目录

[Gradient Descent For Linear Regression 2](#_Toc482262906)

[Gradient Descent in Practice I - Feature Scaling 3](#_Toc482262907)

[Gradient Descent in Practice II - Learning Rate 4](#_Toc482262908)

[Features and Polynomial Regression 5](#_Toc482262909)

[Polynomial Regression 5](#_Toc482262910)

[Normal Equation 6](#_Toc482262911)

[Normal Equation Noninvertibility 6](#_Toc482262912)

[Cost Function 7](#_Toc482262913)

[Simplified Cost Function and Gradient Descent 7](#_Toc482262914)

[Advanced Optimization 8](#_Toc482262915)

[Multiclass Classification: One-vs-all 9](#_Toc482262916)

[The Problem of Overfitting 9](#_Toc482262917)

[Cost Function 10](#_Toc482262918)

[Regularized Linear Regression 11](#_Toc482262919)

[Regularized Logistic Regression 12](#_Toc482262920)

[Model Representation I 13](#_Toc482262921)

[Model Representation II 15](#_Toc482262922)

[Examples and Intuitions I 16](#_Toc482262923)

[Examples and Intuitions II 18](#_Toc482262924)

[Multiclass Classification 19](#_Toc482262925)

[Cost Function 20](#_Toc482262926)

[Backpropagation Algorithm 21](#_Toc482262927)

[Backpropagation Intuition 23](#_Toc482262928)

[Implementation Note: Unrolling Parameters 24](#_Toc482262929)

[Gradient Checking 25](#_Toc482262930)

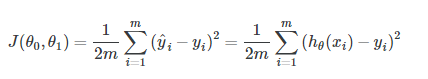
[Random Initialization 25](#_Toc482262931)

[Putting it Together 26](#_Toc482262932)

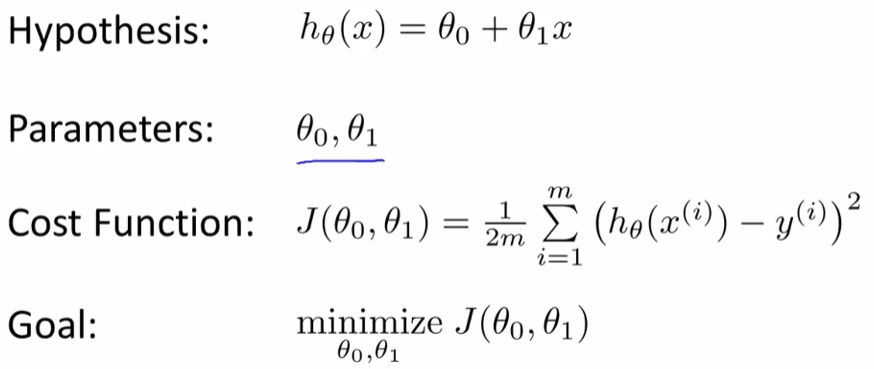
Tom Mitchell provides a more modern definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

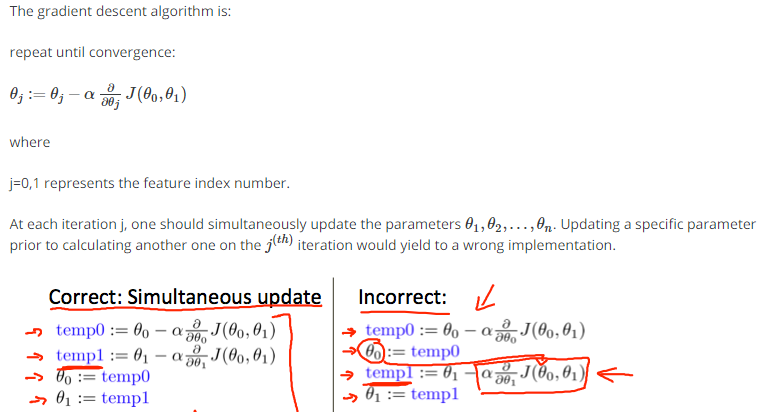
Supervised learning problems are categorized into "regression" and "classification" problems.

We can measure the accuracy of our hypothesis function by using a **cost function**.



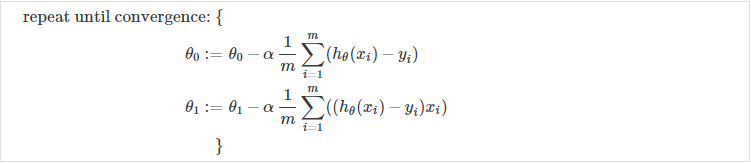
This function is otherwise called the "Squared error function", or "Mean squared error". The mean is halved (12)as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the 12 term.

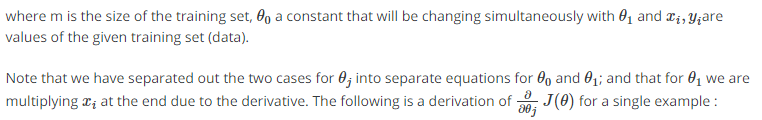


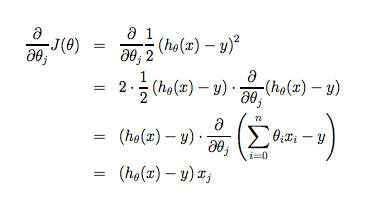


# Gradient Descent For Linear Regression

When specifically applied to the case of linear regression, a new form of the gradient descent equation can be derived. We can substitute our actual cost function and our actual hypothesis function and modify the equation to :

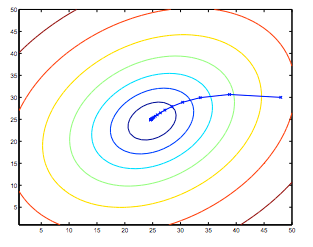






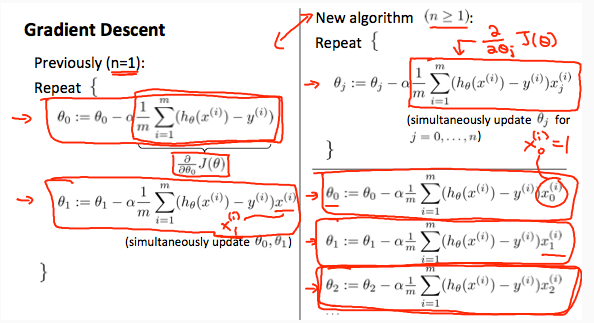
The point of all this is that if we start with a guess for our hypothesis and then repeatedly apply these gradient descent equations, our hypothesis will become more and more accurate.

So, this is simply gradient descent on the original cost function J. This method looks at every example in the entire training set on every step, and is called **batch gradient descent**. Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima; thus gradient descent always converges (assuming the learning rate α is not too large) to the global minimum. Indeed, J is a convex quadratic function. Here is an example of gradient descent as it is run to minimize a quadratic function.



The ellipses shown above are the contours of a quadratic function. Also shown is the trajectory taken by gradient descent, which was initialized at (48,30). The x’s in the figure (joined by straight lines) mark the successive values of θ that gradient descent went through as it converged to its minimum.

**Multiple Variables**



# Gradient Descent in Practice I - Feature Scaling

**Note:** [6:20 - The average size of a house is 1000 but 100 is accidentally written instead]

We can speed up gradient descent by having each of our input values in roughly the same range. This is because θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven.

The way to prevent this is to modify the ranges of our input variables so that they are all roughly the same. Ideally:

−1 ≤ *x*(*i*) ≤ 1

or

−0.5 ≤ *x*(*i*) ≤ 0.5

These aren't exact requirements; we are only trying to speed things up. The goal is to get all input variables into roughly one of these ranges, give or take a few.

Two techniques to help with this are **feature scaling** and **mean normalization**. Feature scaling involves dividing the input values by the range (i.e. the maximum value minus the minimum value) of the input variable, resulting in a new range of just 1. Mean normalization involves subtracting the average value for an input variable from the values for that input variable resulting in a new average value for the input variable of just zero. To implement both of these techniques, adjust your input values as shown in this formula:

*xi*:=(*xi*−*μi) / si*

Where *μi* is the **average** of all the values for feature (i) and *si* is the range of values (max - min), or *si* is the standard deviation.

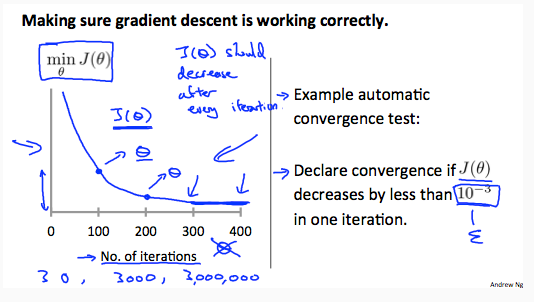
Note that dividing by the range, or dividing by the standard deviation, give different results. The quizzes in this course use range - the programming exercises use standard deviation.

For example, if *xi* represents housing prices with a range of 100 to 2000 and a mean value of 1000, then, *xi*:=(*price*−1000)/1900.

# Gradient Descent in Practice II - Learning Rate

**Debugging gradient descent.** Make a plot with number of iterations on the x-axis. Now plot the cost function, J(θ) over the number of iterations of gradient descent. If J(θ) ever increases, then you probably need to decrease α.

**Automatic convergence test.** Declare convergence if J(θ) decreases by less than E in one iteration, where E is some small value such as 10−3. However in practice it's difficult to choose this threshold value.



It has been proven that if learning rate α is sufficiently small, then J(θ) will decrease on every iteration.

# Features and Polynomial Regression

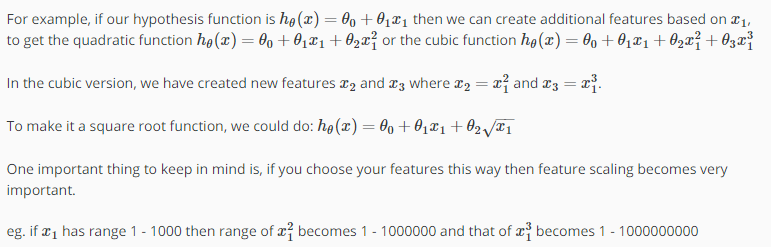
We can improve our features and the form of our hypothesis function in a couple different ways.

We can **combine** multiple features into one. For example, we can combine *x*1 and *x*2 into a new feature *x*3 by taking *x*1⋅*x*2.

### ****Polynomial Regression****

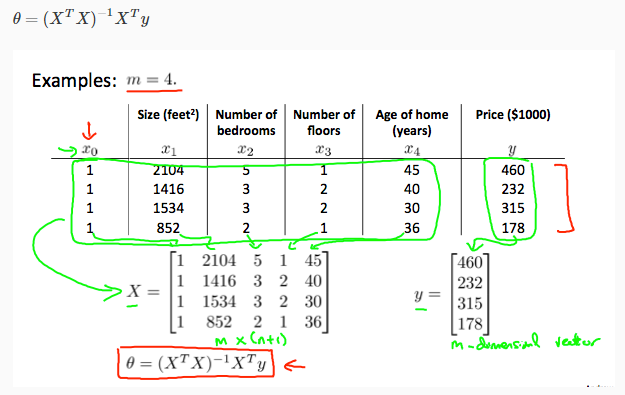
Our hypothesis function need not be linear (a straight line) if that does not fit the data well.

We can **change the behavior or curve** of our hypothesis function by making it a quadratic, cubic or square root function (or any other form).



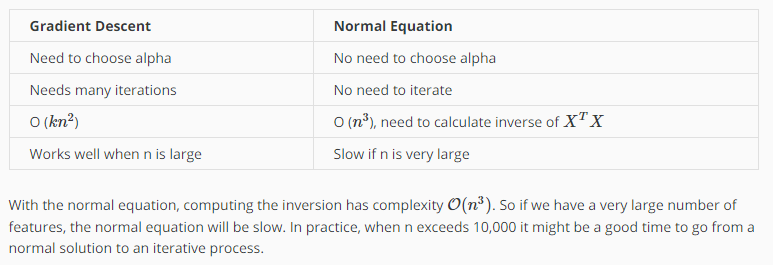
# Normal Equation

Gradient descent gives one way of minimizing J. Let’s discuss a second way of doing so, this time performing the minimization explicitly and without resorting to an iterative algorithm. In the "Normal Equation" method, we will minimize J by explicitly taking its derivatives with respect to the θj ’s, and setting them to zero. This allows us to find the optimum theta without iteration. The normal equation formula is given below:



There is **no need** to do feature scaling with the normal equation.

The following is a comparison of gradient descent and the normal equation:



# Normal Equation Noninvertibility

When implementing the normal equation in octave we want to use the 'pinv' function rather than 'inv.' The 'pinv' function will give you a value of *θ* even if *XTX* is not invertible.

If *XTX* is **noninvertible,** the common causes might be having :

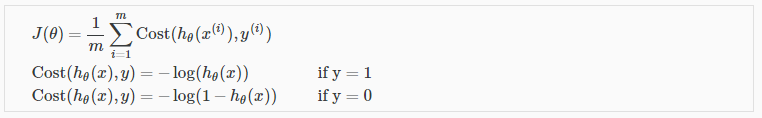
* Redundant features, where two features are very closely related (i.e. they are linearly dependent)
* Too many features (e.g. m ≤ n). In this case, delete some features or use "regularization" (to be explained in a later lesson).

Solutions to the above problems include deleting a feature that is linearly dependent with another or deleting one or more features when there are too many features.

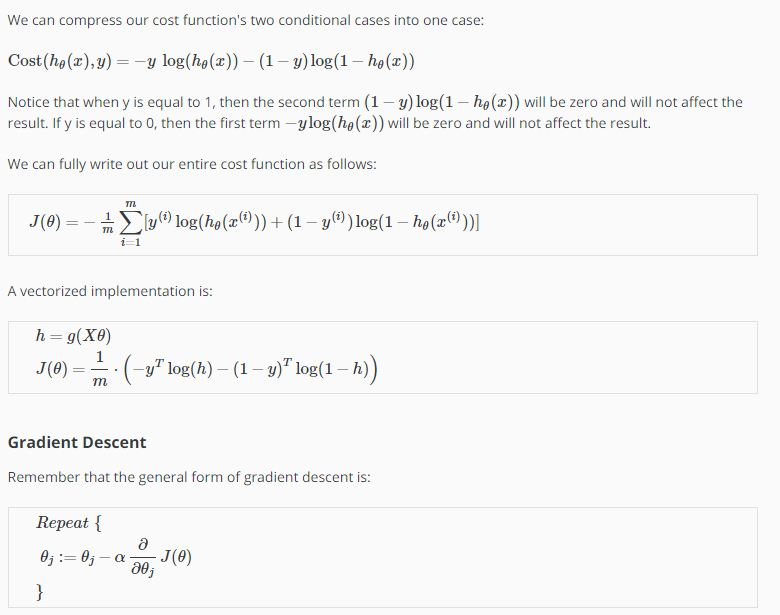
# Cost Function

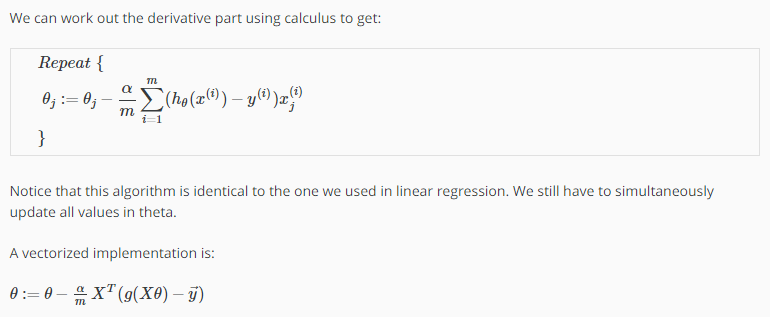
We cannot use the same cost function that we use for linear regression because the Logistic Function will cause the output to be wavy, causing many local optima. In other words, it will not be a convex function.

Instead, our cost function for logistic regression looks like:

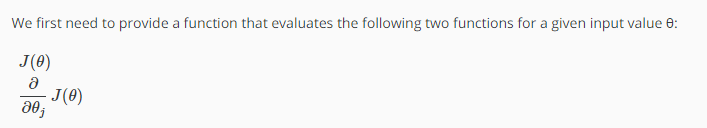


# Simplified Cost Function and Gradient Descent





# Advanced Optimization



We can write a single function that returns both of these:

function [jVal, gradient] = costFunction(theta)

jVal = [...code to compute J(theta)...];

gradient = [...code to compute derivative of J(theta)...];

end

Then we can use octave's "fminunc()" optimization algorithm along with the "optimset()" function that creates an object containing the options we want to send to "fminunc()". (Note: the value for MaxIter should be an integer, not a character string - errata in the video at 7:30)

options = optimset('GradObj', 'on', 'MaxIter', 100);

initialTheta = zeros(2,1);

[optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialTheta, options);

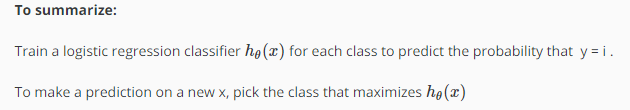
# Multiclass Classification: One-vs-all

Now we will approach the classification of data when we have more than two categories. Instead of y = {0,1} we will expand our definition so that y = {0,1...n}.

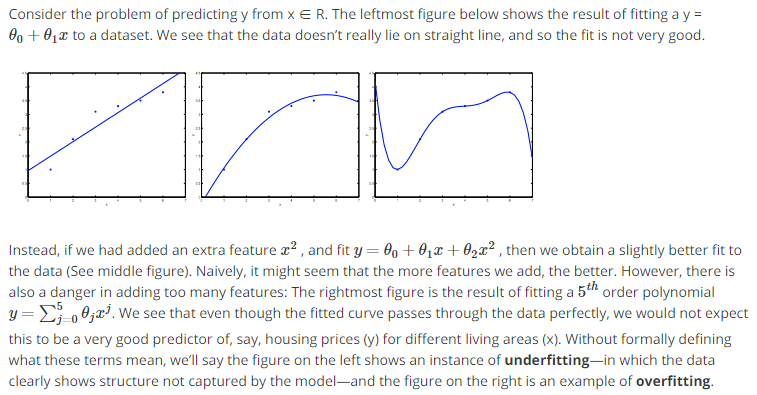
Since y = {0,1...n}, we divide our problem into n+1 (+1 because the index starts at 0) binary classification problems; in each one, we predict the probability that 'y' is a member of one of our classes.



We are basically choosing one class and then lumping all the others into a single second class. We do this repeatedly, applying binary logistic regression to each case, and then use the hypothesis that returned the highest value as our prediction.



# The Problem of Overfitting



Underfitting, or high bias, is when the form of our hypothesis function h maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features. At the other extreme, overfitting, or high variance, is caused by a hypothesis function that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.

This terminology is applied to both linear and logistic regression. There are two main options to address the issue of overfitting:

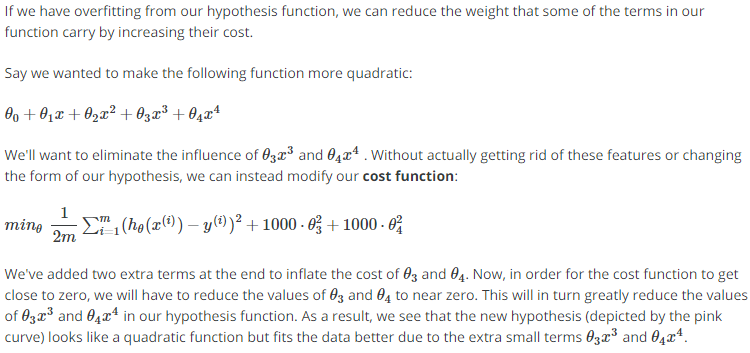
1) Reduce the number of features:

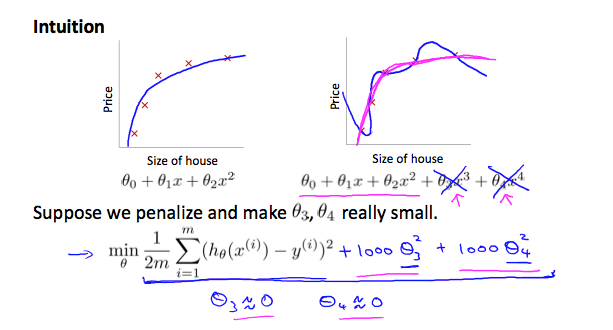
* Manually select which features to keep.
* Use a model selection algorithm (studied later in the course).

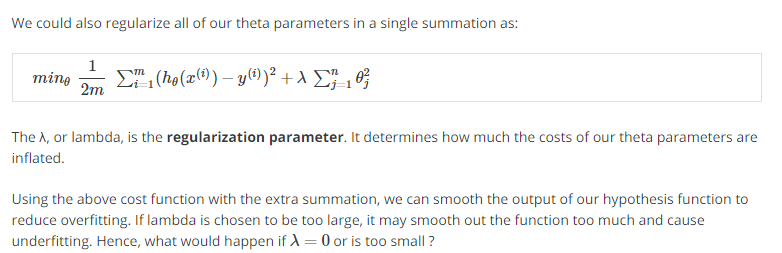
2) Regularization

* Keep all the features, but reduce the magnitude of parameters *θj*.
* Regularization works well when we have a lot of slightly useful features.

# Cost Function

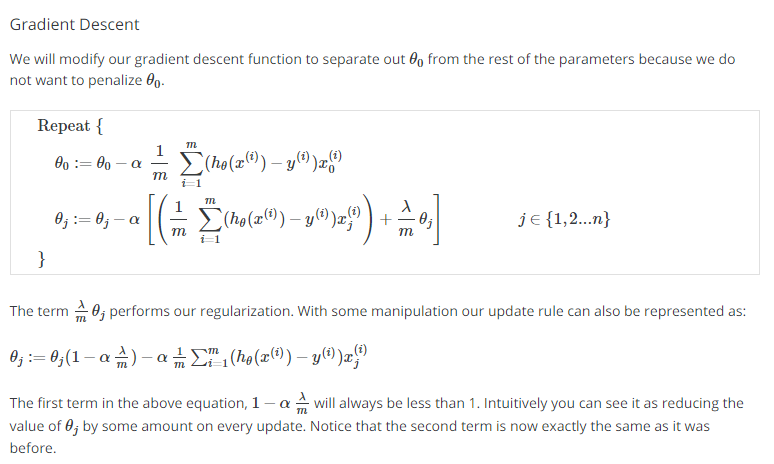


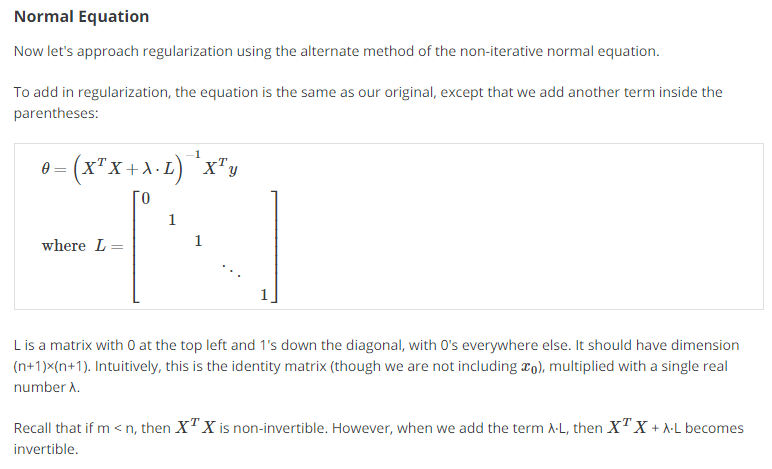




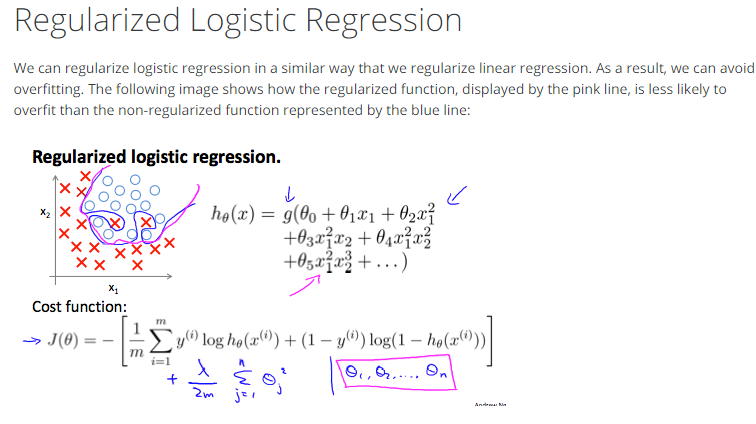
# Regularized Linear Regression

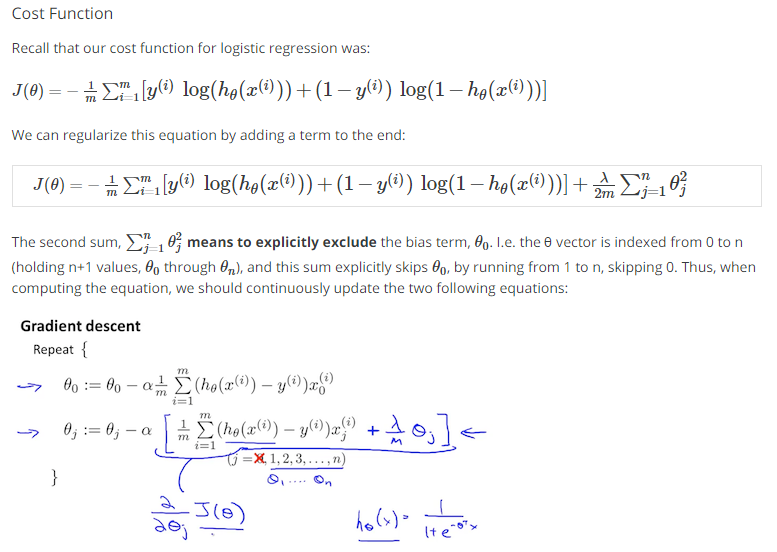
We can apply regularization to both linear regression and logistic regression. We will approach linear regression first.



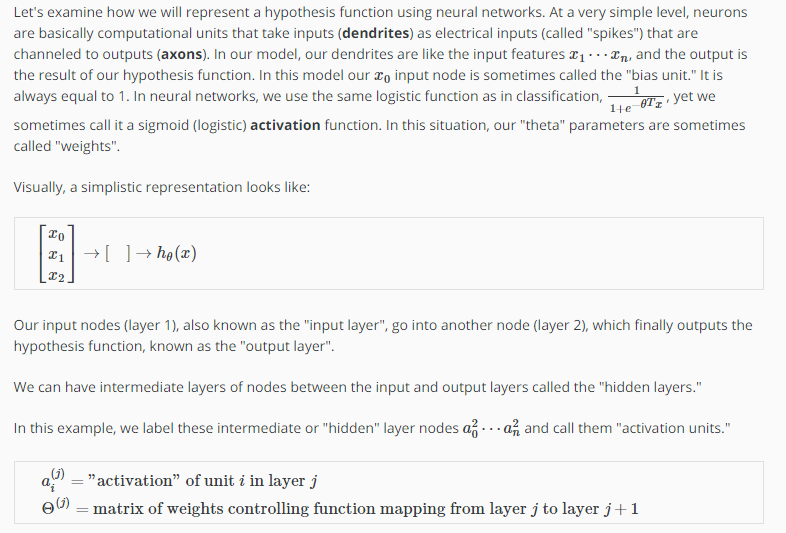


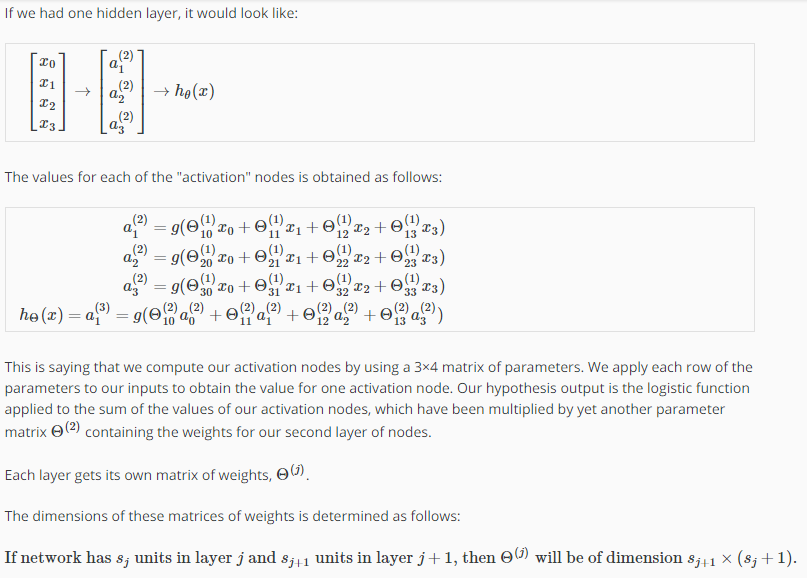
# Regularized Logistic Regression

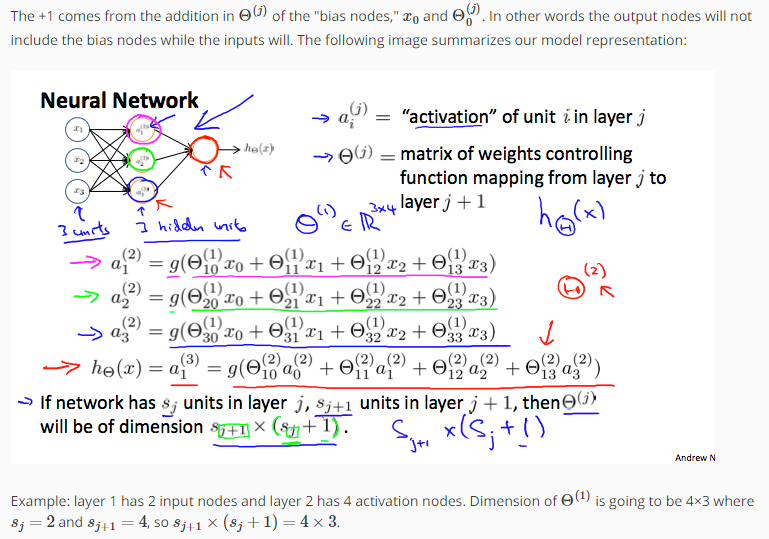




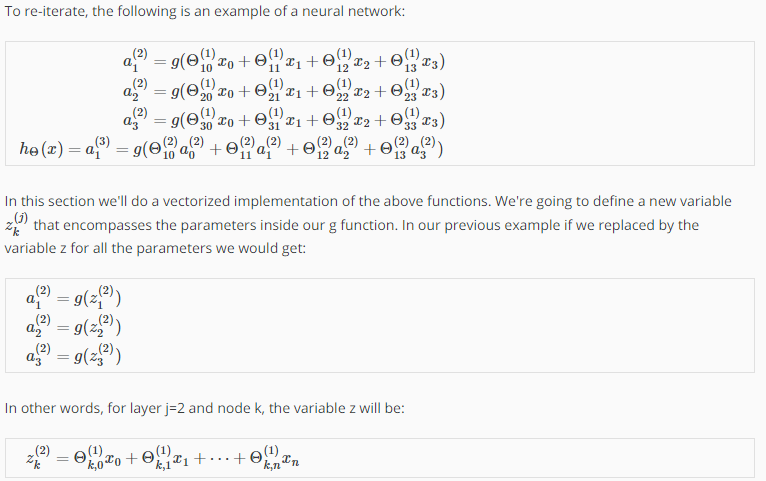
# Model Representation I

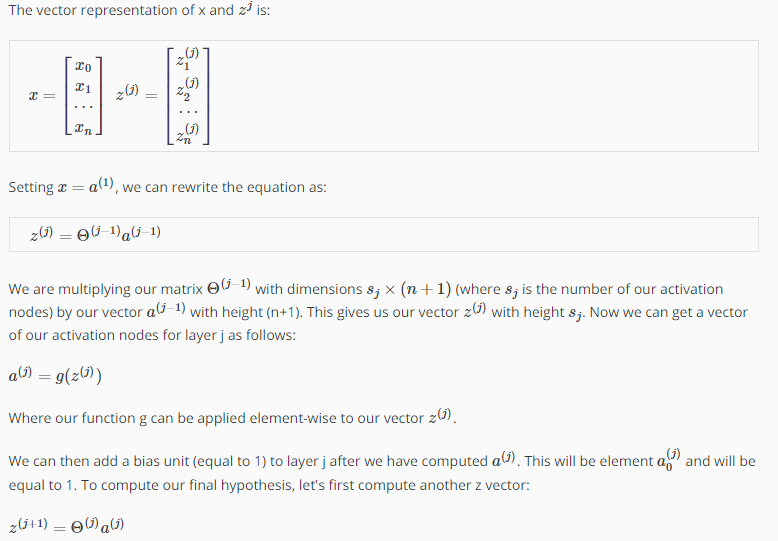


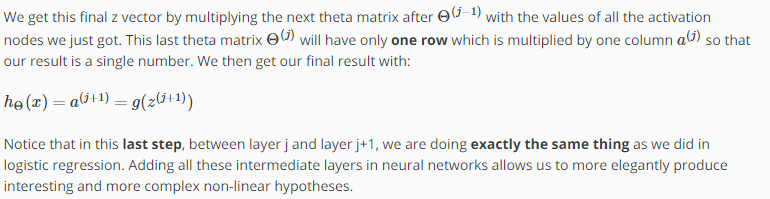




# Model Representation II

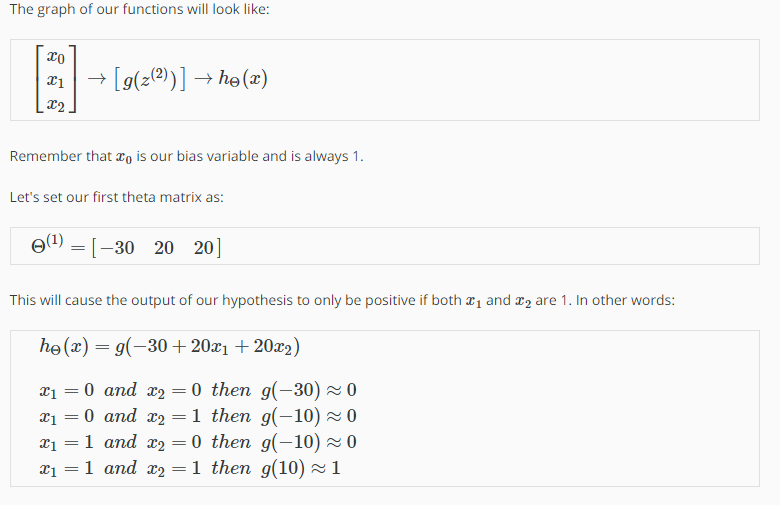






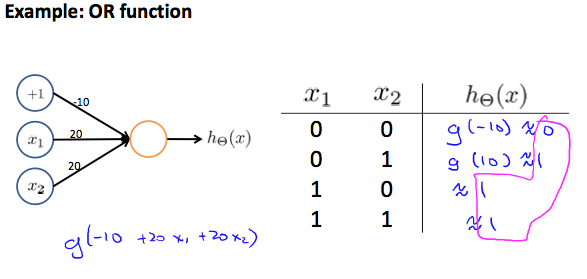
# Examples and Intuitions I

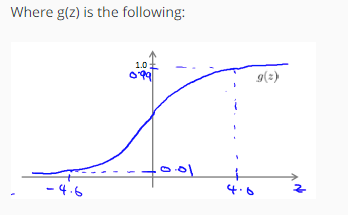
A simple example of applying neural networks is by predicting *x*1 AND *x*2, which is the logical 'and' operator and is only true if both *x*1 and *x*2 are 1.



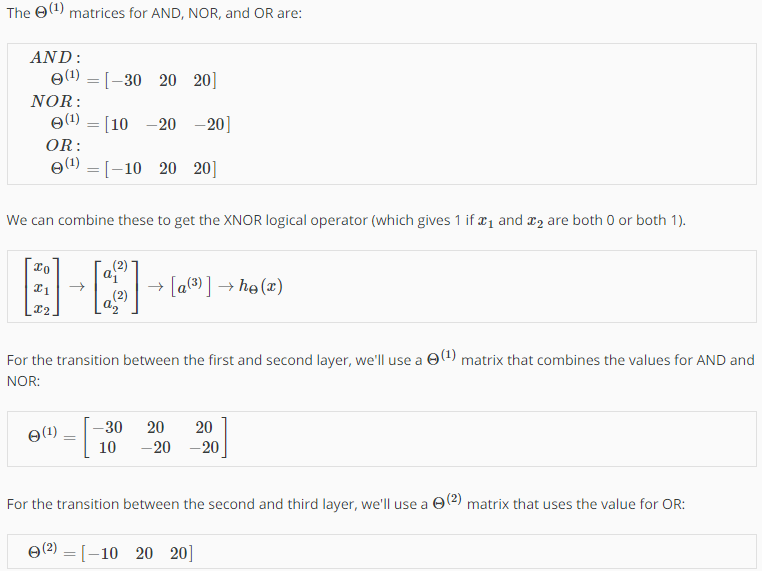
So we have constructed o

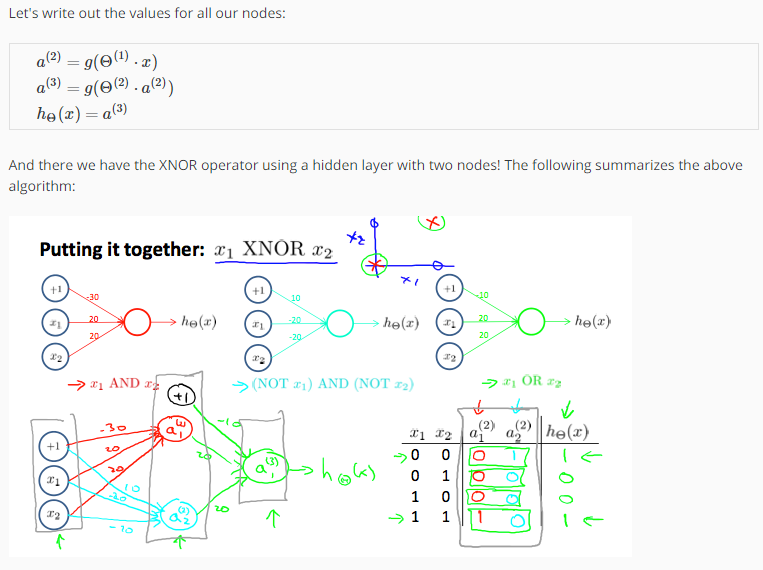
ne of the fundamental operations in computers by using a small neural network rather than using an actual AND gate. Neural networks can also be used to simulate all the other logical gates. The following is an example of the logical operator 'OR', meaning either *x*1 is true or *x*2 is true, or both:





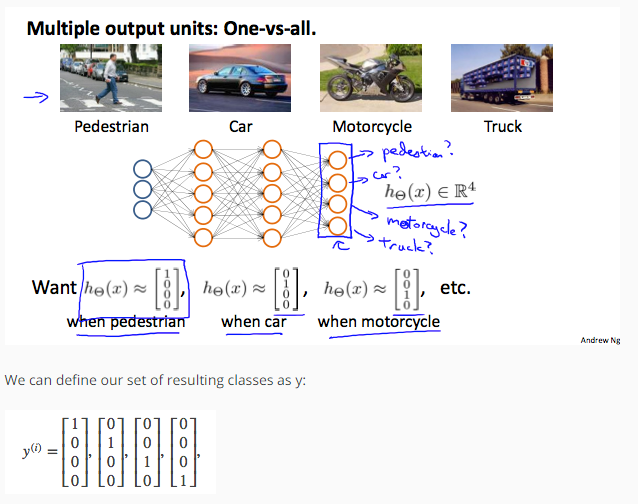
# Examples and Intuitions II

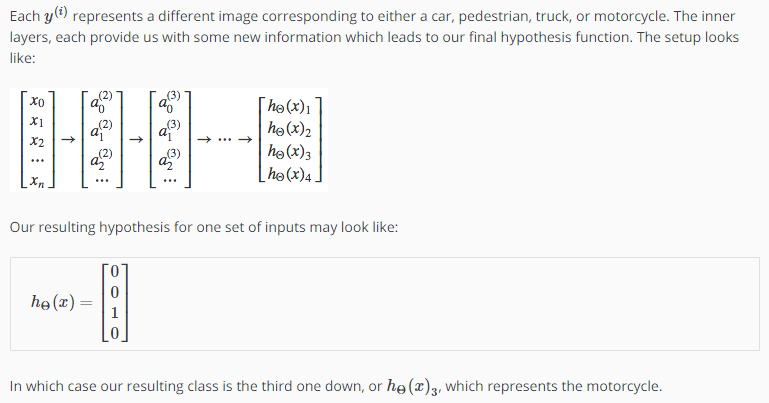




# Multiclass Classification

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:

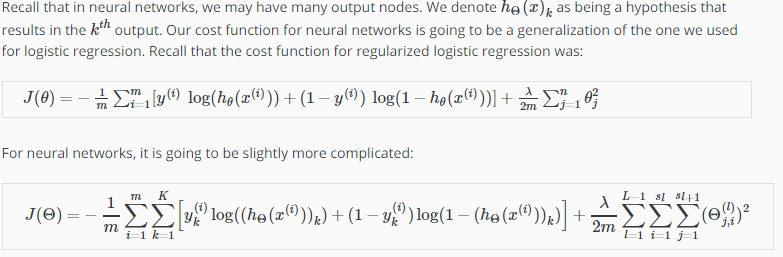




# Cost Function

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

* L = total number of layers in the network
* *sl* = number of units (not counting bias unit) in layer l
* K = number of output units/classes



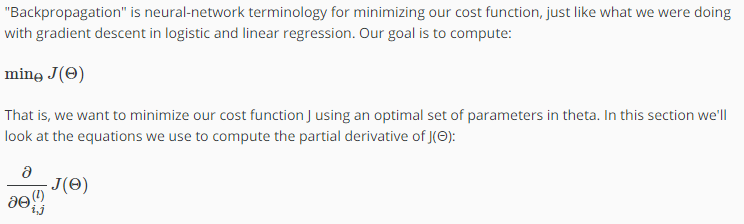
We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.

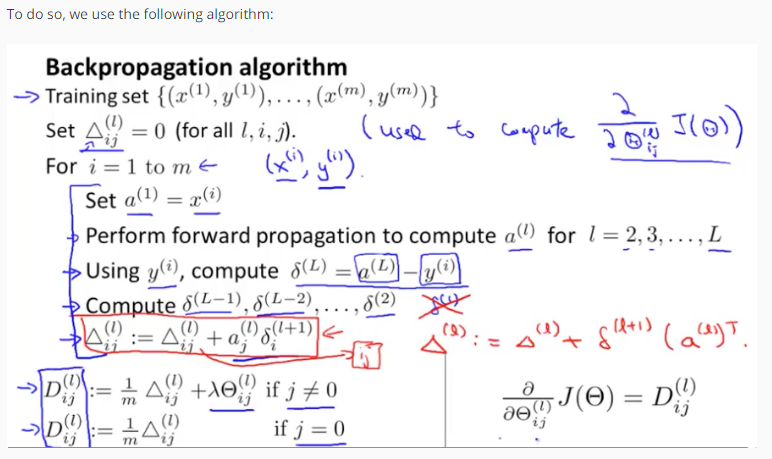
In the regularization part, after the square brackets, we must account for multiple theta matrices. The number of columns in our current theta matrix is equal to the number of nodes in our current layer (including the bias unit). The number of rows in our current theta matrix is equal to the number of nodes in the next layer (excluding the bias unit). As before with logistic regression, we square every term.

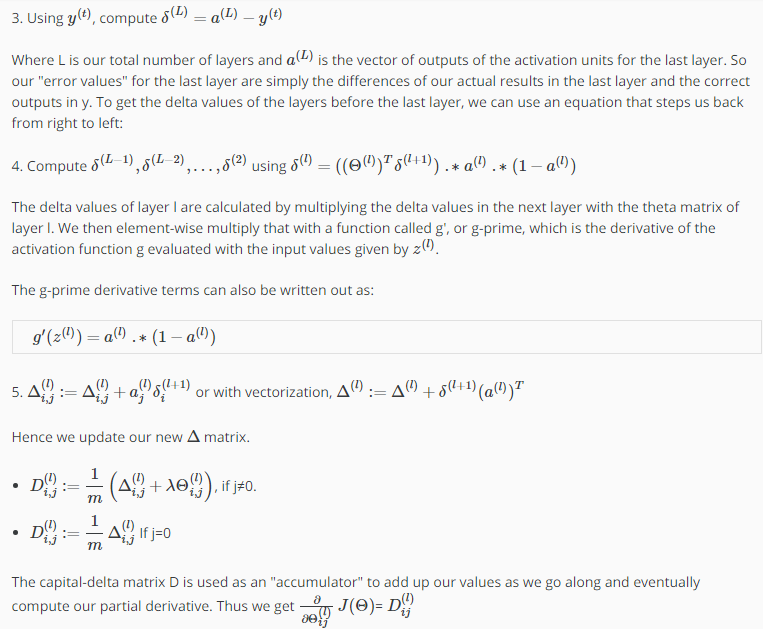
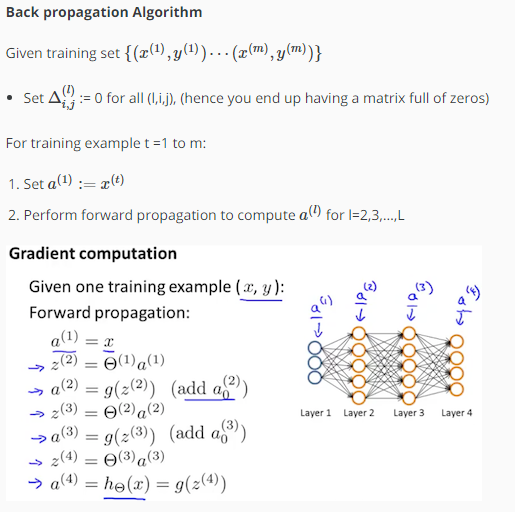
Note:

* 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

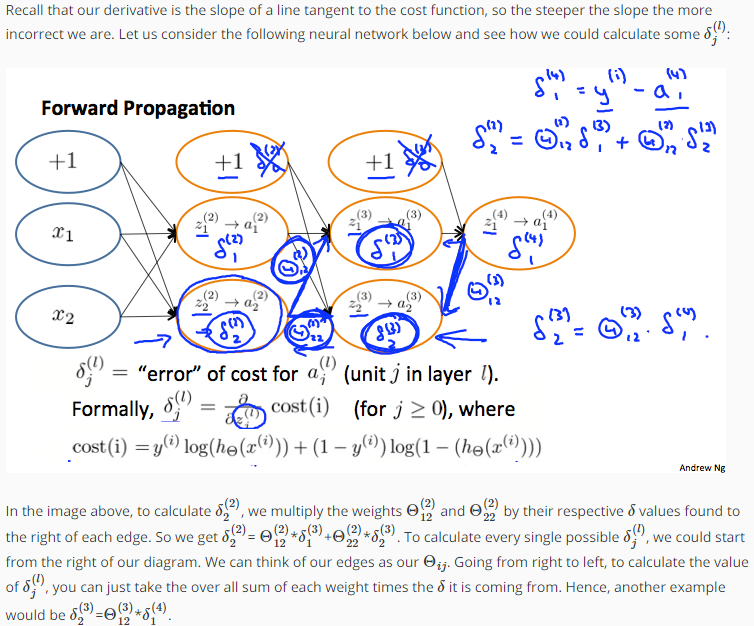
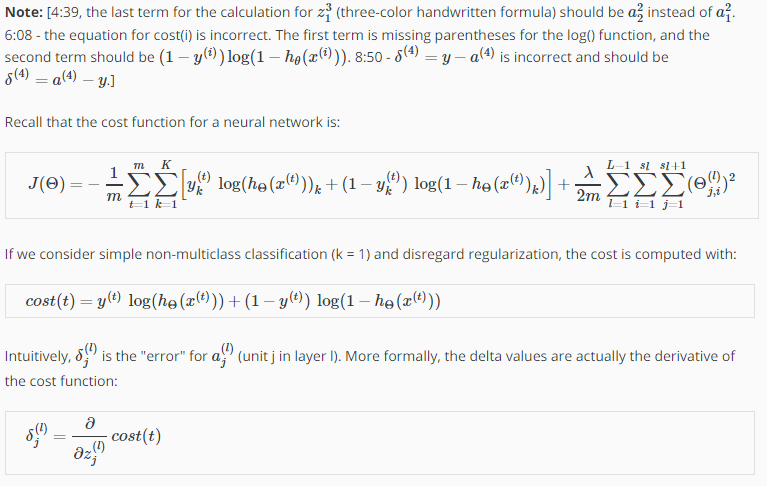
# Backpropagation Algorithm



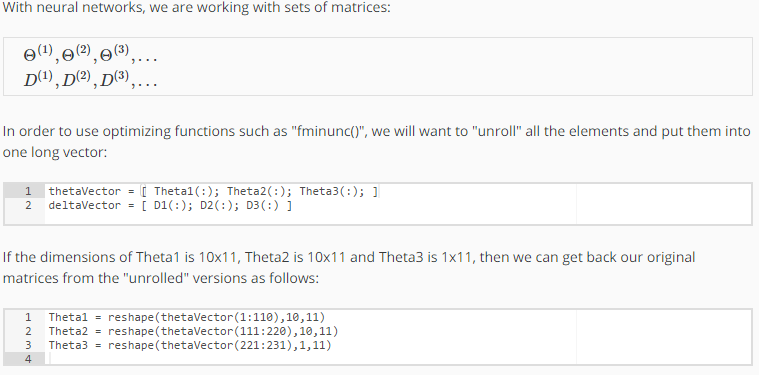


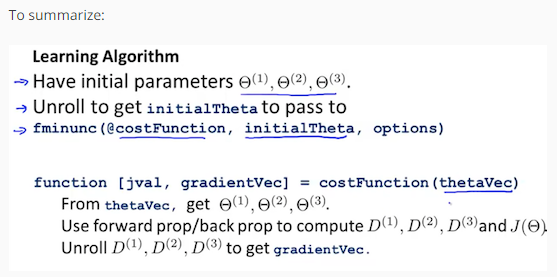


# Backpropagation Intuition

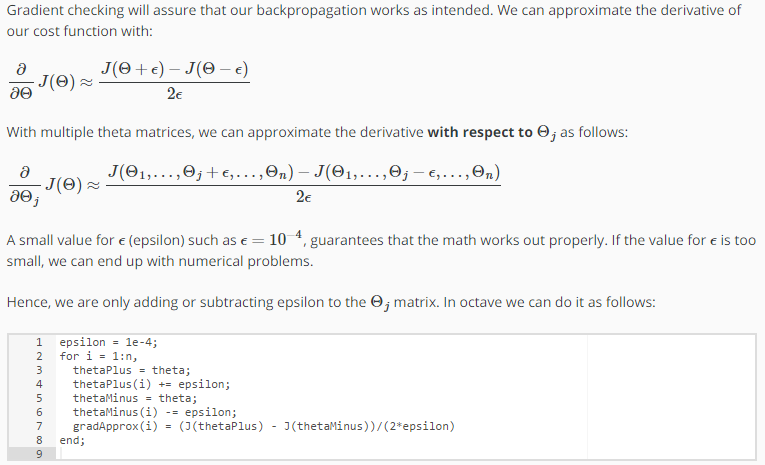


# Implementation Note: Unrolling Parameters





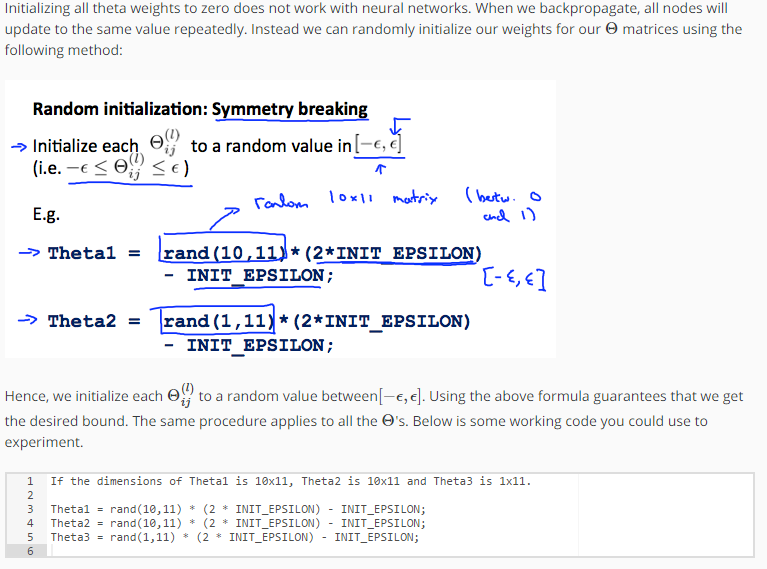
# Gradient Checking



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



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)

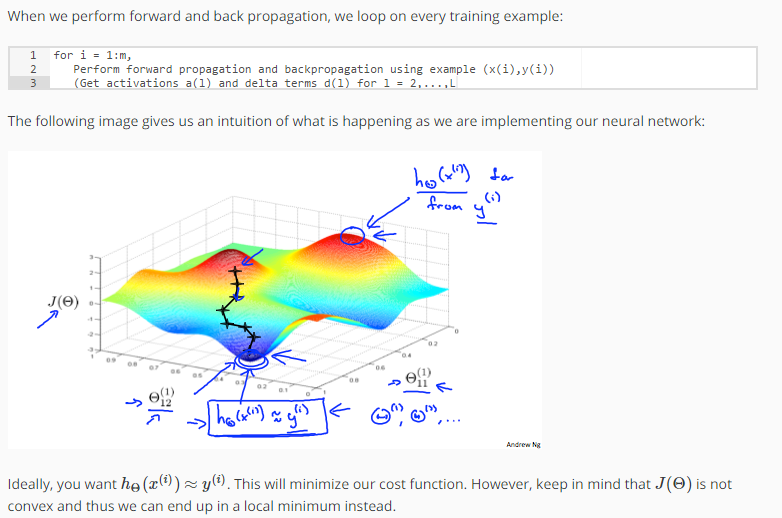
# Putting it Together

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 *h*Θ(*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.



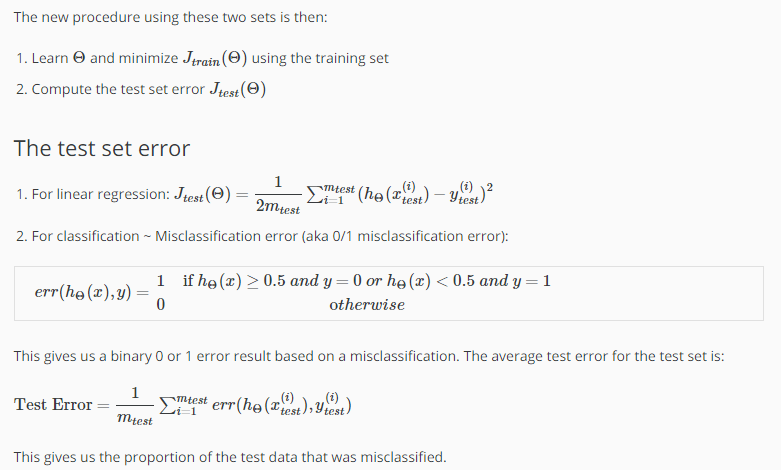
# Evaluating a Hypothesis

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 %.



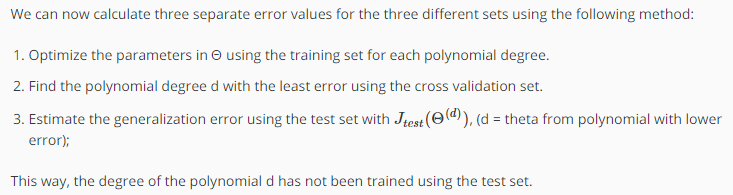
# Model Selection and Train/Validation/Test Sets

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%



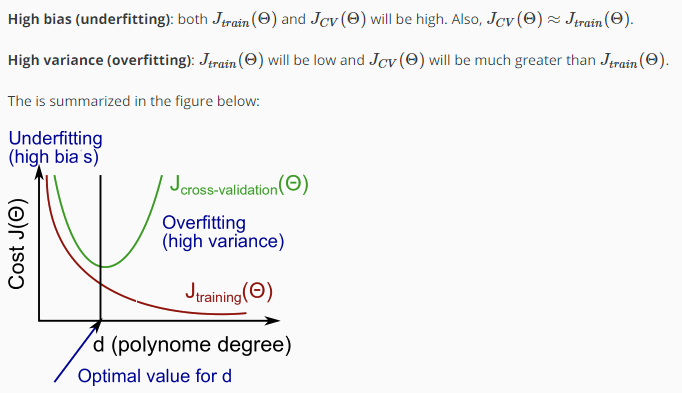
# Diagnosing Bias vs. Variance

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

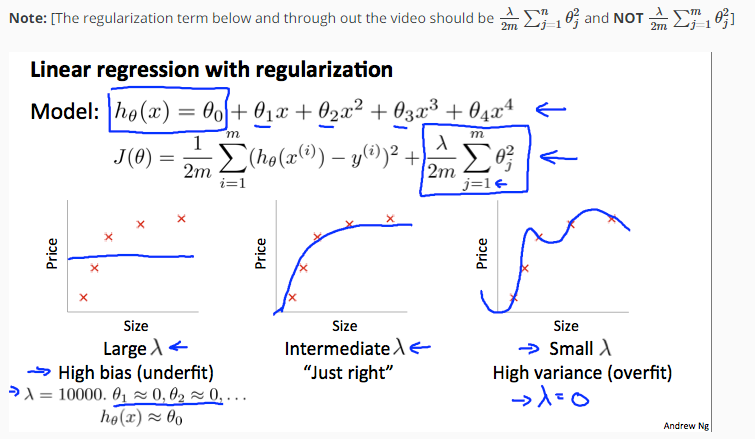
* We need to distinguish whether **bias** or **variance** is the problem contributing to bad predictions.
* High bias is underfitting and high variance is overfitting. Ideally, we need to find a golden mean between these two.

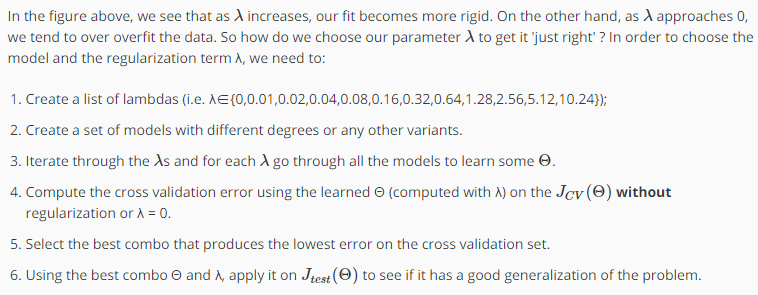
The training error will tend to **decrease** as we increase the degree d of the polynomial.

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.



# Regularization and Bias/Variance

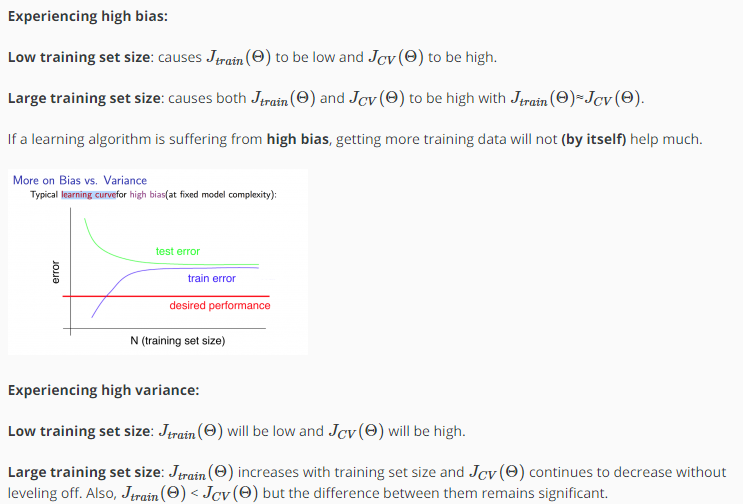


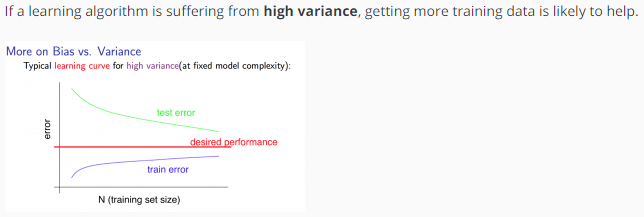


# Learning Curves

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:

* As the training set gets larger, the error for a quadratic function increases.
* 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.

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.

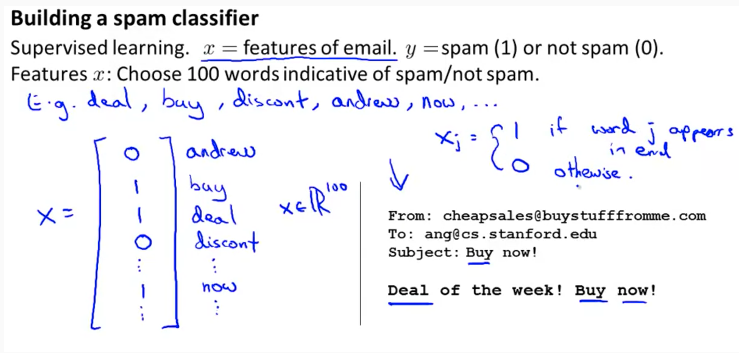
**Model Complexity Effects:**

* 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.

# 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).

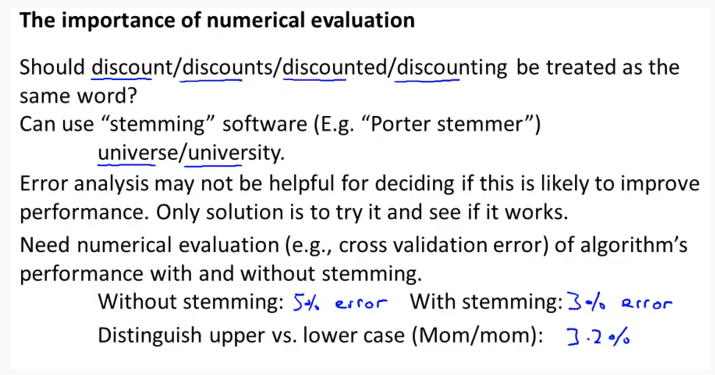
It is difficult to tell which of the options will be most helpful.

# 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 analyze 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.