Machine Learning

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:

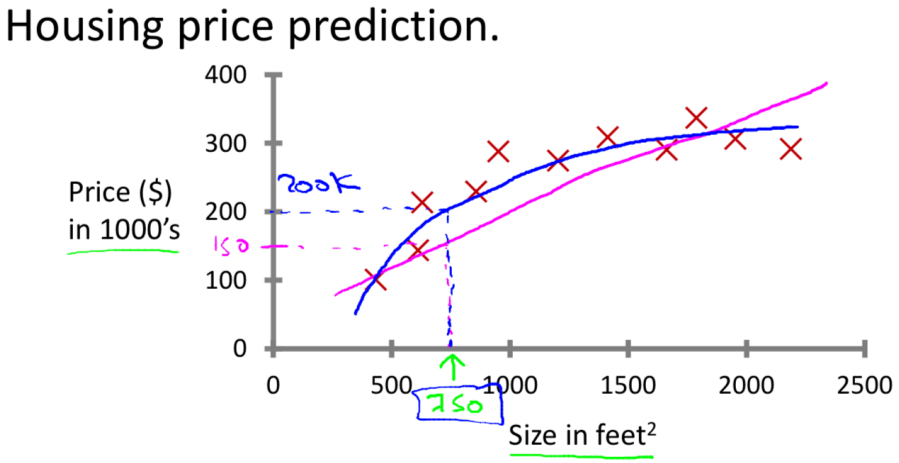
> **Supervised learning** and **Unsupervised learning**.

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.

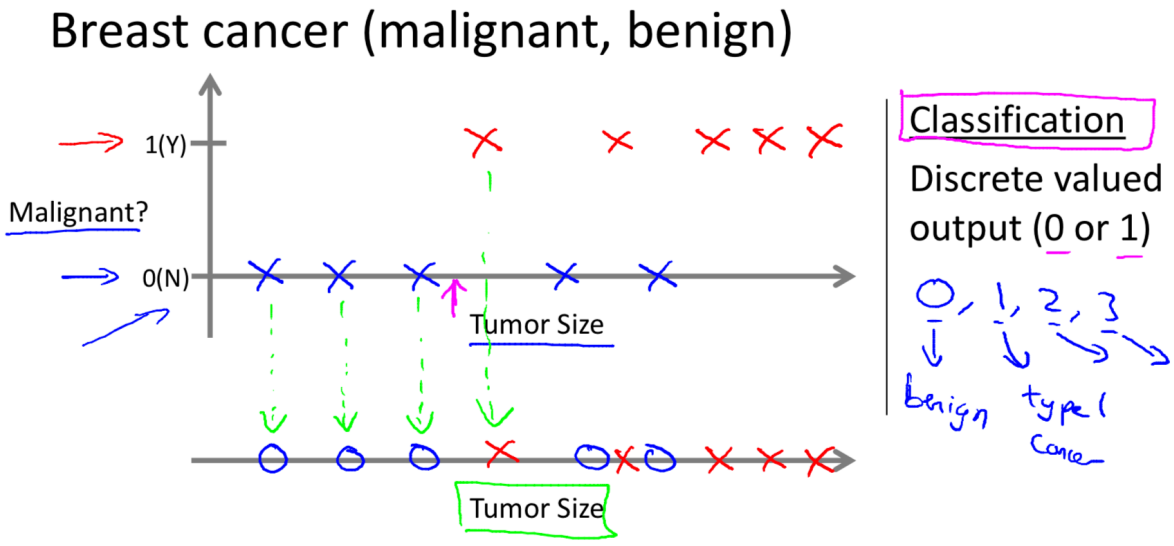


Supervised Learning
"right answers" given

Regression: Predict continuous valued output (price)

- 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:
  - Regression - Given a picture of a person, we have to predict their age on the basis of the given picture
  - Classification - Given a patient with a tumor, we have to predict whether the tumor is malignant or benign.

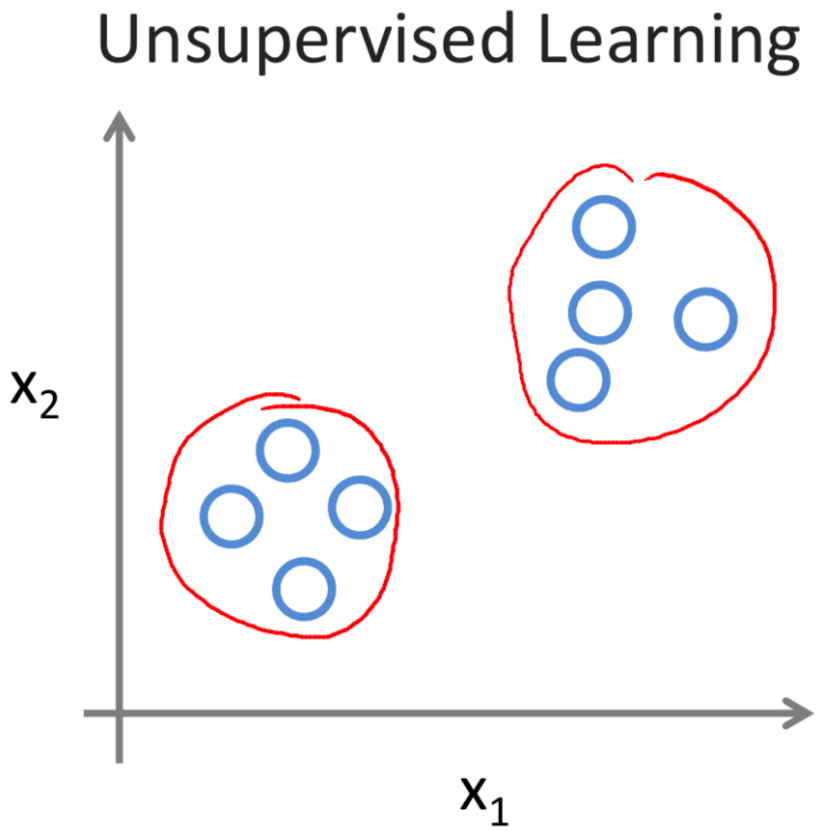


Unsupervised Learning

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.



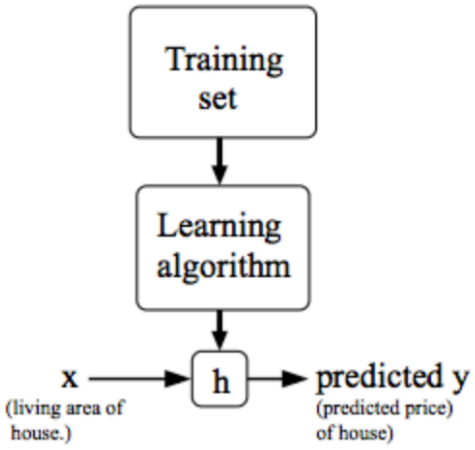
Example:

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.

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 $(x^{(i)},y^{(i)});i=1,\ldots,m$—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, $X = Y = \mathbb{R}$.

To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function $h : X \rightarrow Y$ 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.

Linear Regression with Multiple Variables

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.

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

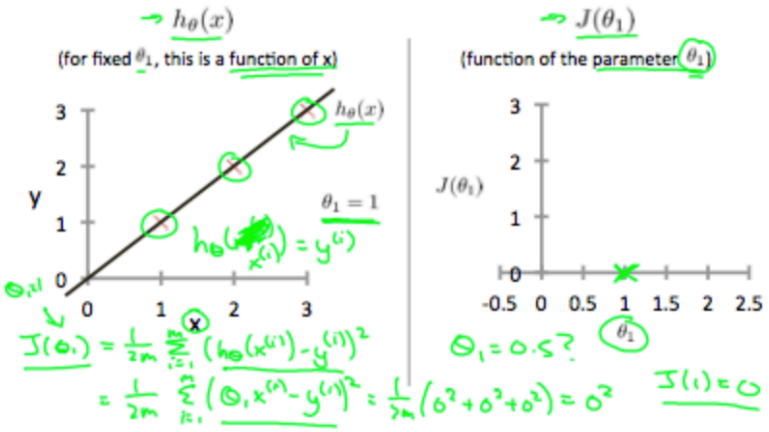
To break it apart, it is $\frac{1}{2}\ \bar{x}$ where $\bar{x}$ is the mean of the squares of $h_\theta(x_i) - y_i$, or the difference between the predicted value and the actual value.

This function is otherwise called the "Squared error function", or "Mean squared error". The mean is halved ($\frac{1}{2}$) 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:

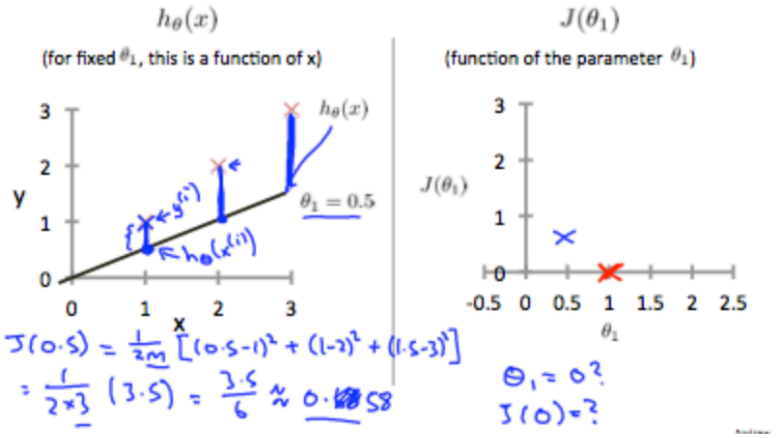
Cost Function - Intuition I

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

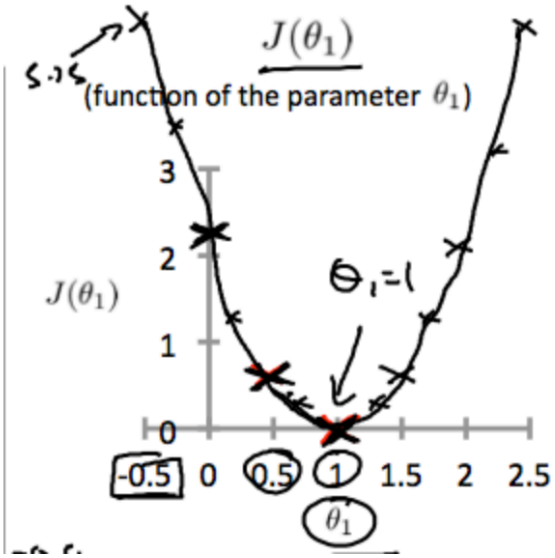
Our objective is to get the best possible line. The best possible line will be such so that the average squared vertical distances of the scattered points from the line will be the least. Ideally, the line should pass through all the points of our training data set. In such a case, the value of $J(\theta_0, \theta_1)$ will be 0. The following example shows 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 our model. Conversely, when θ_1=0.5, we see the vertical distance from our fit to the data points increase.



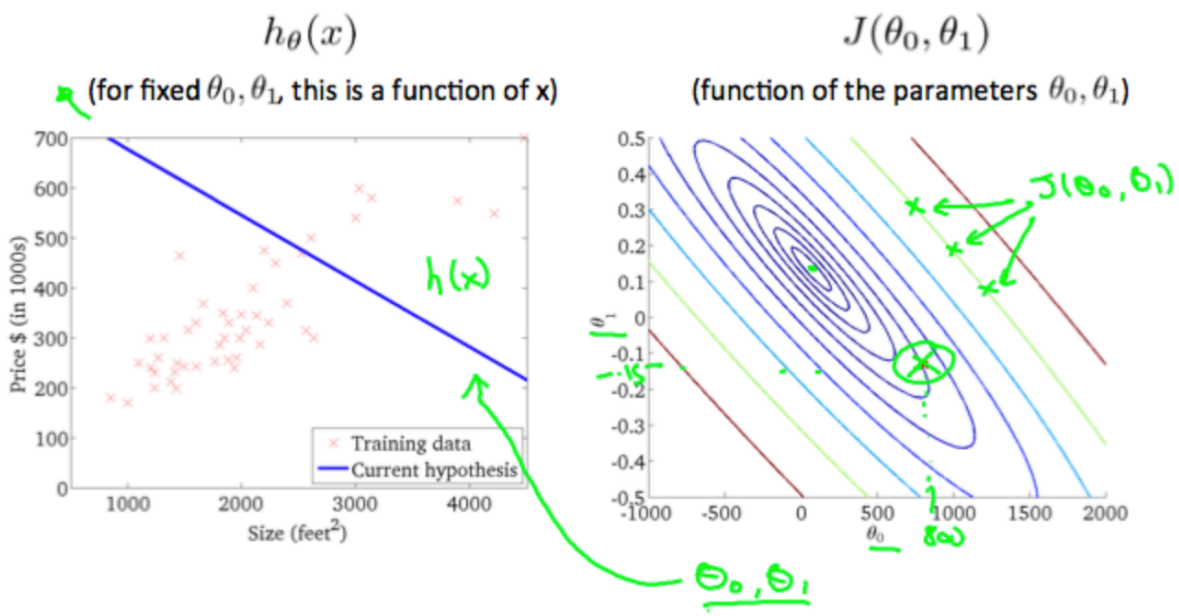
This increases our cost function to 0.58. Plotting several other points yields to the following graph:



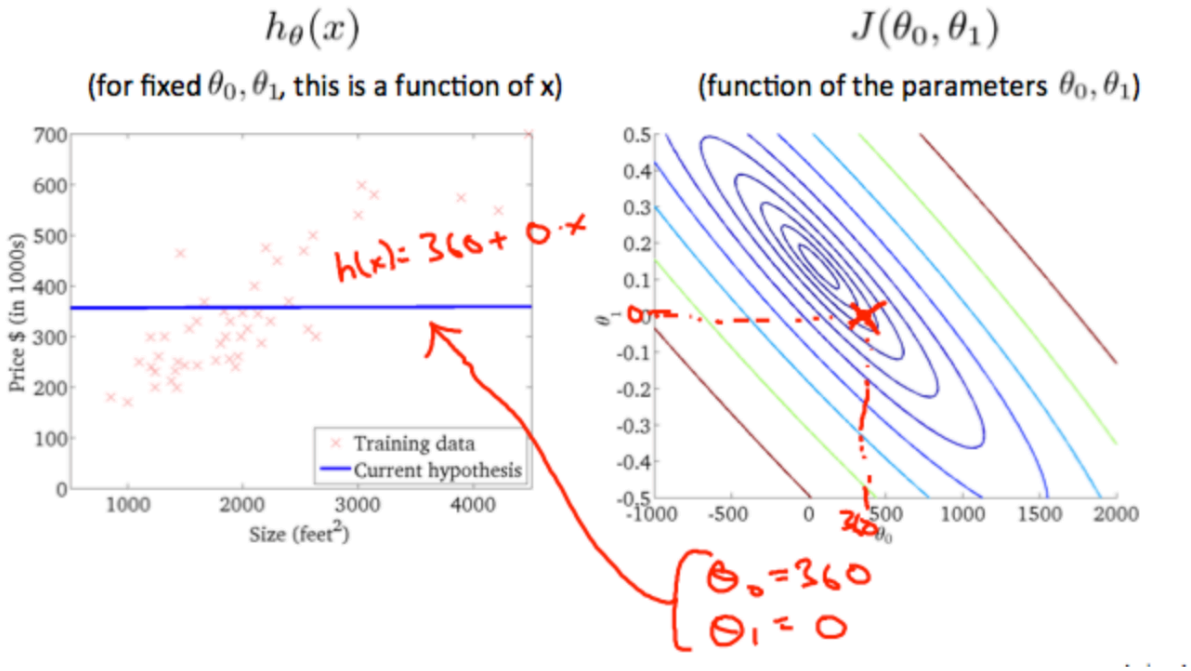
Thus as a goal, we should try to minimize the cost function. In this case, $\theta_1 = 1$ is our global minimum.

Cost Function - Intuition II

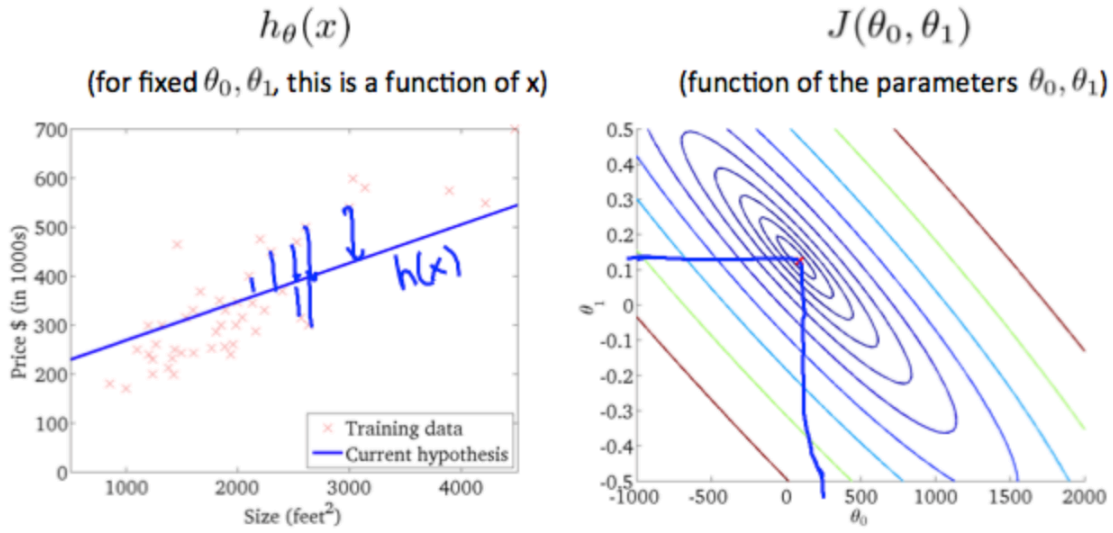
A contour plot is a graph that contains many contour lines. A contour line of a two variable function has a constant value at all points of the same line. An example of such a graph is the one to the right below.



Taking any color and going along the 'circle', one would expect to get the 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 value of $J(\theta_0, \theta_1)$ in the contour plot gets closer to the center thus reducing the cost function error. Now giving our hypothesis function a slightly positive slope results in a better fit of the data.



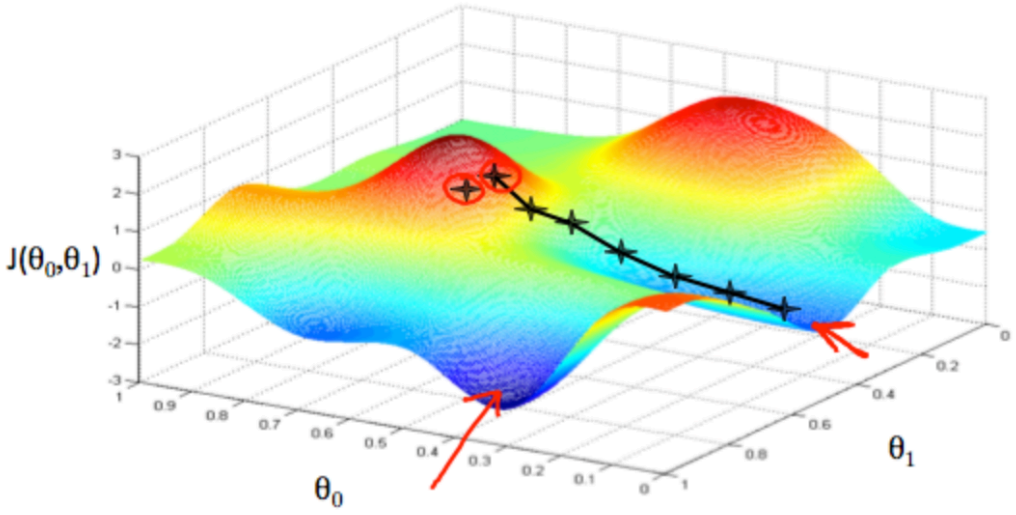
The graph above minimizes the cost function as much as possible and consequently, the result of θ_1 and θ_0 tend to be around 0.12 and 250 respectively. Plotting those values on our graph to the right seems to put our point in the center of the inner most 'circle'.

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.

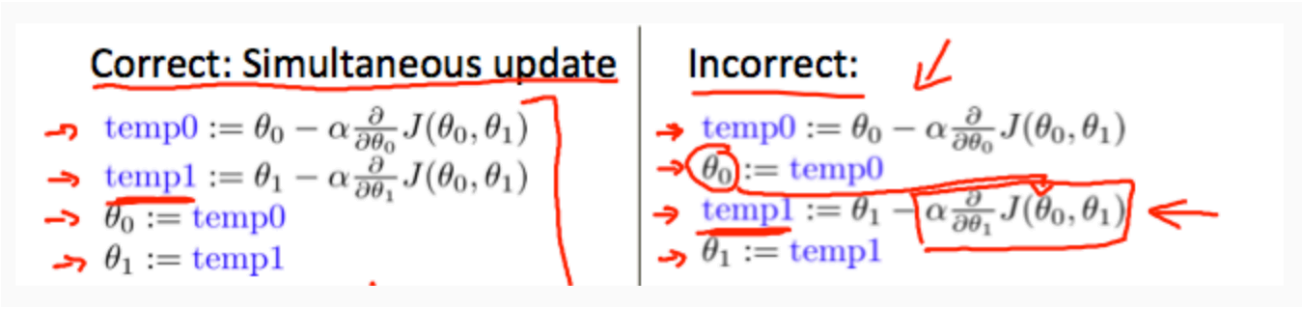
For example, the distance between each 'star' in the graph above represents a step determined by our parameter α. A smaller α would result in a smaller step and a larger α results in a larger step. The direction in which the step is taken is determined by the partial derivative of $J(\theta_0, \theta_1)$. 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:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta j} J(\theta_0, \theta_1)$$

where
j=0,1 represents the feature index number.
At each iteration j, one should simultaneously update the parameters $\theta_1, \theta_2, ..., \theta_n$. Updating a specific parameter prior to calculating another one on the $j^{(th)}$ iteration would yield to a wrong implementation.



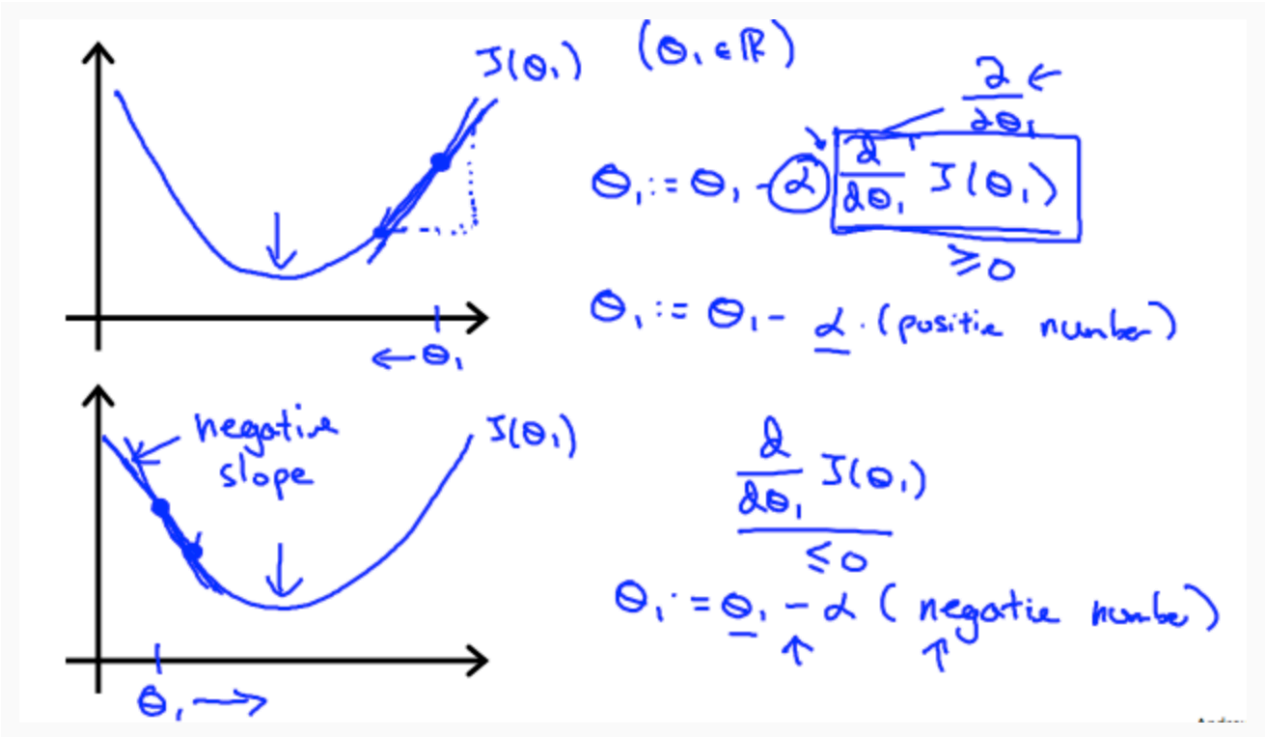
Gradient Descent Intuition

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

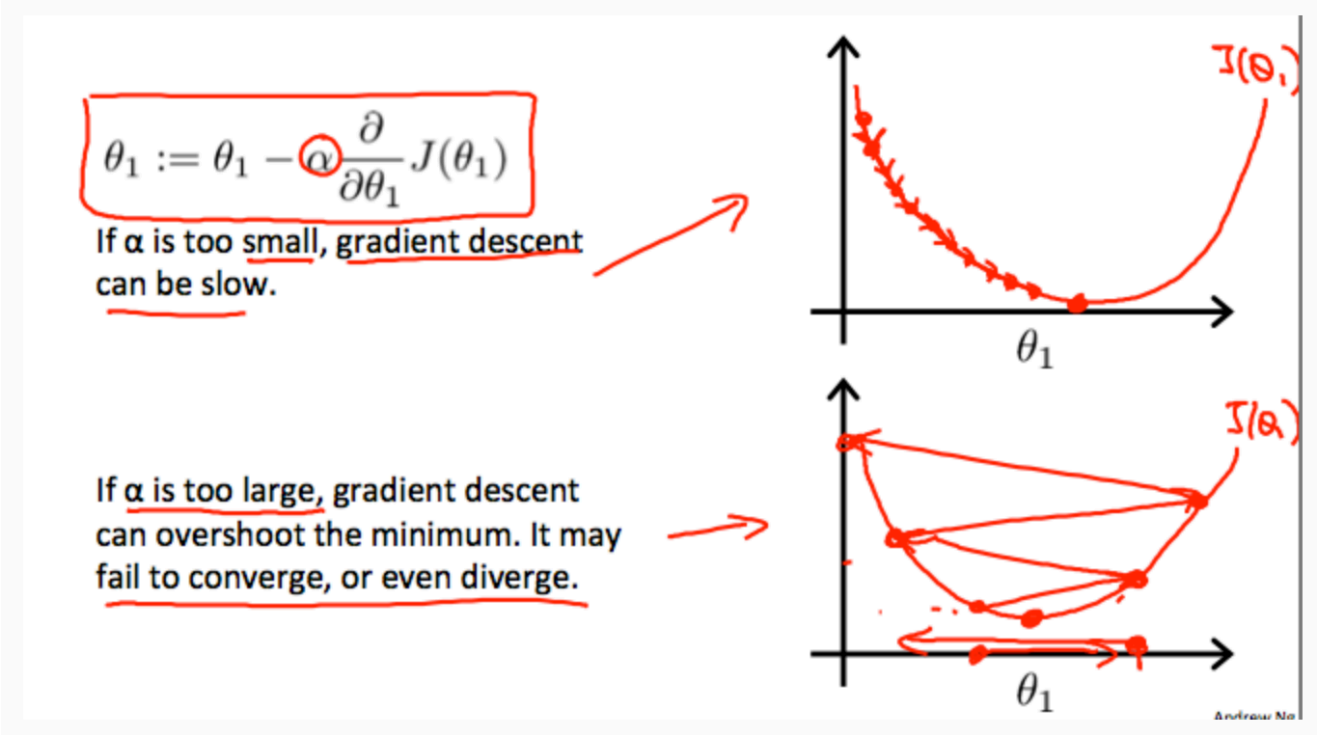
Repeat until convergence:

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

Regardless of the slope's sign for \frac{d}{dθ_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 θ1 decreases.



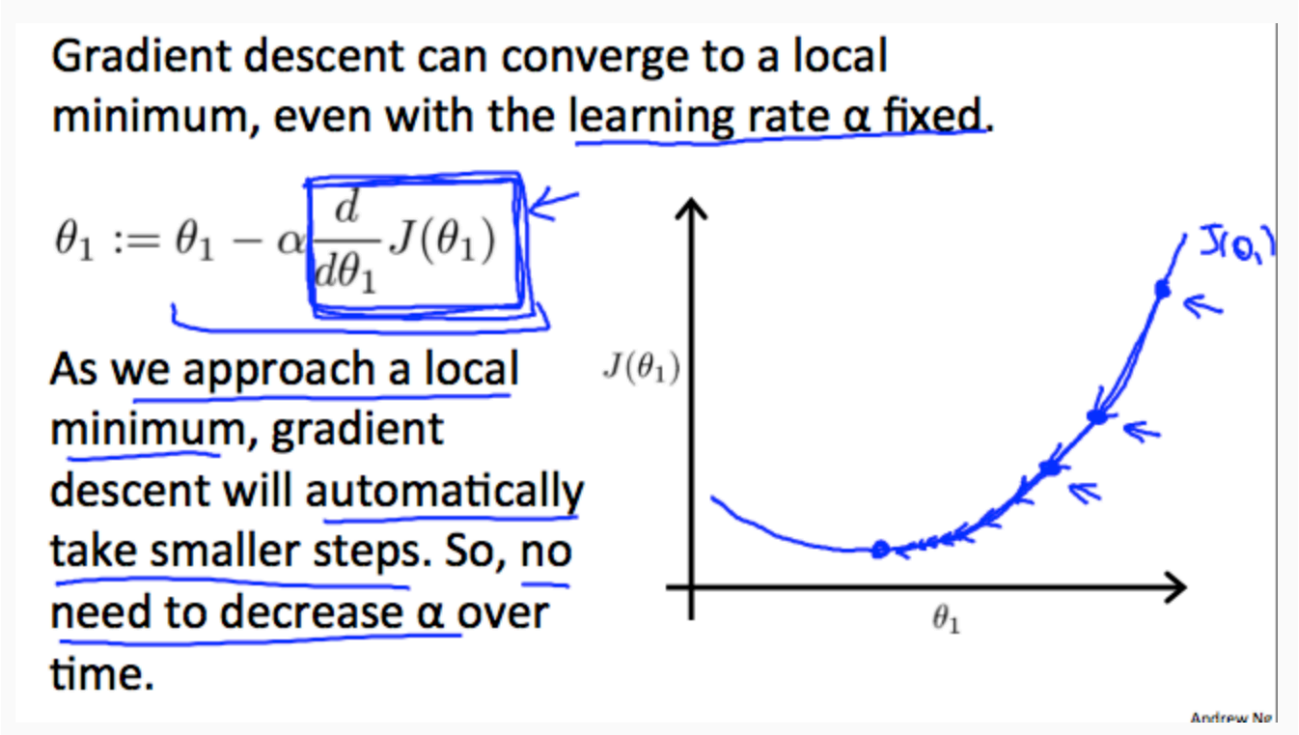
On a side note, we should adjust our parameter α to ensure that the gradient descent algorithm converges in a reasonable time. Failure to converge or too much time to obtain the minimum value imply that our step size is wrong.



How does gradient descent converge with a fixed step size α?

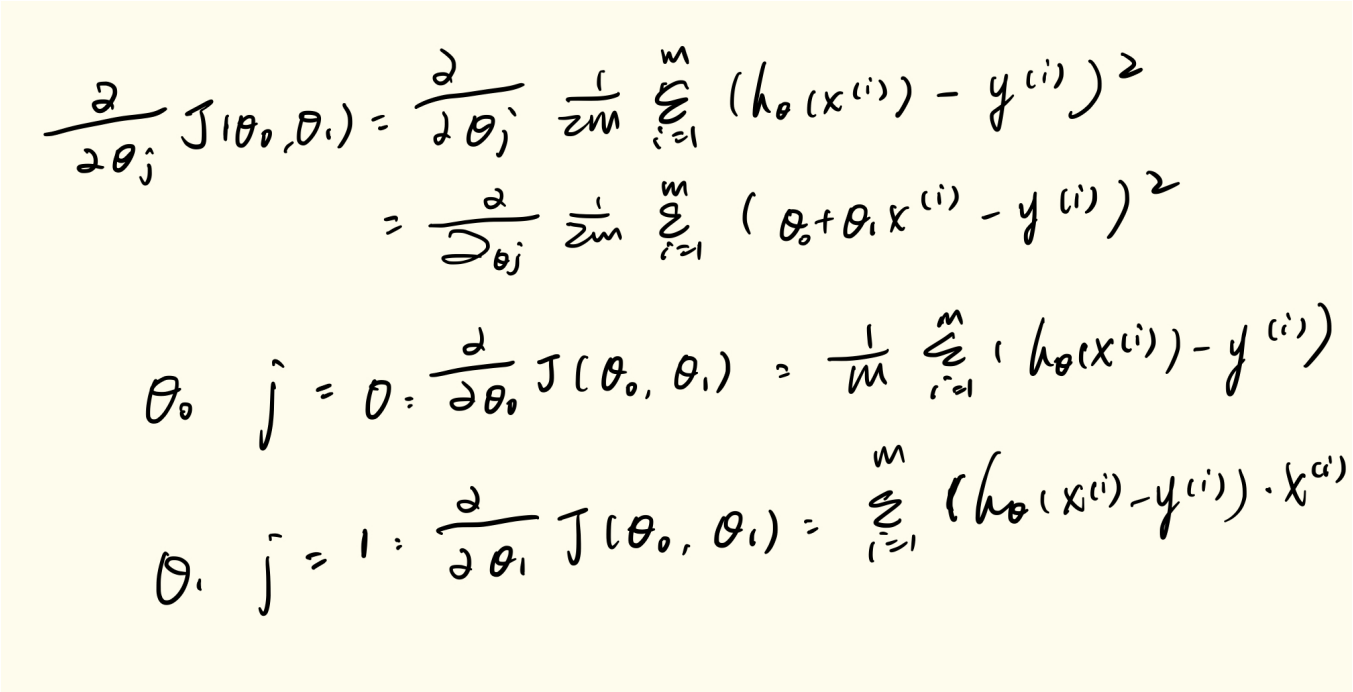
The intuition behind the convergence is that $\frac{d}{d\theta_1} J(\theta_1)$ approaches 0 as we approach the bottom of our convex function. At the minimum, the derivative will always be 0 and thus we get:

$$\theta_1 := \theta_1 - \alpha * 0$$



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 :



repeat until convergence:{

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} ((h_\theta(x_i) - y_i) x_i)$$

}

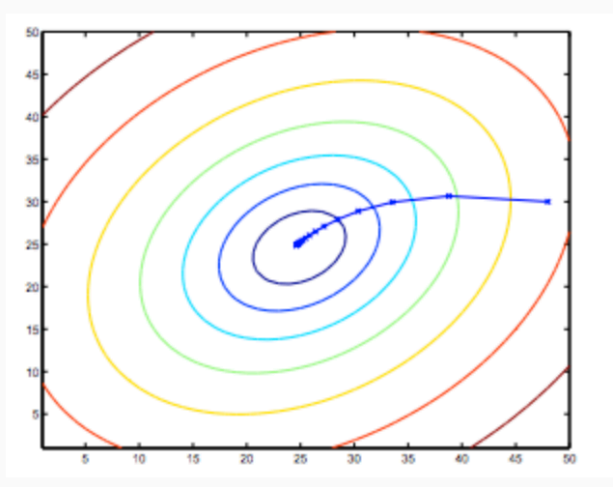
where m is the size of the training set, θ_0 a constant that will be changing simultaneously with θ1 and xi,yiare values of the given training set (data).

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 :

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

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.

Matrices and Vectors

Matrices are 2-dimensional arrays:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{pmatrix}$$

The above matrix has four rows and three columns, so it is a 4 x 3 matrix.

A vector is a matrix with one column and many rows:

$$\begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix}$$

So vectors are a subset of matrices. The above vector is a 4 x 1 matrix.

Notation and terms:

- A_{ij} refers to the element in the ith row and jth column of matrix A.
- A vector with 'n' rows is referred to as an 'n'-dimensional vector.
- v_i refers to the element in the ith row of the vector.
  In general, all our vectors and matrices will be 1-indexed. Note that for some programming languages, the arrays are 0-indexed.
- Matrices are usually denoted by uppercase names while vectors are lowercase.
- "Scalar" means that an object is a single value, not a vector or matrix.
- $\mathbb{R}$ refers to the set of scalar real numbers.
- $\mathbb{R}^n$ refers to the set of n-dimensional vectors of real numbers.

```
% The ; denotes we are going back to a new row.
A = [1, 2, 3; 4, 5, 6; 7, 8, 9; 10, 11, 12]

% Initialize a vector
v = [1;2;3]

% Get the dimension of the matrix A where m = rows and n = columns
[m,n] = size(A)

% You could also store it this way
dim_A = size(A)

% Get the dimension of the vector v
dim_v = size(v)

% Now let's index into the 2nd row 3rd column of matrix A
A_23 = A(2,3)
```

Addition and Scalar Multiplication

Addition and subtraction are element-wise, so you simply add or subtract each corresponding element:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a+w & b+x \\ c+y & d+z \end{bmatrix}$$

Subtracting Matrices:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} - \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a-w & b-x \\ c-y & d-z \end{bmatrix}$$

To add or subtract two matrices, their dimensions must be the same.

In scalar multiplication, we simply multiply every element by the scalar value:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} * x = \begin{bmatrix} a*x & b*x \\ c*x & d*x \end{bmatrix}$$

In scalar division, we simply divide every element by the scalar value:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} /x = \begin{bmatrix} a/x & b/x \\ c/x & d/x \end{bmatrix}$$

Experiment below with the Octave/Matlab commands for matrix addition and scalar multiplication. Feel free to try out different commands. Try to write out your answers for each command before running the cell below.

```
% Initialize matrix A and B
A = [1, 2, 4; 5, 3, 2]
B = [1, 3, 4; 1, 1, 1]

% Initialize constant s
s = 2

% See how element-wise addition works
add_AB = A + B

% See how element-wise subtraction works
sub_AB = A - B

% See how scalar multiplication works
mult_As = A * s

% Divide A by s
div_As = A / s

% What happens if we have a Matrix + scalar?
add_As = A + s
```

Matrix-Vector Multiplication

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a*x & b*y \\ c*x & d*y \\ e*x & f*y \end{bmatrix}$$

Below is an example of a matrix-vector multiplication. Make sure you understand how the multiplication works. Feel free to try different matrix-vector multiplications.

```
% Initialize matrix A
A = [1, 2, 3; 4, 5, 6;7, 8, 9]

% Initialize vector v
v = [1; 1; 1]

% Multiply A * v
```

```
Av = A * v
```

Matrix-Matrix Multiplication

We multiply two matrices by breaking it into several vector multiplications and concatenating the result.

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} * \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} a*w+b*y & a*x+b*z \\ c*w+d*y & c*x+d*z \\ e*w+f*y & e*x+f*z \end{bmatrix}$$

An m x n matrix multiplied by an n x o matrix results in an m x o matrix. In the above example, a 3 x 2 matrix times a 2 x 2 matrix resulted in a 3 x 2 matrix.

To multiply two matrices, the number of columns of the first matrix must equal the number of rows of the second matrix.

For example:

```
% Initialize a 3 by 2 matrix
A = [1, 2; 3, 4;5, 6]

% Initialize a 2 by 1 matrix
B = [1; 2]

% We expect a resulting matrix of (3 by 2)*(2 by 1) = (3 by 1)
mult_AB = A*B

% Make sure you understand why we got that result
```

Matrix Multiplication Properties

- Matrices are not commutative: $A*B \neq B*A$
- Matrices are associative: $(A*B)*C=A*(B*C)$

The identity matrix, when multiplied by any matrix of the same dimensions, results in the original matrix. It's just like multiplying numbers by 1. The identity matrix simply has 1's on the diagonal (upper left to lower right diagonal) and 0's elsewhere.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When multiplying the identity matrix after some matrix ($A*I$), the square identity matrix's dimension should match the other matrix's columns. When multiplying the identity matrix before some other matrix ($I*A$), the square identity matrix's dimension should match the other matrix's rows.

```
% Initialize random matrices A and B
A = [1,2;4,5]
B = [1,1;0,2]

% Initialize a 2 by 2 identity matrix
I = eye(2)

% The above notation is the same as I = [1,0;0,1]

% What happens when we multiply I*A ?
IA = I*A
```

```
12
13 % How about A*I ?
14 AI = A*I
15
16 % Compute A*B
17 AB = A*B
18
19 % Is it equal to B*A?
20 BA = B*A
21
22 % Note that IA = AI but AB != BA
```

Inverse and Transpose

$$A = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

In other words:

$$A_{ij} = A^T_{ji}$$

```
1 % Initialize matrix A
2 A = [1,2,0;0,5,6;7,0,9]
3
4 % Transpose A
5 A_trans = A'
6
7 % Take the inverse of A
8 A_inv = inv(A)
9
10 % What is A^(-1)*A?
11 A_invA = inv(A)*A
```

multivariate Linear Analytically

Multiple Features

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.

- $x^{(i)}_j$ = 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 multivariable form of the hypothesis function accommodating these multiple features is as follows:

$$h_\theta(x) = \theta_0 + \theta_1 x1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$$

In order to develop intuition about this function, we can think about θ0 as the basic price of a house, θ1 as the price per square meter, θ2 as the price per floor, etc. x1 will be the number of square meters in the house, x2 the number of floors, etc.

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

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

This is a vectorization of our hypothesis function for one training example; see the lessons on vectorization to learn more.

Remark: Note that for convenience reasons in this course we assume $x_0^{(i)} = 1$ for (i∈1,...,m). This allows us to do matrix operations with theta 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 it for our 'n' features:

repeat until convergence:{

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)}$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)}$$

$$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_2^{(i)}$$

$$\cdots$$

}

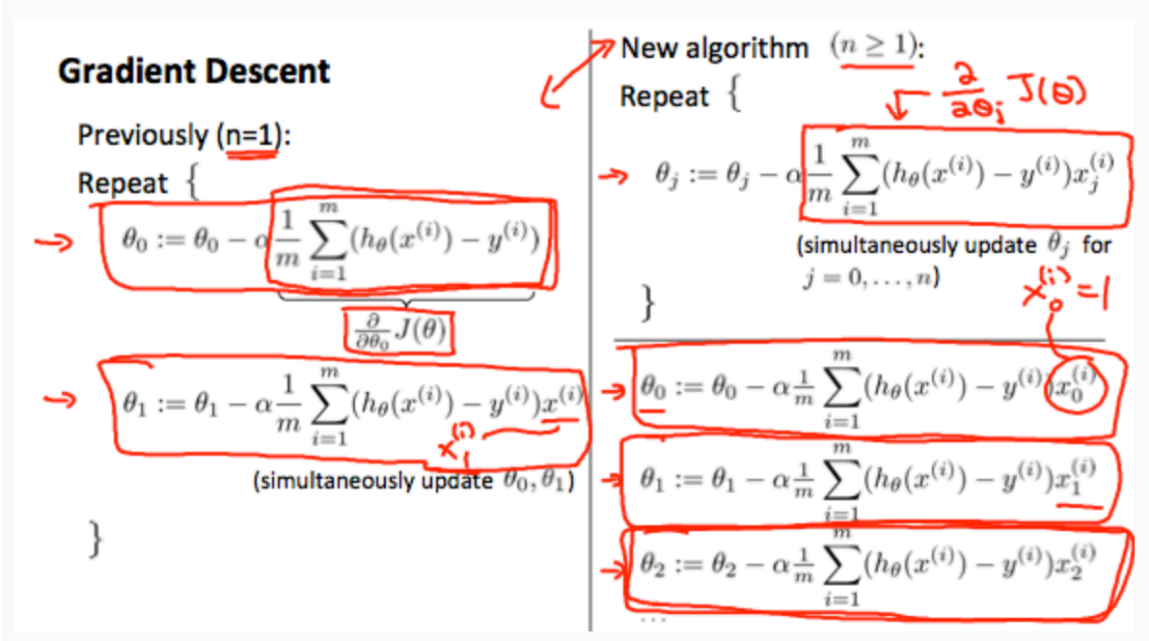
In other words:

repeat until convergence: {

$$\theta_j := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad for\ j := 0 \cdots n$$

}

The following image compares gradient descent with one variable to gradient descent with multiple variables:



Gradient Descent in Practice I - 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 our input variables so that they 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:

$$x_i := \frac{x_i - \mu_i}{si}$$

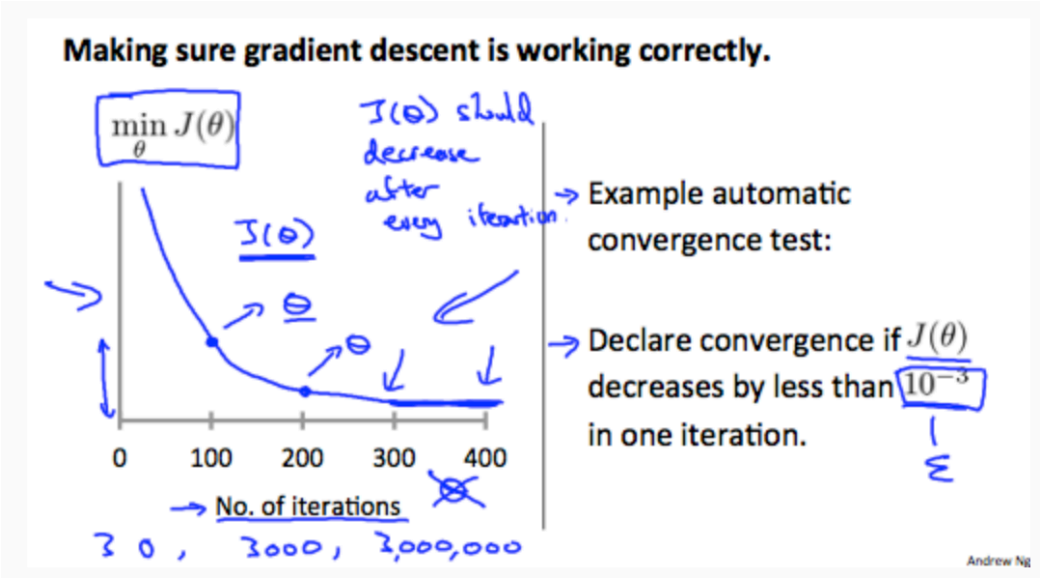
Where μi is the average of all the values for feature (i) and si is the range of values (max - min), or si is the standard deviation.

For example, if xi represents housing prices with a range of 100 to 2000 and a mean value of 1000, then, $x_i := \frac{price - 1000}{1900}$.

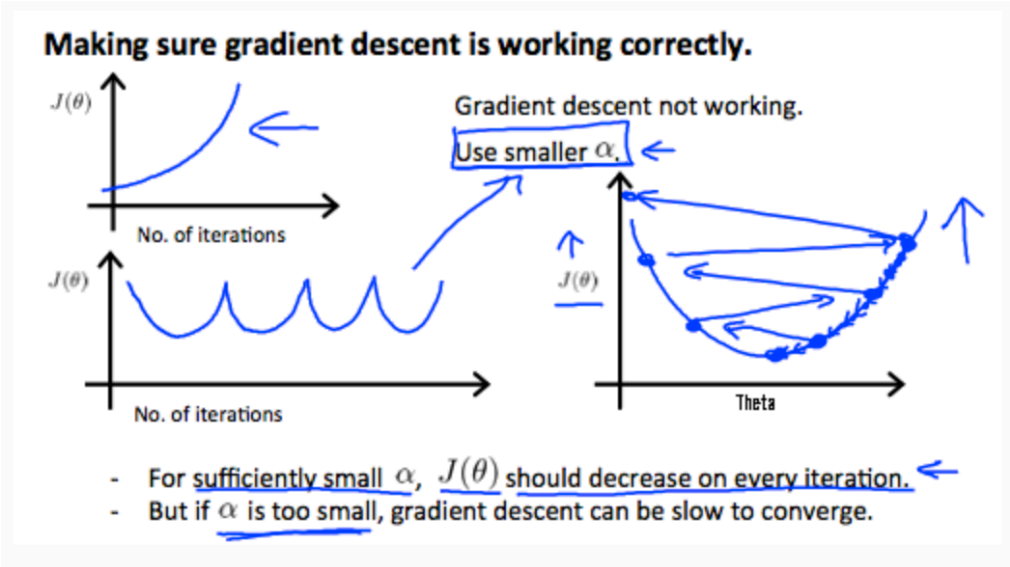
Gradient Descent in Practice II - Learning Rate

Debugging gradient descent. Make a plot with number of iterations on the x-axis. Now plot the cost function, $J_{(\theta)}$ over 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 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 $J_{(\theta)}$ will decrease on every iteration.



To summarize:

- If α is too small: slow convergence.
- If α is too large: may not decrease on every iteration and thus 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 x3 by

taking $x_1 \cdot x_2$.

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 $h_\theta(x) = \theta_0 + \theta_1 x_1$ then we can create additional features based on x1, to get the quadratic function $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2$ or the cubic function $h\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$
In the cubic version, we have created new features x_2 and x_3 where $x_2 = x_1^2$ and $x_3 = x_1^3$.

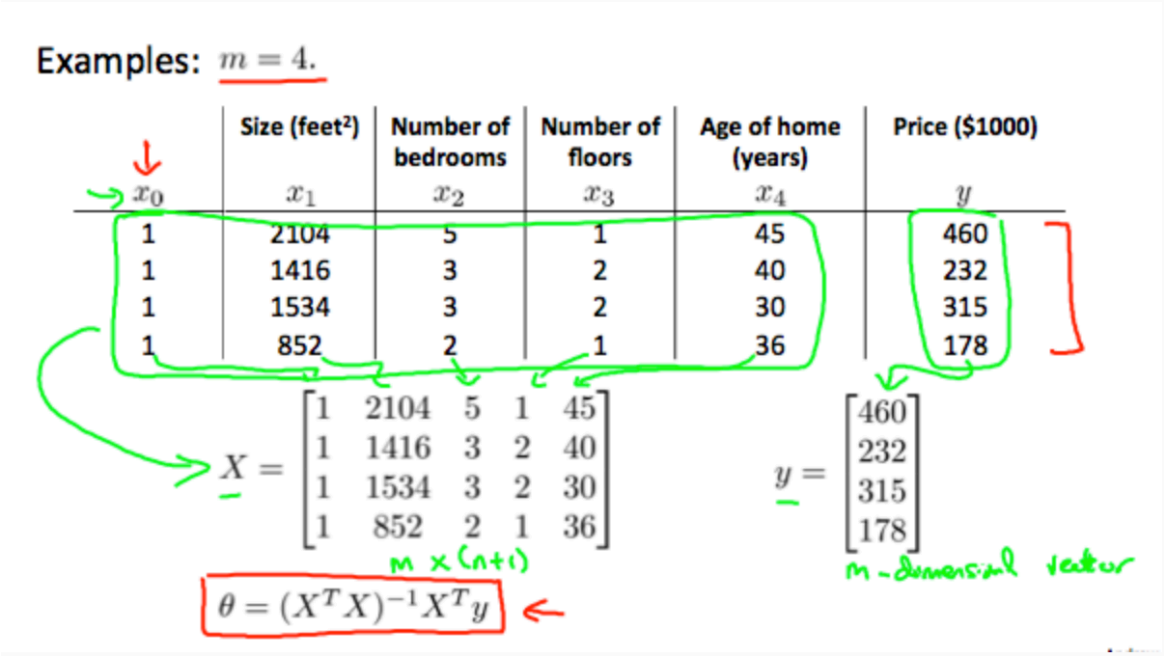
To make it a square root function, we could do: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 \sqrt{x_1}$
One important thing to keep in mind is, if you choose your features this way then feature scaling becomes very important.

eg. if x_1 has range 1 - 1000 then range of x_1^2 becomes 1 - 1000000 and that of x_1^3 becomes 1 - 1000000000

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:

$$\theta = (X^T X)^{-1} X^T y$$



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 alpha	No need to choose alpha

Needs many iterations	No need to iterate
$O(kn^2)$	$O(n^3)$,need to calculate inverse of X^TX
Works well when n is large	Slow if n is very large

With the normal equation, computing the inversion has complexity $O(n^3)$. 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 to use the 'pinv' function rather than 'inv.' The 'pinv' function will give you a value of θ even if X^TX is not invertible.

If X^TX 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.

Logistic Regression

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 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.

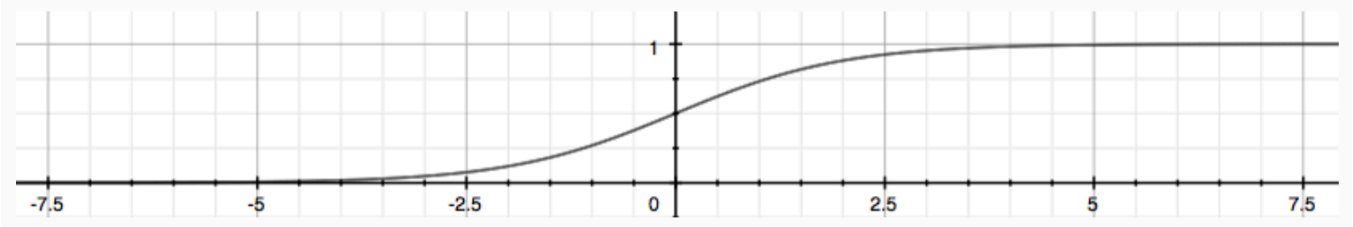
Hypothesis Representation

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

Our new form 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:



The function g(z), shown here, maps any real number to the (0, 1) interval, making it useful for transforming an arbitrary-valued function into a function better suited for classification.

hθ(x) will give us the probability that our output is 1. For example, $h_\theta(x) = 0.7$ gives us a probability of 70% that our output is 1. Our probability that our prediction 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%).

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

Decision Boundary

In order to get our 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 zero, its output is greater than or equal to 0.5:

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

Remember.

$$z = 0, e^0 = 1 \Rightarrow g(z) = \frac{1}{2}$$
$$z \rightarrow \infty, e^{-\infty} \rightarrow 0 \Rightarrow g(z) = 1$$
$$z \rightarrow -\infty, e^{\infty} \rightarrow \infty \Rightarrow g(z) = 0$$

Again, the input to the sigmoid function g(z) (e.g. θTX) **doesn't need to be linear**, and could be a function that describes a circle (e.g. $z = \theta_0 + \theta_1 x_2^1 + \theta_2 x_2^2$) or any shape to fit our data.

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.

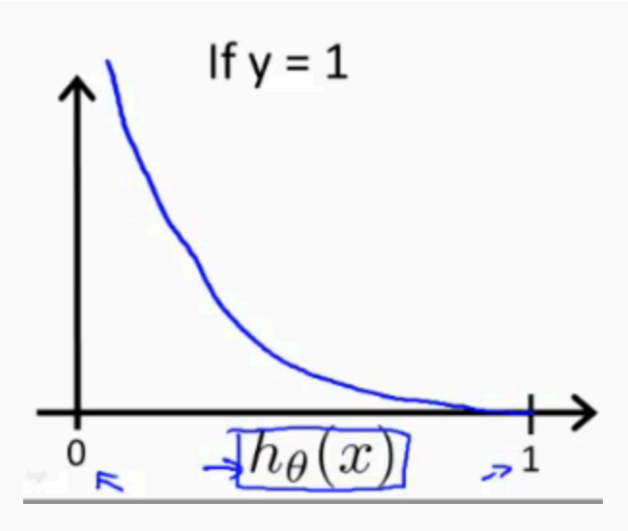
Instead, our cost function for logistic regression looks like:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)})$$

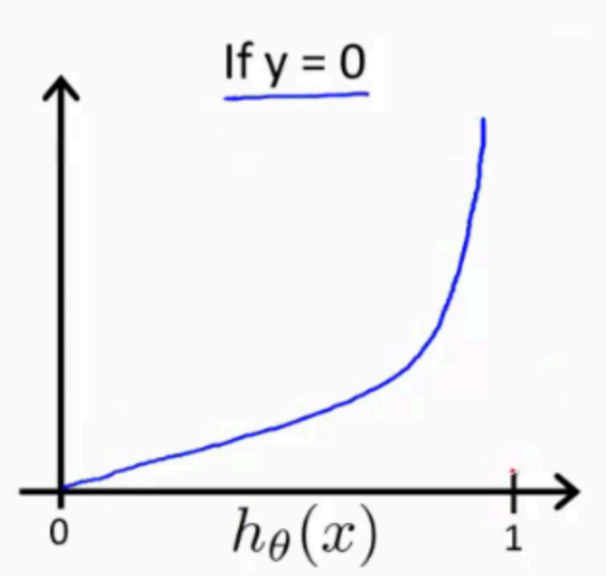
$$Cost(h\theta(x), y) = -log(h_\theta(x)) \qquad if\ y = 1$$

$$Cost(h\theta(x), y) = -log(1 - h_\theta(x)) \qquad if\ y = 0$$

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

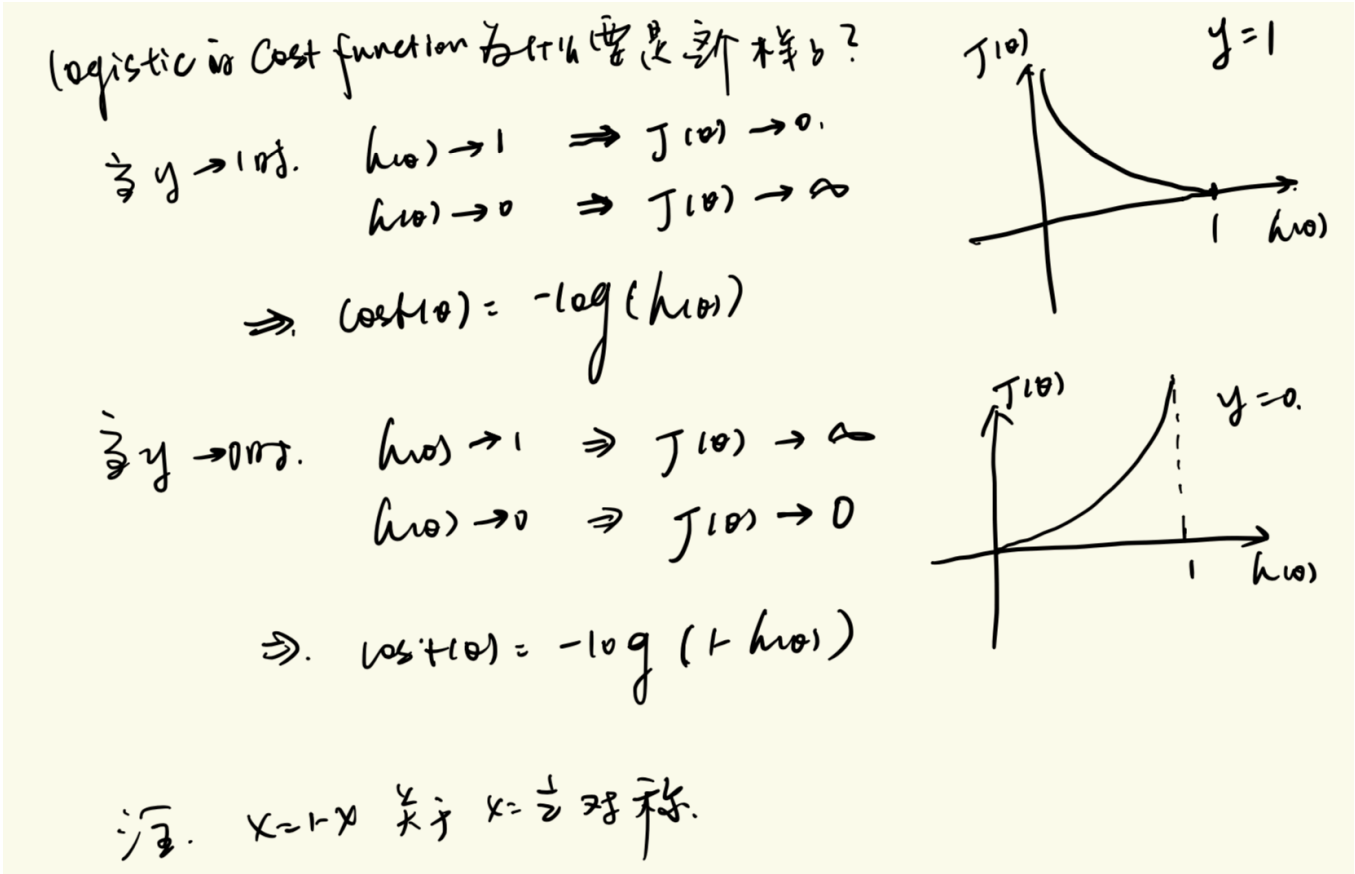


Similarly, when y = 0, we get the following plot for $J(\theta)$ vs $h_\theta(x)$:



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(θ) is convex for logistic regression.

Simplified Cost Function and Gradient Descent

We can compress our cost function's two conditional cases into one case:

$$Cost(h_\theta(x), y) = -ylog(h_\theta(x)) - (1-y)log(1-h_\theta(x))$$

Notice that when y is equal to 1, then the second term $(1-y)log(1-h_\theta(x))$ will be zero and will not affect the result. If y is equal to 0, then the first term $-ylog(h_\theta(x))$ will be zero and will not affect the result.

We can fully write out our entire cost function as follows:

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

Gradient Descent

Remember that the general form of gradient descent is:

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

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

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

Multiclass Classification: One-vs-all

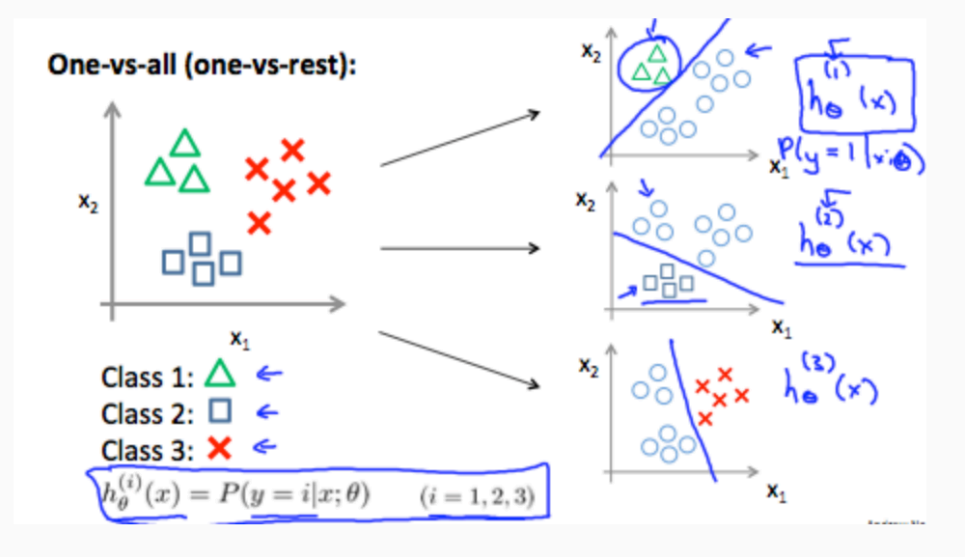
Now we will approach the classification of data when we have more than two categories. Instead of y = {0,1} we will expand our definition so that y = {0,1...n}.

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.

$$
\begin{aligned}
&y \in \{0, 1...n\} \\
&h_\theta^{(0)}(x) = P(y = 0|x; \theta) \\
&h_\theta^{(1)}(x) = P(y = 1|x; \theta) \\
&\cdots \\
&h_\theta^{(n)}(x) = P(y = n|x; \theta) \\
&prediction = \max_i(h_\theta^{(i)}(x))
\end{aligned}
$$

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 following image shows how one could classify 3 classes:

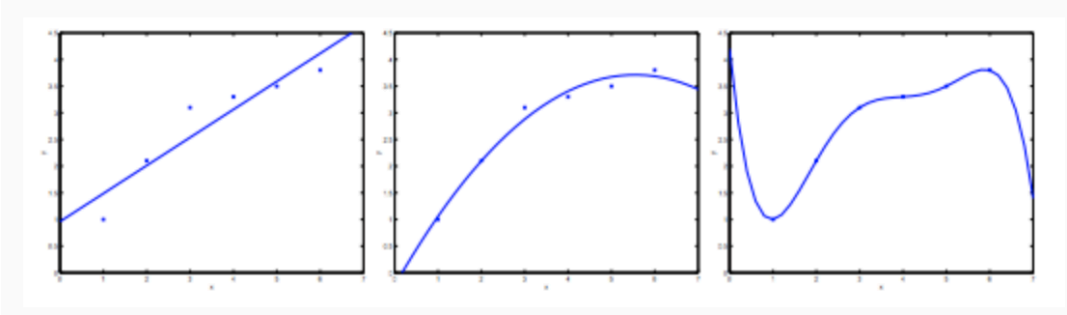


To summarize:
Train a logistic regression classifier $h_\theta(x)$ for each class to predict the probability that y = i, To make a prediction on a new x, pick the class that maximizes $h_\theta(x)$.

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 $y = \theta_0 + \theta_1 x$ to a dataset. We see that the data doesn't really lie on straight line, and so the fit is not very good.



Instead, if we had added an extra feature x_2 , and fit $y = \theta_0 + \theta_1 x + \theta_2 x_2$, then 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 5th order polynomial $y = \sum_{j=0}^{5} \theta_j x_j$. We see that even though the fitted 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 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.
- **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**:

- Reduce the number of features:
  - Manually select which features to keep.
  - Use a model selection algorithm (studied later in the course).
- Regularization
  - Keep all the features, but reduce the magnitude of 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:

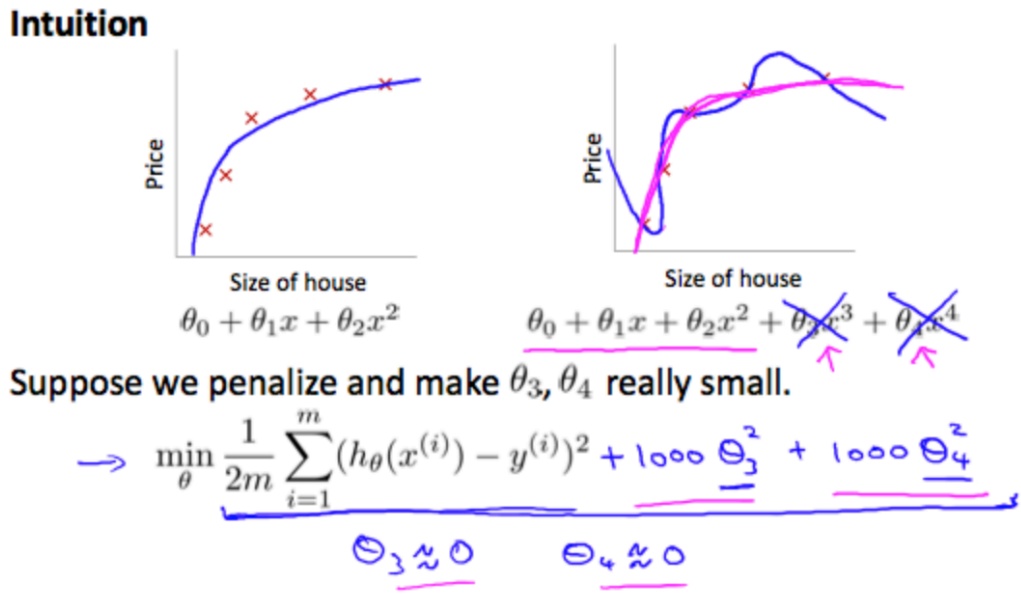
$$\theta_0 + \theta_1 x + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4$$

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**:

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

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 close to zero, we will have to reduce the values of θ_3 and θ_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

pink 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 theta parameters in a single summation as:

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

The λ, or lambda, is the **regularization parameter**. It determines how much the costs of our theta 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 lambda is chosen to be too large, it may smooth out the function too much and cause underfitting. Hence, what would happen if λ=0 or is too small ?

Regularized Linear Regression

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

repeat until convergence:{

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$$

$$\theta_j := \theta_j - \alpha \left[\left(\frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \right] \qquad j \in \{1, 2...n\}$$

}

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

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

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 is now 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 our original, except that we add another term inside the parentheses:

$$\theta = \left(X^TX + \lambda \cdot L\right)^{(-1)}X^Ty$$

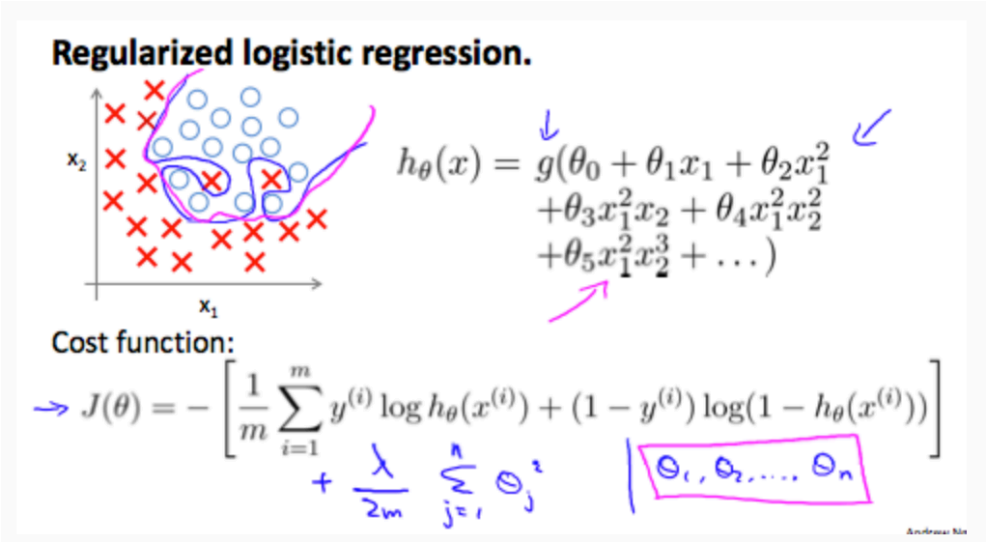
$$where\ L = \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}$$

L is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else. It should have dimension (n+1)×(n+1). Intuitively, this is the identity matrix (though we are not including x_0), multiplied with a single real number \lambda.

Recall that if m < n, then X^TX is non-invertible. However, when we add the term $\lambda \cdot l$, then $X^TX + \lambda \cdot L$ becomes 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:



Cost Function

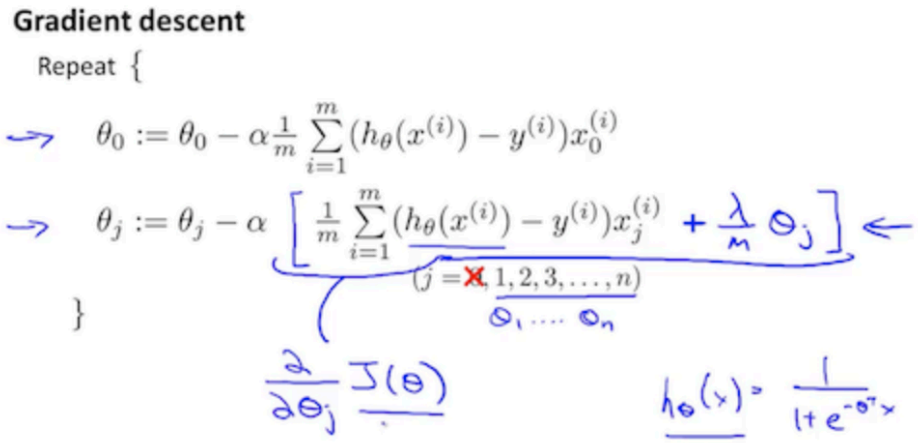
Recall that our cost function for logistic regression was:

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

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

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

he 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 two following equations:



Neural Networks

Model Representation

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 $x_1 \ldots x_n$, and the output is the result of our 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 use the same logistic function as in classification, $\frac{1}{1+e^{-\theta^T x}}$, yet we sometimes call it a sigmoid (logistic) **activation** function. In this situation, our "theta" parameters are sometimes called "**weights**".

Visually, a simplistic representation looks like:

$$\begin{bmatrix}x_0\\x_1\\x_2\\x_3\end{bmatrix} \rightarrow \begin{bmatrix}a_1^{(2)}\\a_2^{(2)}\\a_3^{(2)}\end{bmatrix} \rightarrow h_\theta(x)$$

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

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

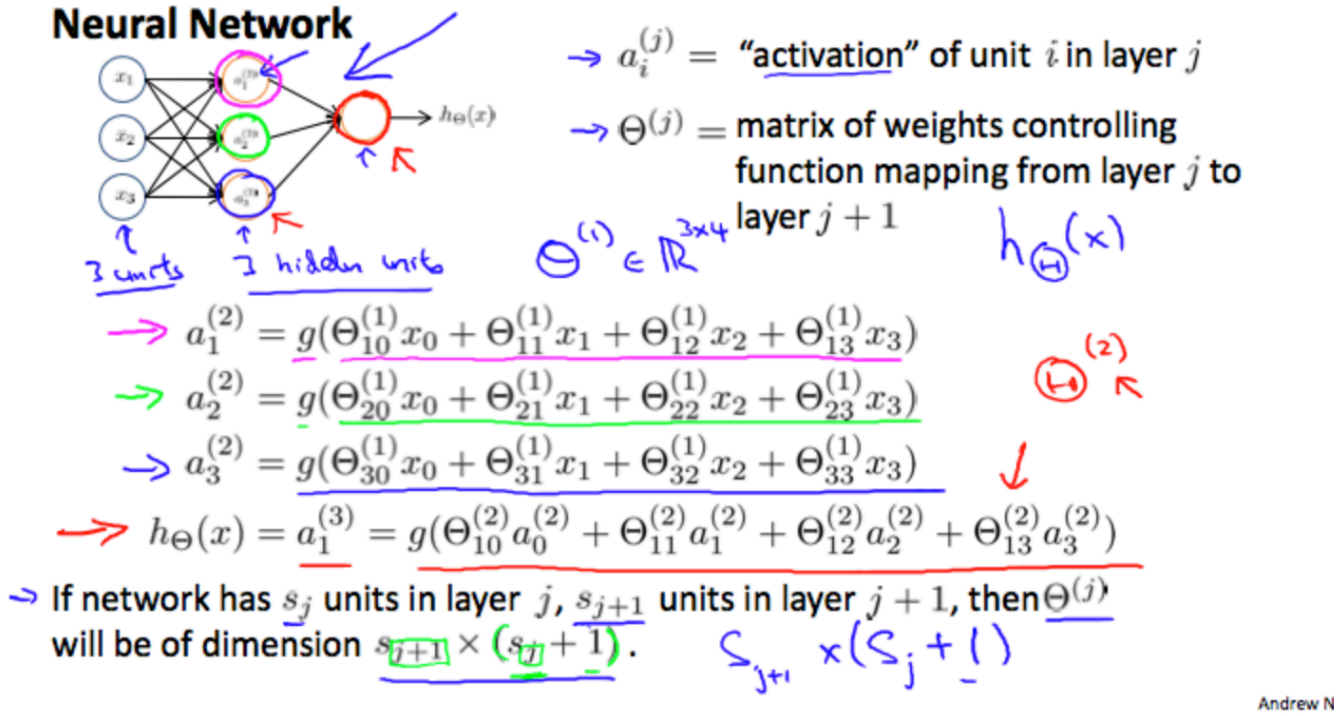
This is saying that we compute our activation nodes by using a 3×4 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been 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 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. Dimension of $\Theta^{(1)}$ is going to be 4×3 where $s_j = 2$ and $s_{j+1} = 4$, so $s_{j+1} \times (s_j + 1) = 4 \times 3$.

Examples and Intuitions I

A simple example of applying neural networks is by predicting x1 AND x2, which is the logical 'and' operator and is only true if both x1 and x2 are 1.

The graph of our functions will look like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \rightarrow \left[g(z^{(2)}) \right] \rightarrow h_\theta(x)$$

Remember that x0 is our bias variable and is always 1.

Let's set our first theta matrix as:

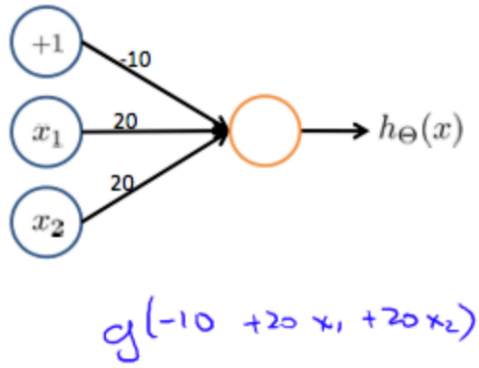
$$\theta^{(1)} = \begin{bmatrix} -30 & 20 & 20 \end{bmatrix}$$

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

$$h\Theta(x) = g(-30 + 20x1 + 20x2)$$
$$x1 = 0 \textit{ and } x2 = 0 \textit{ then } g(-30) \approx 0$$
$$x1 = 0 \textit{ and } x2 = 1 \textit{ then } g(-10) \approx 0$$
$$x1 = 1 \textit{ and } x2 = 0 \textit{ then } g(-10) \approx 0$$
$$x1 = 1 \textit{ and } x2 = 1 \textit{ then } g(10) \approx 1$$

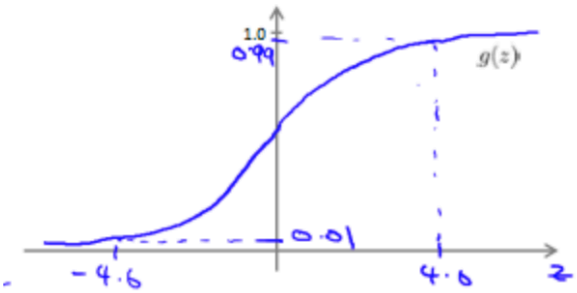
So we have constructed one of the fundamental operations in computers by using a small neural network rather than using an actual AND gate. Neural networks can also be used to simulate all the other logical gates. The following is an example of the logical operator 'OR', meaning either x_1 is true or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

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:

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

We can combine these to get the XNOR logical operator (which gives 1 if x1 and x2 are both 0 or both 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:

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

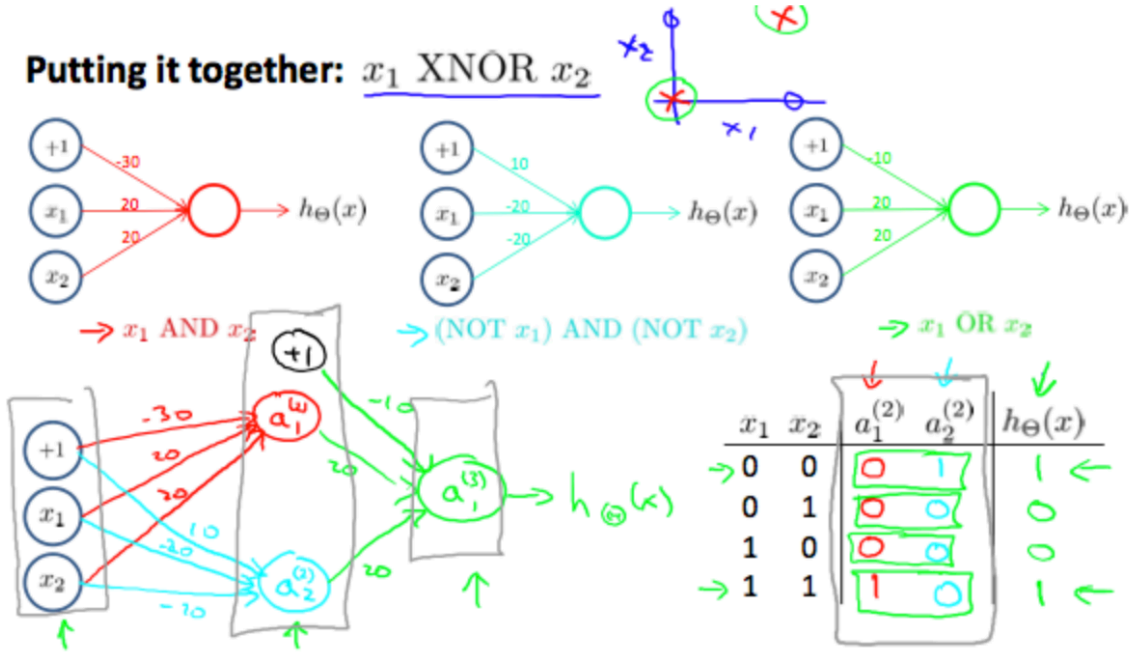
For the transition between the second and third layer, we'll use a $\Theta^(2)$ matrix that uses the value for OR:

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

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

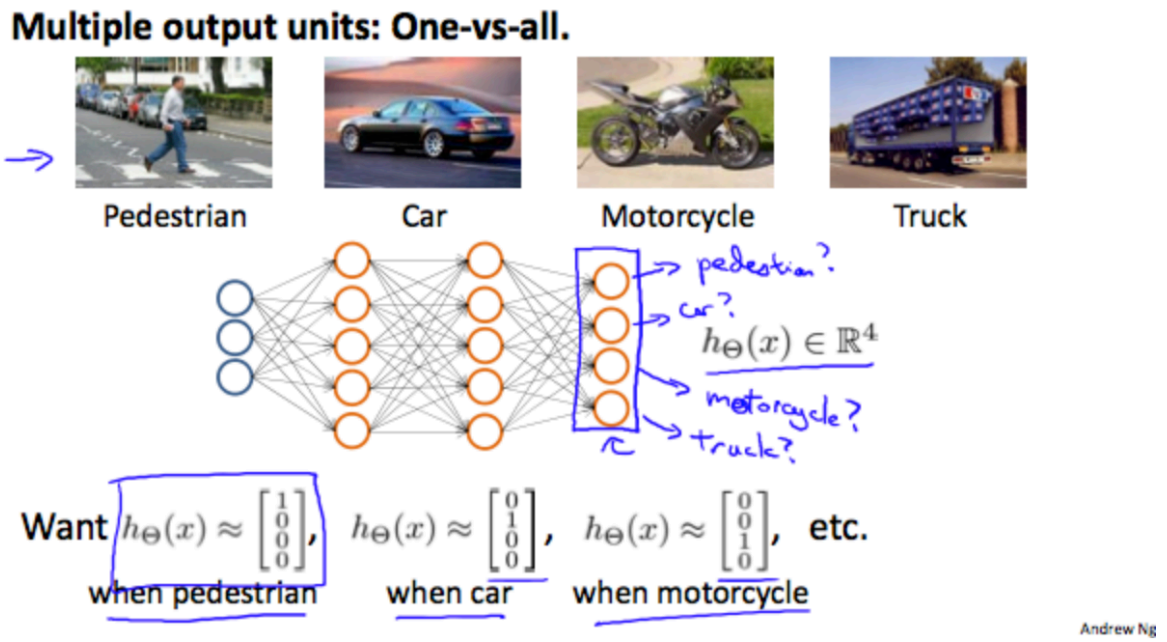
$$
\begin{aligned}
a^{(2)} &= g(\Theta^{(1)} \cdot x) \\
a^{(3)} &= g(\Theta(2) \cdot a^{(2)}) \\
h_\Theta^{(x)} &= a^{(3)}
\end{aligned}
$$

And there we have the XNOR operator 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 final hypothesis function. The setup looks like:

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

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.

Cost Function

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

- L = total number of layers in the network
- s_l = number of units (not counting bias unit) in layer l

- K = number of output units/classes

Recall that in neural networks, we may have many output nodes. We denote $h_\Theta(x)_k$ as being a hypothesis that results in the kth output. Our cost function for neural networks is going to be a generalization of the one we used for logistic regression. Recall that the cost function for regularized logistic regression was:

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

For neural networks, it is going to be slightly more complicated:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{k}\left[y^{(i)}log(h_\theta(x^{(i)})) + (1-y^{(i)})log(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{L-1}\sum_{i=1}^{s_j}\sum_{j=1}^{s_j+1}\left(\Theta_{j,i}^{(l)}\right)$$

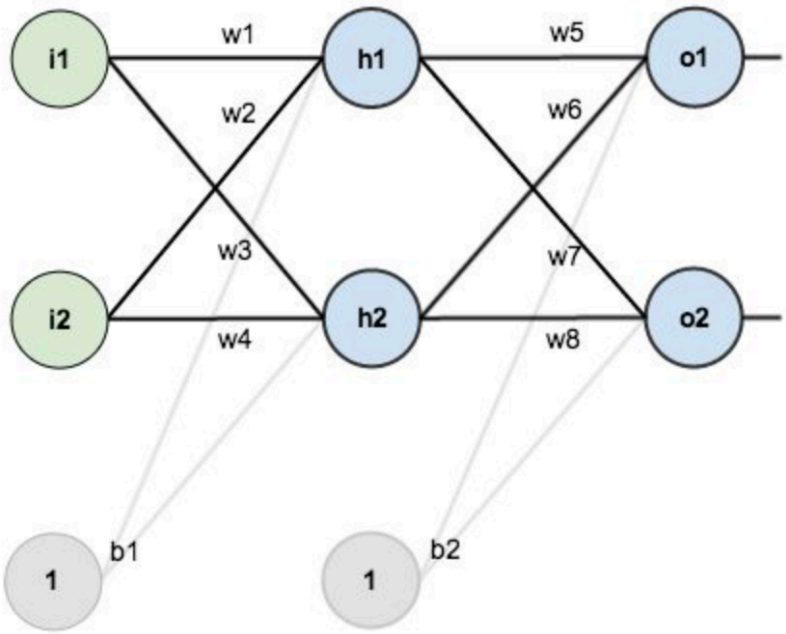
We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, before the square brackets, we have an additional nested summation that loops through the number of output nodes.

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

反向传播算法



- **前向传播**

> 对于节点 h_1 来说， h_1 的净输入 net_{h_1} 如下：
> $$net_{h_1} = w_1 \times i_1 + w_2 \times i_2 + b_1 \times 1$$
> 接着对 net_{h_1} 做一个sigmoid函数得到节点 h_1 的输出：
> $$out_{h_1} = \frac{1}{1 + e^{-net_{h_1}}}$$
> 类似的，我们能得到节点 h_2 、 o_1 、 o_2 的输出 out_{h_2} 、 out_{o_1} 、 out_{o_2} 。

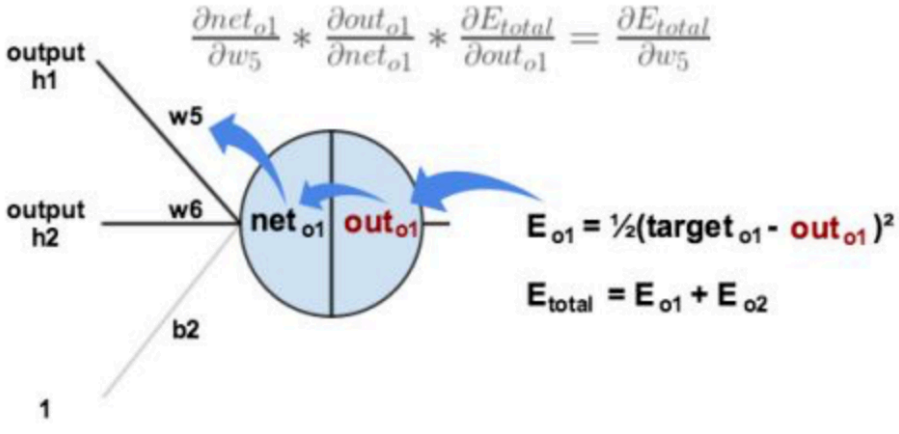
- **误差**

> 得到结果后，整个神经网络的输出误差可以表示为：
> $$E_{total} = \sum \frac{1}{2}(target - output)^2$$
> 其中 $output$ 就是刚刚通过前向传播算出来的 out_{o_1} 、 out_{o_2} ； $target$ 是节点 o_1 、 o_2 的目标值。 E_{total} 用来衡量二者的误差。
> 这个 E_{total} 也可以认为是cost function，不过这里省略了防止overfit的regularization term（ $\sum w_i^2$ ）
> 展开得到
> $$E_{total} = Eo_1 + Eo_2 = \frac{1}{2}(target_{o_1} - out_{o_1})^2 + \frac{1}{2}(target_{o_2} - out_{o_2})^2$$

- **后向传播**

对输出层的 w_5

> 通过梯度下降调整 w_5 ，需要求 $\frac{\partial E_{total}}{\partial w_5}$ ，由链式法则：
> $$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} \frac{\partial out_{o_1}}{\partial net_{o_1}} \frac{\partial net_{o_1}}{\partial w_5}$$ ，
> 如下图所示：



$$\frac{\partial E_{total}}{\partial out_{o_1}} = \frac{\partial}{\partial out_{o_1}}(\frac{1}{2}(target_{o_1} - out_{o_1})^2 + \frac{1}{2}(target_{o_2} - out_{o_2})^2) = -(target_{o_1} - out_{o_1})$$

$$\frac{\partial out_{o_1}}{\partial net_{o_1}} = \frac{\partial}{\partial net_{o_1}}\frac{1}{1+e^{-net_{o_1}}} = out_{o_1}(1 - out_{o_1})$$

$$\frac{\partial net_{o_1}}{\partial w_5} = \frac{\partial}{\partial w_5}(w_5 \times out_{h_1} + w_6 \times out_{h_2} + b_2 \times 1) = out_{h_1}$$

以上3个相乘得到梯度 $\frac{\partial E_{total}}{\partial w_5}$，之后就可以用这个梯度训练了：

$$w_5^+ = w_5 - \eta\frac{\partial E_{total}}{\partial w_5}$$

很多教材比如Stanford的课程，会把中间结果 $\frac{\partial E_{total}}{\partial net_{o_1}} = \frac{\partial E_{total}}{\partial out_{o_1}}\frac{\partial out_{o_1}}{\partial net_{o_1}}$ 记做 δ_{o_1}，表示这个节点对最终的误差需要负多少责任。。所以有 $\frac{\partial E_{total}}{\partial w_5} = \delta_{o_1} out_{h_1}$ 。

通过梯度下降调整 w_1，需要求 $\frac{\partial E_{total}}{\partial w_1}$，由链式法则：

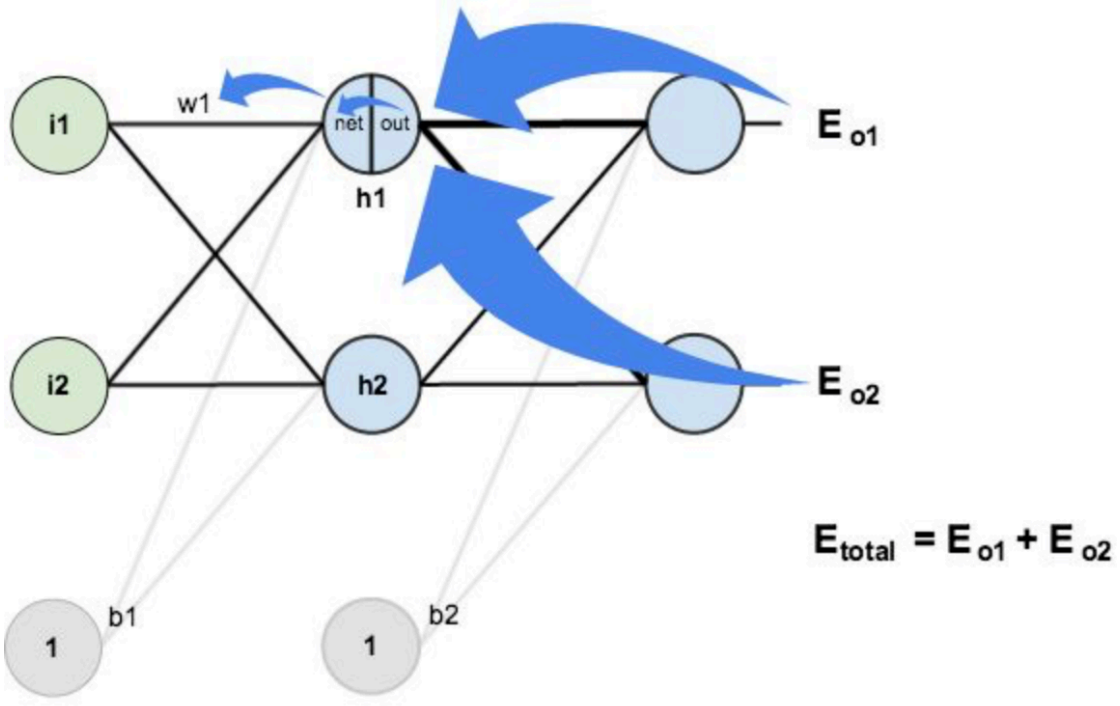
$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}}\frac{\partial out_{h_1}}{\partial net_{h_1}}\frac{\partial net_{h_1}}{\partial w_1},$$

如下图所示：

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\downarrow$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



求解每个部分：

$$\frac{\partial E_{total}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial out_{h_1}} + \frac{\partial E_{o_2}}{\partial out_{h_1}}$$ ，

其中

$$\frac{\partial E_{o_1}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial net_{o_1}} \times \frac{\partial net_{o_1}}{\partial out_{h_1}} = \delta_{o_1} \times \frac{\partial net_{o_1}}{\partial out_{h_1}} = \delta_{o_1} \times \frac{\partial}{\partial out_{h_1}}(w_5 \times out_{h_1} + w_6 \times out_{h_2} + b_2 \times 1) = \delta_{o_1} w_5$$

，这里 δ_{o_1} 之前计算过。

$\frac{\partial E_{o_2}}{\partial out_{h_1}}$ 的计算也类似，所以得到

$\frac{\partial E_{total}}{\partial out_{h_1}} = \delta_{o_1} w_5 + \delta_{o_2} w_7$ 。

$\frac{\partial E_{total}}{\partial w_1}$ 的链式中其他两项如下：

$$\frac{\partial out_{h_1}}{\partial net_{h_1}} = out_{h_1}(1 - out_{h_1})$$ ，

$$\frac{\partial net_{h_1}}{\partial w_1} = \frac{\partial}{\partial w_1}(w_1 \times i_1 + w_2 \times i_2 + b_1 \times 1) = i_1$$

相乘得到

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} \frac{\partial out_{h_1}}{\partial net_{h_1}} \frac{\partial net_{h_1}}{\partial w_1} = (\delta_{o_1} w_5 + \delta_{o_2} w_7) \times out_{h_1}(1 - out_{h_1}) \times i_1$$

得到梯度后，就可以对 w_1 迭代了：

$w_1^+ = w_1 - \eta \frac{\partial E_{total}}{\partial w_1}$ 。

在前一个式子里同样可以对 δ_{h_1} 进行定义，

$$\delta_{h_1} = \frac{\partial E_{total}}{\partial out_{h_1}} \frac{\partial out_{h_1}}{\partial net_{h_1}} = (\delta_{o_1} w_5 + \delta_{o_2} w_7) \times out_{h_1}(1 - out_{h_1}) = (\sum_o \delta_o w_{ho}) \times out_{h_1}(1 - out_{h_1})$$

，所以整个梯度可以写成 $\frac{\partial E_{total}}{\partial w_1} = \delta_{h_1} \times i_1$

引入**delta**

利用链式法则来更新权重你会发现其实这个方法简单，但过于冗长。由于更新的过程可以看做是从网络的输入层到输出层从前往后更新，每次更新的时候都需要重新计算节点的误差，因此会存在一些不必要的重复计算。其实对于已经计算完毕的节点我们完全可以直接拿来用，因此我们可以重新看待这个问题，从后往前更新。先更新后边的权重，之后再在此基础上利用更新后边的权重产生的中间值来更新较靠前的参数。这个中间变量就是下文要介绍的delta变量，一来简化公式，二来减少计算量，有点动态规划的赶脚。

接下来用事实说话，大家仔细观察一下在第四部分链式求导部分误差对于输出层的w11以及隐藏层的w11求偏导以及偏置的求偏导的过程，你会发现，三个公式存在相同的部分，同时隐藏层参数求偏导的过程会用到输出层参数求偏导的部分公式，这正是引入了中间变量delta的原因（其实红框的公式就是delta的定义）。

$$\frac{\partial e_{o1}}{\partial w_{11}^2} = (a_1^3 - y_1) \cdot sigmoid(z_1^3) \cdot (1 - sigmoid(z_1^3)) \cdot a_1^2$$

$$\frac{\partial e_{o1}}{\partial w_{11}^1} = (a_1^3 - y_1) \cdot sigmoid(z_1^3) \cdot (1 - sigmoid(z_1^3)) \cdot w_{11}^2 \cdot sigmoid(z_1^2) \cdot (1 - sigmoid(z_1^2)) \cdot a_1^1$$

$$\frac{\partial e_{o1}}{\partial b_1^3} = (a_1^3 - y_1) \cdot sigmoid(z_1^3) \cdot (1 - sigmoid(z_1^3))$$

大家看一下经典书籍《神经网络与深度学习》中对于delta的描述为在第l层第j个神经元上的误差，定义为误差对于当前带权输入求偏导，数学公式如下：

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}$$

因此输出层的误差可以表示为（上图红色框公式）：

$$\delta_1^3 = (a_1^3 - y_1) \cdot sigmoid(z_1^3) \cdot (1 - sigmoid(z_1^3)) = \nabla_a C \odot \sigma'(z_1^3)$$

隐藏层的误差可以表示为（上图蓝色框公式）：

$$\delta_1^2 = (a_1^3 - y_1) \cdot sigmoid(z_1^3) \cdot (1 - sigmoid(z_1^3)) \cdot w_{11}^2 \cdot sigmoid(z_1^2) \cdot (1 - sigmoid(z_1^2)) = ((w_{11}^2)^T \delta_1^3) \odot \sigma'(z_1^2)$$

同时对于权重更新的表示为（上图绿色框公式）：

$$\frac{\partial e_{o1}}{\partial w_{11}^2} = (a_1^3 - y_1) \cdot sigmoid(z_1^3) \cdot (1 - sigmoid(z_1^3)) \cdot a_1^2 = a_1^2 \cdot \delta_1^3$$

其实对于偏置的更新表示为（上图红色框）：

$$\frac{\partial e_{o1}}{\partial b_1^3} = (a_1^3 - y_1) \cdot sigmoid(z_1^3) \cdot (1 - sigmoid(z_1^3)) = \delta_1^3$$

上述4个公式其实就是《神经网络与深度学习》书中传说的反向传播4大公式（详细推导证明可移步此书）：

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

Advice for Applying Machine Learning

Evaluating a Hypothesis

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

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

We can move on to evaluate our new hypothesis.

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

The new procedure using these two sets is then:

Learn Θ and minimize $J_{train}(\Theta)$ using the training set
Compute the test set error $J_{test}(\Theta)$

The test set error

1. For linear regression:

$$J_{test}(\Theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_\Theta(x^{(i)}_{test}) - y^{(i)}_{test})^2$$

For classification ~ Misclassification error (aka 0/1 misclassification error):

$$err(h_\Theta(x), y) = \begin{cases} 1 & if\ h_\Theta(x) \geq 0.5\ and\ y = 0\ or\ h_\Theta(x) < 0.5\ and\ y = 1 \\ 0 & otherwise \end{cases}$$

This gives us a binary 0 or 1 error result based on a misclassification. The average test error for the test set is:

$$TestError) = \frac{1}{m_{test}} \sum_{i=1}^{m_{test}} err(h_\Theta(x_{test}^{(i)}), y_{test}^{(i)})^2$$

This gives us the proportion of the test data that was misclassified.

Model Selection and Train/Validation/Test Sets

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

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

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

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

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

1. Optimize the parameters in Θ using the training set for each polynomial degree.
2. Find the polynomial degree d with the least error using the cross validation set.
3. Estimate the generalization error using the test set with $J_{test}(\Theta^{(d)})$, (d = theta from polynomial with lower error);

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

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.

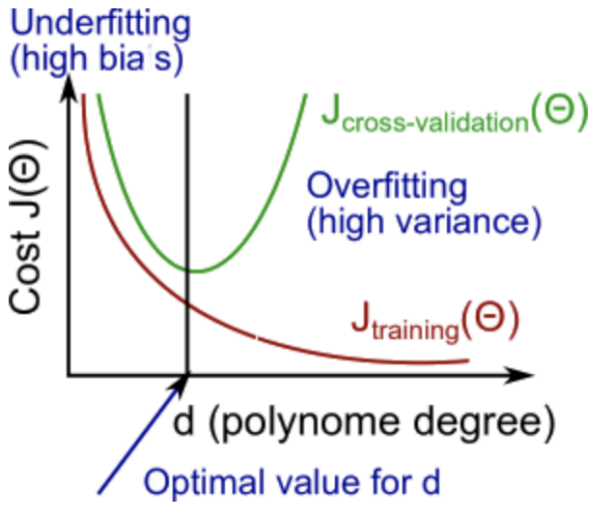
- We need to distinguish whether bias or variance is the problem contributing to bad predictions.
- **High bias** is underfitting and **high variance** is overfitting. Ideally, we need to find a golden mean between these two.

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

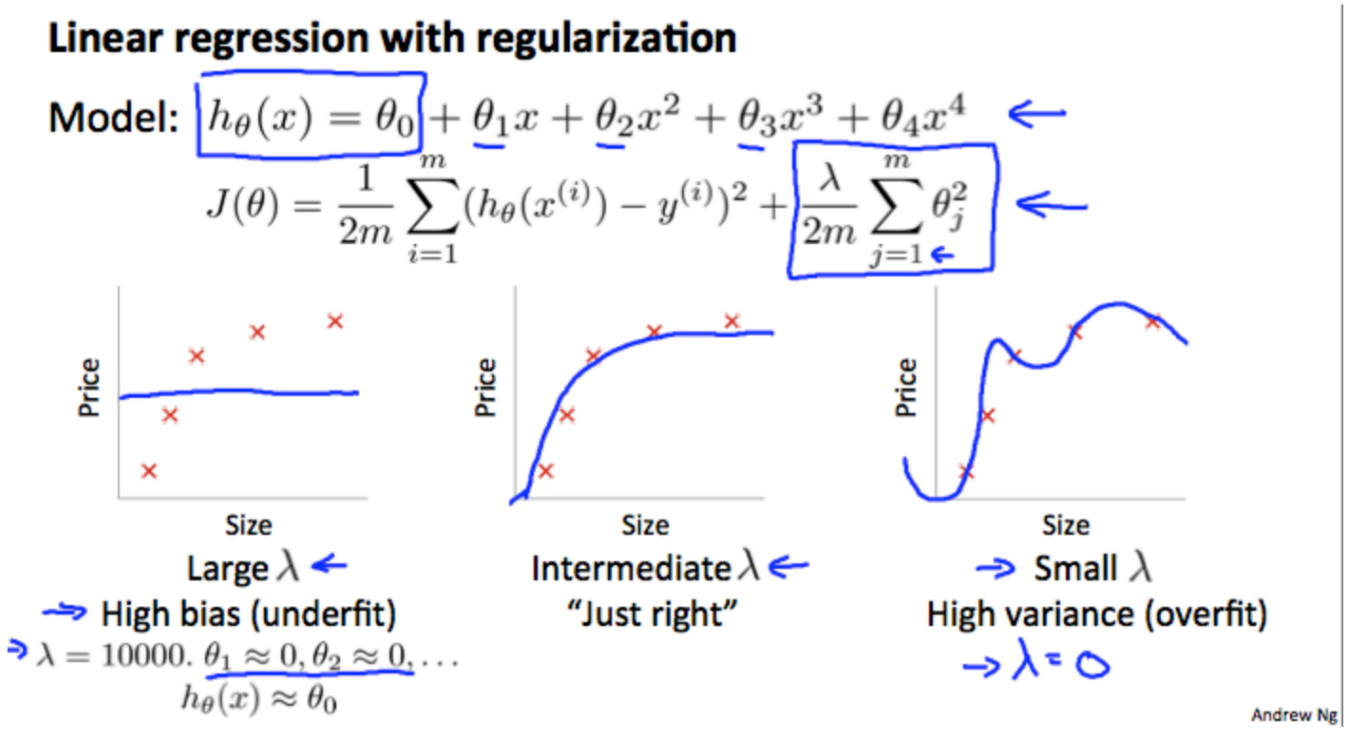
At the same time, the cross validation error will tend to **decrease** as we increase d up to a point, and then it will **increase** as d is increased, forming a convex curve.

- High bias (underfitting): both $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ will be high. Also, $J_{CV}(\Theta) \approx J_{train}(\Theta)$.
- High variance (overfitting): $J_{train}(\Theta)$ will be low and $J_{CV}(\Theta)$ will be much greater than $J_{train}(\Theta)$.

The is summarized in the figure below:



Regularization and Bias/Variance



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

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

Learning Curves

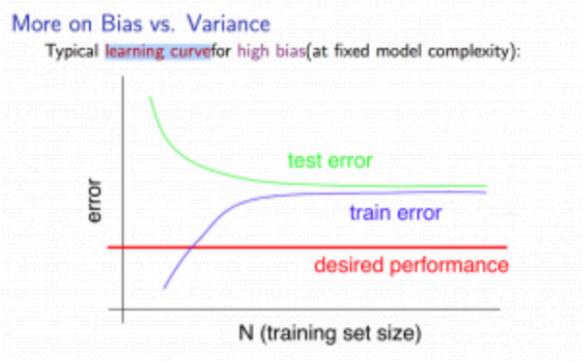
Training an algorithm on a very few number of data points (such as 1, 2 or 3) will easily have 0 errors because we can always find a quadratic curve that touches exactly those number of points. Hence:

- 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.

Experiencing high bias:

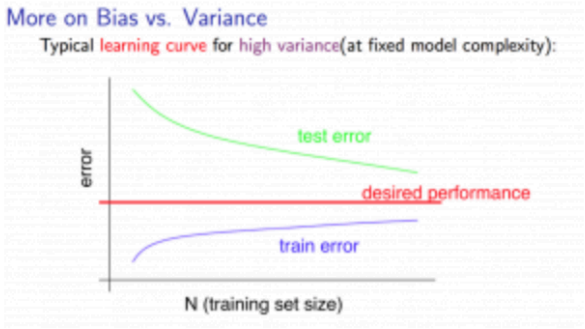
- **Low training set size**: causes $J_{train}(\Theta)$ to be low and $J_{CV}(\Theta)$ to be high.
- **Large training set size**: causes both $J_{train}(\Theta)$ and $J_CV(\Theta)$ to be high with $J_{train}(\Theta) \approx J_{CV}(\Theta)$.

> If a learning algorithm is suffering from high bias, getting more training data will not (by itself) help much.



Experiencing high variance:

- **Low training set size**: causes $J_{train}(\Theta)$ to be low and $J_{CV}(\Theta)$ to be high.
- **Large training set size**: $J_{train}(\Theta)$ increases with training set size and $J_{CV}(\Theta)$ continues to decrease without leveling off. Also, $J_{train}(\Theta) < J_{CV}(\Theta)$ but the difference between them remains significant.

> If a learning algorithm is suffering from high variance, getting more training data is likely to help.
>
> 
>

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.

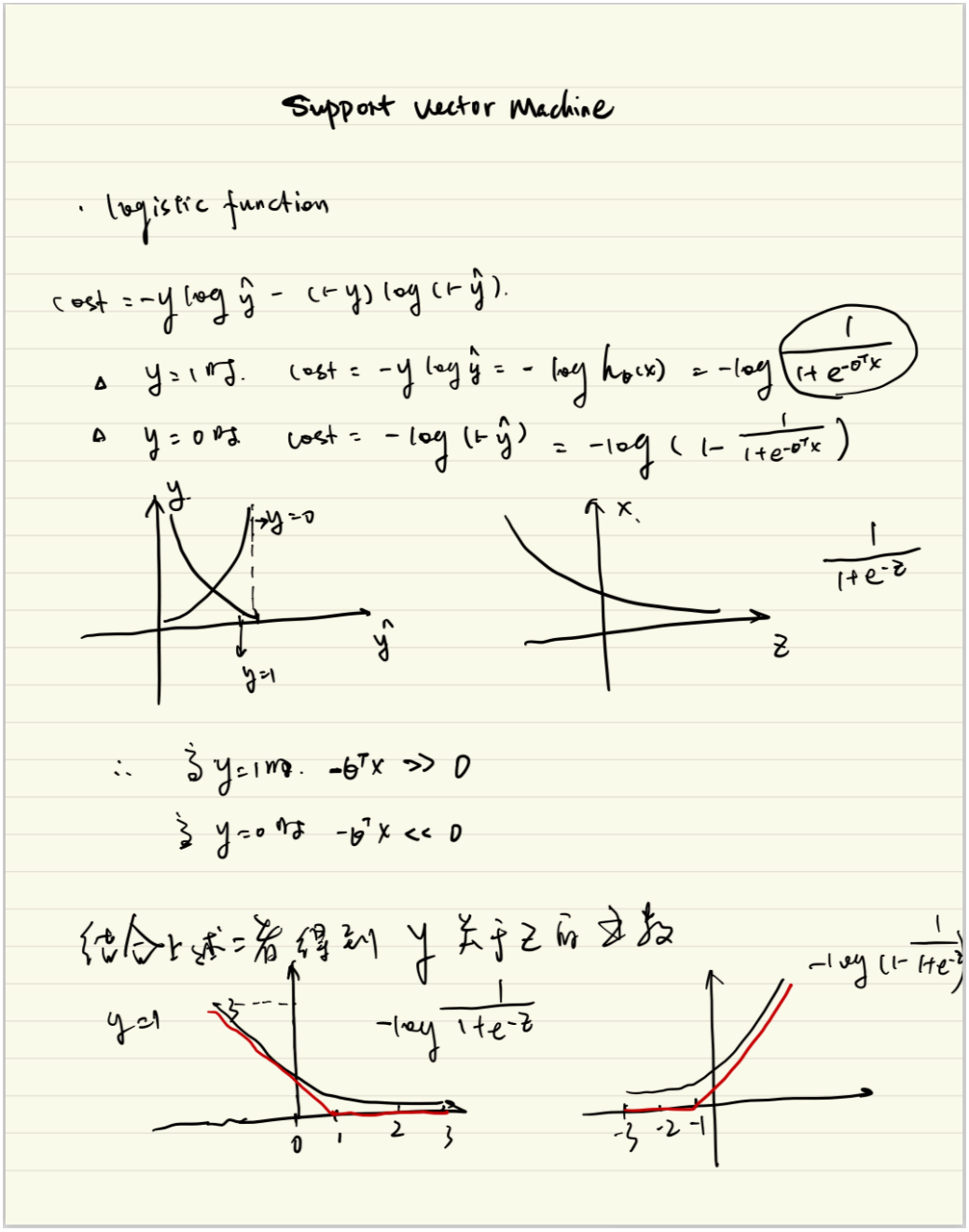
Diagnosing Neural Networks

- 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.

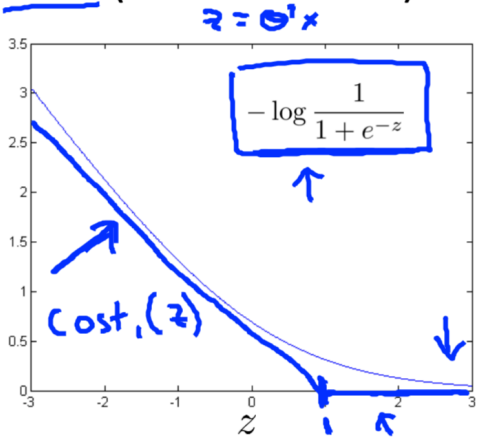
Support Vector Machines

from logistic to SVM

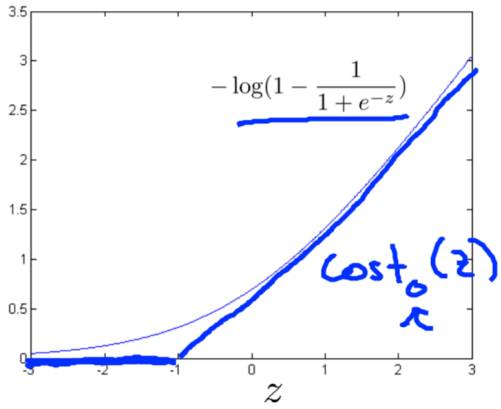


Cost Function

If $y = 1$ (want $\theta^T x \gg 0$):



If $y = 0$ (want $\theta^T x \ll 0$):



recall the cost function of logistic Regression:

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

We replace some terms with new terms and delete some terms which will not influence the results:

$$\min_\theta C\sum_{i=1}^{m}[y^{(i)}cost_1(\theta^T x^{(i)})) + (1-y^{(i)})cost_0(\theta^T x^{(i)})] + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$$

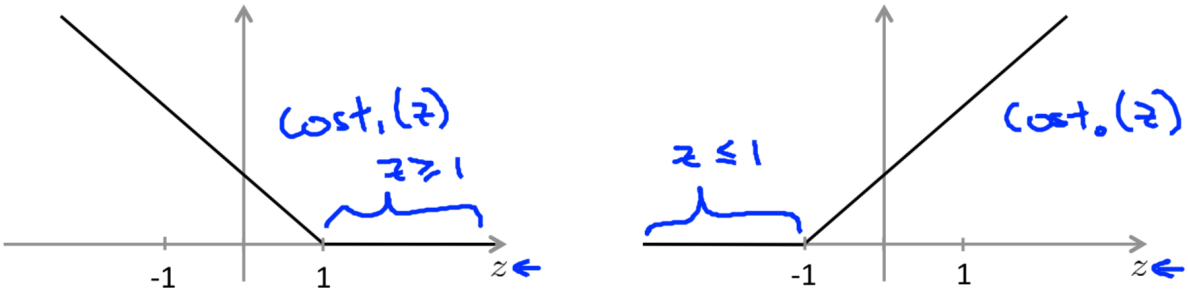
Hypothesis

$$h_\theta = \begin{cases} 0 & if\ \theta^T X > 0 \\ 1 & otherwise \end{cases}$$

Large Margin Intuion

Support Vector Machine

$$\min_\theta C\sum_{i=1}^{m}\left[y^{(i)}cost_1(\theta^T x^{(i)}) + (1-y^{(i)})cost_0(\theta^T x^{(i)})\right] + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$$



If $y = 1$, we want $\theta^T x \geq 1$ (not just ≥ 0) $\quad \theta^T x \geqslant 1$

If $y = 0$, we want $\theta^T x \leq -1$ (not just < 0) $\quad \theta^T x \leqslant -1$

$C = 100{,}000$

SVM Decision Boundary

$$\min_\theta C\sum_{i=1}^{m}\left[y^{(i)}cost_1(\theta^T x^{(i)}) + (1-y^{(i)})cost_0(\theta^T x^{(i)})\right] + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$$

Whenever $y^{(i)} = 1$:

$\theta^T x^{(i)} \geq 1$

$\min_\theta \; 0 + \frac{1}{2}\sum_{i=1}^{n}\theta_j^2$

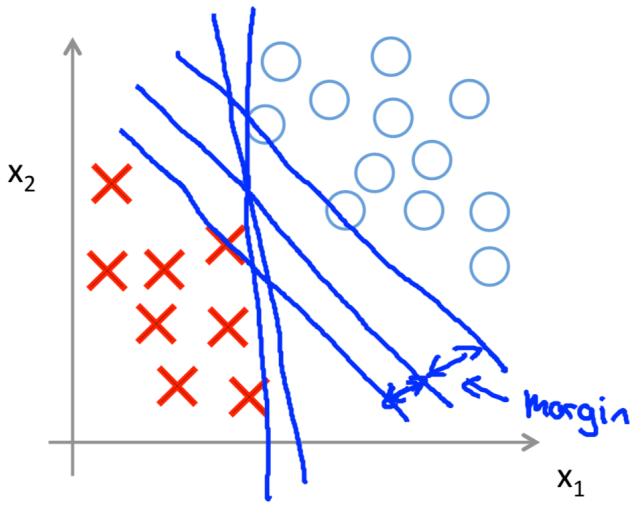
s.t. $\theta^T x^{(i)} \geq 1$ if $y^{(i)} = 1$

$\theta^T x^{(i)} \leq -1$ if $y^{(i)} = 0$

Whenever $y^{(i)} = 0$:

$\theta^T x^{(i)} \leq -1$

SVM Decision Boundary: Linearly separable case

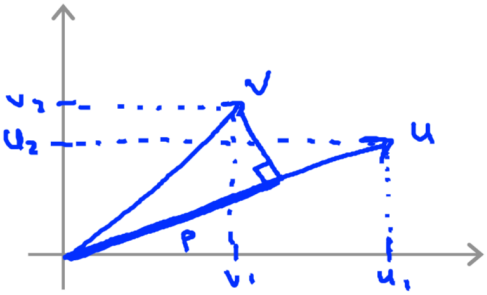


Large margin classifier

Vector inner product

Vector Inner Product

$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$ $v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$



$u^T v = ?$ $\begin{bmatrix} u_1 & u_2 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$

$\|u\|$ = length of vector u

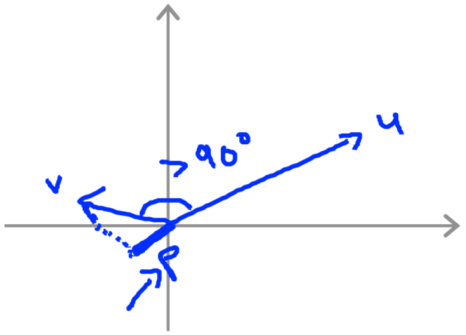
$= \sqrt{u_1^2 + u_2^2} \in \mathbb{R}$

p = length of projection of v onto u.

Signed

$u^T v = p \cdot \|u\| \leftarrow = v^T u$

$= u_1 v_1 + u_2 v_2 \leftarrow \quad p \in \mathbb{R}$



$u^T v = p \cdot \|u\|$

$p < 0$

Andrew

SVM Decision Boundary

$\omega = (\sqrt{\omega})^2$

$$\min_\theta \frac{1}{2} \sum_{j=1}^{n} \theta_j^2 \quad = \frac{1}{2}(\theta_1^2 + \theta_2^2) = \frac{1}{2}\left(\sqrt{\theta_1^2 + \theta_2^2}\right)^2 = \frac{1}{2}\|\theta\|^2$$

$= \|\theta\|$

$$\text{s.t.} \quad \theta^T x^{(i)} \geq 1 \quad \text{if } y^{(i)} = 1$$

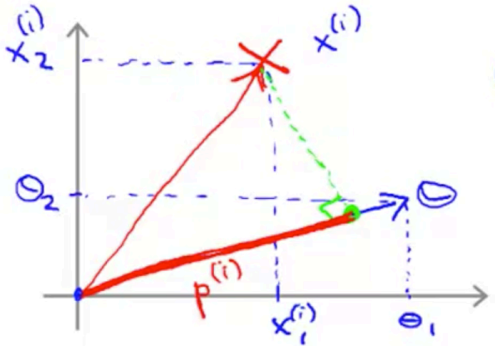
$$\theta^T x^{(i)} \leq -1 \quad \text{if } y^{(i)} = 0$$

$\begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix}$ $\theta_0 = 0$

Simplification: $\theta_0 = 0$. $n = 2$

$\theta^T x^{(i)} = ?$

$u^T v$



$\theta^T x^{(i)} = p^{(i)} \cdot \|\theta\| \leftarrow$

$= \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} \leftarrow$

Andrew Ng

SVM Decision Boundary

$$\min_\theta \frac{1}{2}\sum_{j=1}^{n}\theta_j^2 = \frac{1}{2}\|\theta\|^2$$

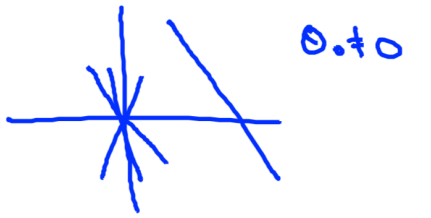
s.t. $p^{(i)} \cdot \|\theta\| \geq 1$ if $y^{(i)} = 1$

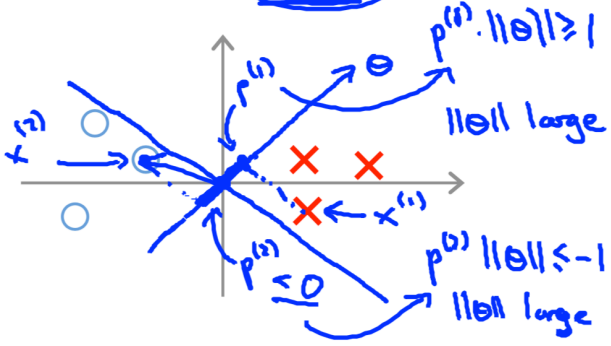
$p^{(i)} \cdot \|\theta\| \leq -1$ if $y^{(i)} = 1$

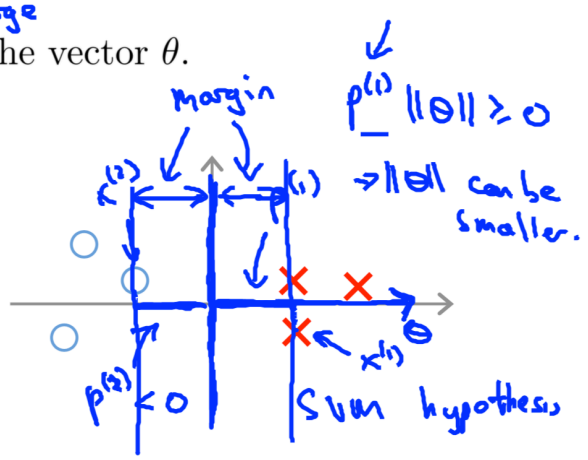
C very large

where $p^{(i)}$ is the projection of $x^{(i)}$ onto the vector θ.

Simplification: $\theta_0 = 0$



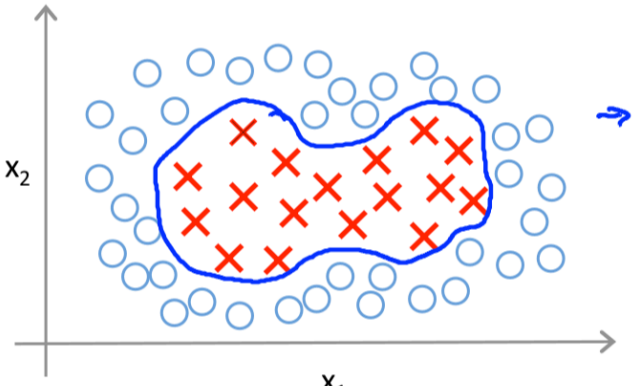




Andrew Ng

kernel

Non-linear Decision Boundary



Predict $y = 1$ if

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 + \theta_5 x_2^2 + \cdots \geq 0$$

$h_\theta(x) = \begin{cases} 1 & \text{if } \theta_0 + \theta_1 x_1 + \cdots \geq 0 \\ 0 & \text{otherwise} \end{cases}$

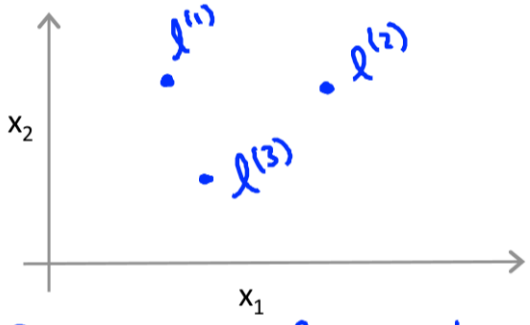
$\Rightarrow \theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 + \cdots$

$f_1 = x_1,\ f_2 = x_2,\ f_3 = x_1 x_2,\ f_4 = x_1^2,\ f_5 = x_1^3, \ldots$

Is there a different / better choice of the features $f_1, f_2, f_3, \ldots$?

Andrew Ng

Kernel



Given x, compute new feature depending on proximity to landmarks $l^{(1)}, l^{(2)}, l^{(3)}$

Given x:

$$f_1 = \text{similarity}(x, l^{(1)}) = \exp\left(-\frac{\|x - l^{(1)}\|^2}{2\sigma^2}\right)$$

$$f_2 = \text{similarity}(x, l^{(1)}) = \exp\left(-\frac{\|x - l^{(2)}\|^2}{2\sigma^2}\right)$$

$$f_3 = \text{similarity}(x, l^{(3)}) = \exp(\dots)$$

Kernel (Gaussian kernels) $k(x, l^{(i)})$

Andrew Ng

Kernels and Similarity

$$f_1 = \text{similarity}(x, l^{(1)}) = \exp\left(-\frac{\|x - l^{(1)}\|^2}{2\sigma^2}\right)$$

If $x \approx l^{(1)}$:

$$f_1 \approx \exp\left(-\frac{0^2}{2\sigma^2}\right) \approx 1$$

If x if far from $l^{(1)}$:

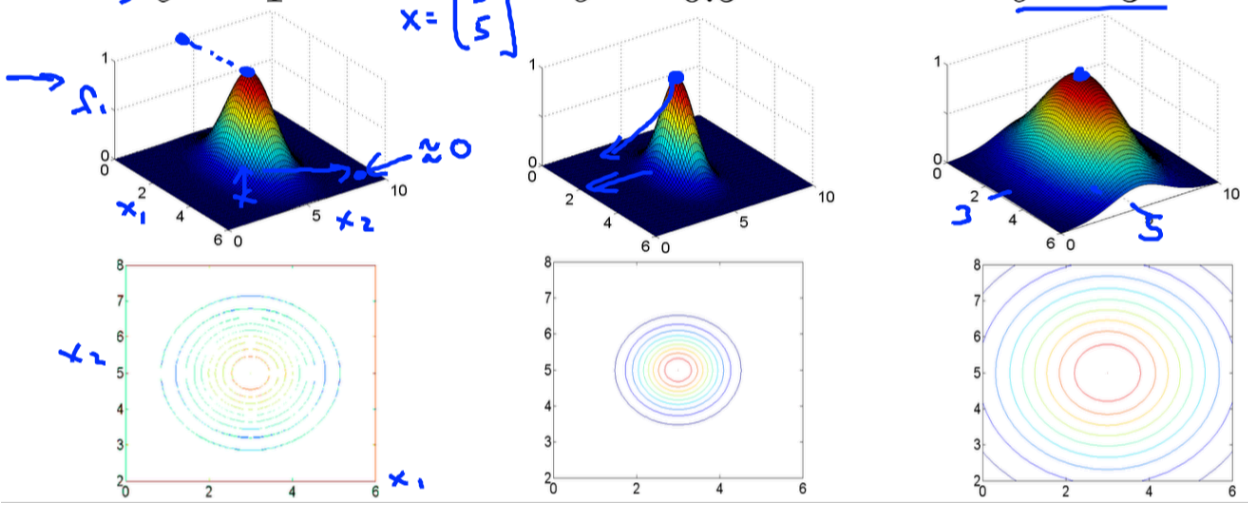
$$f_1 = \exp\left(-\frac{(\text{large number})^2}{2\sigma^2}\right) \approx 0.$$

$l^{(1)} \to f_1$
$l^{(2)} \to f_2$
$l^{(3)} \to f_3$.

Example:

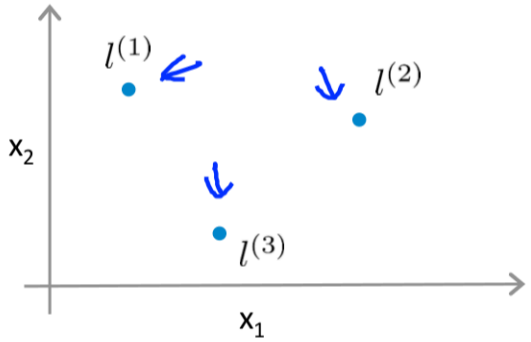
$l^{(1)} = \begin{bmatrix} 3 \\ 5 \end{bmatrix}$, $f_1 = \exp\left(-\frac{\|x - l^{(1)}\|^2}{2\sigma^2}\right)$

$\sigma^2 = 1$ $\quad$ $\sigma^2 = 0.5$ $\quad$ $\sigma^2 = 3$



Andrew Ng

Choosing the landmarks



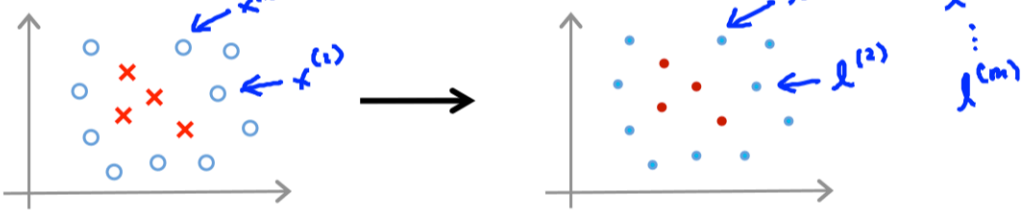
Given x:

$f_i = \text{similarity}(x, l^{(i)})$

$= \exp\left(-\frac{||x - l^{(i)}||^2}{2\sigma^2}\right)$

Predict $y = 1$ if $\theta_0 + \theta_1 f_1 + \theta_2 f_2 + \theta_3 f_3 \geq 0$

Where to get $l^{(1)}, l^{(2)}, l^{(3)}, \ldots$?



Andrew Ng

SVM with Kernels

Hypothesis: Given x, compute features $f \in \mathbb{R}^{m+1}$

→ Predict "y=1" if $\theta^T f \geq 0$

Training:

$$\min_\theta C \sum_{i=1}^{m} y^{(i)} cost_1(\theta^T f^{(i)}) + (1 - y^{(i)}) cost_0(\theta^T f^{(i)}) + \frac{1}{2} \sum_{j=1}^{m} \theta_j^2$$

Andrew Ng

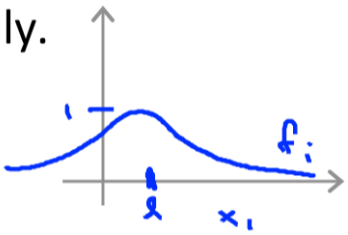
SVM parameters:

C ($= \frac{1}{\lambda}$). → Large C: Lower bias, high variance.

→ Small C: Higher bias, low variance.

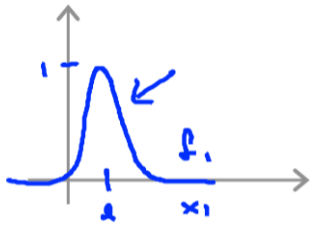
σ^2 Large σ^2: Features f_i vary more smoothly.

→ Higher bias, lower variance.



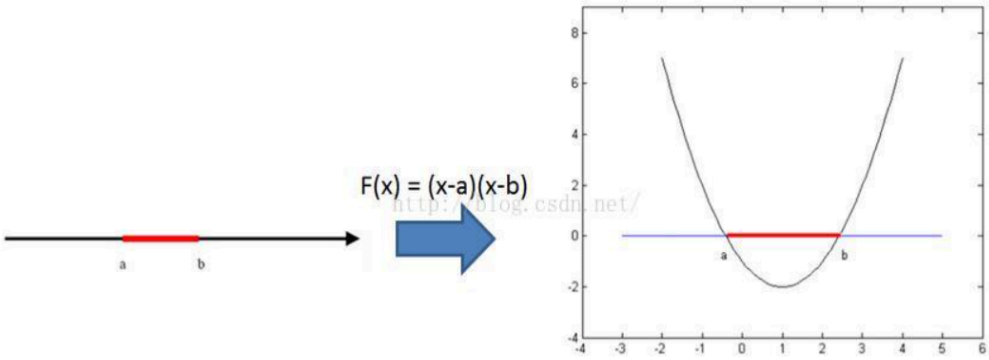
Small σ^2: Features f_i vary less smoothly.

Lower bias, higher variance.



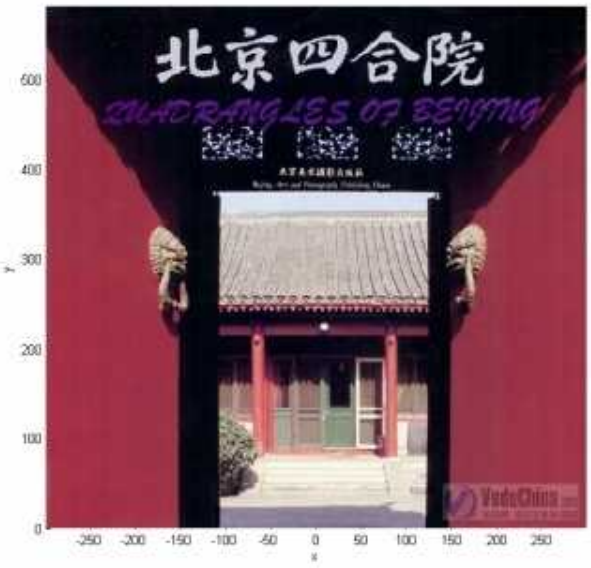
低维线性不可分到高维线性可分的简单例子

那么在高维空间中，样本数据很有可能线性可分，这个特点可以用下图做一个很好的说明：

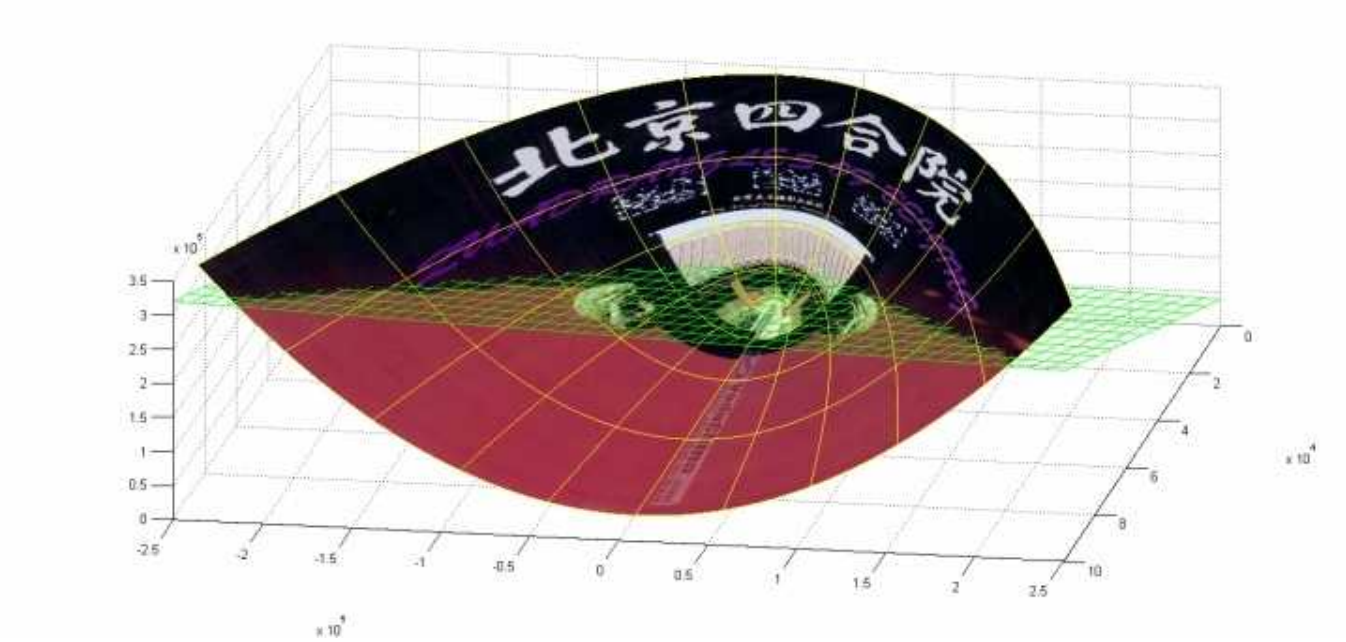


一个核函数把低维空间映射到高维空间的例子

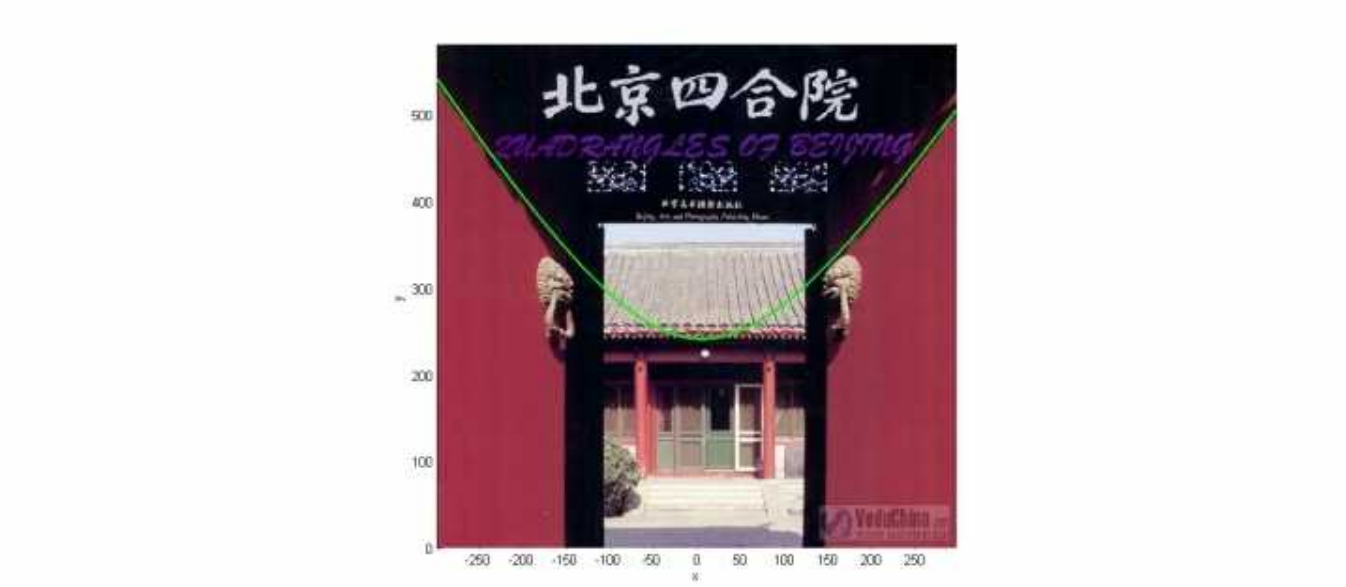
下面这张图位于第一、二象限内。我们关注红色的门，以及“北京四合院”这几个字下面的紫色的字母。我们把红色的门上的点看成是“+”数据，紫色字母上的点看成是“-”数据，它们的横、纵坐标是两个特征。显然，在这个二维空间内，“+”“-”两类数据不是线性可分的。



我们现在考虑核函数，即“内积平方”。这里面是二维空间中的两个点。这个核函数对应着一个二维空间到三维空间的映射，它的表达式是：可以验证，在P这个映射下，原来二维空间中的图在三维空间中的像是这个样子：



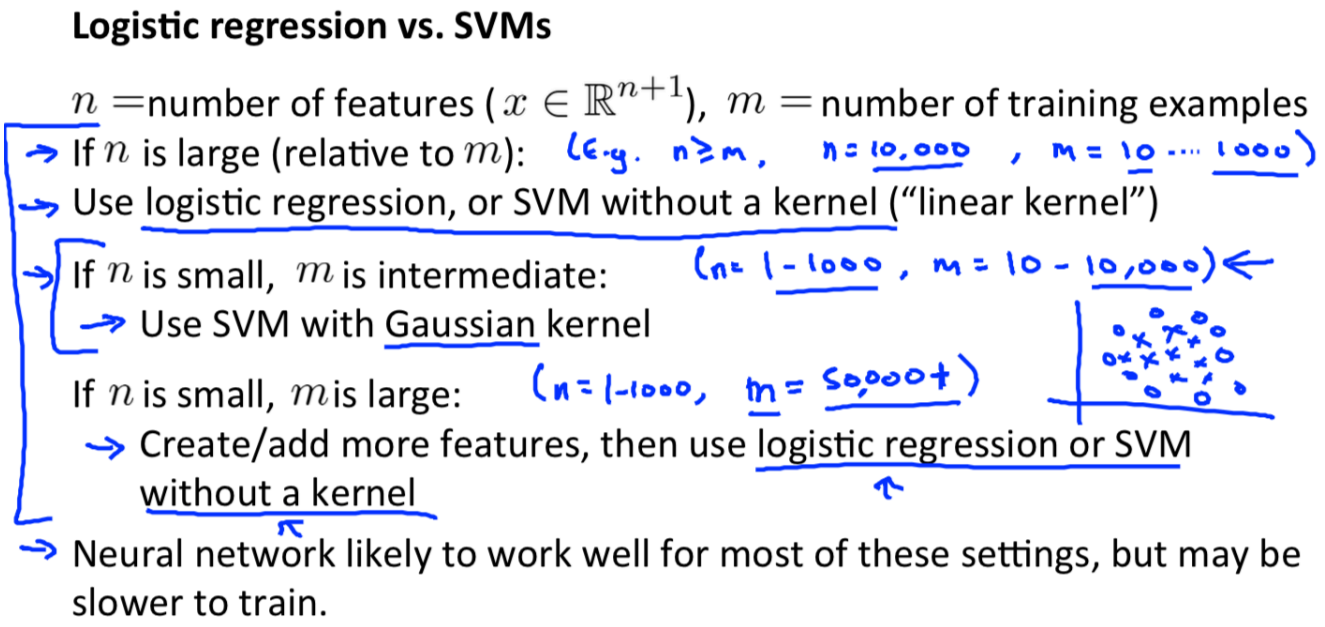
（前后轴为x轴，左右轴为y轴，上下轴为z轴）注意到绿色的平面可以完美地分割红色和紫色，也就是说，两类数据在三维空间中变成线性可分的了。而三维中的这个判决边界，再映射回二维空间中是这样的：



这是一条双曲线，它不是线性的。

核函数的作用就是隐含着一个从低维空间到高维空间的映射，而这个映射可以把低维空间中线性不可分的两类点变成线性可分的。当然，我举的这个具体例子强烈地依赖于数据在原始空间中的位置。事实中使用的核函数往往比这个例子复杂得多。它们对应的映射并不一定能够显式地表达出来；它们映射到的高维空间的维数也比我举的例子（三维）高得多，甚至是无穷维的。这样，就可以期待原来并不线性可分的两类点变成线性可分的了。

SVM or Logistic Regression



- 如果特征维数很高，往往线性可分（SVM解决非线性分类问题的思路就是将样本映射到更高维的特征空间中），可以采用LR或者线性核的SVM；
- 如果样本数量很多，由于求解最优化问题的时候，目标函数涉及两两样本计算内积，使用高斯核明显计算量会大于线性核，所以手动添加一些特征，使得线性可分，然后可以用LR或者线性核的SVM；
- 如果不满足上述两点，即特征维数少，样本数量正常，可以使用高斯核的SVM。

Clustering

K-means algorithm

K-means算是一个很简单的聚类算法，而聚类与决策树、SVM等不同，是一种无监督的学习，所谓无监督学习（Unsupervised learning）是和监督学习相对应的，不同于监督学习，无监督学习所给的训练集是不包含标签的，所有数据集都只包括特征xi
而没有标签yi
。
聚类的主要目的就是将这些没有标签的数据分为N个簇(cluster)，其主要的应用有市场划分、社交网络分析、天文学中的数据分析等等。

K-Means的描述如下：

1. Initialize **cluster centroids** $\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^n$ randomly.

2. Repeat until convergence: {

   For every i, set

   $$c^{(i)} := \arg\min_j ||x^{(i)} - \mu_j||^2.$$

   For each j, set

   $$\mu_j := \frac{\sum_{i=1}^{m} 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} 1\{c^{(i)} = j\}}.$$

   }

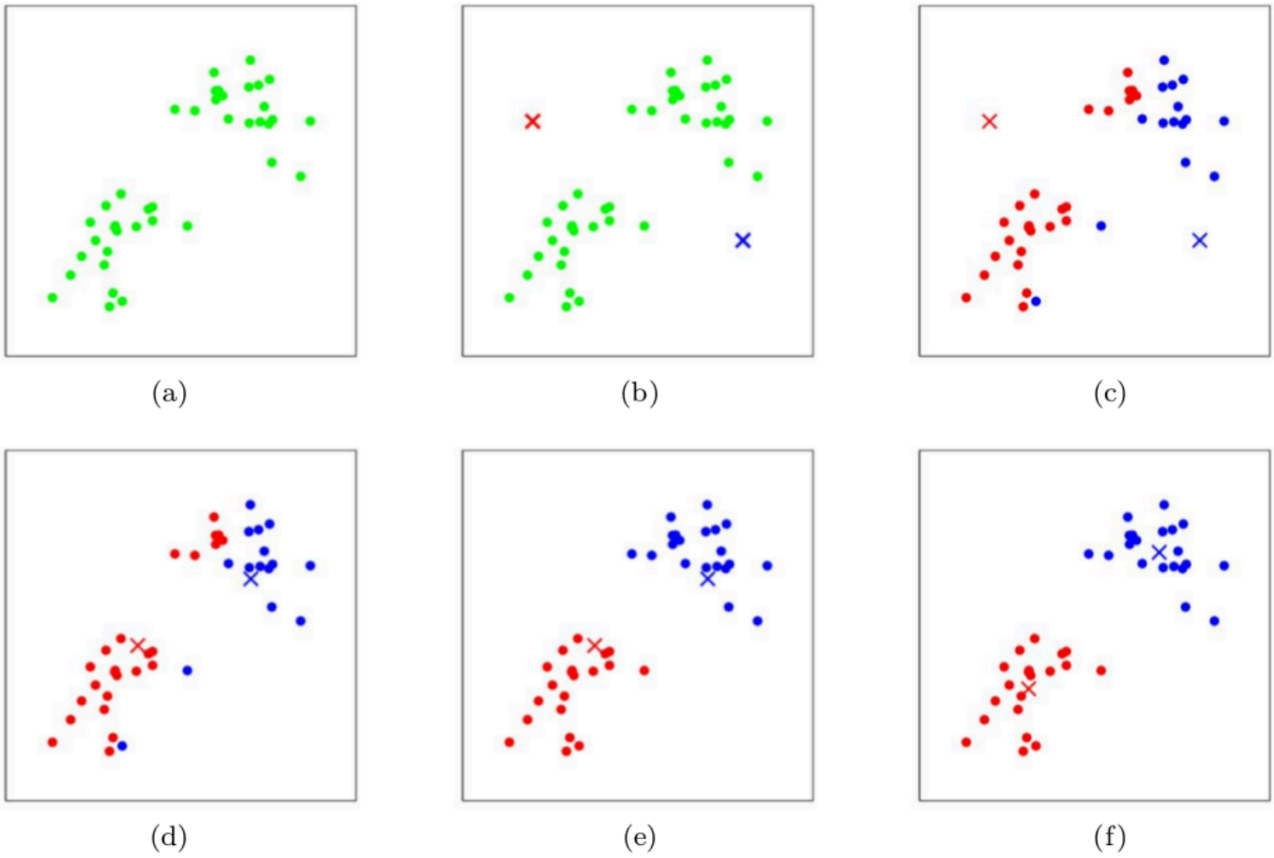
先对参数进行说明：

- $x^{(i)}$ 为第i个数据点；
- $c^{(i)}$ 为x^{(i)}的簇；
- u_j 为第j个簇的质心点；

在对算法进行说明：

1. 首先需要初始化质心点，在K-Means中，通常采用随机的方法对质心点进行初始化。更好的办法是：随机选择m(m>k)个数据，再从中选择k个数据点作为质心点；
2. 第一个for循环主要用于给数据点$x^{(i)}$ 赋值$c^{(i)}$ ，称为 cluster assignment steps，对每一个数据点，都会计算她与所有质心点的距离，而后将数据点分配到与它距离最近的簇；
3. 第二个for循环主要用于更新质心点的位置，称为move centroid steps，而u_j 这里的计算公式所代表的意思就是，分母：统计所有$c_i = j$的点的个数；分子是所有$c_i = j$的点的坐标和。那整体的意思就很明确了，就是求这些点的平均值，作为新的质心点的位置。

当c和u收敛之后，就可以结束整个迭代过程。下面看一个实例：

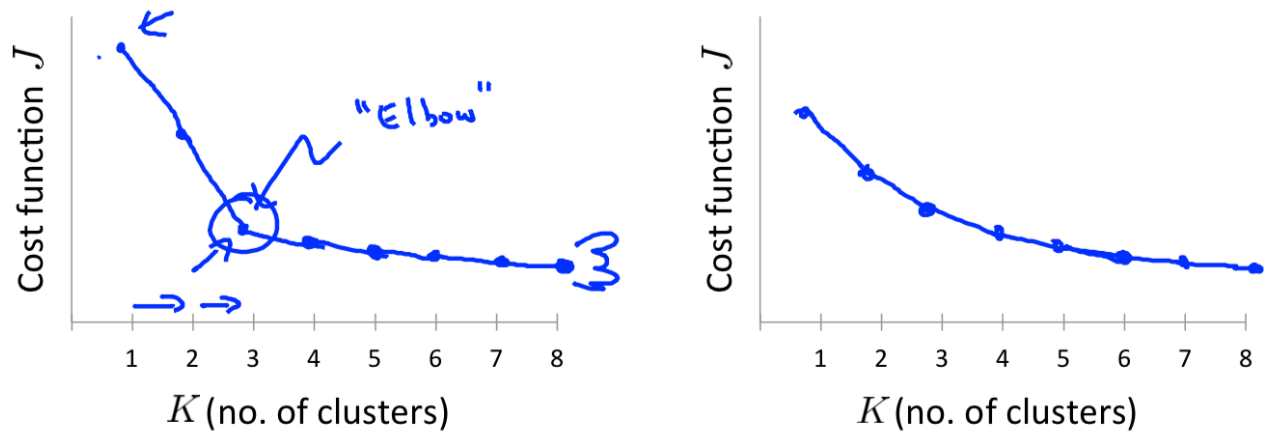


优化目标

在上一部分中，我们说最终的目的是要达到一个收敛，那我们就用一个失真函数（distortion function）来衡量。

$$J(c,\mu) = \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2$$

J(c,u)实际上是一个单调递减的函数，且是一个非凸函数，只要我们能找到拐点，那我们就已经达到了收敛，又称这种方法为elbow function。偶尔也有可能陷入局部最优情况，或出现震荡情况，这样一定是有问题了。



总结

K-Means虽然简单，容易实现，但是也会收敛到局部最小值，这种情况下可以采用K-Means的改进算法：二分K-均值算法。算法的思想就是：首先将所有点做为一个簇，然后将该簇一分为二。之后选择其中一个簇进行继续划分，选择哪一个簇进行划分则取决于对其划分是否可以最大程度降低SSE的值，不断划分直到达到用户所指定的K值。

```
#! /usr/bin/env python
# -*-coding:utf8 -*-

from numpy import *


def loadDataSet(fileName):
    dataMat = []
    fr = open(fileName)
    for line in fr.readlines():
        curLine = line.strip().split('/t')
        fltLine = map(float, curLine)
        dataMat.append(fltLine)
    return dataMat


def distEclud(vecA, vecB):

    return sqrt(sum(power(vecA - vecB), 2))


def randCent(dataSet, k):
    n = shape(dataSet)[1]
    centroids = mat(zeros(k,n))
    for j in range(n):
        minJ = min(dataSet[:, j])
        rangeJ = float(max(dataSet[:, j]) - minJ)
        centroids[:, j] = minJ + rangeJ*random.rand(k, 1)
    return centroids

def kMeans(dataSet, k, distMeas = distEclud, createCent = randCent):
    # 确定数据集的大小
    m = shape(dataSet)[0]
    # 建立矩阵来存储c(i)和x(i)距离u(c(i))的值
    clusterAssment = mat(zeros(m,2))
    #随机初始化质心
    centroids= createCent(dataSet, k)
    # 设置标志位
    clusterChanged = True
    while clusterChanged:
        clusterChanged = False
        for i in range(m):
            minDist = inf; minIndex = -1
            for j in range(k):
                distJI = distMeas(centroids[k, :], dataSet[i, :])
                if distJI < minDist :
                    minDist = distJI; minIndex = j
                if clusterAssment[i, 0] != minIndex:
                    clusterChanged = True
                clusterAssment[i, :] = minIndex,minDist**2
            print centroids
```

```
        for cent in range(k):
            ptsInClust = dataSet[nonzero(clusterAssment[:, 0].A == cent)[0]]
            centroids[cent, :] = mean(ptsInClust, axis=0)
        return centroids, clusterAssment
```

Dimensionality Reducion

why we need to do Dimensionality Reducion?

- data compression
- visually

PCA实例

现在假设有一组数据如下：

	x	y
	2.5	2.4
	0.5	0.7
	2.2	2.9
	1.9	2.2
Data =	3.1	3.0
	2.3	2.7
	2	1.6
	1	1.1
	1.5	1.6
	1.1	0.9

行代表了样例，列代表特征，这里有10个样例，每个样例两个特征。可以这样认为，有10篇文档，x是10篇文档中“learn”出现的TF-IDF，y是10篇文档中“study”出现的TF-IDF。

1. 分别求x和y的平均值，然后对于所有的样例，都减去对应的均值。这里x的均值是1.81，y的均值是1.91，那么一个样例减去均值后即为（0.69,0.49），得到

	x	y
	0.69	0.49
	-1.31	-1.21
	0.39	0.99
	0.09	0.29
DataAdjust =	1.29	1.09
	0.49	0.79
	0.19	-0.31
	-0.81	-0.81
	-0.31	-0.31
	-0.71	-1.01

2. 求特征协方差矩阵，如果数据是3维，那么协方差矩阵是：

$$C = \begin{pmatrix} cov(x,x) & cov(x,y) & cov(x,z) \\ cov(y,x) & cov(y,y) & cov(y,z) \\ cov(z,x) & cov(z,y) & cov(z,z) \end{pmatrix}$$

这里只有x和y，求解得

$$cov = \begin{pmatrix} .616555556 & .615444444 \\ .615444444 & .716555556 \end{pmatrix}$$

对角线上分别是x和y的方差，非对角线上是协方差。协方差是衡量两个变量同时变化的变化程度。协方差大于0表示x和y若一个增，另一个也增；小于0表示一个增，一个减。如果 x 和 y 是统计独立的，那么二者之间的协方差就是 0；但是协方差是 0，并不能说明 x 和 y 是独立的。协方差绝对值越大，两者对彼此的影响越大，反之越小。协方差是没有单位的量，因此，如果同样的两个变量所采用的量纲发生变化，它们的协方差也会产生树枝上的变化。

3. 求协方差的特征值和特征向量，得到：

$$eigenvalues = \begin{pmatrix} .0490833989 \\ 1.28402771 \end{pmatrix}$$

$$eigenvectors = \begin{pmatrix} -.735178656 & -.677873399 \\ .677873399 & -.735178656 \end{pmatrix}$$

上面是两个特征值，下面是对应的特征向量，特征值0.0490833989对应特征向量为，这里的特征向量都归一化为单位向量。

4. 将特征值按照从大到小的顺序排序，选择其中最大的k个，然后将其对应的k个特征向量分别作为列向量组成特征向量矩阵。这里特征值只有两个，我们选择其中最大的那个，这里是1.28402771，对应的特征向量是(-0.677873399, -0.735178656)T。

5. 第五步，将样本点投影到选取的特征向量上。假设样例数为m，特征数为n，减去均值后的样本矩阵为DataAdjust(m*n*)，*协方差矩阵是*nn，选取的k个特征向量组成的矩阵为EigenVectors(n*k)。那么投影后的数据FinalData为

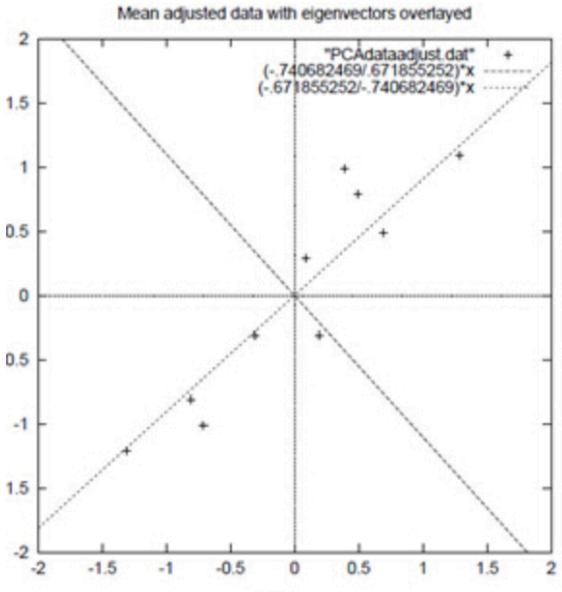
   FinalData(10*1*) = *DataAdjust(10*2矩阵) x 特征向量(-0.677873399, -0.735178656)T

   得到的结果是：

Transformed Data (Single rigenvector)
x
-0.827970186
1.77758033
-0.992197494
-0.274210416
-1.67580142
-0.912949103
0.991094375
1.14457216
0.438046137
1.22382056

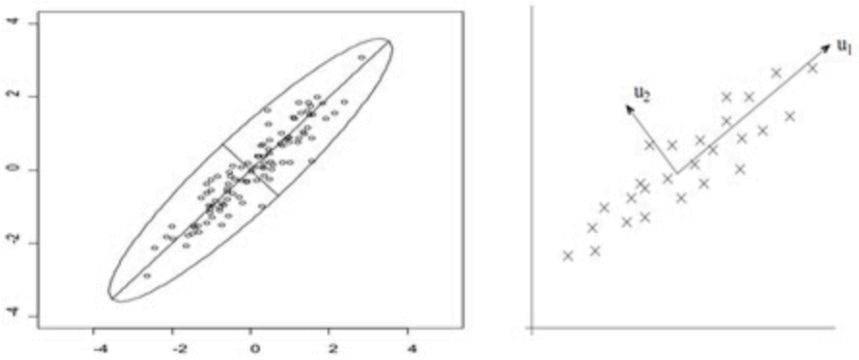
这样，就将原始样例的n维特征变成了k维，这k维就是原始特征在k维上的投影。

上面的数据可以认为是learn和study特征融合为一个新的特征叫做LS特征，该特征基本上代表了这两个特征。上述过程如下图2描述：



正号表示预处理后的样本点，斜着的两条线就分别是正交的特征向量（由于协方差矩阵是对称的，因此其特征向量正交），最后一步的矩阵乘法就是将原始样本点分别往特征向量对应的轴上做投影。

pca的理解



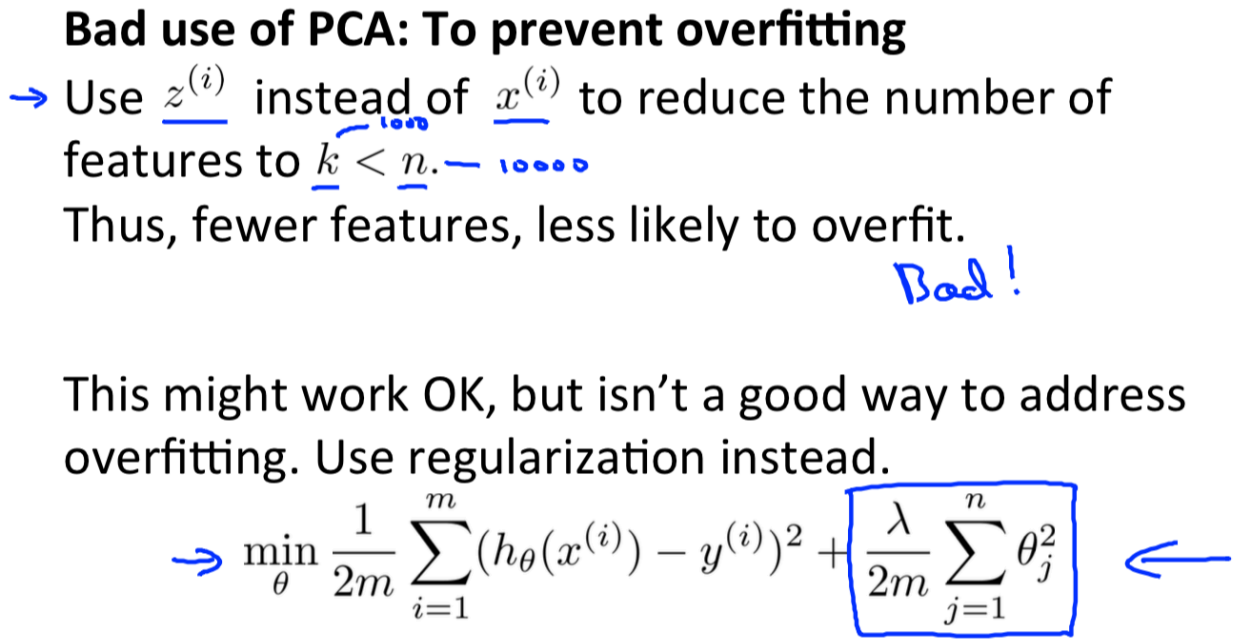
在第一部分中，我们举了一个学生成绩的例子，里面的数据点是六维的，即每个观测值是6维空间中的一个点。我们希望将6维空间用低维空间表示。

先假定只有二维，即只有两个变量，它们由横坐标和纵坐标所代表；因此每个观测值都有相应于这两个坐标轴的两个坐标值；如果这些数据形成一个椭圆形状的点阵，那么这个椭圆有一个长轴和一个短轴。在短轴方向上，数据变化很少；在极端的情况，短轴如果退化成一点，那只有在长轴的方向才能够解释这些点的变化了；这样，由二维到一维的降维就自然完成了。

上图中，u1就是主成分方向，然后在二维空间中取和u1方向正交的方向，就是u2的方向。则n个数据在u1轴的离散程度最大（方差最大），数据在u1上的投影代表了原始数据的绝大部分信息，即使不考虑u2，信息损失也不多。而且，u1、u2不相关。只考虑u1时，二维降为一维。

椭圆的长短轴相差得越大，降维也越有道理。

PCA 算法的应用



Application of PCA

- Compression
  - Reduce memory/disk needed to store data
  - Speed up learning algorithm ←

Choose k by % of variance retain

- Visualization

$k=2$ or $k=3$

Anomaly detection

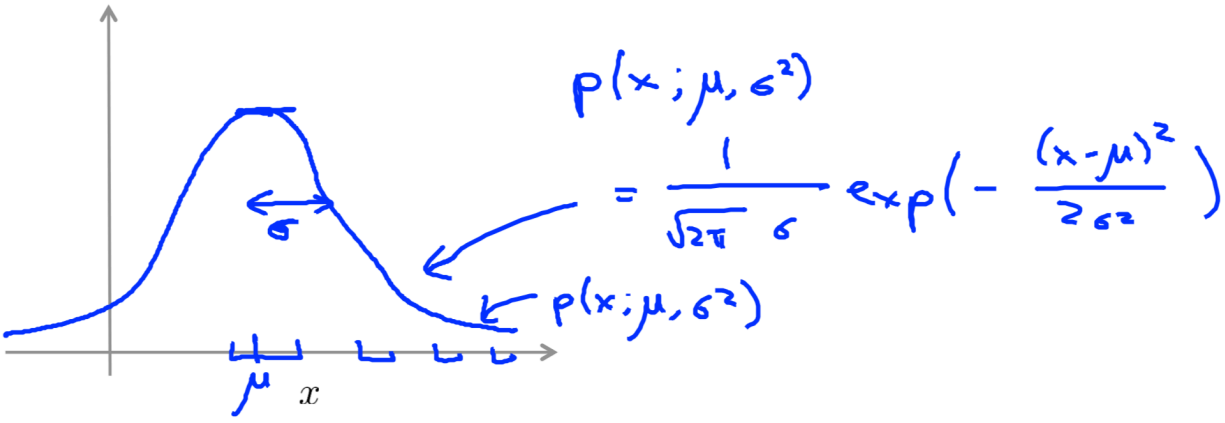
Gaussian (Normal) distribution

Say $x \in \mathbb{R}$. If x is a distributed Gaussian with mean μ, variance σ^2.

$x \sim \mathcal{N}(\mu, \sigma^2)$

↑ "distributed as"

σ standard deviation

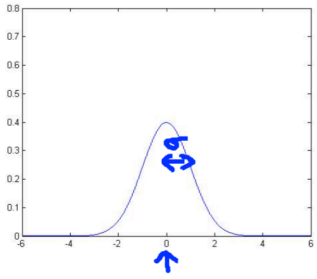


$$p(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi}\,\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

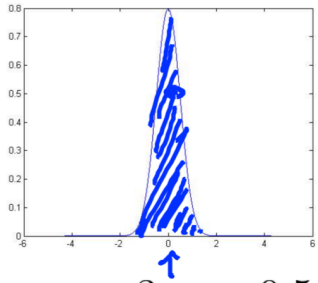
$p(x;\mu,\sigma^2)$

Gaussian distribution example

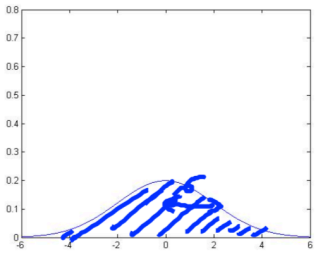
$\mu = 0, \sigma = 1$



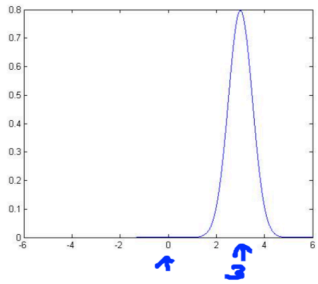
$\mu = 0, \sigma = 0.5$ $\sigma^2 = 0.25$



$\mu = 0, \sigma = 2$



$\mu = 3, \sigma = 0.5$



Andrew Ng

Algorithm

Density estimation

Training set: $\{x^{(1)}, \ldots, x^{(m)}\}$

Each example is $x \in \mathbb{R}^n$

$x_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$

$x_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$

$x_3 \sim \mathcal{N}(\mu_3, \sigma_3^2)$

$p(x)$

$= p(x_1; \mu_1, \sigma_1^2)\, p(x_2; \mu_2, \sigma_2^2)\, p(x_3; \mu_3, \sigma_3^2) \cdots p(x_n; \mu_n, \sigma_n^2)$

$= \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$

$\sum_{i=1}^{n} i = 1+2+3+\ldots+n$

$\prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \ldots \times n$

Andrew Ng

Anomaly detection algorithm

1. Choose features x_i that you think might be indicative of anomalous examples.
2. Fit parameters $\mu_1, \ldots, \mu_n, \sigma_1^2, \ldots, \sigma_n^2$

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2$$

3. Given new example x, compute $p(x)$:

$$p(x) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2) = \prod_{j=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_j} \exp\left(-\frac{(x_j - \mu_j)^2}{2\sigma_j^2}\right)$$

Anomaly if $p(x) < \varepsilon$

Andrew Ng

Algorithm evaluation

Fit model $p(x)$ on training set $\{x^{(1)}, \ldots, x^{(m)}\}$

On a cross validation/test example x, predict

$$y = \begin{cases} 1 & \text{if } p(x) < \varepsilon \text{ (anomaly)} \\ 0 & \text{if } p(x) \geq \varepsilon \text{ (normal)} \end{cases}$$

Possible evaluation metrics:

- True positive, false positive, false negative, true negative
- Precision/Recall
- F_1-score

Can also use cross validation set to choose parameter ε

Andrew Ng

Anomaly detection VS. Supervised learning

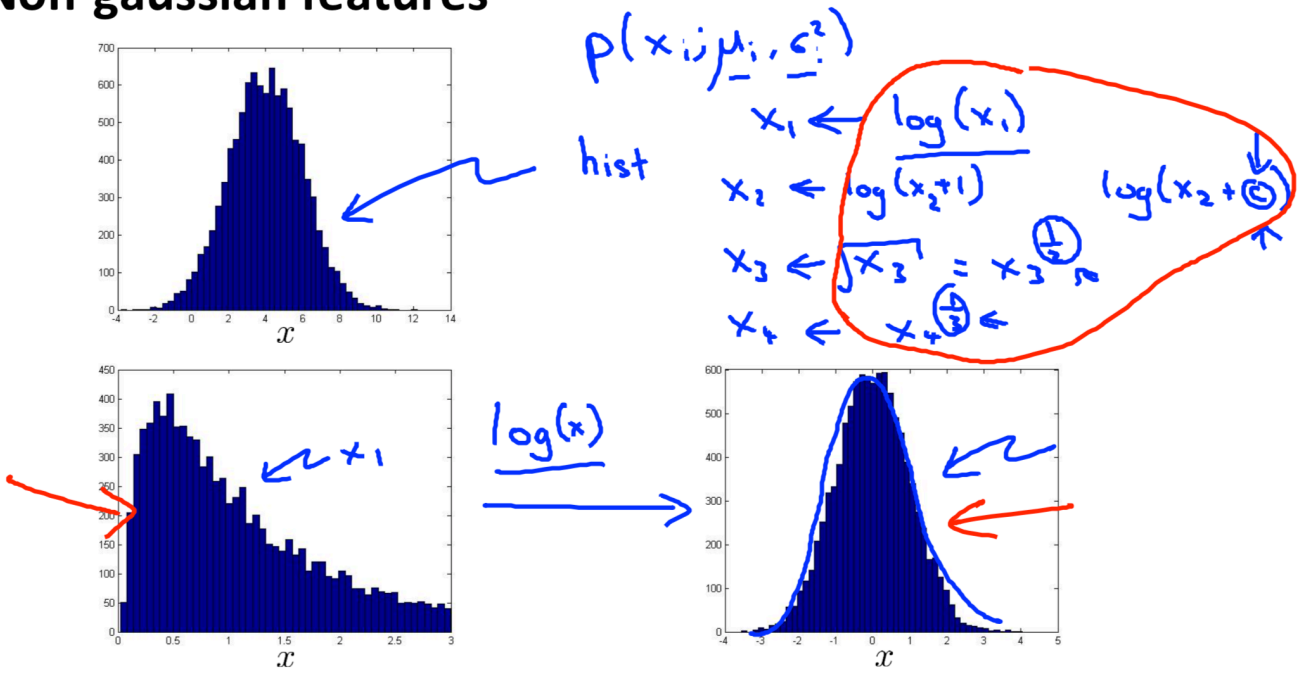
Very small number of positive examples ($y = 1$). (0-20 is common). Large number of negative ($y = 0$) examples. Many different "types" of anomalies. Hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like any of the anomalous examples we've seen so far.	Large number of positive and negative examples. Enough positive examples for algorithm to get a sense of what positive examples are like, future positive examples likely to be similar to ones in training set.
• Fraud detection • Manufacturing (e.g. aircraft engines) • Monitoring machines in a data center ⋮	• Email spam classification • Weather prediction (sunny/ rainy/etc). • Cancer classification ⋮

Choosing what features to use

主要思路是将非高斯分布的特征经过变换转换成高斯分布的特征

$$eg. x \rightarrow log(x)$$

Non-gaussian features



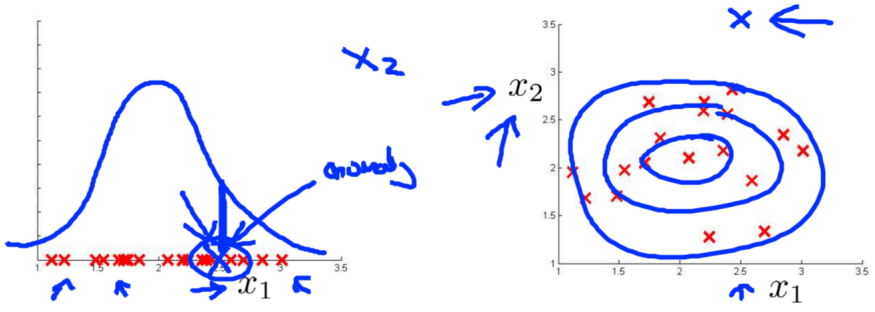
Error analysis for anomaly detection

利用误差分析寻找新特征

Want $p(x)$ large for normal examples x.
$p(x)$ small for anomalous examples x.

Most common problem:

$p(x)$ is comparable (say, both large) for normal and anomalous examples



Monitoring computers in a data center

Choose features that might take on unusually large or small values in the event of an anomaly.

- x_1 = memory use of computer
- x_2 = number of disk accesses/sec
- x_3 = CPU load
- x_4 = network traffic

$x_5 = \frac{\text{CPU load}}{\text{network traffic}}$

$x_6 = \frac{(\text{CPU load})^2}{\text{network traffic}}$

Recommender Systems

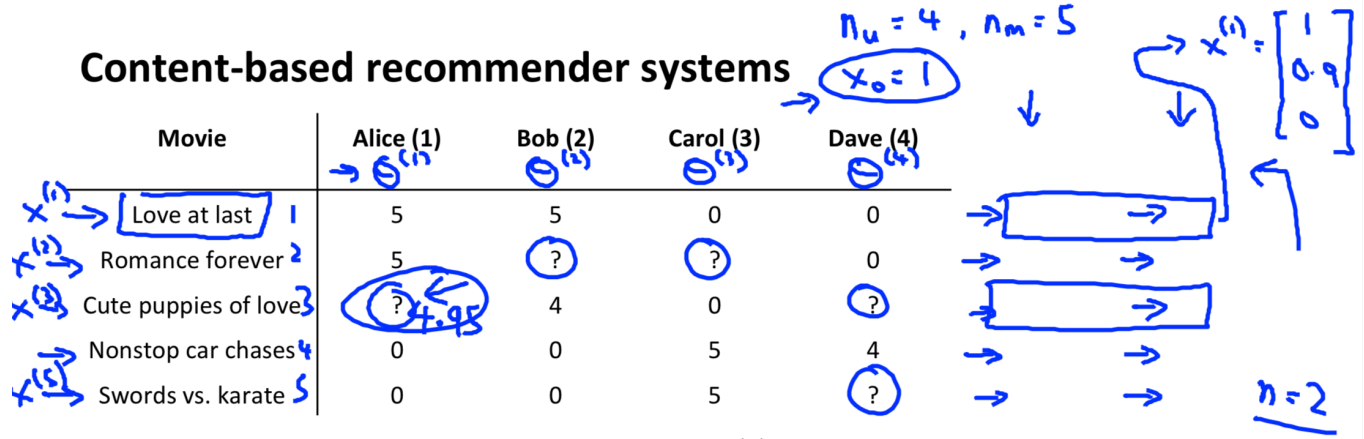
Problem motivation

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
Love at last	5	5	0	0	0.9	0
Romance forever	5	?	?	0	1.0	0.01
Cute puppies of love	?	4	0	?	0.99	0
Nonstop car chases	0	0	5	4	0.1	1.0
Swords vs. karate	0	0	5	?	0	0.9

已知电影的特征x_1, x_2，对每一个用户拟合线性回归模型$\theta^T X$, 此时每个电影是一个样本，优化目标是电影的评分。

Content-based recommender systems

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)
Love at last	5	5	0	0
Romance forever	5	?	?	0
Cute puppies of love	?	4	0	?
Nonstop car chases	0	0	5	4
Swords vs. karate	0	0	5	?



For each user j, learn a parameter $\theta^{(j)} \in \mathbb{R}^3$. Predict user j as rating movie i with $(\theta^{(j)})^T x^{(i)}$ stars.

$x^{(3)} = \begin{bmatrix} 1 \\ 0.99 \\ 0 \end{bmatrix}$ $\leftrightarrow$ $\theta^{(1)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix}$ $(\theta^{(1)})^T x^{(3)} = 5 \times 0.99 = 4.95$

Andrew Ng

Optimization objective:

To learn $\theta^{(j)}$ (parameter for user j):

$$\min_{\theta^{(j)}} \frac{1}{2} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

To learn $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(n_u)}$:

$$\min_{\theta^{(1)},\ldots,\theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} \left((\theta^{(j)})^T x^{(i)} - y^{(i,j)} \right)^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

$\theta^{(1)}, \ldots, \theta^{(n_u)}$

Optimization algorithm:

$$\min_{\theta^{(1)},\ldots,\theta^{(n_u)}} \frac{1}{2}\sum_{j=1}^{n_u}\sum_{i:r(i,j)=1}\left((\theta^{(j)})^T x^{(i)} - y^{(i,j)}\right)^2 + \frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}(\theta_k^{(j)})^2$$

$J(\theta^{(1)},\ldots,\theta^{(n_u)})$

Gradient descent update:

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \sum_{i:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})x_k^{(i)} \quad (\text{for } k = 0)$$

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha\left(\sum_{i:r(i,j)=1}((\theta^{(j)})^T x^{(i)} - y^{(i,j)})x_k^{(i)} + \lambda\theta_k^{(j)}\right) \quad (\text{for } k \neq 0)$$

$\frac{1}{m^{(j)}}$ $\frac{\partial}{\partial \theta_k^{(j)}} J(\theta^{(1)},\ldots,\theta^{(n_u)})$

Andrew Ng

已知用户对电影类型的喜好 θ, 对每一个用户，拟合线性回归模型 $\theta^T X$, 优化目标是用户对电影的评分，因此可以求得X,即电影的特征

Problem motivation

Movie	Alice (1) $\theta^{(1)}$	Bob (2) $\theta^{(2)}$	Carol (3) $\theta^{(3)}$	Dave (4) $\theta^{(4)}$	x_1 (romance)	x_2 (action)
Love at last $x^{(1)}$	5	5	0	0	1.0	0.0
Romance forever	5	?	?	0	?	?
Cute puppies of love	?	4	0	?	?	?
Nonstop car chases	0	0	5	4	?	?
Swords vs. karate	0	0	5	?	?	?

$x_0 = 1$

$x^{(1)} = \begin{bmatrix} 1 \\ 1.0 \\ 0.0 \end{bmatrix}$

$\theta^{(1)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix}, \theta^{(2)} = \begin{bmatrix} 0 \\ 5 \\ 0 \end{bmatrix}, \theta^{(3)} = \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix}, \theta^{(4)} = \begin{bmatrix} 0 \\ 0 \\ 5 \end{bmatrix}$

$\theta^{(j)}$

$(\theta^{(1)})^T x^{(1)} \approx 5$

$(\theta^{(2)})^T x^{(1)} \approx 5$

$(\theta^{(3)})^T x^{(1)} \approx 0$

$(\theta^{(4)})^T x^{(1)} \approx 0$

Andrew Ng

Optimization algorithm

Given $\theta^{(1)}, \ldots, \theta^{(n_u)}$, to learn $x^{(i)}$:

$$\min_{x^{(i)}} \frac{1}{2} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{k=1}^{n} (x_k^{(i)})^2$$

Given $\theta^{(1)}, \ldots, \theta^{(n_u)}$, to learn $x^{(1)}, \ldots, x^{(n_m)}$:

$$\min_{x^{(1)},\ldots,x^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2$$

Collaborative filtering algorithm

根据用户之间的相似度来推荐

Collaborative filtering optimization objective

Given $x^{(1)}, \ldots, x^{(n_m)}$, estimate $\theta^{(1)}, \ldots, \theta^{(n_u)}$:

$$\min_{\theta^{(1)},\ldots,\theta^{(n_u)}} \frac{1}{2} \sum_{j=1}^{n_u} \sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

Given $\theta^{(1)}, \ldots, \theta^{(n_u)}$, estimate $x^{(1)}, \ldots, x^{(n_m)}$:

$$\min_{x^{(1)},\ldots,x^{(n_m)}} \frac{1}{2} \sum_{i=1}^{n_m} \sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2$$

Minimizing $x^{(1)}, \ldots, x^{(n_m)}$ and $\theta^{(1)}, \ldots, \theta^{(n_u)}$ simultaneously:

$$J(x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)}) = \frac{1}{2} \sum_{(i,j):r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)})^2 + \frac{\lambda}{2} \sum_{i=1}^{n_m} \sum_{k=1}^{n} (x_k^{(i)})^2 + \frac{\lambda}{2} \sum_{j=1}^{n_u} \sum_{k=1}^{n} (\theta_k^{(j)})^2$$

$$\min_{\substack{x^{(1)},\ldots,x^{(n_m)} \\ \theta^{(1)},\ldots,\theta^{(n_u)}}} J(x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)})$$

$\theta \to x \to \theta \to x \to \ldots$

Andrew Ng

Collaborative filtering algorithm

1. Initialize $x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)}$ to small random values.
2. Minimize $J(x^{(1)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)})$ using gradient descent (or an advanced optimization algorithm). E.g. for every $j = 1, \ldots, n_u, i = 1, \ldots, n_m$:

$$x_k^{(i)} := x_k^{(i)} - \alpha \left(\sum_{j:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) \theta_k^{(j)} + \lambda x_k^{(i)} \right)$$

$$\theta_k^{(j)} := \theta_k^{(j)} - \alpha \left(\sum_{i:r(i,j)=1} ((\theta^{(j)})^T x^{(i)} - y^{(i,j)}) x_k^{(i)} + \lambda \theta_k^{(j)} \right)$$

3. For a user with parameters θ and a movie with (learned) features x, predict a star rating of $\theta^T x$.

Andrew Ng

Finding related movies

For each product i, we learn a feature vector $x^{(i)} \in \mathbb{R}^n$.

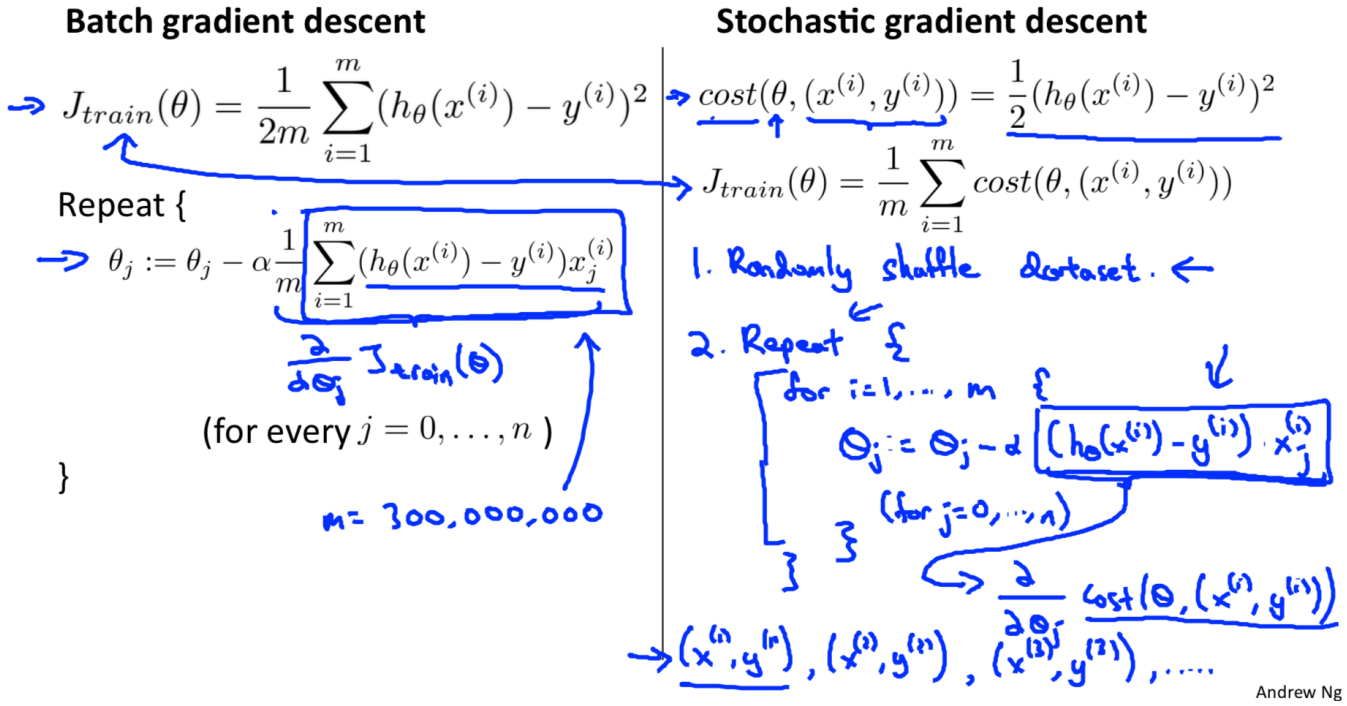
How to find movies j related to movie i?

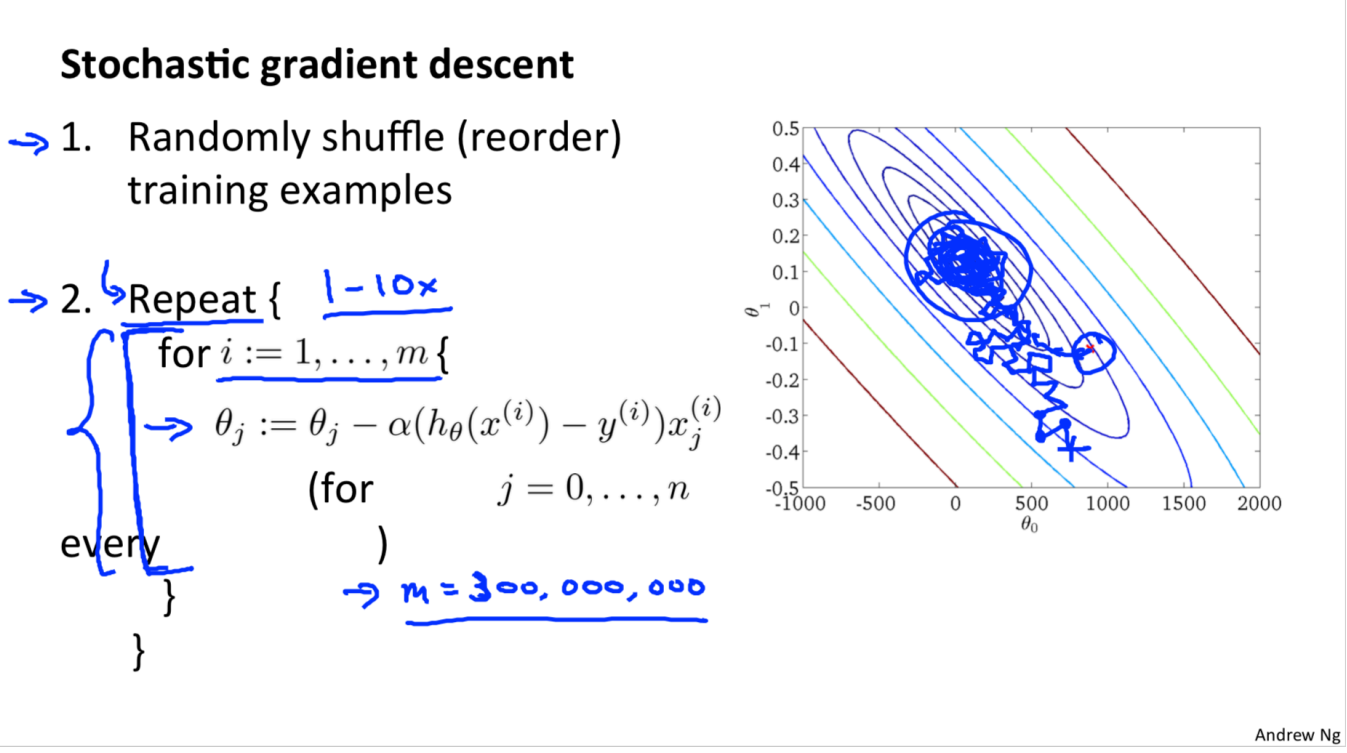
5 most similar movies to movie i:
Find the 5 movies j with the smallest $\|x^{(i)} - x^{(j)}\|$.

Large scale machine learning

Stochastic gradient descent

每次只用一个样本来计算梯度





Mini-batch gradient descent

每次利用 n 个样本来计算梯度

Mini-batch gradient descent

Say $b = 10, m = 1000$.

Repeat {

for $i = 1, 11, 21, 31, \ldots, 991$ {

$$\theta_j := \theta_j - \alpha \frac{1}{10} \sum_{k=i}^{i+9} (h_\theta(x^{(k)}) - y^{(k)}) x_j^{(k)}$$

(for every $j = 0, \ldots, n$)

}

}

→ b examples

→ 1 example

Vectorization

$m = 300,000,000$

$b = 10$

Online learning

抛弃固有数据集的概念，每次用新来的样本来更新梯度，然后抛弃这个样本。

Shipping service website where user comes, specifies origin and destination, you offer to ship their package for some asking price, and users sometimes choose to use your shipping service ($y = 1$), sometimes not ($y = 0$).

Features x capture properties of user, of origin/destination and asking price. We want to learn $p(y = 1|x; \theta)$ to optimize price.

price

logistic regression

Repeat forever {

Get (x, y) corresponding to user.

Update θ using (x, y).

$\theta_j := \theta_j - \alpha (h_\theta(x) - y) x_j$ $(j = 0, \ldots, n)$

}

→ Can adapt to changing user preference.

Andrew Ng