Machine Learning

Coursera Notes for the Stanford Course by Andrew Ng

Week 1

Introduction

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 "regressions" 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 out 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

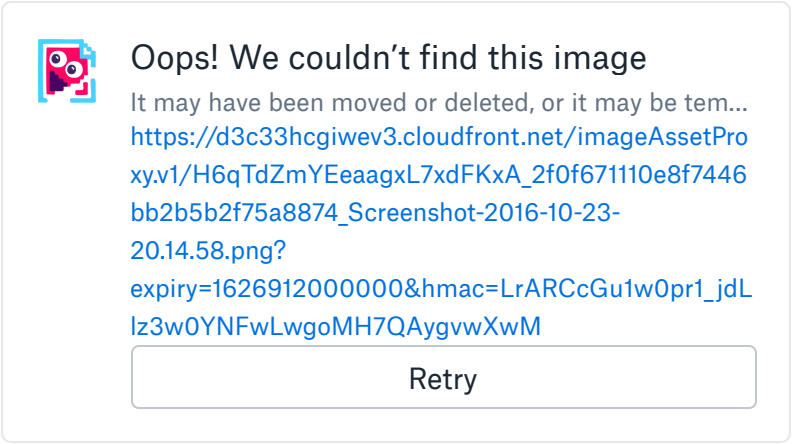
Unsupervised learning allows us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables.
We can derive this structure by clustering the data based on relationships among the variables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

Model and Cost function

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 variables that we are trying to predict (price). A pair $\left(x^{(i)}, y^{(i)}\right)$ is called a training example, and the dataset that we'll be using to learn—a list of m training examples $\left(x^{(i)}, y^{(i)}\right) ; i = 1, \ldots, m$—is called a training set. Note that the superscript (i) in the notation is a simply an index into the training set, and has nothing to do with exponentiation. We will also use X to denote the space if input values, and Y to denote the space of output values. In this example, $X = Y = \Re$.
To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function $h : X \to Y$ so that $h(x)$ is a "good" predictor for

the corresponding value of y. For historic reasons, the function h is called a hypothesis. Seen pictorially, the process is therefor 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 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 is a classification problem.

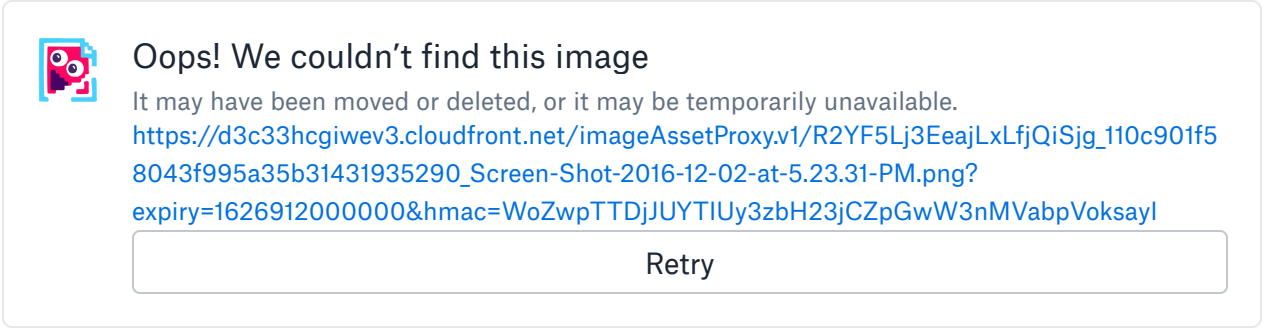
Cost Function

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 outputs y's.

$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)^2$

To break it apart, it is $\frac{1}{2}\bar{x}$ where $\bar{x}$ is the mean of the squares of $h_\theta(x_i) - y_i$, or the difference between the predicted value and the actual value.

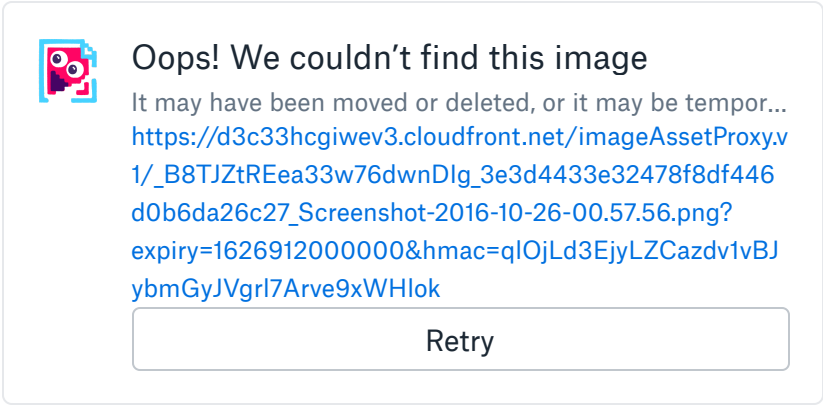
This function is otherwise called the "Squared error function", or "Mean squared error". This mean is halved as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the $\frac{1}{2}$ term The following image summarizes what the cost function does:



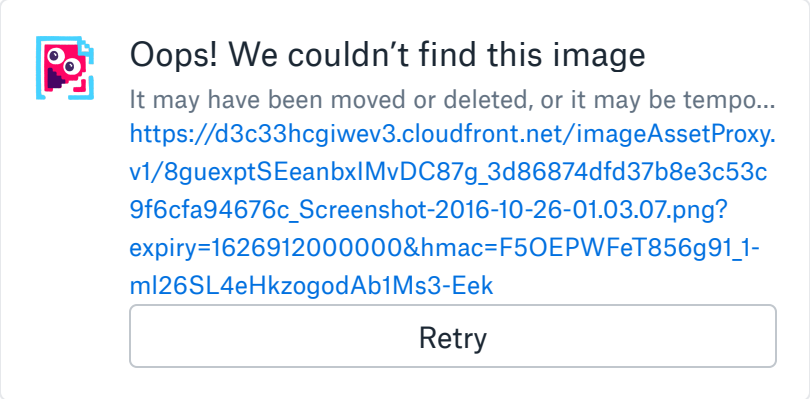
Cost function - Intuition I

If we try to think of it in visual terms, out training data set is scattered on the x-y plane. We are trying to make a straight line (defined by $h_\theta(x)$) which passed through these scattered data points.

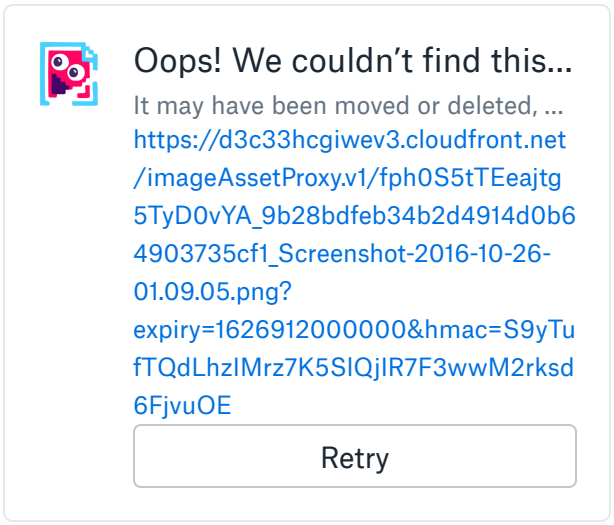
Our objective is to get the best possible line. The best possible line will be such so that the average squared vertical distances of the scattered points from the line will be the least. Ideally, the line should pass through all the points of our training data set. In such a case, the values of $J(\theta_0, \theta_1)$ will be 0. The following examples show the ideal situation where we have a cost function of 0.



When $\theta_1 = 1$, we get a slope of 1 which goes through every single data point in out model. Conversely, when $\theta_1 = 0.5$, we see the vertical distance from out fit to the data points increase.



This increases our cost function to 0.58. Plotting several other points yields to the following graph:



Thus as a goal, we should try to minimize the cost function. In this case, $\theta_1 = 1$ is out global minimum.

Cost Function - Intuition II

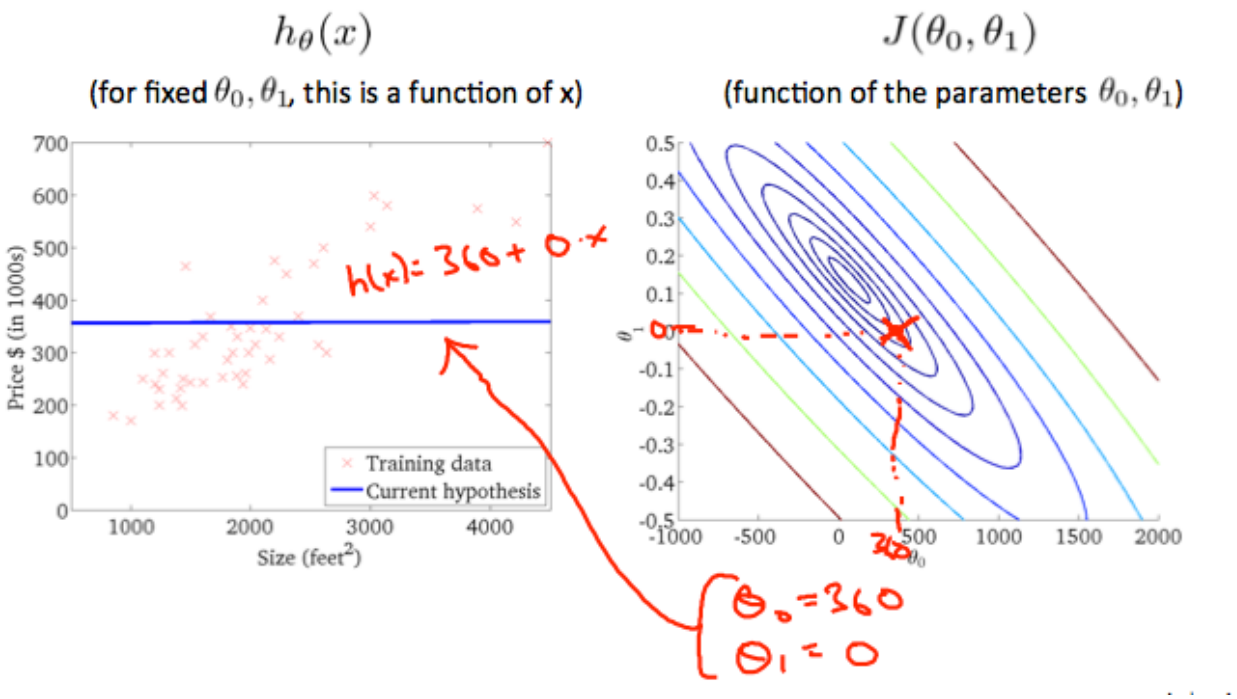
A contour plot is a graph that contains many contour lines. A contour line of a two variable function has a constant value at all points of the same line. An example of such a graph is the one to the right below.

Oops! We couldn't find this image

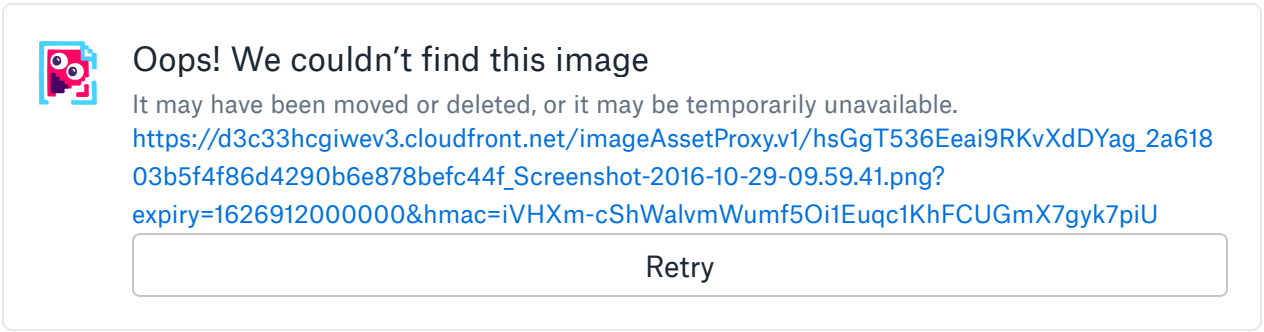
It may have been moved or deleted, or it may be temporarily unavailable.
https://d3c33hcgiwev3.cloudfront.net/imageAssetProxy.v1/N2oKYp2wEeaVChLw2Vaaug_d4d1c5b1c90578b32a6672e3b7e4b3a4_Screenshot-2016-10-29-01.14.37.png?expiry=1626912000000&hmac=LxtUyj0movw0jfyHTljgZrKdtRZCyo4AcaMZH7r5Rvw

Retry

Taking any color and going along the "circle", one would expect to get he same value of the cost function. For example, the three green points found on the green line above have the same value for $J(\theta_0, \theta_1)$ and as a result, they are found along the same line. The circled x displays the value of the cost function for the graph on the left when $\theta_0 = 800$ and $\theta_1 = -0.15$. Taking another $h(x)$ and plotting its contour plot, one gets the following graphs:



When $\theta_0 = 360$ and $\theta_1 = 0$, the values of $J(\theta_0, \theta_1)$ in the contour plot gets closer to the center thus reducing the cost function error. Now giving out hypothesis function a slightly positive slope results in a better fit of the data.



The graph above minimizes the cost function as much as possible and consequently, the results of θ_1 and θ_0 tend to be around 0.12 and 250 respectively. Plotting those values on our graph to the right seems to put our point on the inner most "circle".

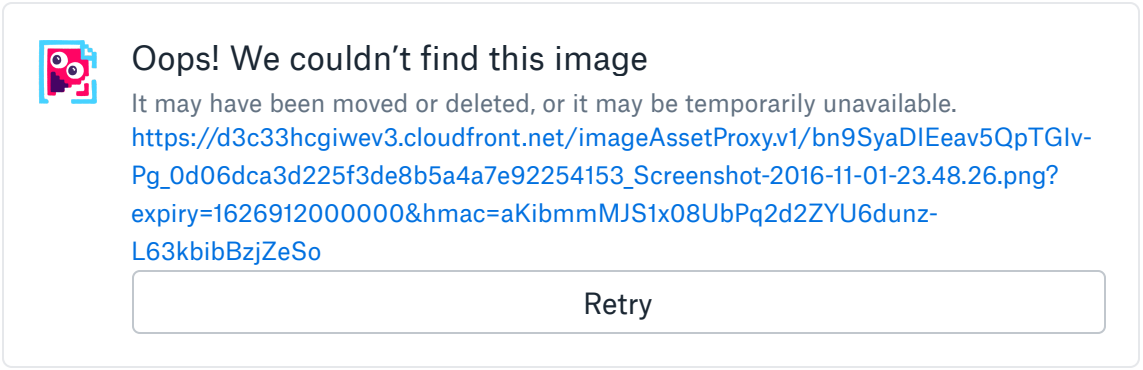
Parameter Learning

Gradient Decent

So we have our hypothesis 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 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 out hypothesis with those specific θ parameters. The graph below depicts such a setup.

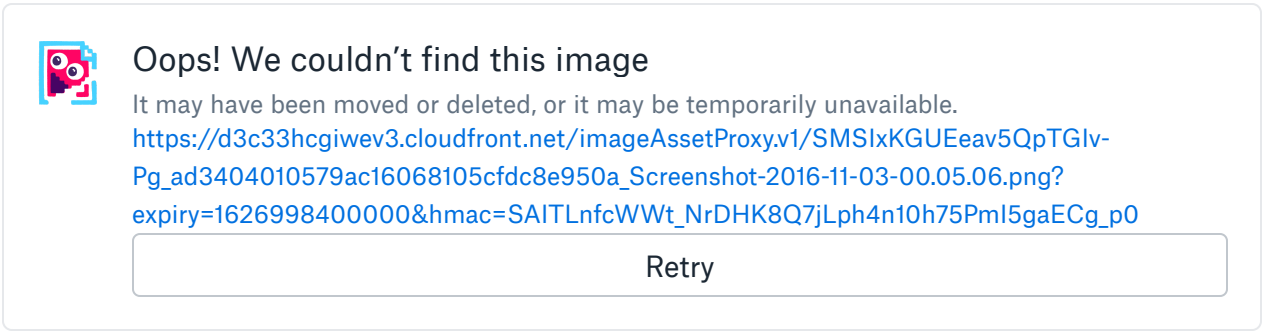


Gradient Decent Intuition

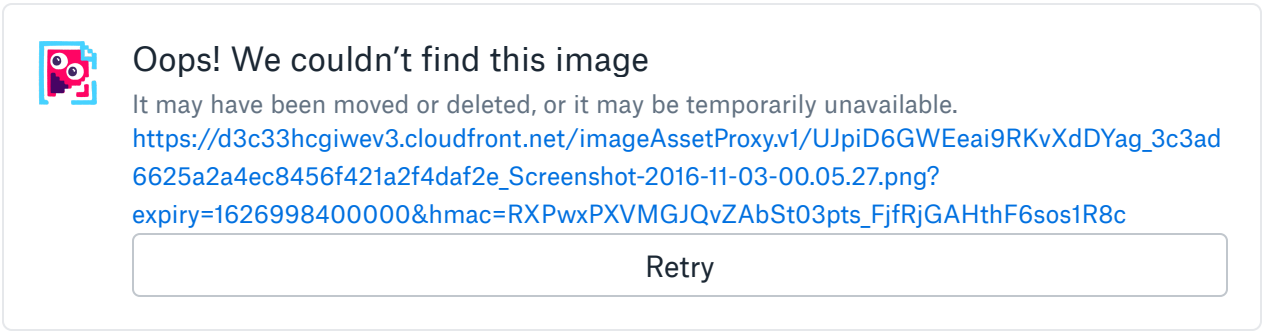
Here explore the scenario with one parameter θ_1 and plot its cost function to implement a gradient descent. Out formula for a single parameter is

$\theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$

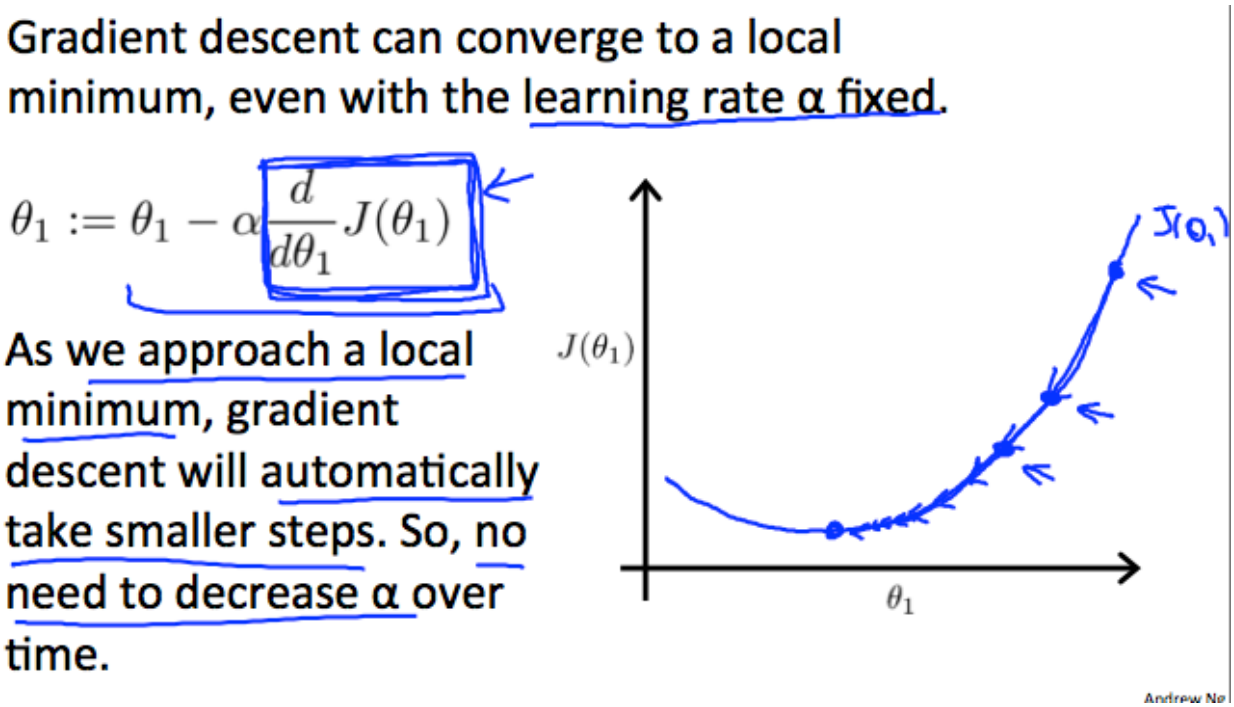
Regardless of the slope sign for $\frac{d}{d\theta_1} J(\theta_1)$, θ_1 eventually converges to its minimum value. The following graph shows that when the slope is negative, the value of θ_1 decreases.



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 implies that our step size is wrong.



How does gradient descent converge with a fixed step size α?
The intuition behind the convergence is that $\frac{d}{d\theta_1}J(\theta_1)$ approaches 0 as we approach the bottom of our convex function. At the minimum, the derivative will always be 0 and thus we get
$\theta_0 := \theta_1 - \alpha \times 0$



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

$$\text{Repeat until convergence:}\{$$
$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x_i\right) - y_i\right)$$
$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x_i\right) - y_i\right)$$
$$\}$$

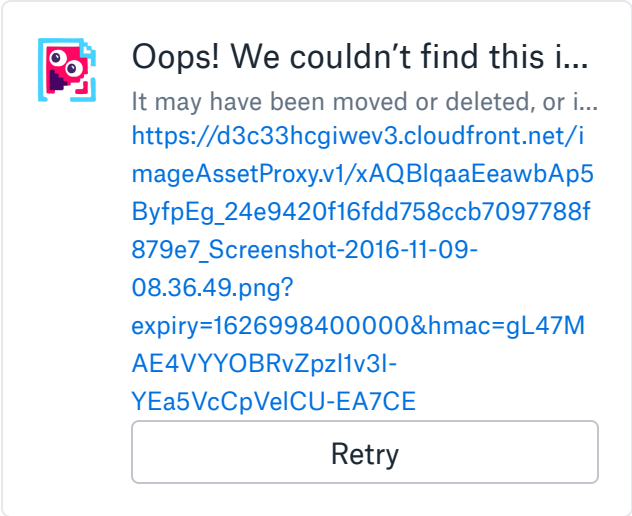
where m is the size of the training set, θ_0 a constant that will be changing simultaneously with θ_1 and x_i, y_i are values of the given training set (data).

Note that we have separated out the two cases for θ_j into separate equations for θ_0 and θ_1; and that for θ_1 we are multiplying x_i at the end due to the derivative. The following is a derivation of $\frac{\partial}{\partial \theta_j} J(\theta)$ for a single example:

$$\frac{\partial}{\partial \theta_j} J(\theta) = \frac{\partial}{\partial \theta_j} \frac{1}{2} \left(h_\theta(x) - y\right)^2$$
$$= 2 \frac{1}{2} \left(h_\theta(x) - y\right) \cdot \frac{\partial}{\partial \theta_j} \left(h_\theta(x) - y\right)$$
$$= \left(h_\theta(x) - y\right) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{i=0}^{n} \theta_i x_i - y\right)$$
$$= \left(h_\theta(x) - y\right) x_j$$

The point of all this is that if we start with a guess for our hypothesis and then repeatedly apply these gradient descent equations, out 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 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 successive values of θ that gradient descent went through as it converged to its minimum.

Week 2

Multivariate Linear Regression

Multiple Features

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

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

$x_j^{(i)}$ = value of feature j in the i^{th} training example

$x^{(i)}$ = the input (features) of the i^{th} training example

$m =$ the number of training examples
$n =$ the number of features
The multivariate form of the hypothesis function accommodating these multiple features is as follows:
$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$
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:

$$h_\theta(x) = \begin{bmatrix}\theta_0 & \theta_1 & \cdots & \theta_n\end{bmatrix} \begin{bmatrix}x_0 \\ x_1 \\ \vdots \\ x_n\end{bmatrix} = \theta^T x$$

This is a vectorization of our hypothesis function for one training example; see lessons on vectorization to learn more.
Remark: Note that for convenience reasons in this course we assume $x_0^{(i)} = 1$ for $(i \in 1, \ldots, m)$. This allows us to do matrix operations with θ and x. Hence making the two vectors θ and $x^{(i)}$ match each other element-wise (that is, have the same number of elements: $n + 1$.

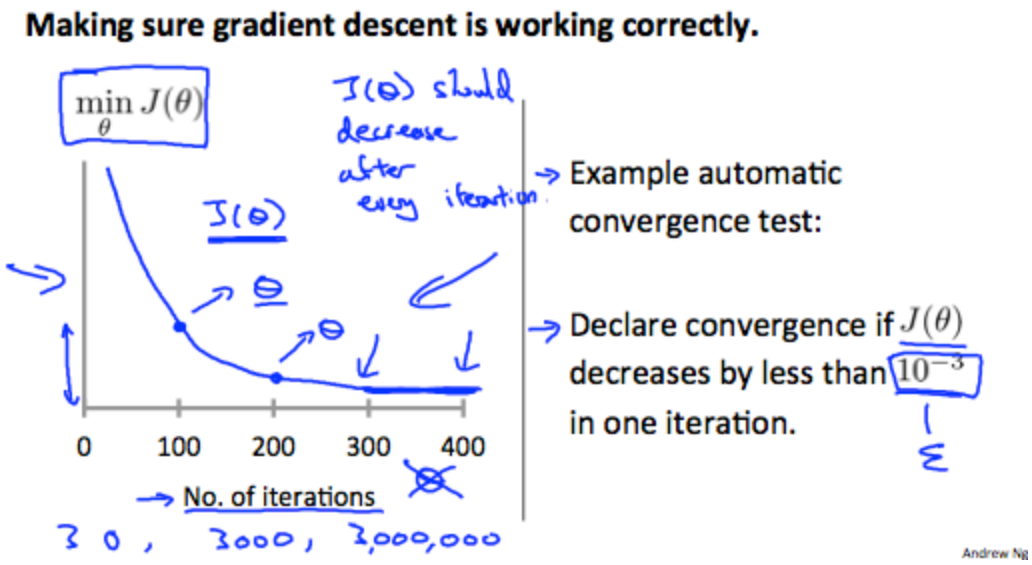
Gradient Descent for Multiple Variables
The gradient descent equation itself is generally the same form; we just have to repeat if for n features.
repeat until convergence: $\{$
$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_0^{(i)}$
$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_1^{(i)}$
$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_2^{(i)}$
$\vdots$
$\}$
In other words:
repeat until convergence: $\{$
$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)}$ for $j := 0 \ldots n$

Gradient Decent in Practice 1 — Feature Scaling

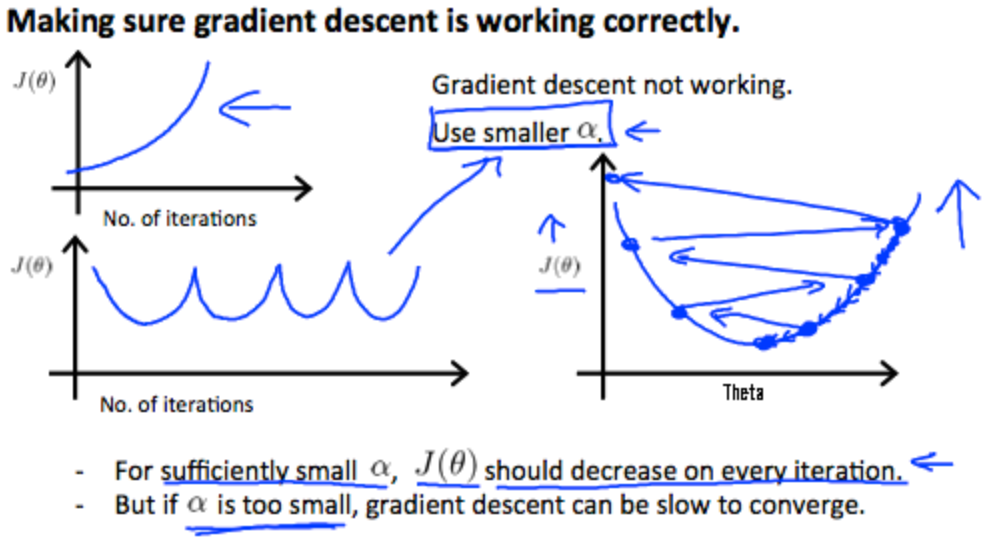
We can speed up gradient descent by having each of our input values in roughly the same range. This is because θ will descend quickly on small ranges and slowly on large ranges, and so will oscillate inefficiently down to the optimum when the variables are very uneven.
The way to prevent this is to modify the ranges of out input variables so that the are all roughly the same. Ideally:
$-1 \leq x_{(i)} \leq 1$
or
$-0.5 \leq x_{(i)} \leq 0.5$
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:
$x_i := \frac{x_i - \mu_i}{s_i}$
Where μ_i is the **average** of all the values for feature (i) and s_i is the range of values (max - min), of s_i is the standard deviation.
Note that dividing by the range, or dividing by the standard deviation, give different results.

Gradient Descent in Practice II — Learning Rate

Debugging gradient descent: Make a plot with *number of iterations* of the x-axis. Now plot the cost function $J(\theta)$ of the number of iterations of gradient descent. If $J(\theta)$ ever increases, then you probably need to decrease α.
Automatic convergence test: Declare convergence if $J(\theta)$ decreases by less than E in one iterations, 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 $J(\theta)$ will decrease on every iteration.



To summarize:

- If α is too small: slow convergence.
- If α is too large $J(\theta)$ may not decrease on every iteration and this may not converge.

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 $x_1 \times x_2$

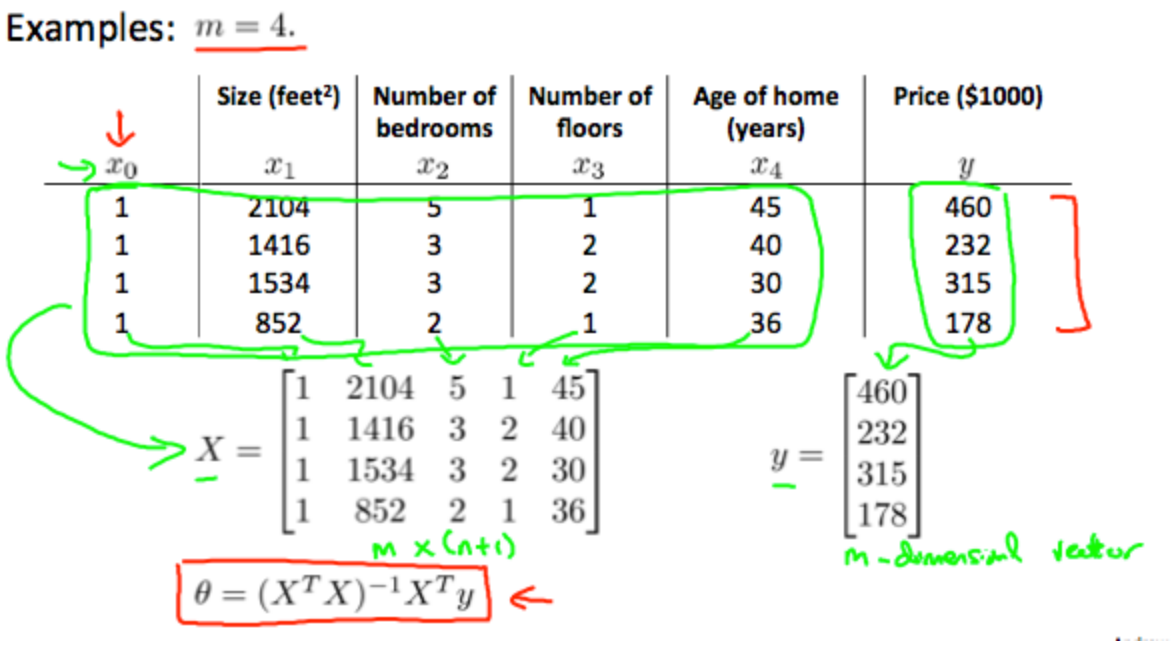
- Polynomial Regression

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).
For example, if our hypothesis function is $h(\theta) = \theta_0 + \theta_1 x_1$ then we can create addition features based on x_1, to get the quadratic function $h(\theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$ or the cubic function $h(\theta) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$.
In the cubic version, we have created new features x_2 and x_3 where $x_2 = x_1^2$ and $x_3 = x_1^3$.
To make it a square root function, we could do: $h(\theta) = \theta_0 + \theta_1 x_1 + \theta_2 \sqrt{x_1}$
One important thing to keep in mind is if you choose your features this way then feature scaling becomes very important.
e.g. if x_1 has a range of 1-1000 then the range of x_2 becomes 1-1000000 and that of x_1^3 becomes 1-1000000000.

Computing Parameters Analytically

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. The the *Normal Equation* method, we will minimize J by explicitly taking its derivatives with respect to θ_j's and setting them to zero. This allows us to find the optimum θ without iterations. The normal equation formula is given below:
$\theta = \left(X^T X\right)^{-1} X^T y$



There is **no need** to do feature scaling with the normal equation.
The following is a comparison of gradient descent and the normal equation:

Gradient Descent	Normal Equation
Need to choose α	No need to choose α
Needs many iterations	No need to iterate
$O\left(kn^2\right)$	$O\left(n^3\right)$, need to calculate the inverse of $X^T X$
Works well when n is large	Slow if n is very large

With the normal equation, computing the inversion has complexity $O\left(n^3\right)$. So if we have a very large number of features, the normal equation will be slow. In practice, when n exceeds 10,000 it might be a good time to go from a normal solution to an iterative process.

Normal Equation Noninvertibility
When implementing the normal equation in octave we want the use the `pinv` function rather than `inv`. The `pinv` function will give you a value of θ even if $X^T X$ is not invertible.
If $X^T X$ is **noninvertible**, the common causes might be having:

- Redundant features, where two features are very closely related (i.e. the are linearly dependent)

- Too many features (e.g. $m \leq n$). 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.

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 1 and all less than 0.5 as 0. However, this method doesn't work well because classification is not actually a linear function.

The classification problem is just like the linear regression problem, except that the values 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 multi-class case.) For instance, if we are trying to build a spam classifier for email, then $x^{(i)}$ may be some feature of a piece of email, and y maybe 1 if it is a piece of spam email, and 0 otherwise. Hence $y \in \{0, 1\}$. 0 is also called the negative, and 1 the positive class, and they are sometime denoted by the symbols "$-$" and "$+$". Given $x^{(i)}$, the corresponding $y^{(i)}$ is also called the label for the training example.

Hypothesis Representation

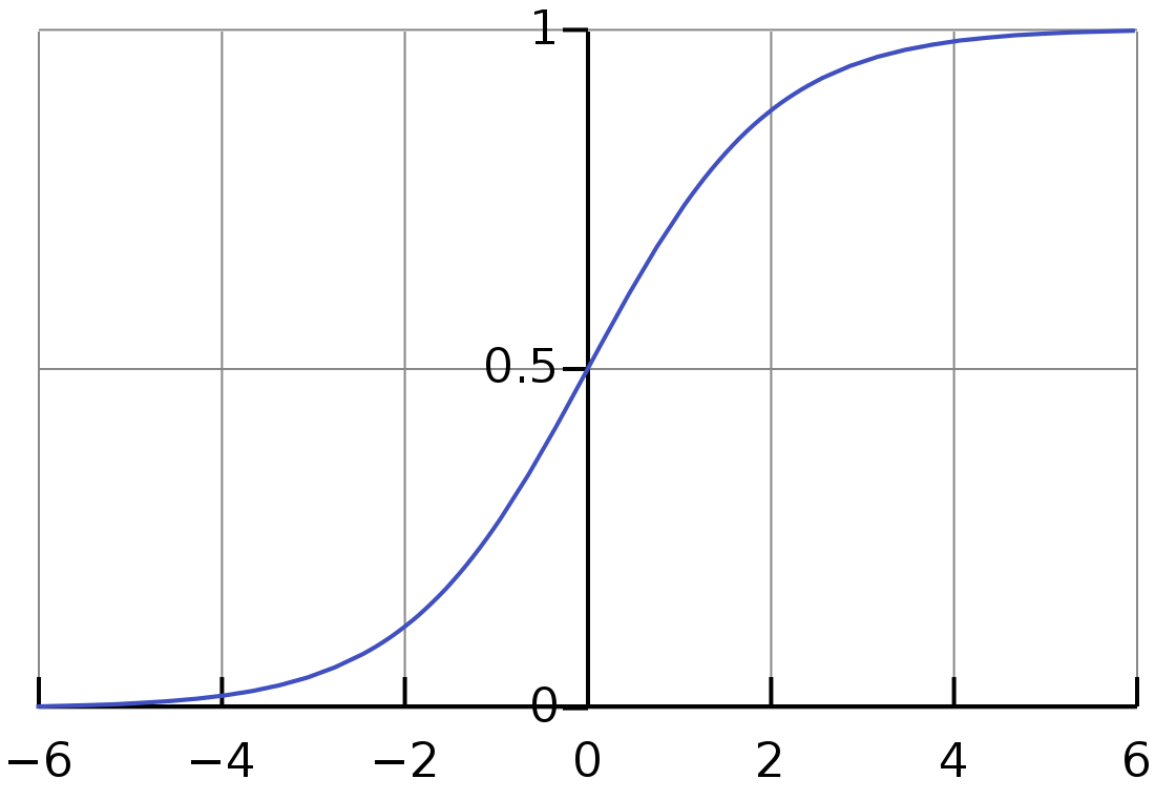
We could approach the classification problem ignoring the fact that y is discrete-valued, and use our old #linear-regression algorithm to try and predict y given x. However, it is easy to construct examples where this method performs poorly. Intuitively, it also doesn't make sense for $h_\theta(x)$
to take values larger than 1 or smaller than 0 when we know that $y \in \{0, 1\}$. To fix this, let's change the form for our hypothesis $h_\theta(x)$ to satisfy $0 \leq h_\theta(x) \leq 1$. This is accomplished by plugging $\theta^T x$ into the logistics function.

Out new for uses the #sigmoid-function, also called the #logistic-function.

$$h_\theta(x) = g(\theta^T x)$$
$$z = \theta^T x$$

$g(z) = \frac{1}{1+e^{-z}}$

The following image shows us what the #sigmoid-function looks like:



Sigmoid function - Wikipedia

The function $g(z)$, shown here, maps any real number to the $(0, 1)$ interval, making it very useful for transforming an arbitrary-valued function into a function better suited for classification.
$h_\theta(x)$ will give us the **probability** that our output is 1. For example, $h_\theta(x) = 0.7$ will give us a probability of 70% that our output is 1. Our probability that our output is 0 is just the complement of our probability that it is 1 (e.g. if probability that it is 1 is 70%, then the probability that it is 0 is 30%).

$h_\theta(x) = P(y = 1|x; \theta) = 1 - P(y = 0|x; \theta)$
$P(y = 1|x; \theta) + P(y = 0|x; \theta) = 1$

Decision Boundary
In order to get out discrete 0 or 1 classification, we can translate the output of the hypothesis function as follows:

$h_\theta(x) \geq 0.5 \rightarrow y = 1$
$h_\theta(x) < 0.5 \rightarrow y = 0$

The way our #logistic-function g behaves is that when its input is greater than or equal to 0, its output is greater than or equal to 0.5:

$g(z) \geq 0.5$
when $z \geq 0$

Remember:

$z = 0, e^0 = 1, g(z) = 0.5$
$z \rightarrow \infty, e^\infty = 0, g(z) = 1$
$z \rightarrow -\infty, e^{-\infty} = \infty, g(z) = 0$

So if our input to g is $\theta^T X$, then it means:

$h_\theta(x) = g(\theta^T x) \geq 0.5$
when $\theta^T x \geq 0$

From these statements we can now say:

$\theta^T x \geq 0, y = 1$
$\theta^T x < 0, y = 0$

The #decision-boundary is the line that separates the area where $y = 0$ and where $y = 1$. It is created by our hypothesis function.
Example

$$\theta = \begin{bmatrix} 5 \\ -1 \\ 0 \end{bmatrix}, X = \begin{bmatrix} 0 \\ x_1 \\ x_2 \end{bmatrix}, h_\theta(x) = \theta^T X$$
$y = 1$ if $\theta_0 + \theta_1 x_1 + \theta_2 x_2 \geq 0$
$5 - x_1 \geq 0$
$-x_1 \geq -5$

$x_1 \leq 5$

In this case, our decision boundary is a straight vertical line place on the graph where $x_1 = 5$ and everything to the left of that denotes $y = 1$, while everything to the right denotes $y = 0$.
Again, the inputs to the #sigmoid-function $g(z)$ (e.g. $\theta^T X$) doesn't need to be linear, and could be a function that describes a circle (e.g. $z = \theta_0 + \theta_1 x_1^2 + \theta_2 x_2^2$) or any shape to fit our data.

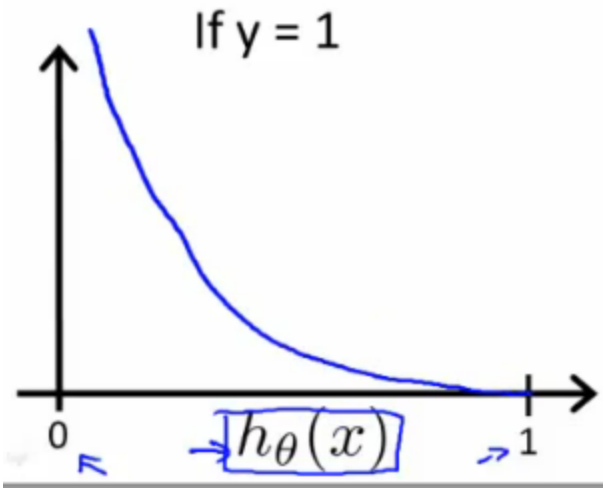
Logistic Regression Model

Cost Function

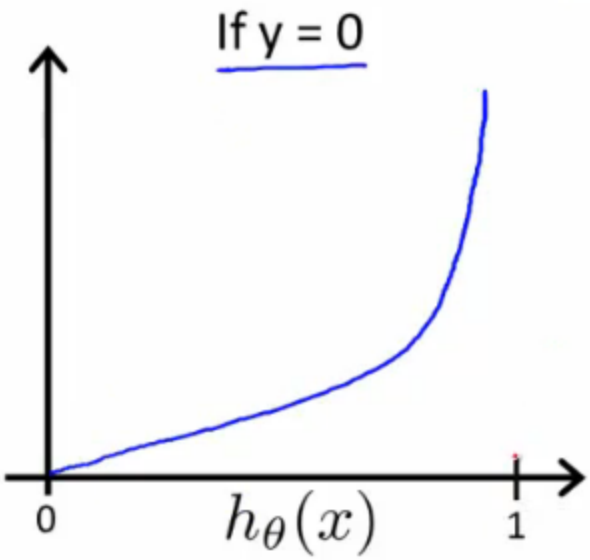
We cannot use the cost function we used for linear regression because the #logistic-function will cause the output to be wavy, causing many local optima. In other words, it will no be a #convex function.
Instead, our cost function for linear regression look like:

$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \frac{1}{2} \text{Cost}(h_\theta(x^{(i)}), (y^{(i)}))$
$\text{Cost}(h_\theta(x^{(i)}), (y^{(i)})) = -\log(h_\theta(x))$ if $y = 1$
$\text{Cost}(h_\theta(x^{(i)}), (y^{(i)})) = -\log(1 - h_\theta(x))$ if $y = 0$

When $y = 1$, we get the following plot for $J(\theta)$ versus $h_\theta(x)$



Similarly, when $y = 0$, we get the following plot for $J(\theta)$ versus $h_\theta(x)$



$\text{Cost}(h_\theta(x), y) = 0$ if $h_\theta(x) = y$
$\text{Cost}(h_\theta(x), y) \to \infty$ if $y = 0$ and $h_\theta(x) \to 1$
$\text{Cost}(h_\theta(x), y) \to \infty$ if $y = 1$ and $h_\theta(x) \to 0$

If our correct answer y is 0, then the #cost-function will be 0 if our hypothesis function also outputs 0. If our hypothesis approaches 1, then the cost function will approach infinity.
If our correct answer y is 1, then the cost function will be 0 if our hypothesis function outputs 1. If our hypothesis approaches 0, then the cost function will approach infinity.
Note that writing the cost function in this way guarantees that $J(\theta)$ is convex for logistic regression.

Simplified Cost Function and Gradient Decent
We can compress out #cost-function's two conditional cases into one case:
$\text{Cost}(h_\theta(x), y) = -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x))$
Notice that when $y = 1, 1 - y = 0$ and the second term will not affect the result. If $y = 0$, the first term equals zero and will not affect the result.
We can fully write out the entire cost function as follows:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right]$$

A vectorized implementation is:

$h = g(X\theta)$
$J(\theta) = \frac{1}{m} \cdot \left(-y^T \log(h) - (1 - y^T) \log(h)\right)$

Gradient Decent
Remember the general form of #gradient-decent is:

Repeat $\{$
$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta} J(\theta)$
$\}$

We can work out the derivative part using calculus to get:

Repeat $\{$
$\theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) x_j^{(i)}$
$\}$

Notice that this algorithm is identical to the one we used in linear regression. We sill have to simultaneously update all values in θ.
A vectorized implementation is:

$\theta := \theta - \frac{\alpha}{m} X^T \left(g(X^T\theta) - y\right)$

Advanced optimization
Conjugate gradient, *BFGS*, and *L-BFGS* are more sophisticated, faster ways to optimize θ that can be used instead of gradient descent. We suggest that you should not write these more sophisticated algorithms yourself (unless you are an expert in numerical computing) but use the libraries instead, as they're already tested and highly optimized. Octave provides them.
We first need to provide a function that evaluates the following two functions for a given input value θ:
$J(\theta)$
$\frac{\partial}{\partial \theta} J(\theta)$
We can a single function that returns both of these:

```
function [jVals, gradient] = costFunction(theta)
  jVals = [...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()`.

```
options = optimset('GradObj', 'on', 'MaxIter', 100);
initialTheta = zeros(2,1);
   [optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialTheta, options);
```

We give to the function `fminunc()` our cost function, our initial vector of theta values, and the `options` object that we created beforehand.

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

$$
\begin{aligned}
&y = \{0, 1, \dots, n\} \\
&h_\theta^{(0)}(x) = P(y = 0 | x : \theta) \\
&h_\theta^{(1)}(x) = P(y = 1 | x : \theta) \\
&\vdots \\
&h_\theta^{(n)}(x) = P(y = n | x : \theta) \\
&\text{prediction} = \max_i \left(h_\theta^{(i)}(x) \right)
\end{aligned}
$$

We are basically choosing one class and lumping all the others into a single second class. We do this repeatedly, applying binary regression to each case, and then use the hypothesis that returned the highest value as our prediction.
The following three images shows how we could classify 3 classes:

Oops! We couldn't find this image

It may have been moved or deleted, or it may be temporarily unava...
https://d3c33hcgiwev3.cloudfront.net/imageAssetProxy.v1/cqmPjanSEeawbAp5ByfpEg_299fcfbd527b6b5a7440825628339c54_Screenshot-2016-11-13-10.52.29.png?expiry=1612483200000&hmac=N0djoJTTkHtGSIifuNLv7Jxv2ggB5Fk5i9mYlfqSGkM

Retry

The summarize, train a logistic regression classifier $h_\theta(x)$ for each class to predict the probability that $y = i$. To make a prediction on a new x, pick the class that maximizes $h_\theta(x)$.

Solving the Problem of Overfitting

The Problem of Overfitting

Consider the problem of predicting y from $x \in R$. The leftmost figure below shows the result of fitting a $y = \theta_0 + \theta_1 x_1$ to a dataset. We see that the data doesn't really lie on the straight line, and so the fit is not very good.
#insert-figure
Instead, if we had added an extra feature x^2, and fit $y = \theta_0 + \theta_1 x + \theta_2 x^2$, when we obtain a slightly better fit to the data (see middle figure). Naively, it might seem that the more features we add, the better. However, there is also a danger in adding too many features: the rightmost figure is the result of fitting a fifth order polynomial $y = \sum_{j=0}^{5} \theta_j x^j$. We see that even though the fitter curve passes through the data perfectly, we would not expect this to be a very good predictor of, say, housing prices y for different living areas x. Without formally defining what these terms mean, we'll say the figure on the left shows an instance of **underfitting**—in which the data clearly shows structure not captured by the model—and the figure on the right is an example of **overfitting**.
Underfitting, or high bias, is when the form of out 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 buy does not generalise well to predict new data. Is 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 features, but reduce the magnitude of the parameters θ_j.
- Regularization works well when we have a lot of slightly useful features.

Cost Function
If we have overfitting from our hypothesis function, we can reduce the weight that some of the terms in our function carry by increasing their cost.
Say we wanted to make the following function more quadratic:

$$\theta_0 + \theta_1 x^2 + \theta_3 x^3 + \theta_4 x^4$$

We'll want to eliminate the influence of $\theta_3 x^3$ and $\theta_4 x^4$. Without actually getting rid of these features or changing the form of our hypothesis, we can instead modify our **cost function**.

$$\min_\theta = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2 + 1000 \cdot \theta_3^2 + 1000 \cdot \theta_4^2$$

We've added two extra terms at the end to inflate the cost of θ_3 and θ_4. Now, in order for the cost function to get clade to zero, we will have to reduce the values of θ_3 and $\theta_4$to near zero. This will in turn greatly reduce the values of $\theta_3 x^3$ and $\theta_4 x^4$ in our hypothesis function. As a result, we see that the new hypothesis (depicted by the pick curve) looks like a quadratic function but fits the data better due to the extra small terms $\theta_3 x^3$ and $\theta_4 x^4$.
We could also regularize all of our θ parameters in a single summation as:

$$\min_\theta = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2 + \lambda \sum_{j=1}^{n} \theta_j^2$$

The λ is the **regularization parameter**. It determines how much the costs of our θ parameters are inflated.
Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If λ is chosen to be too large, it may smooth out the function too much and cause underfitting. Hence, what would happen if $\lambda = 0$ or is too small?

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

Gradient Descent
We will modify our gradient descent function to separate out θ_0 from the rest of the parameters because we do not want to penalize θ_0.

#error

The term $\frac{\lambda}{m}\theta_j$ performs our regularization. With some manipulation our update rule can also be represented as:

$$\theta_j = \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}$$

The first term in the above equation, $1 - \alpha\frac{\lambda}{m}$ will always be less than 1. Intuitively you can see it as reducing the value of θ_j by some amount on every update. Notice that the second term ius noe exactly the same as it was before.

Normal Equation
Now let's approach regularization using the alternate method of the non-iterative normal equation.

To add in regularization, the equation is the same as out original, except that we add another term inside the parentheses:

#error

L is a matric with 0 at the top left and 1's down the diagonal, with 0's

everywhere else. It should have dimension $(n+1) \times (n+1)$. Intuitively, this is the identity matrix (though we are not including x_0) multiplied with a single real number λ.
Recall that is $m < n$, the $X^T X$ is non-invertible. However, when we add the term $\lambda \cdot L$
the $X^T X + \lambda \cdot L$ become invertible.

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:

#image

Cost Function
Recall that our cost function for logistic regression was:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log \left(h_\theta \left(x^{(i)} \right) \right) + \left(1 - y^{(i)} \right) \log \left(1 - h_\theta \left(x^{i} \right) \right) \right]$$

We can regularize this equation by adding a term to the end:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log \left(h_\theta \left(x^{(i)} \right) \right) + \left(1 - y^{(i)} \right) \log \left(1 - h_\theta \left(x^{i} \right) \right) \right] + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2$$

The second sum, $\sum_{j=1}^{n} \theta_j^2$ **means to explicitly exclude** the bias term, θ_0. i.e. the θ vector is indexed from 0 to n (holding $n+1$ values, θ_0 through θ_n), and this sum explicitly skips θ_0, by running from 1 to n, skipping 0. Thus, when computing the equation, we should continuously update the following equations:

#image

Week 4

Neural Networks

Model Representation I

Let's examine how we will represent a hypothesis function using neural networks. At a very simple level, neurons are basically computational units that take inputs (**dendrites**) as electrical inputs (called "spikes") that are channeled to outputs (**axons**). In our model, our dendrites are like the input features $x_1 \cdots x_n$ and the output is the result if out hypothesis function. In this model our x_0 input node is sometimes called the "bias unit". It is always equal to 1. In neural networks, we used the same logistic function as in classification, $\frac{1}{1+e^{-\theta^T x}}$, yet we sometimes call is a sigmoid (logistic) **activation** function. In this situation, our θ parameters are sometimes called "weights".
Visually, a simplistic representation looks like:
$[x_0 x_1 x_2] \rightarrow [\ ] \rightarrow h_\theta(x)$

Our input nodes (layer 1), also known as the "input layer", go into another node (layer 2), which finally outputs the hypothesis function, known as the "output layer".
We can have intermediate layers of nodes between the input and output layers called the "hidden layers".

In thus example, we label these intermediate or "hidden" layer nodes $a_0^{(2)} \cdots a_n^{(2)}$ and call them "activation units".

$a_i^{(j)} = \text{"activation" of unit } i \text{ in layer } j$
$\Theta^{(j)} = \text{matrix of weights controlling function mapping from layer } j \text{ to layer } j+1$

If we had one hidden layer, it would look like:

$[x_0 x_1 x_2 x_3] \rightarrow [a_1^{(2)} a_2^{(2)} a_3^{(2)}] \rightarrow h_\theta(x)$

The values for each of the "activation" nodes is obtained as follows:

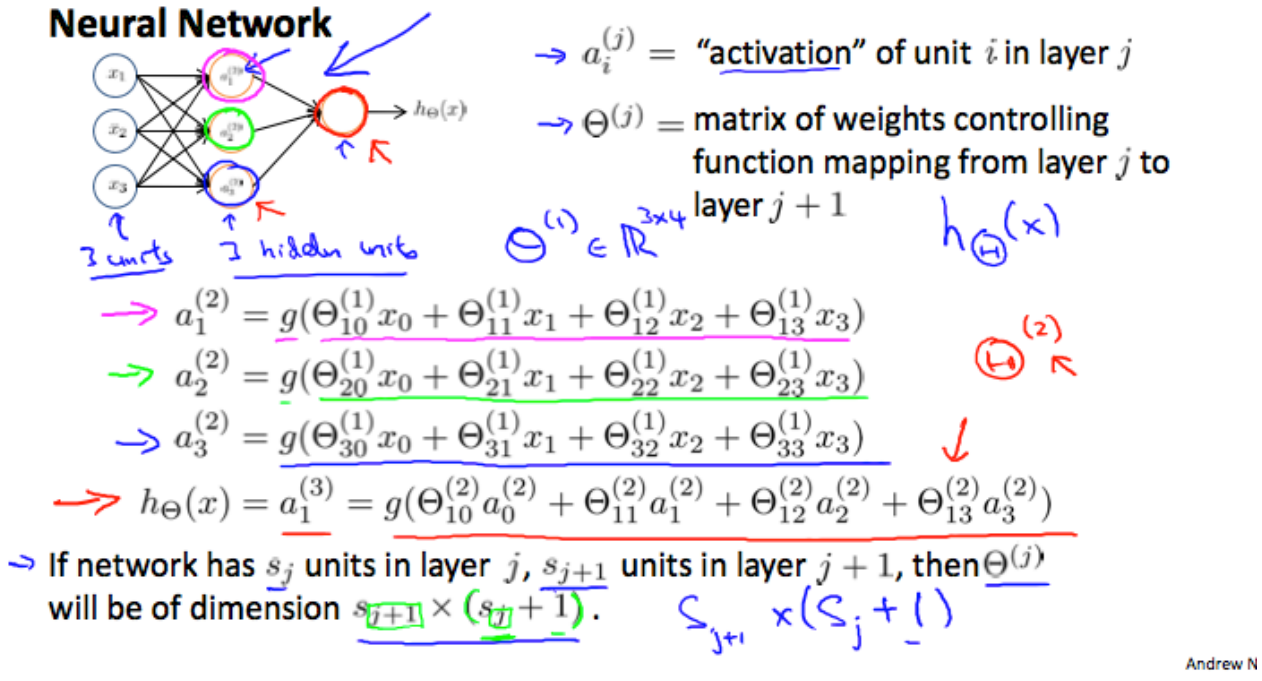
$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$
$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$
$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$
$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$

This is saying that we compute our activation nodes by using a 3x4 matrix of parameters. We apply each row of the parameters to our input to obtain the value for

one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied by yet another parameter matrix $\Theta^{(2)}$ containing the weights for our second layer of nodes.
Each layer gets its own matrix of weights, $\Theta^{(j)}$.
The dimensions of these matrices of weights is determined as follows

If the network has s_j units in layer j and s_{j+1} units in layer $j+1$, then $\Theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$

The +1 comes from the addition in $\Theta^{(j)}$ of the "bias nodes", x_0 and $\Theta_0^{(j)}$. In other words the output nodes will not include the bias nodes while the inputs will, The following image summarizes our model representation:



Example: If layer 1 has 2 input nodes and layer 2 has 4 activation nodes the dimension of $\Theta^{(1)}$ is going to be 4x3 where $s_j = 2$ and $s_{j+1} = 4$, so $s_{j+1} \times (s_j + 1) = 4 \times 3$.

Model Representation II
To re-iterate, the following is an example of a neural network:

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$

$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

In this section we'll do a vectorized implementation of the above functions. We're going to define a new variable $z_k^{(j)}$ that encompasses the parameters inside our g function. In our previous example if we replace the variable z for all the parameters we would get:

$$a_1^{(2)} = g(z_1^{(2)})$$
$$a_2^{(2)} = g(z_2^{(2)})$$
$$a_3^{(2)} = g(z_3^{(2)})$$

In other words, for layer $j+2$ and node k, the variable z will be:

$$z_k^{(2)} = \Theta_{k,0}^{(1)}x_0 + \Theta_{k,1}^{(1)}x_1 + \cdots + \Theta_{k,n}^{(1)}x_n$$

The vector representation of x and z^j is:

$$x = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix} \; z^{(j)} = \begin{bmatrix} z_1^{(j)} \\ z_2^{(j)} \\ \vdots \\ z_n^{(j)} \end{bmatrix}$$

Setting $x = a^{(1)}$, we can rewrite the equation as:

$$z^{(j)} = \Theta^{(j-1)}a^{(j-1)}$$

We are multiplying our matrix $\Theta^{(j-1)}$ with dimensions $s_j \times (n+1)$ (where s_j is the number of our activation nodes) by our vector $a^{(j-1)}$ with height $(n+1)$. This gives us our vector $z^{(j)}$ with height s_j. Now we can get a vector of our activation nodes fro layer j as follows:

$$a^{(j)} = g(z^{(j)})$$

Where our function g can be applied element-wise to our vector $z^{(j)}$.
We can then add a bias unit (equal to 1) to layer j after we have computed $a^{(j)}$. This will be element $a_0^{(j)}$ and will be equal to 1. To compute our final hypothesis, let's first compute another z vector:

$$z^{(j+1)} = \Theta^{(j)}a^{(j)}$$

We get this final z vector by multiplying the next Θ matrix after $\Theta^{(j-1)}$ with the values of all the activation nodes we just got. The last matrix $\Theta^{(j)}$ will have only **one row** which is multiplued by one column $a^{(j)}$ so that our result is a single number. We then get our final result with:

$$h_\Theta(x) = a^{(j+1)} = g(z^{(j+1)})$$

Notice that in this **last step**, between layer j and layer $j+1$, we are doing **exactly the same thing** as we did in logistic regression. Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypotheses.

Applications

Examples and Intuition I

A simple example of applying neural networks is by predicting $x_1 \text{ AND } x_2$, which is the logical "and" operator and is only true if both x_1 and x_2 are 1.
The graph of our function will look like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \left[g(z^{(2)}) \right] \rightarrow h_\Theta(x)$$

Remember that x_0 is our bias variable and is always 1.
Let's set our first theta matrix as:

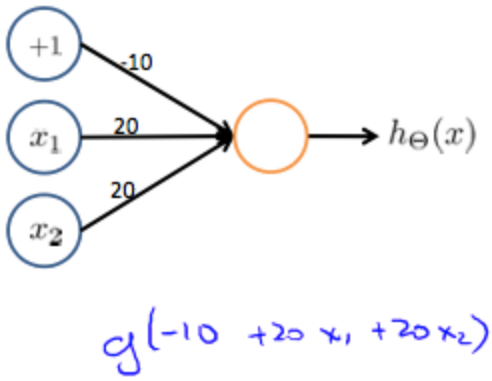
$$\Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \end{bmatrix}$$

This will cause the output of our hypothesis to only be positive if both x_1 and x_2 are 1. In other words:

$h_\Theta(x) = g(-30 + 20x_1 + 20x_2)$
$x_1 = 0 \text{ and } x_2 = 0 \text{ then } g(-30) \approx 0$
$x_1 = 1 \text{ and } x_2 = 0 \text{ then } g(-10) \approx 0$
$x_1 = 0 \text{ and } x_2 = 1 \text{ then } g(-10) \approx 0$
$x_1 = 1 \text{ and } x_2 = 1 \text{ then } g(10) \approx 1$

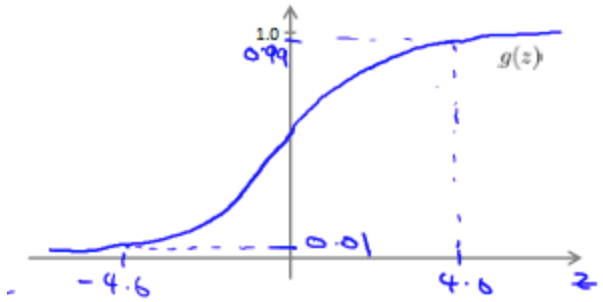
So we have constructed one of the fundamental operations in computers by using a small neural network rather than an actual AND gate. Neural networks can also be used to simulate all the other logical gates. The following is an example of the logical operator OR meaning wither x_1 is true, or x_2 is true, or both:

Example: OR function



x_1	x_2	$h_\Theta(x)$
0	0	g(-10) ≈ 0
0	1	g(10) ≈ 1
1	0	≈ 1
1	1	≈ 1

Where $g(z)$ is the following:



Examples and Intuitions II
The $\Theta^{(1)}$ matrices for AND, NOR, and OR are:

$AND:\ \ \Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \end{bmatrix}$
$NOR:\ \ \Theta^{(1)} = \begin{bmatrix} 10 & -20 & -20 \end{bmatrix}$
$OR:\ \ \Theta^{(1)} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix}$

We can combine these to get the XNOR logical operator (which gives 1 if x_1 and x_2 are both 0 or 1).

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} a^{(3)} \end{bmatrix} \rightarrow h_\Theta(x)$$

For the transition between the first and second layer, we'll use a $\Theta^{(1)}$ matrix that combines the values for AND and NOR:

$$\Theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 & 10 & -20 & -20 \end{bmatrix}$$

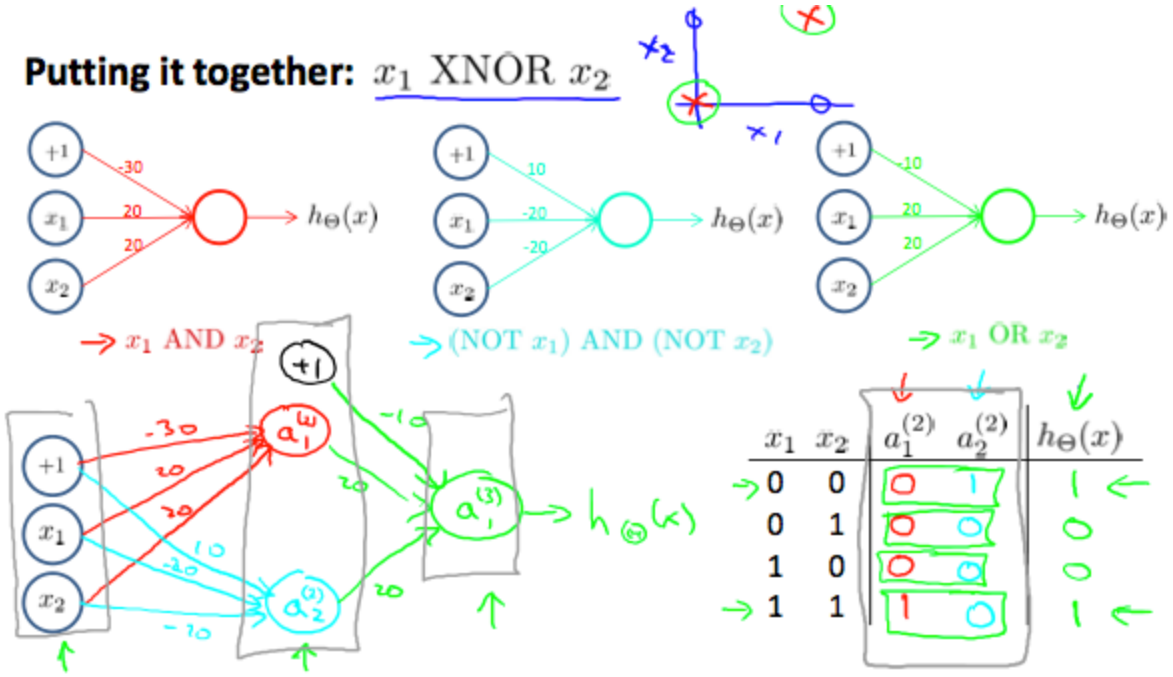
For the transition between the second and third layer, we'll use a $\Theta^{(2)}$ matrix that uses the value for OR:

$$\Theta^{(2)} = \begin{bmatrix} -10 & 20 & 20 \end{bmatrix}$$

Let's write out the value for all our nodes:

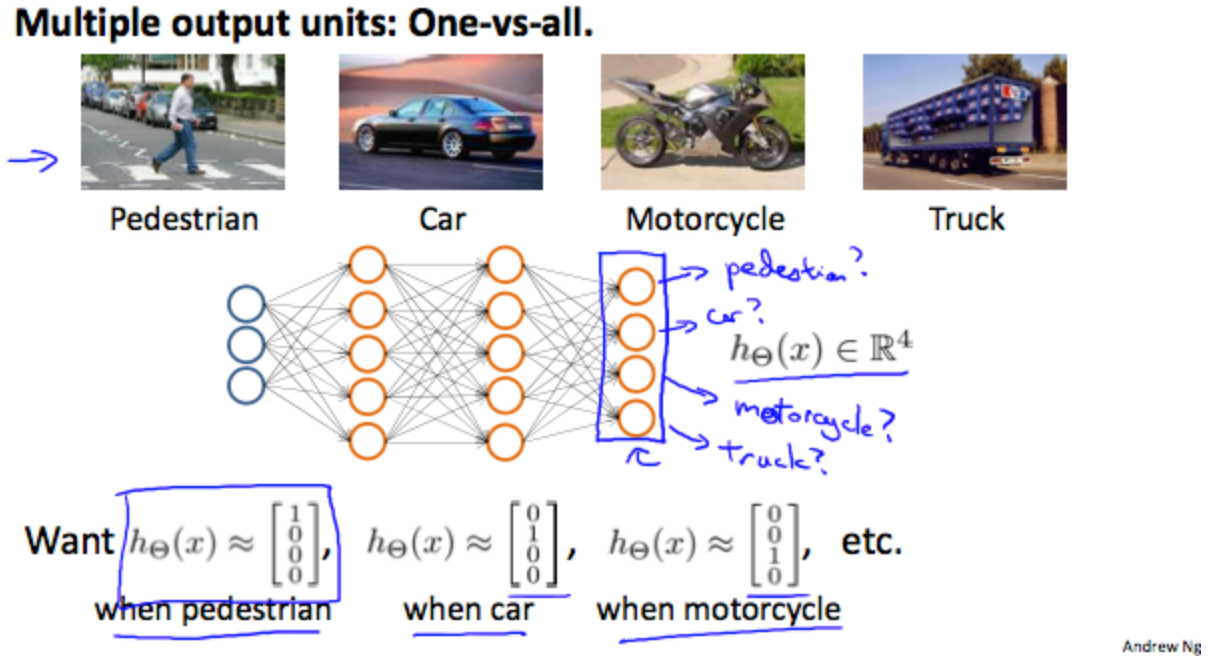
$a^{(2)} = g(\Theta^{(1)} \cdot x)$
$a^{(3)} = g(\Theta^{(2)} \cdot a^{(2)})$
$h_\Theta(x) = a^{(3)}$

And there we have the XNOR operation using a hidden layer with two nodes! The following summarizes the above algorithm:



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:



We can define our set of resulting classes as y:

$$y^{(i)} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix},$$

Each $y^{(i)}$ represents a different image corresponding to either a car, pedestrian, truck, or motorcycle. The inner layers, each provide us with some new information which leads to our hypothesis function. The set up looks like:

$$\begin{bmatrix}x_0\\x_1\\x_2\\\cdots\\x_n\end{bmatrix} \rightarrow \begin{bmatrix}a_0^{(2)}\\a_1^{(2)}\\a_2^{(2)}\\\cdots\end{bmatrix} \rightarrow \begin{bmatrix}a_0^{(3)}\\a_1^{(3)}\\a_2^{(3)}\\\cdots\end{bmatrix} \rightarrow \cdots \rightarrow \begin{bmatrix}h_\Theta(x)_1\\h_\Theta(x)_2\\h_\Theta(x)_3\\h_\Theta(x)_4\end{bmatrix}$$

Our resulting hypothesis for one set of inputs may look like:

$$h_\Theta(x) = \begin{bmatrix}0 & 0 & 1 & 0\end{bmatrix}$$

In which case our resulting class is the third one down, or $h_\Theta(x)_3$, which represents the motorcycle.