias or variance** is the problem contributing to bad predictions.
* High bias is underfitting and high variance is overfitting. Ideally, we need to find a golden mean between these two.

The training error will tend to **decrease** as we increase the degree d of the polynomial.

At the same time, the cross validation error will tend to **decrease** as we increase d up to a point, and then it will **increase** as d is increased, forming a convex curve.



## Regularisation and Bias/Variance



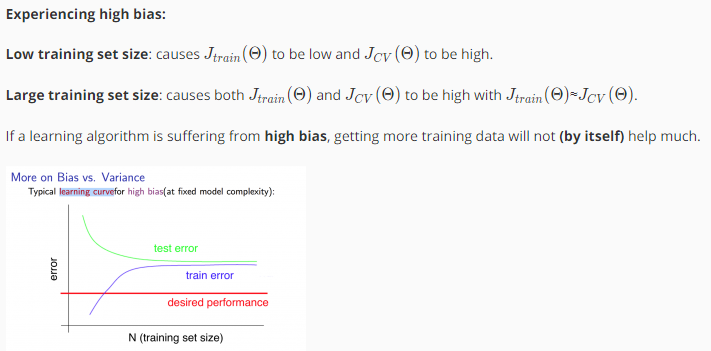
In the figure above, we see that as λ increases, our fit becomes more rigid. On the other hand, as λ approaches 0, we tend to over overfit the data. So how do we choose our parameter λ to get it 'just right'? In order to choose the model and the regularization term λ, we need to:

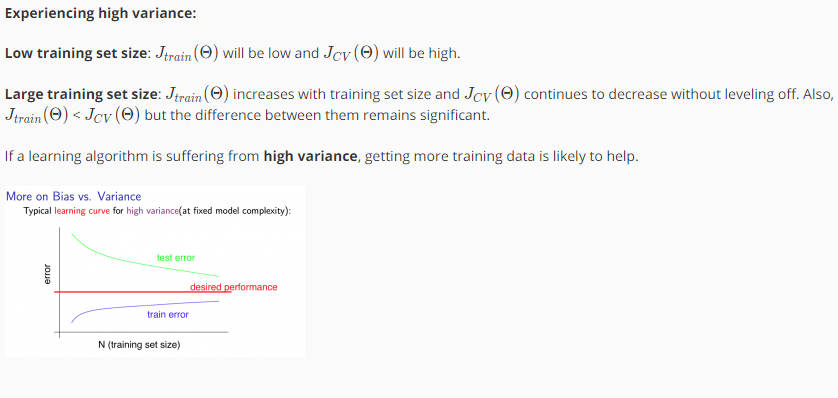
1. Create a list of lambdas (i.e. λ∈{0,0.01,0.02,0.04,0.08,0.16,0.32,0.64,1.28,2.56,5.12,10.24});
2. Create a set of models with different degrees or any other variants.
3. Iterate through the λs and for each λ go through all the models to learn some Θ.
4. Compute the cross validation error using the learned Θ (computed with λ) on the JCV​(Θ) without regularization or λ = 0.
5. Select the best combo that produces the lowest error on the cross validation set.
6. Using the best combo Θ and λ, apply it on Jtest​(Θ) to see if it has a good generalization of the problem.

## Learning Curves

Training an algorithm on a very few number of data points (such as 1, 2 or 3) will easily have 0 errors because we can always find a quadratic curve that touches exactly those number of points. Hence:

1. As the training set gets larger, the error for a quadratic function increases.
2. The error value will plateau out after a certain m, or training set size.





## Deciding what to do next revisited

Our decision process can be broken down as follows:

* **Getting more training examples:** Fixes high variance
* **Trying smaller sets of features:** Fixes high variance
* **Adding features:** Fixes high bias
* **Adding polynomial features:** Fixes high bias
* **Decreasing λ:** Fixes high bias
* **Increasing λ:** Fixes high variance.

**Diagnosing Neural Networks**

A neural network with fewer parameters is **prone to underfitting**. It is also **computationally cheaper**.

A large neural network with more parameters is **prone to overfitting**. It is also **computationally expensive**. In this case you can use regularization (increase λ) to address the overfitting.

Note: - Using a single hidden layer is a good starting default. You can train your neural network on a number of hidden layers using your cross validation set. You can then select the one that performs best.

**Model Complexity Effects:**

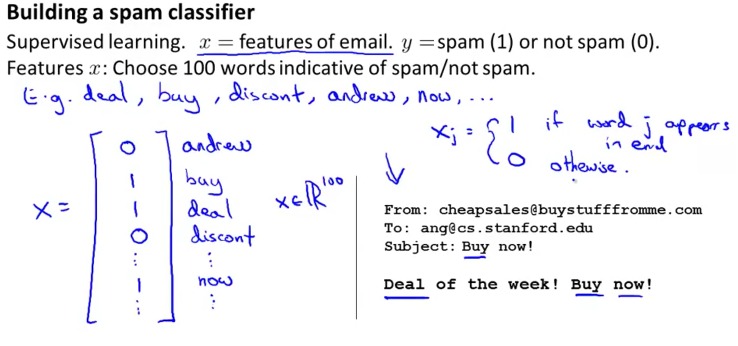
* Lower-order polynomials (low model complexity) have high bias and low variance. In this case, the model fits poorly consistently.
* Higher-order polynomials (high model complexity) fit the training data extremely well and the test data extremely poorly. These have low bias on the training data, but very high variance.
* In reality, we would want to choose a model somewhere in between, that can generalize well but also fits the data reasonably well.

# Building a Spam Classifier

## Prioritizing what to work on

**System Design Example:**

Given a data set of emails, we could construct a vector for each email. Each entry in this vector represents a word. The vector normally contains 10,000 to 50,000 entries gathered by finding the most frequently used words in our data set. If a word is to be found in the email, we would assign its respective entry a 1, else if it is not found, that entry would be a 0. Once we have all our x vectors ready, we train our algorithm and finally, we could use it to classify if an email is a spam or not.



**So how could you spend your time to improve the accuracy of this classifier?**

* Collect lots of data (for example "honeypot" project but doesn't always work)
* Develop sophisticated features (for example: using email header data in spam emails)
* Develop algorithms to process your input in different ways (recognizing misspellings in spam).

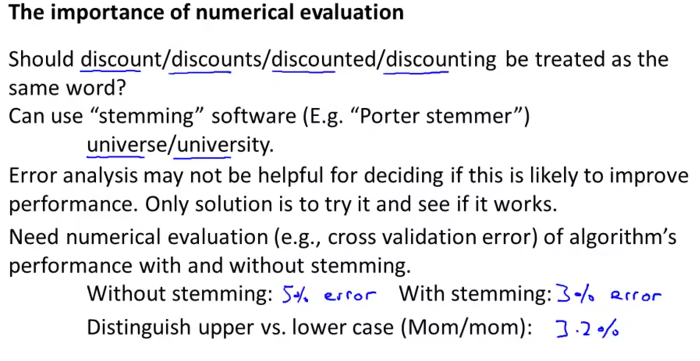
It is difficult to tell which of the options will be most helpful.

## Error Analysis

The recommended approach to solving machine learning problems is to:

* Start with a simple algorithm, implement it quickly, and test it early on your cross validation data.
* Plot learning curves to decide if more data, more features, etc. are likely to help.
* Manually examine the errors on examples in the cross validation set and try to spot a trend where most of the errors were made.

For example, assume that we have 500 emails and our algorithm misclassifies a 100 of them. We could manually analyse the 100 emails and categorize them based on what type of emails they are. We could then try to come up with new cues and features that would help us classify these 100 emails correctly. Hence, if most of our misclassified emails are those which try to steal passwords, then we could find some features that are particular to those emails and add them to our model. We could also see how classifying each word according to its root changes our error rate:



It is very important to get error results as a single, numerical value. Otherwise it is difficult to assess your algorithm's performance. For example if we use stemming, which is the process of treating the same word with different forms (fail/failing/failed) as one word (fail), and get a 3% error rate instead of 5%, then we should definitely add it to our model. However, if we try to distinguish between upper case and lower case letters and end up getting a 3.2% error rate instead of 3%, then we should avoid using this new feature. Hence, we should try new things, get a numerical value for our error rate, and based on our result decide whether we want to keep the new feature or not.