Week 1

What is Machine Learning?

Two definitions of Machine Learning are offered. Arthur Samuel described it as: "the field of study that gives computers the ability to learn without being explicitly programmed." This is an older, informal definition.

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."

Example: playing checkers.

E = the experience of playing many games of checkers

T = the task of playing checkers.

P = the probability that the program will win the next game.

In general, any machine learning problem can be assigned to one of two broad classifications:

Supervised learning and Unsupervised learning.

## Supervised Learning

In supervised learning, we are given a data set and already know what our correct output should look like, having the idea that there is a relationship between the input and the output.

Supervised learning problems are categorized into "regression" and "classification" problems. In a regression problem, we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function. In a classification problem, we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories.

**Example 1:**

Given data about the size of houses on the real estate market, try to predict their price. Price as a function of size is a continuous output, so this is a regression problem.

We could turn this example into a classification problem by instead making our output about whether the house "sells for more or less than the asking price." Here we are classifying the houses based on price into two discrete categories.

**Example 2**:

(a) Regression - Given a picture of a person, we have to predict their age on the basis of the given picture

(b) Classification - Given a patient with a tumor, we have to predict whether the tumor is malignant or benign.

## Unsupervised Learning

Unsupervised learning allows us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables.

We can derive this structure by clustering the data based on relationships among the variables in the data.

With unsupervised learning there is no feedback based on the prediction results.

**Example:**

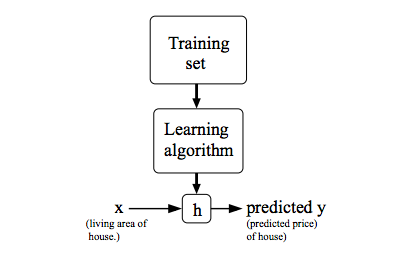
Clustering: Take a collection of 1,000,000 different genes, and find a way to automatically group these genes into groups that are somehow similar or related by different variables, such as lifespan, location, roles, and so on.

Non-clustering: The "Cocktail Party Algorithm", allows you to find structure in a chaotic environment. (i.e. identifying individual voices and music from a mesh of sounds at a [cocktail party](https://en.wikipedia.org/wiki/Cocktail_party_effect)).

# **Model Representation**

To establish notation for future use, we’ll use *x*(*i*) to denote the “input” variables (living area in this example), also called input features, and *y*(*i*) to denote the “output” or target variable that we are trying to predict (price). A pair (*x*(*i*),*y*(*i*)) is called a training example, and the dataset that we’ll be using to learn—a list of m training examples (*x*(*i*),*y*(*i*));*i*=1,...,*m*—is called a training set. Note that the superscript “(i)” in the notation is simply an index into the training set, and has nothing to do with exponentiation. We will also use X to denote the space of input values, and Y to denote the space of output values. In this example, X = Y = ℝ.

To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function h : X → Y so that h(x) is a “good” predictor for the corresponding value of y. For historical reasons, this function h is called a hypothesis. Seen pictorially, the process is therefore like this:



When the target variable that we’re trying to predict is continuous, such as in our housing example, we call the learning problem a regression problem. When y can take on only a small number of discrete values (such as if, given the living area, we wanted to predict if a dwelling is a house or an apartment, say), we call it a classification problem.

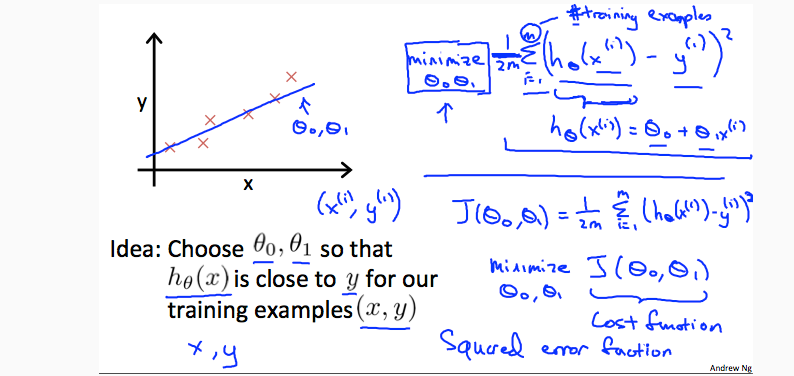
# **Cost Function**

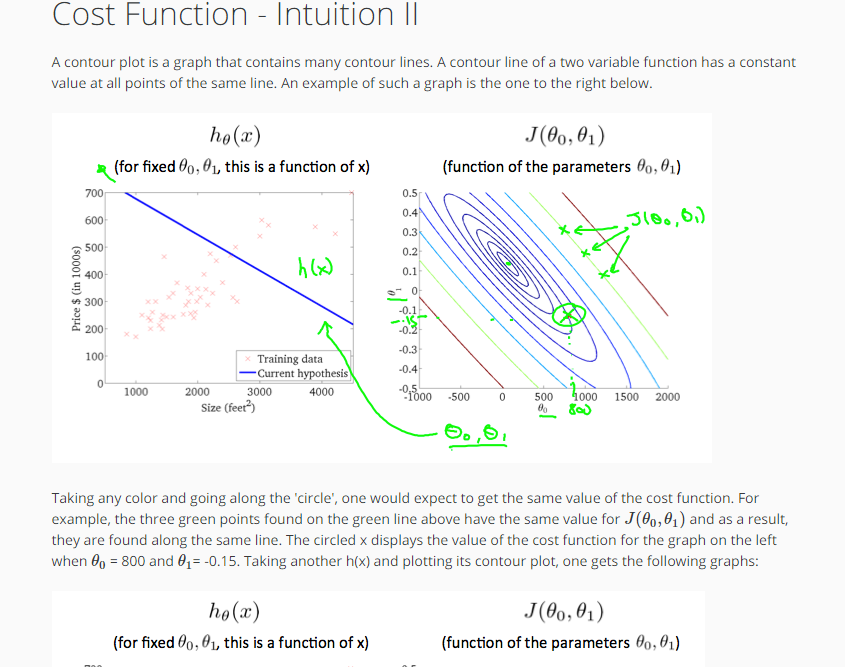
We can measure the accuracy of our hypothesis function by using a **cost function**. This takes an average difference (actually a fancier version of an average) of all the results of the hypothesis with inputs from x's and the actual output y's.

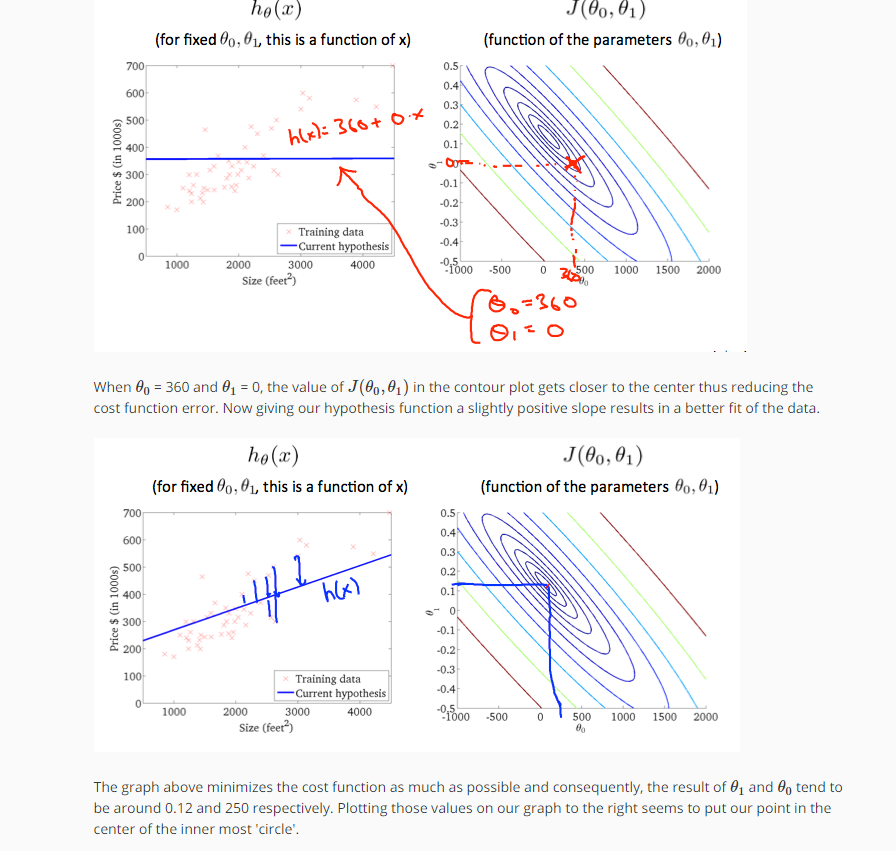
*J*(*θ*0,*θ*1)=12*m*∑*i*=1*m*(*y*^*i*−*yi*)2=12*m*∑*i*=1*m*(*hθ*(*xi*)−*yi*)2

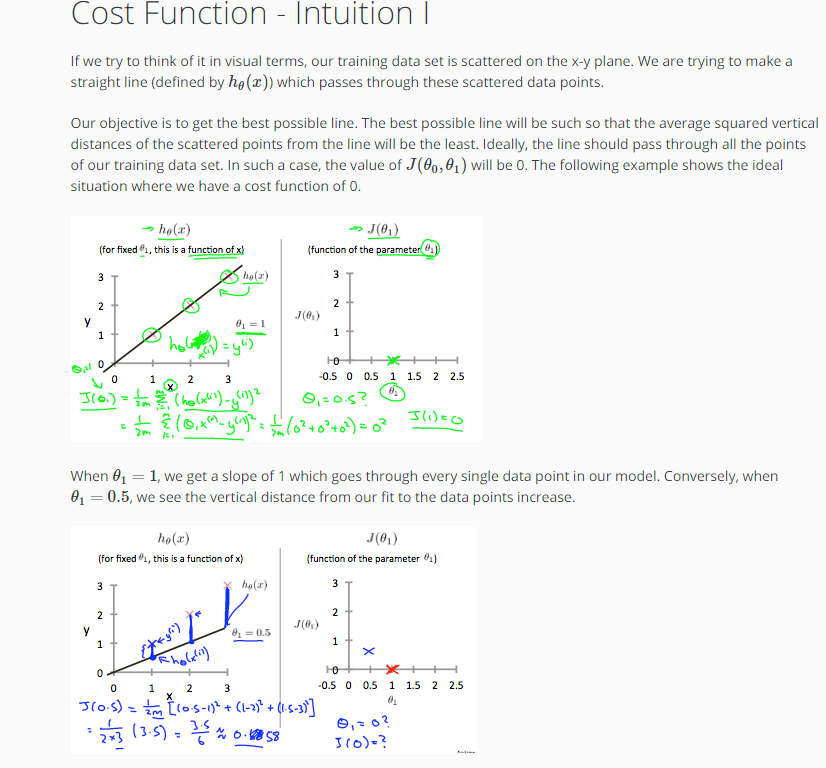
To break it apart, it is 12 *x*¯ where *x*¯ is the mean of the squares of *hθ*(*xi*)−*yi* , or the difference between the predicted value and the actual value.

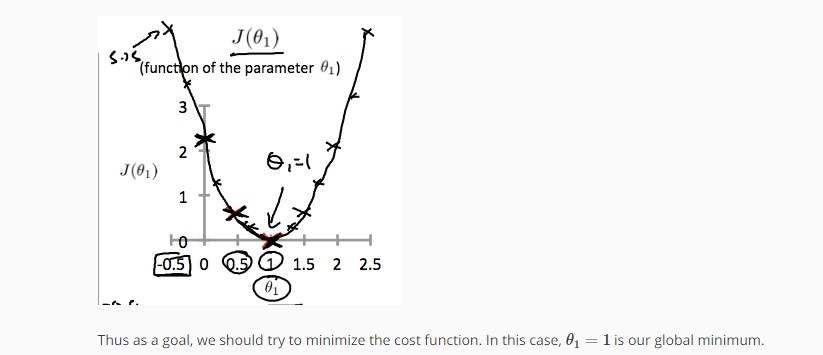
This function is otherwise called the "Squared error function", or "Mean squared error". The mean is halved (12)as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the 12 term. The following image summarizes what the cost function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









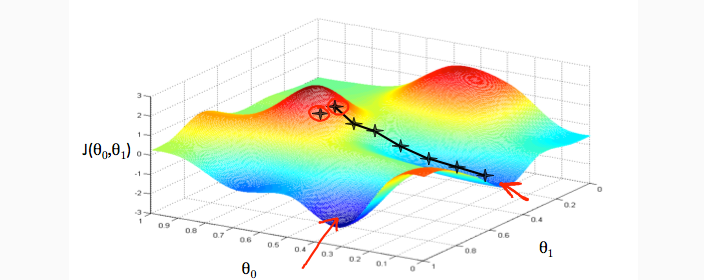


# **Gradient Descent**

So we have our hypothesis function and we have a way of measuring how well it fits into the data. Now we need to estimate the parameters in the hypothesis function. That's where gradient descent comes in.

Imagine that we graph our hypothesis function based on its fields *θ*0 and *θ*1 (actually we are graphing the cost function as a function of the parameter estimates). We are not graphing x and y itself, but the parameter range of our hypothesis function and the cost resulting from selecting a particular set of parameters.

We put *θ*0 on the x axis and *θ*1 on the y axis, with the cost function on the vertical z axis. The points on our graph will be the result of the cost function using our hypothesis with those specific theta parameters. The graph below depicts such a setup.



We will know that we have succeeded when our cost function is at the very bottom of the pits in our graph, i.e. when its value is the minimum. The red arrows show the minimum points in the graph.

The way we do this is by taking the derivative (the tangential line to a function) of our cost function. The slope of the tangent is the derivative at that point and it will give us a direction to move towards. We make steps down the cost function in the direction with the steepest descent. The size of each step is determined by the parameter α, which is called the learning rate.

For example, the distance between each 'star' in the graph above represents a step determined by our parameter α. A smaller α would result in a smaller step and a larger α results in a larger step. The direction in which the step is taken is determined by the partial derivative of *J*(*θ*0,*θ*1). Depending on where one starts on the graph, one could end up at different points. The image above shows us two different starting points that end up in two different places.

The gradient descent algorithm is:

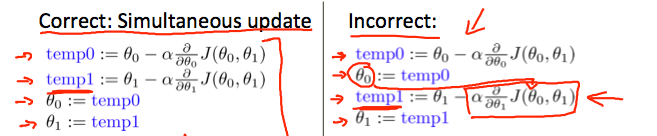
repeat until convergence:

*θj*:=*θj*−*α*∂/∂*θjJ*(*θ*0,*θ*1)

where

j=0,1 represents the feature index number.

At each iteration j, one should simultaneously update the parameters *θ*1,*θ*2,...,*θn*. Updating a specific parameter prior to calculating another one on the *j*(*th*) iteration would yield to a wrong implementation.

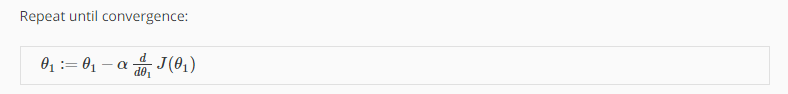




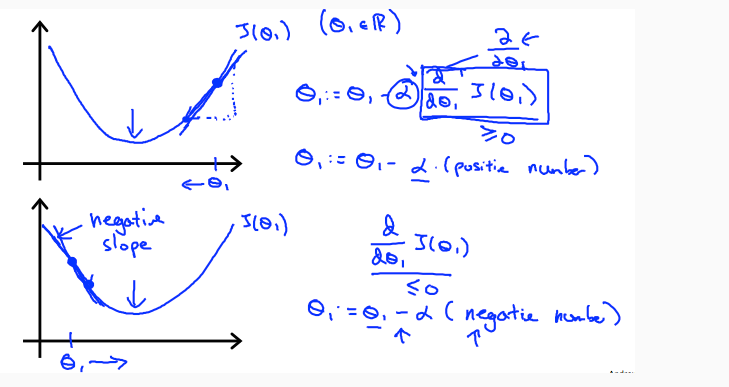
# **Gradient Descent Intuition**

In this video we explored the scenario where we used one parameter *θ*1 and plotted its cost function to implement a gradient descent. Our formula for a single parameter was :

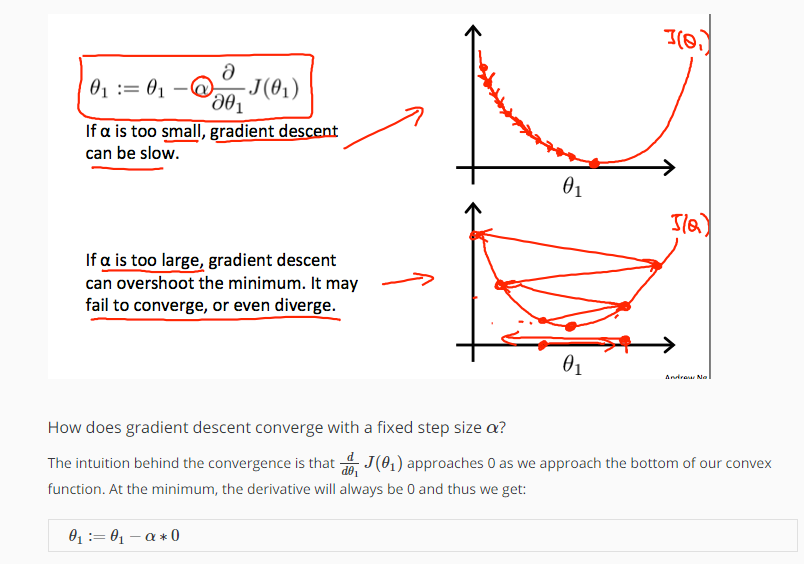
Repeat until convergence:

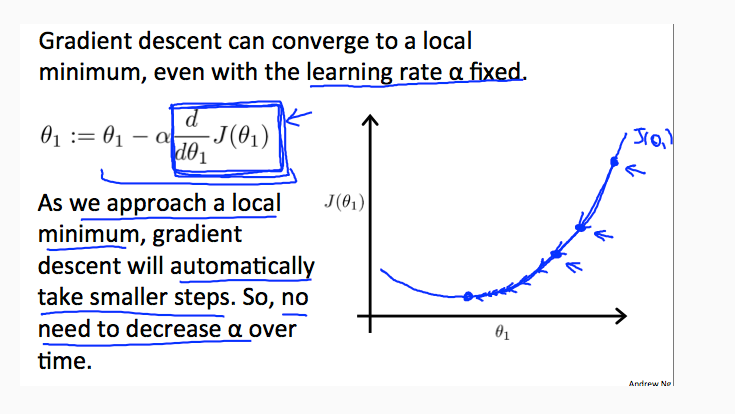


Regardless of the slope's sign for *ddθ*1*J*(*θ*1), *θ*1 eventually converges to its minimum value. The following graph shows that when the slope is negative, the value of *θ*1 increases and when it is positive, the value of *θ*1 decreases.



On a side note, we should adjust our parameter *α* to ensure that the gradient descent algorithm converges in a reasonable time. Failure to converge or too much time to obtain the minimum value imply that our step size is wrong

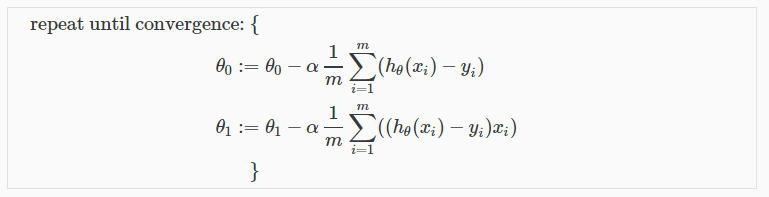


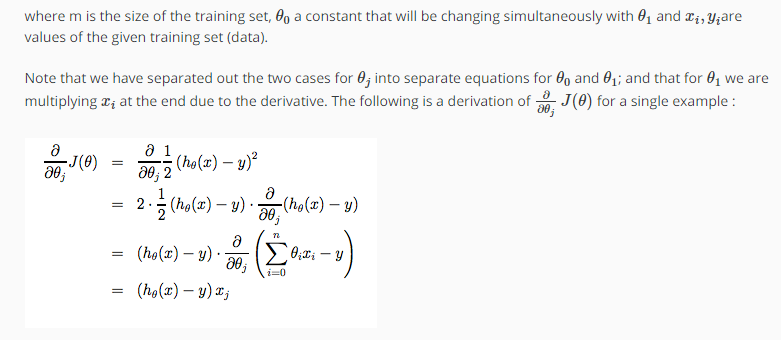


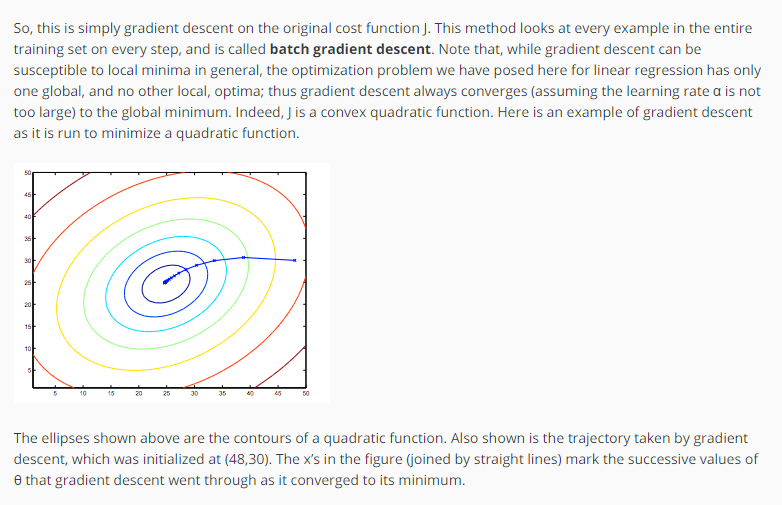
# **Gradient Descent For Linear Regression**

**Note:** [At 6:15 "h(x) = -900 - 0.1x" should be "h(x) = 900 - 0.1x"]

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 :







Week 2

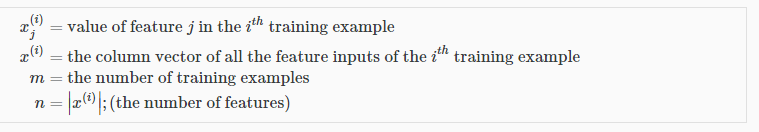
### Multivariate Linear Regression

# **Multiple Features**

**Note:** [7:25 - *θT* is a 1 by (n+1) matrix and not an (n+1) by 1 matrix]

Linear regression with multiple variables is also known as "multivariate linear regression".

We now introduce notation for equations where we can have any number of input variables.

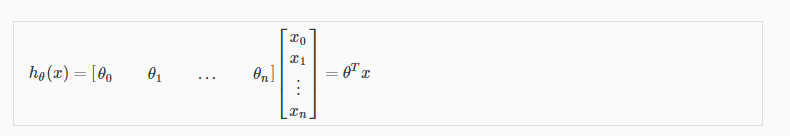


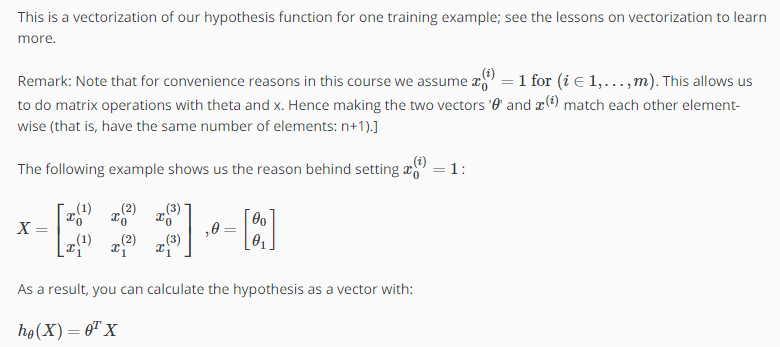
The multivariable form of the hypothesis function accommodating these multiple features is as follows:

*hθ*(*x*)=*θ*0+*θ*1*x*1+*θ*2*x*2+*θ*3*x*3+⋯+*θnxn*

In order to develop intuition about this function, we can think about *θ*0 as the basic price of a house, *θ*1 as the price per square meter, *θ*2 as the price per floor, etc. *x*1 will be the number of square meters in the house, *x*2 the number of floors, etc.

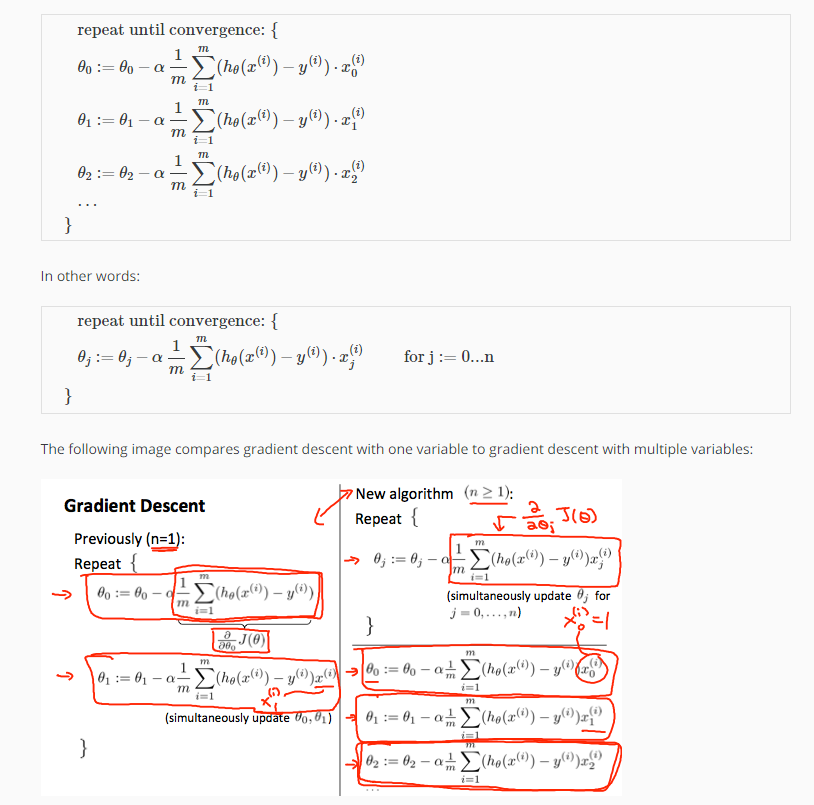
Using the definition of matrix multiplication, our multivariable hypothesis function can be concisely represented as:





## ****Gradient Descent for Multiple Variables****

The gradient descent equation itself is generally the same form; we just have to repeat it for our 'n' features:



# **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:

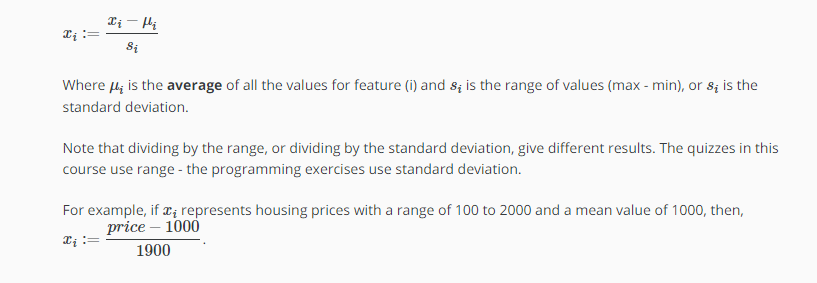
−1 ≤ *x*(*i*) ≤ 1

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:

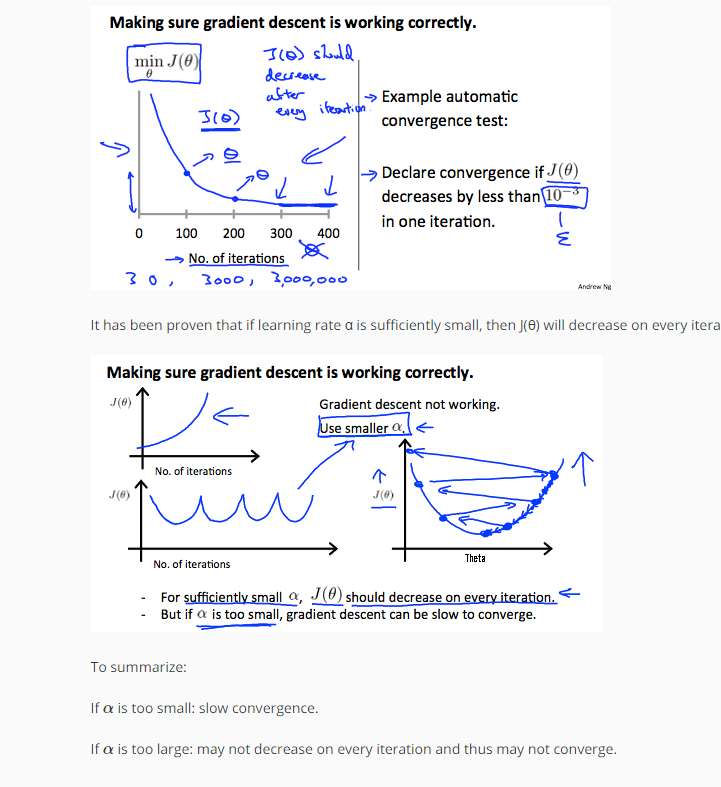


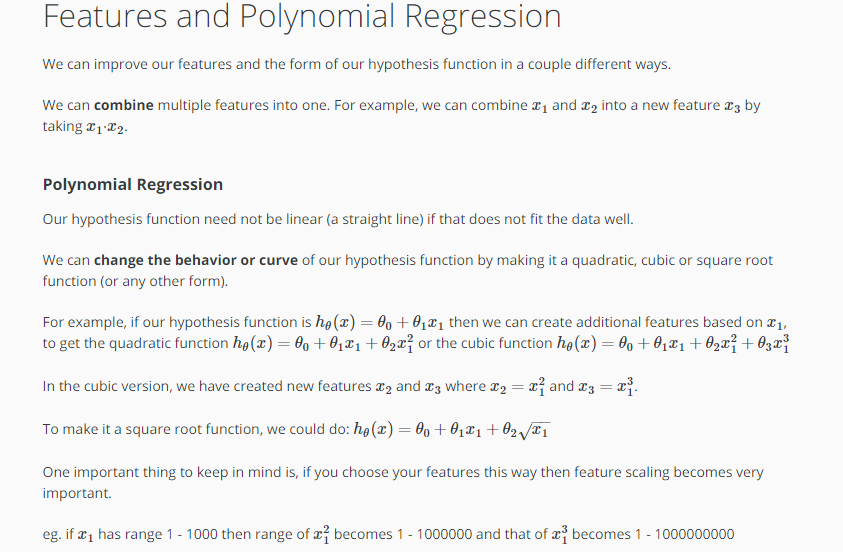
# **Gradient Descent in Practice II - Learning Rate**

**Note:** [5:20 - the x -axis label in the right graph should be *θ* rather than No. of iterations ]

**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.



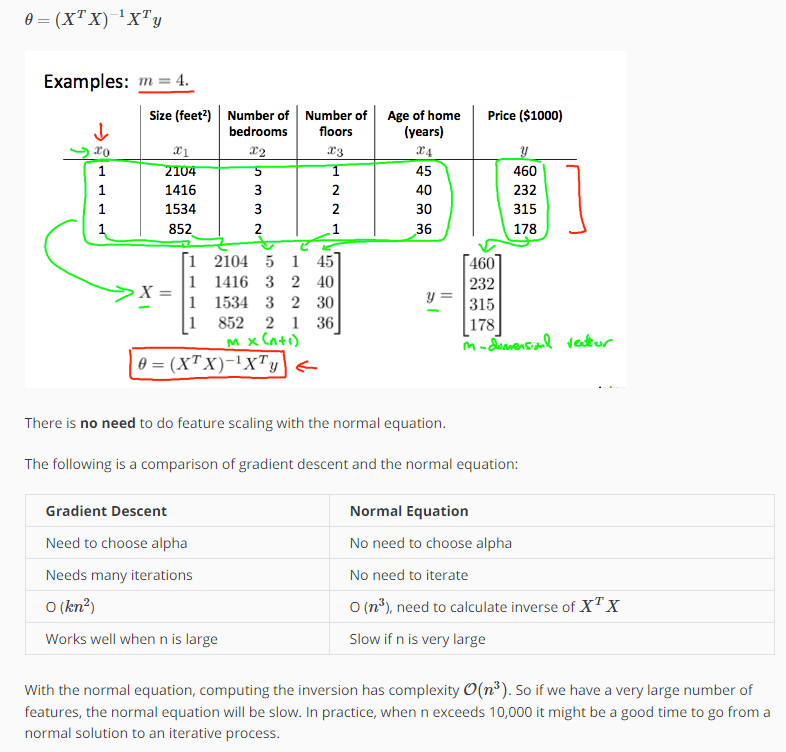


Computing Parameters Analytically

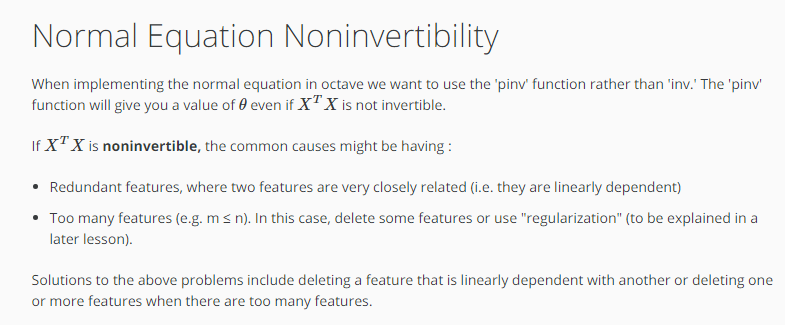
# **Normal Equation**

**Note:** [8:00 to 8:44 - The design matrix X (in the bottom right side of the slide) given in the example should have elements x with subscript 1 and superscripts varying from 1 to m because for all m training sets there are only 2 features *x*0 and *x*1. 12:56 - The X matrix is m by (n+1) and NOT n by n. ]

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:



# **Normal Equation Noninvertibility**



Week 3

Classification and Representation

# **Classification**

To attempt classification, one method is to use linear regression and map all predictions greater than 0.5 as a 1 and all less than 0.5 as a 0. However, this method doesn't work well because classification is not actually a linear function.

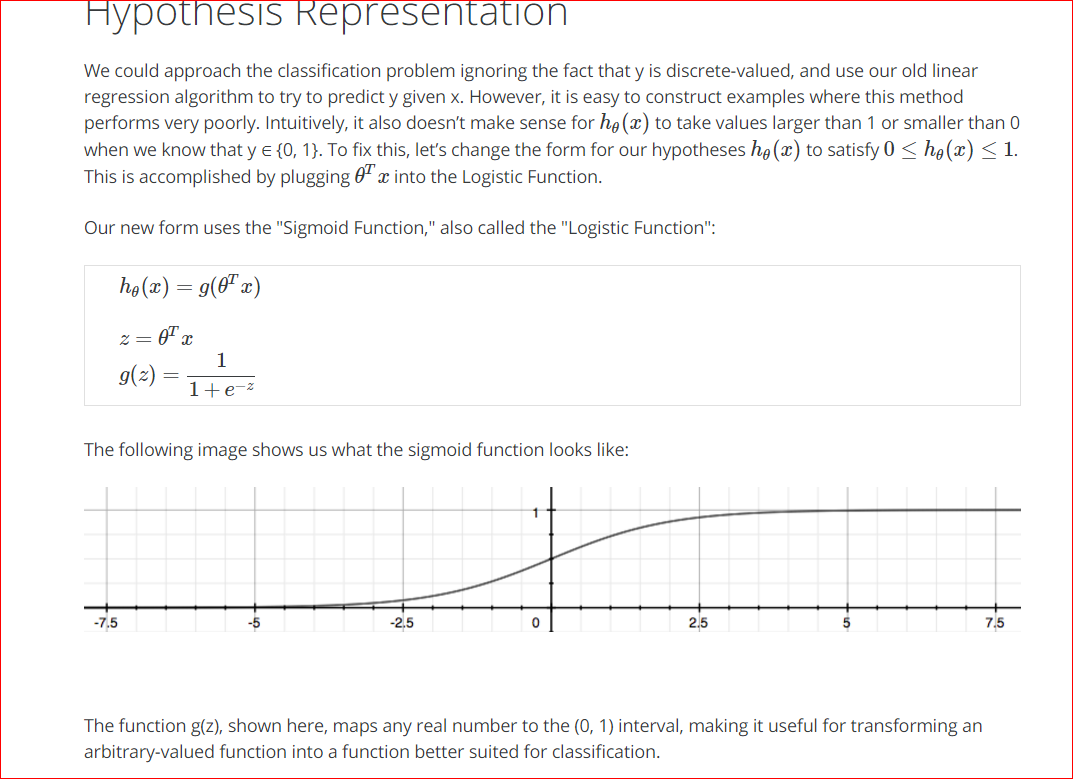
The classification problem is just like the regression problem, except that the values y we now want to predict take on only a small number of discrete values. For now, we will focus on the **binary classification** **problem** in which y can take on only two values, 0 and 1. (Most of what we say here will also generalize to the multiple-class case.) For instance, if we are trying to build a spam classifier for email, then *x*(*i*) may be some features of a piece of email, and y may be 1 if it is a piece of spam mail, and 0 otherwise. Hence, y∈ {0,1}. 0 is also called the negative class, and 1 the positive class, and they are sometimes also denoted by the symbols “-” and “+.”

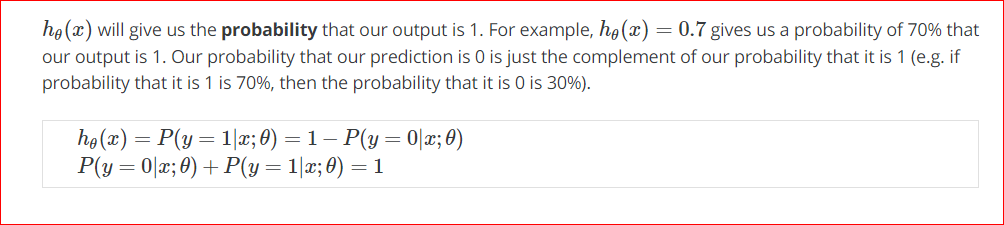
Given *x* (*i*), the corresponding *y* (*i*) is also called the label for the training example.

**Logistic Regression Model (is a classification problem)**

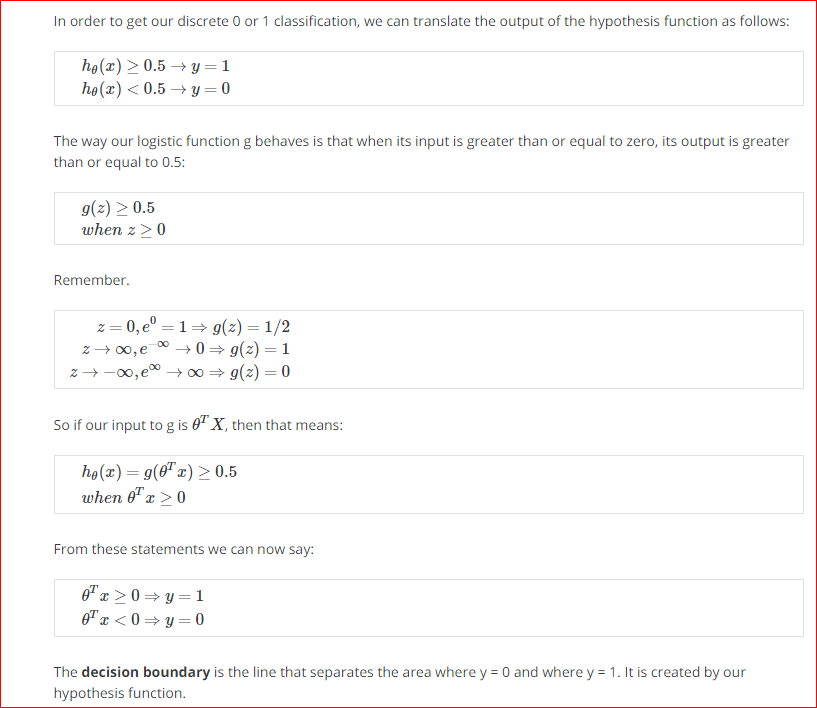
**Hypothesis Representation**

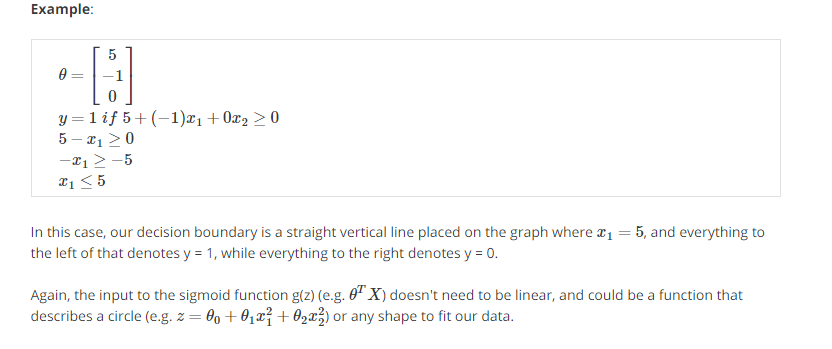
**Decision Boundary**





# **Decision Boundary**



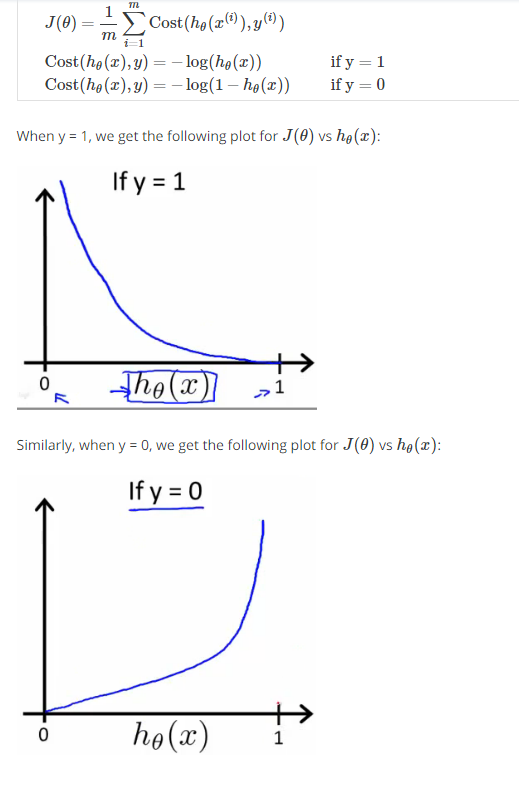


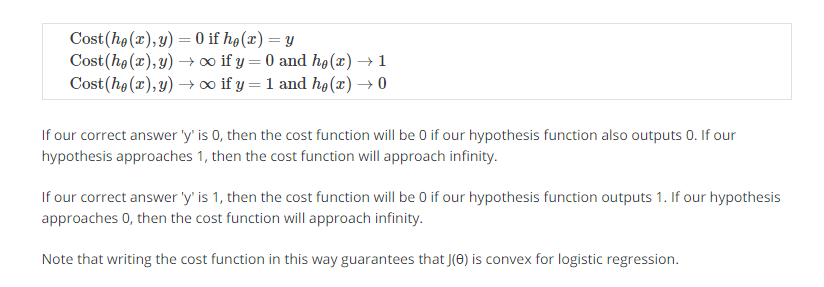
Logistic Regression Model

# **Cost Function**

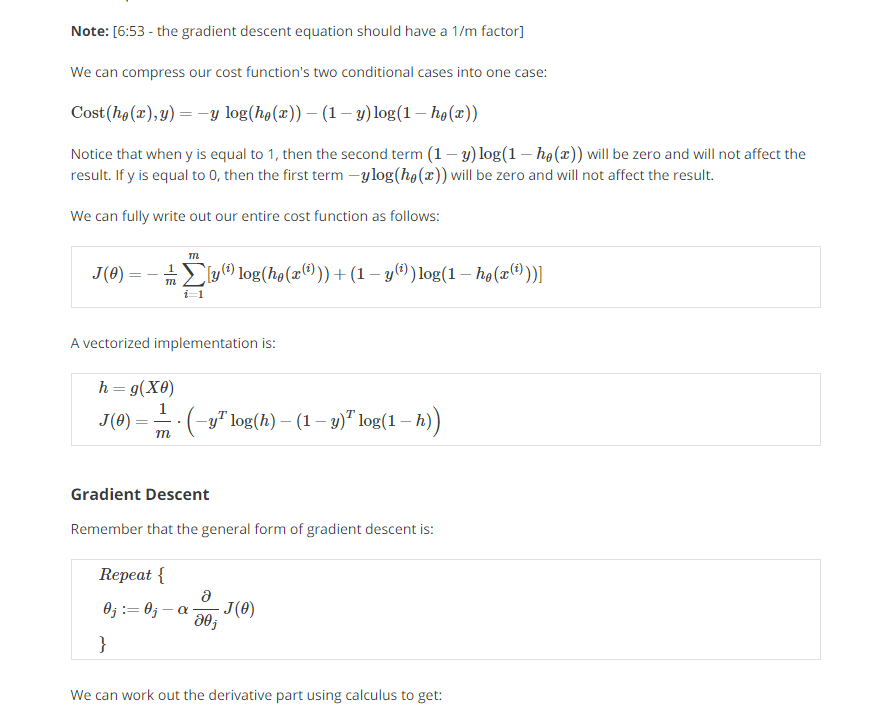
We cannot use the same cost function that we use for linear regression because the Logistic Function will cause the output to be wavy, causing many local optima. In other words, it will not be a convex function.

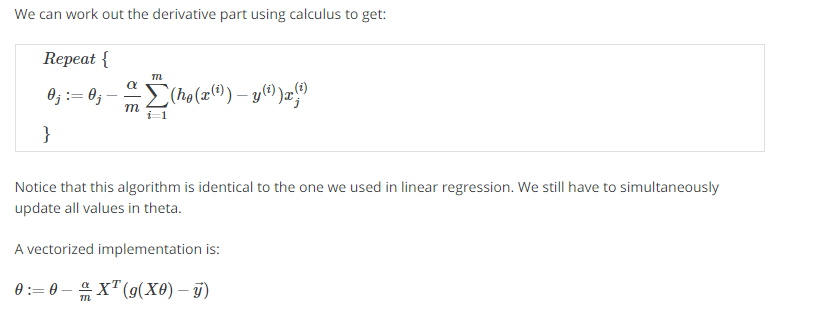
Instead, our cost function for logistic regression looks like:



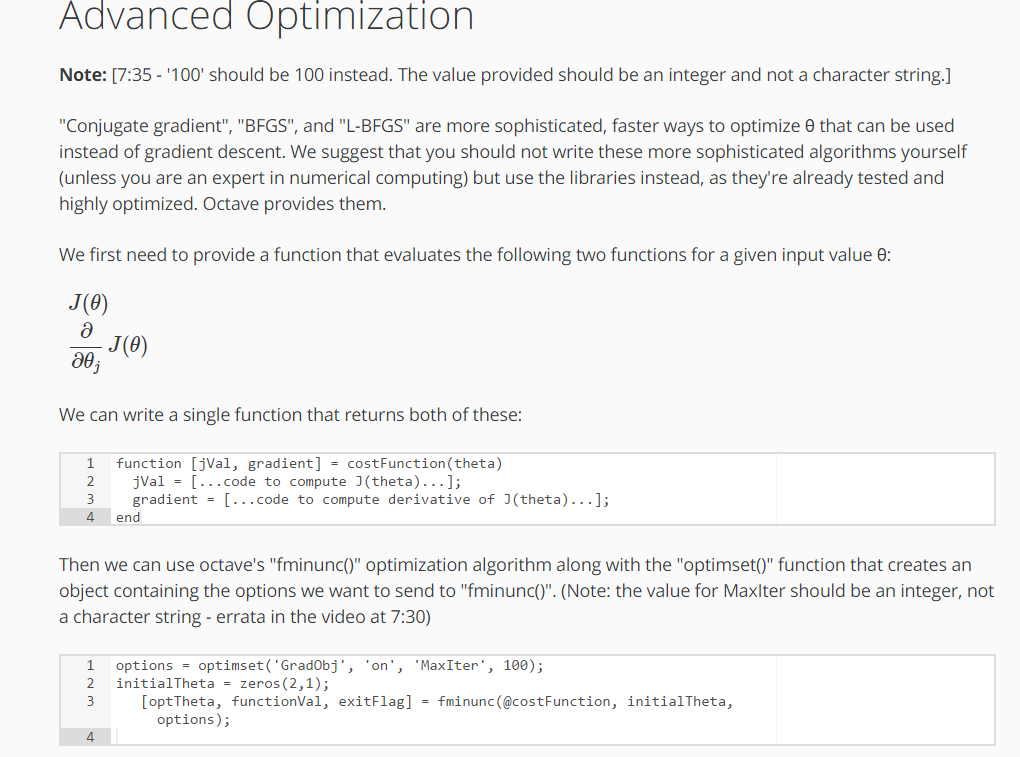


Simplified Cost Function and Gradient Descent





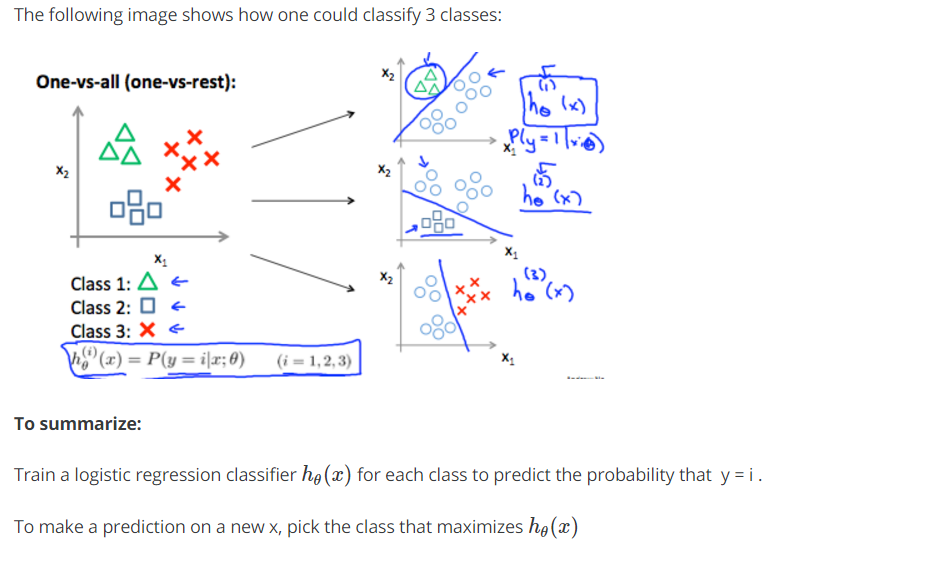
Advanced Optimization



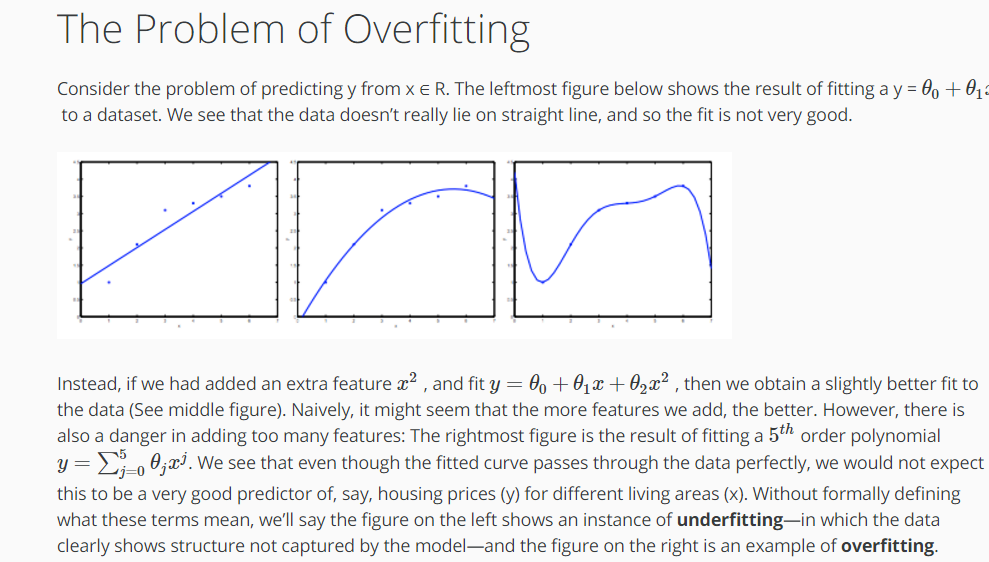


# **Multiclass Classification: One-vs-all**





Solving 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:

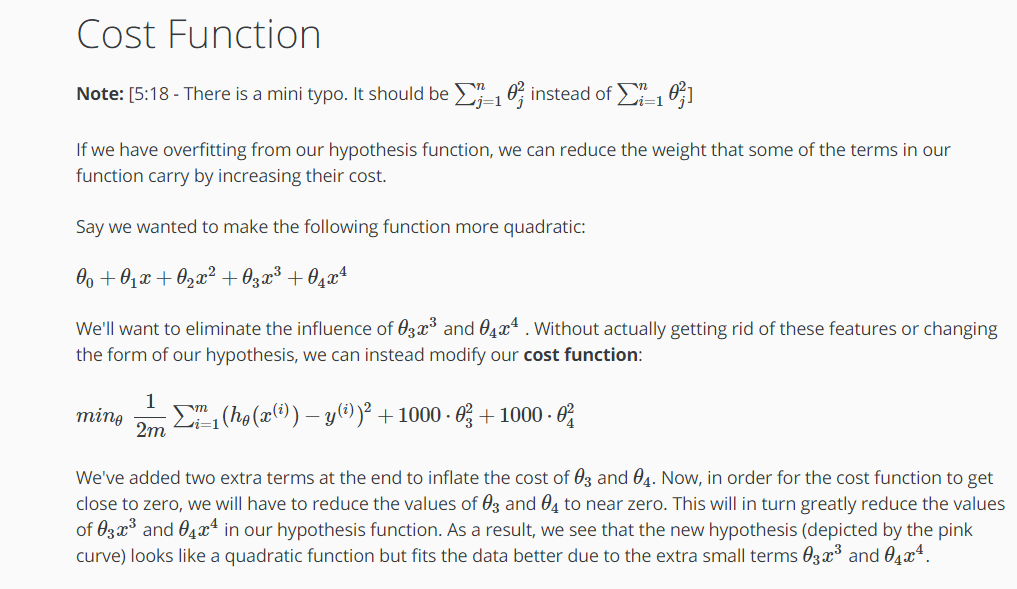
1) Reduce the number of features:

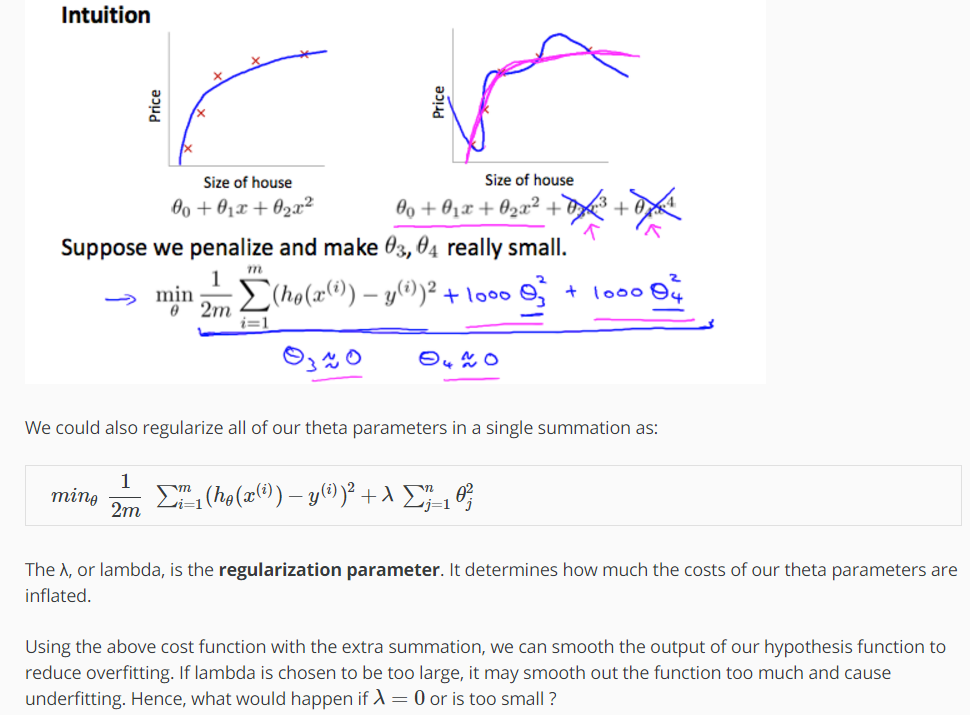
* Manually select which features to keep.
* Use a model selection algorithm (studied later in the course).

2) Regularization

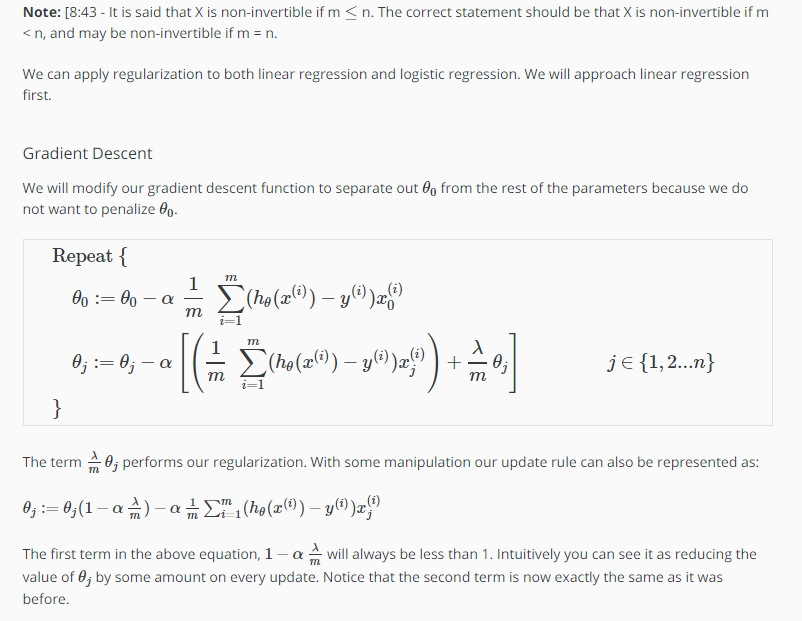
* Keep all the features, but reduce the magnitude of parameters *θj*.
* Regularization works well when we have a lot of slightly useful features.

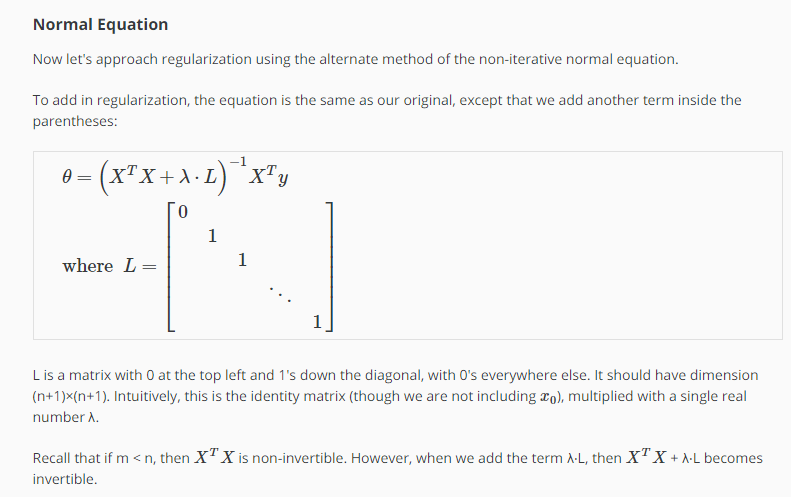
Cost Function





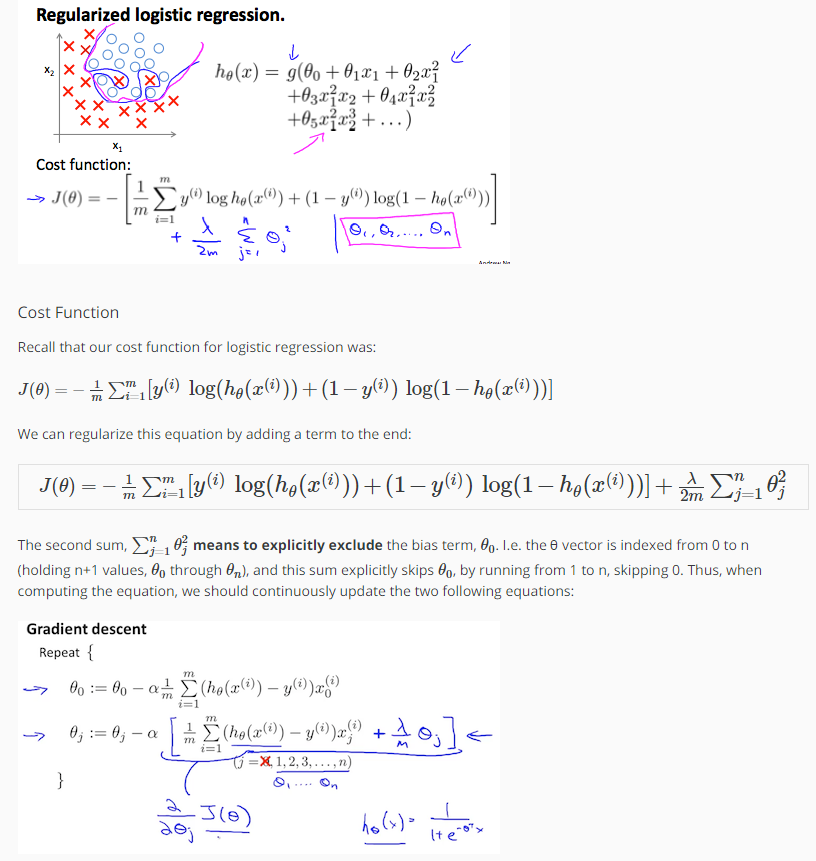
# **Regularized Linear Regression**





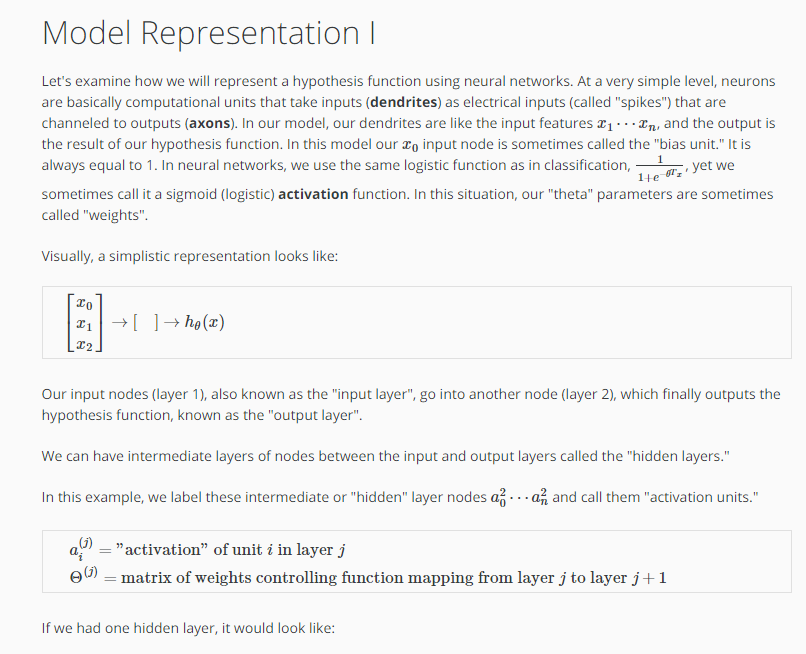
# **Regularized Logistic Regression**

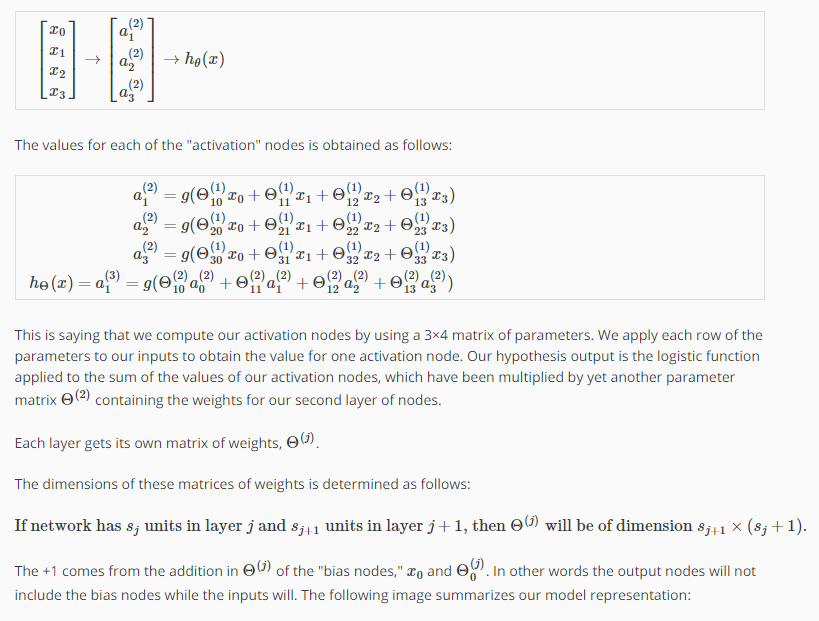
We can regularize logistic regression in a similar way that we regularize linear regression. As a result, we can avoid overfitting. The following image shows how the regularized function, displayed by the pink line, is less likely to overfit than the non-regularized function represented by the blue line:

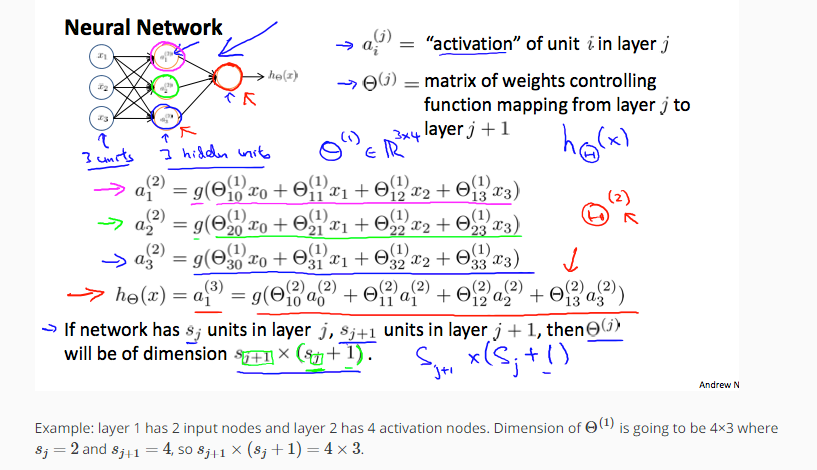


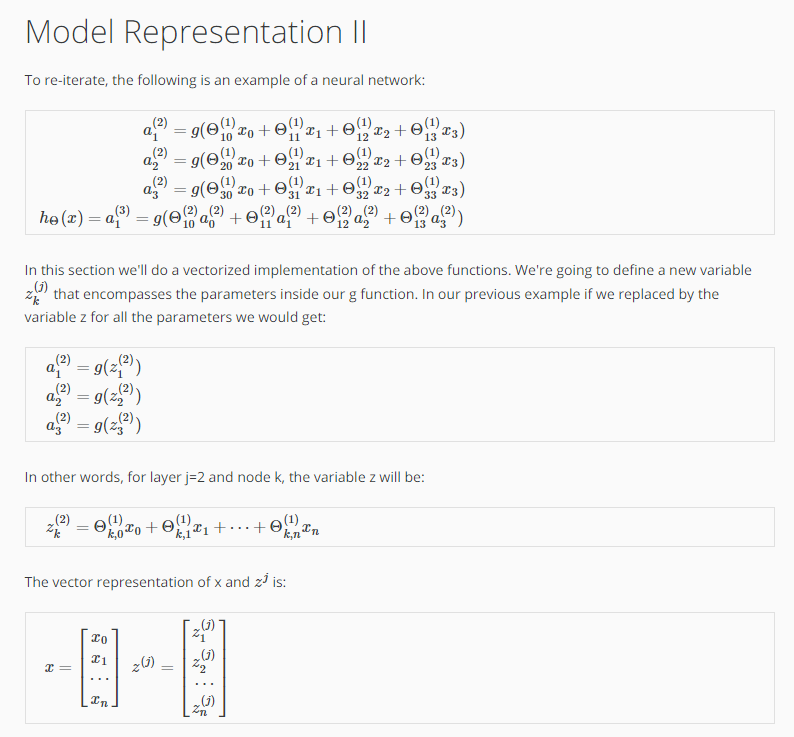
Week 4

Neural Networks

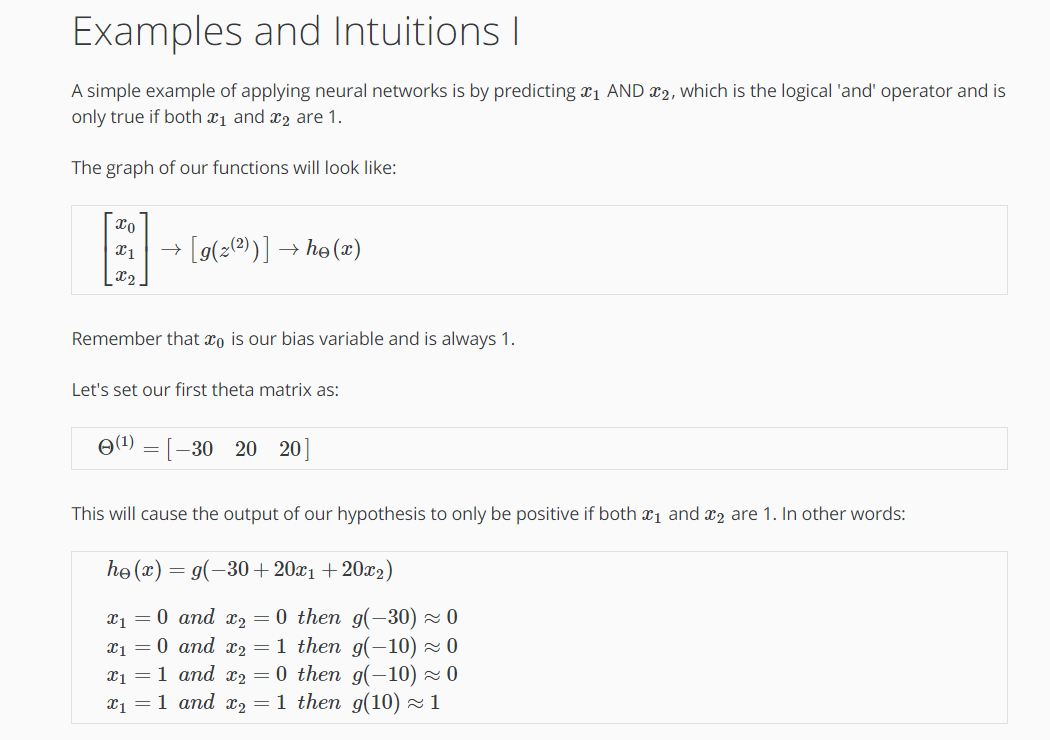


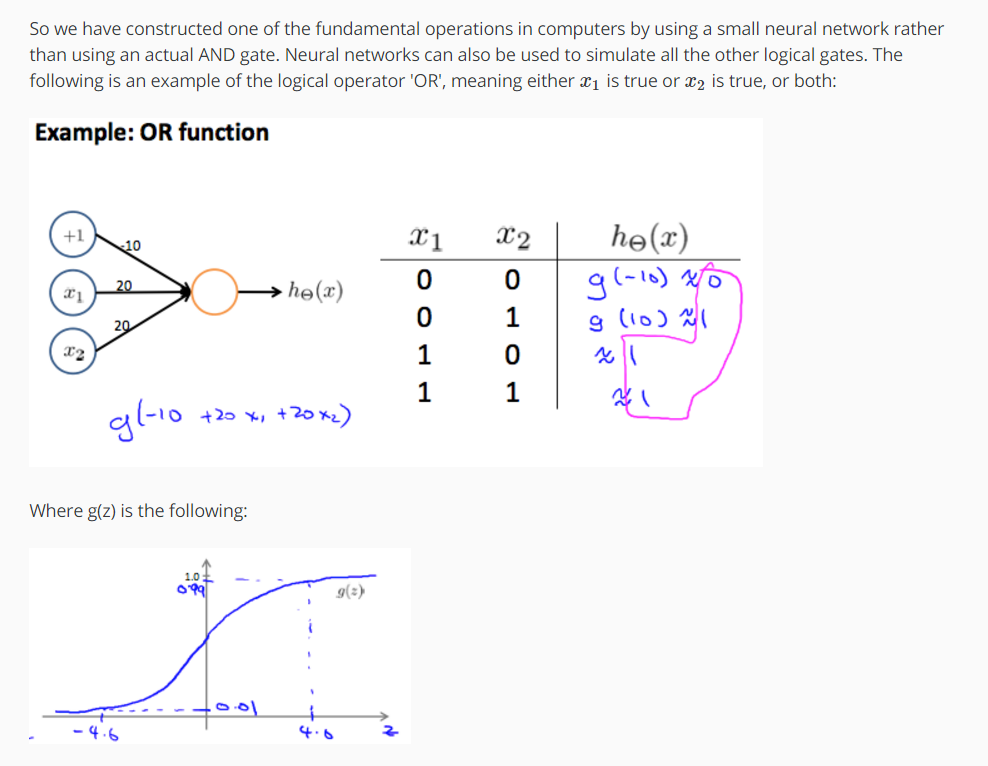


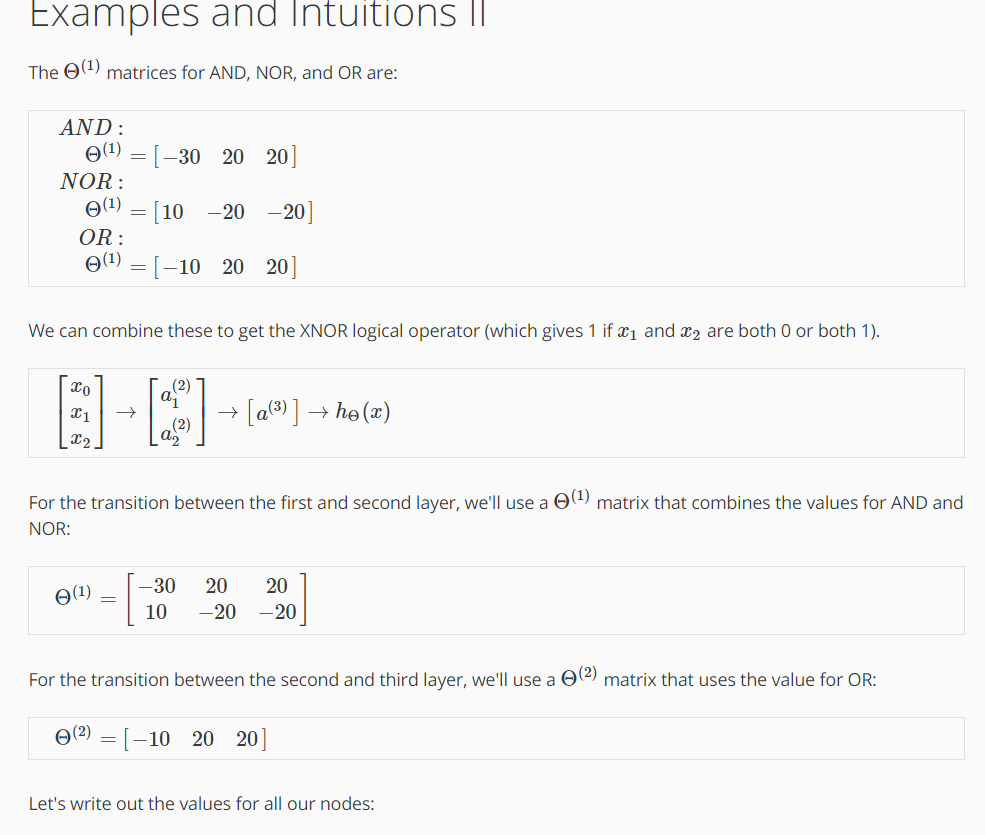


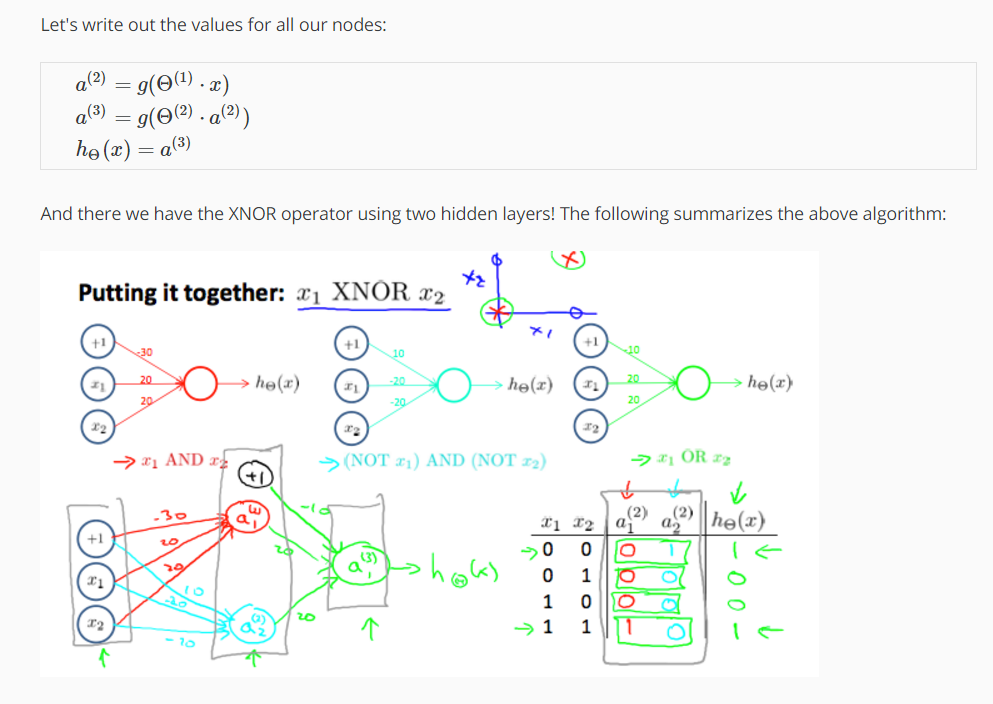


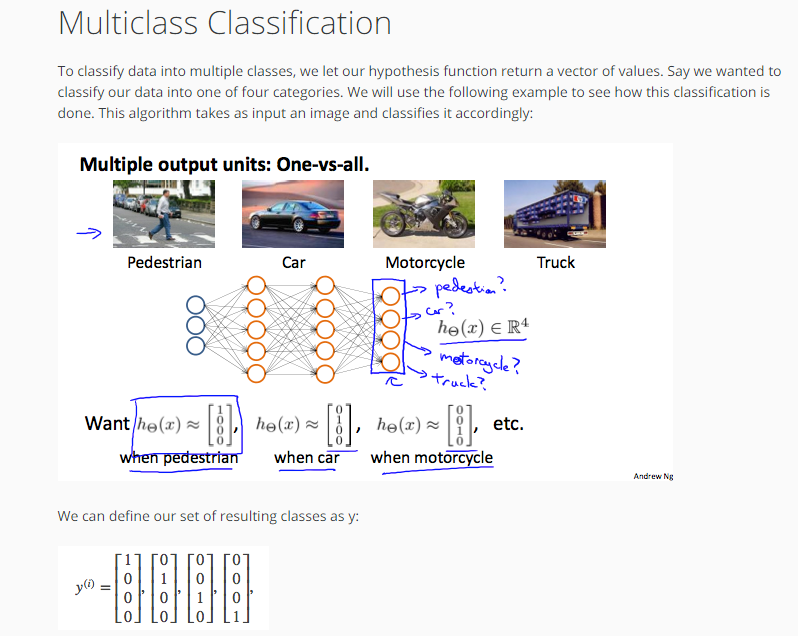


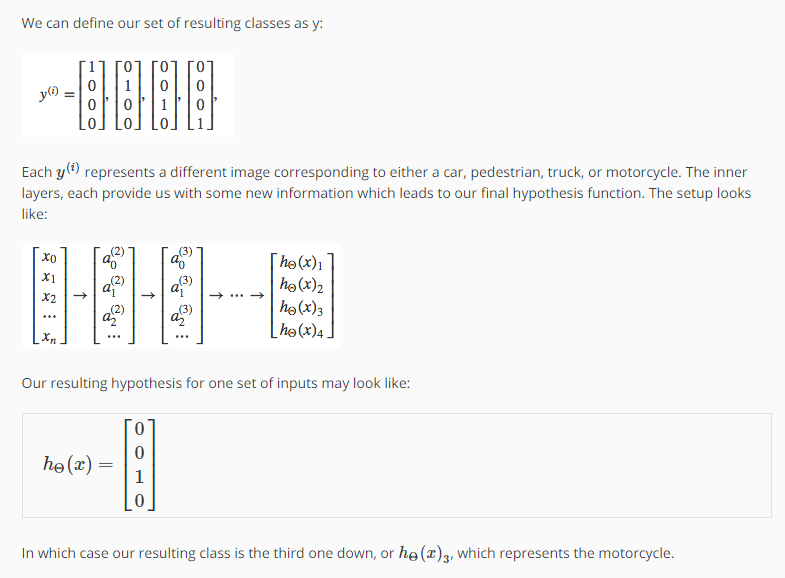






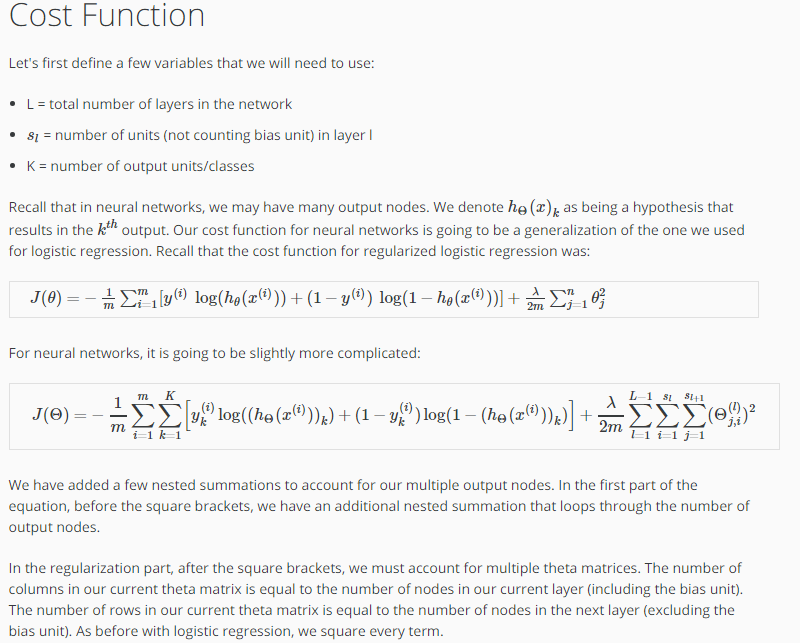


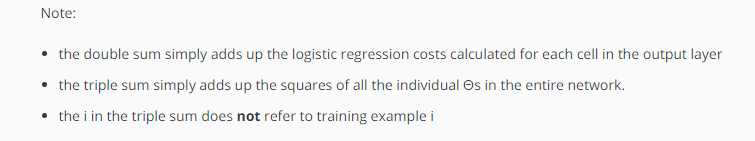


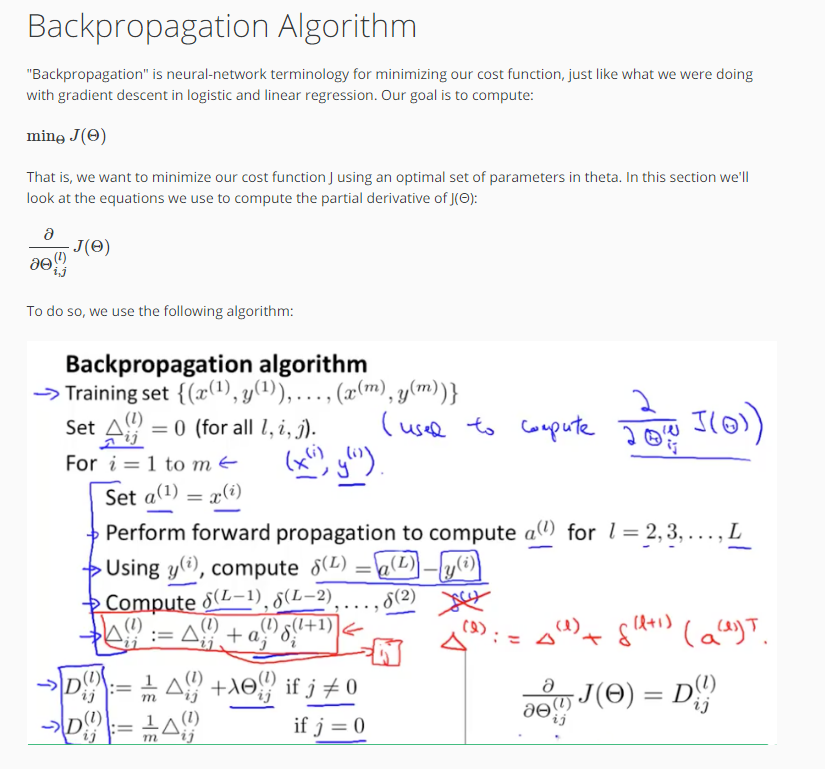


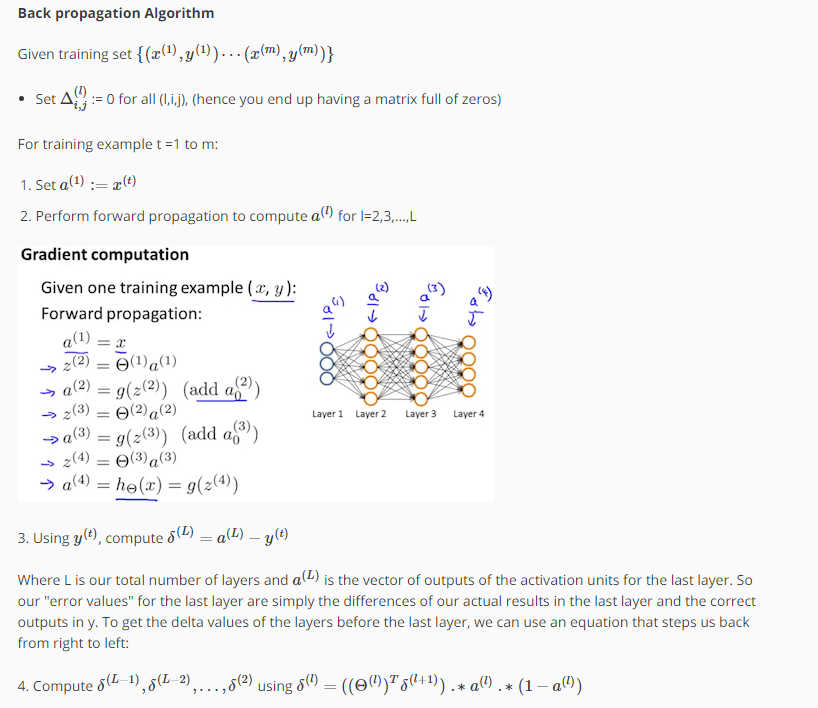
Week 5

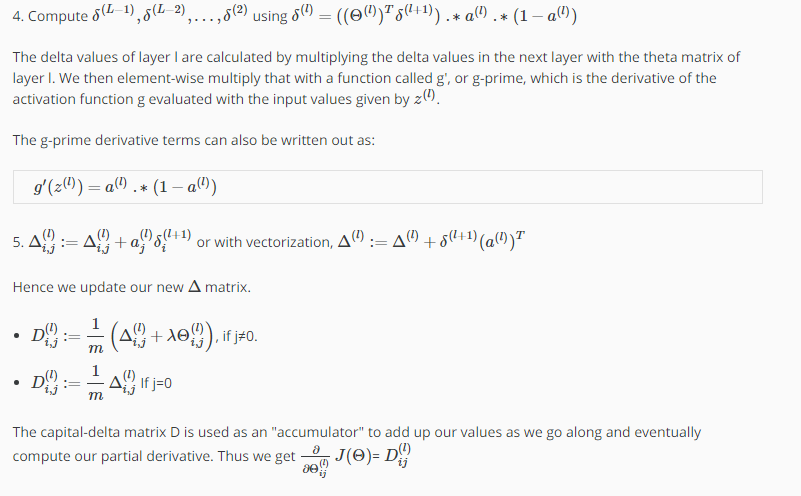
#### **Cost Function and Backpropagation**

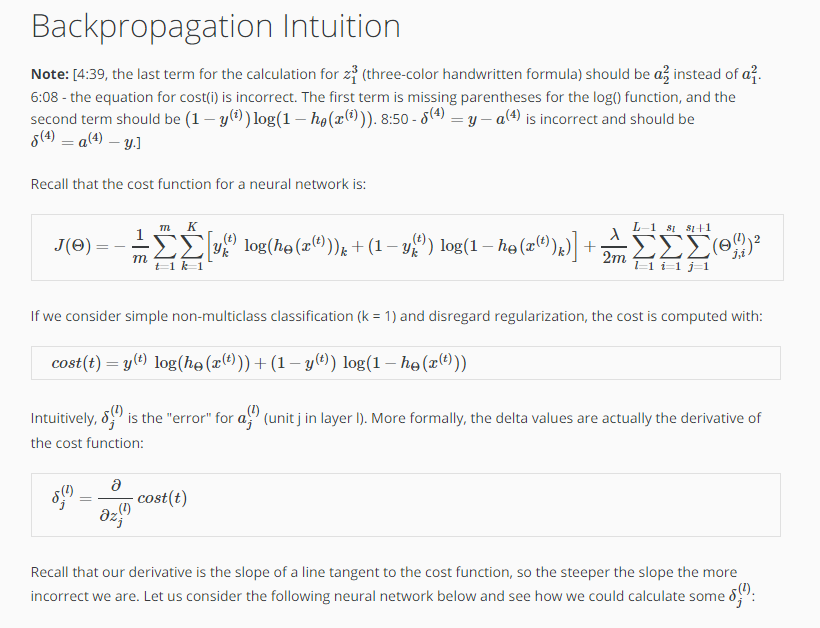














#### **Backpropagation in practice**

