ML:Introduction

What is Machine Learning?

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

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

Example: playing checkers.

E = the experience of playing many games of checkers

T = the task of playing checkers.

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

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

supervised learning, OR

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. Here is a description on Math is Fun on Continuous and Discrete Data.

Example 1:

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

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

Example 2 :

(a) Regression - Given a picture of Male/Female, We have to predict his/her age on the basis of given picture.

(b) Classification - Given a picture of Male/Female, We have to predict Whether He/She is of High school, College, Graduate age. Another Example for Classification - Banks have to decide whether or not to give a loan to someone on the basis of his credit history.

Unsupervised Learning

Unsupervised learning, on the other hand, 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, i.e., there is no teacher to correct you.

Example:

Clustering: Take a collection of 1000 essays written on the US Economy, and find a way to automatically group these essays into a small number that are somehow similar or related by different variables, such as word frequency, sentence length, page count, and so on.

Non-clustering: The "Cocktail Party Algorithm", which can find structure in messy data (such as the identification of individual voices and music from a mesh of sounds at a cocktail party (https://en.wikipedia.org/wiki/Cocktail_party_effect)). Here is an answer on Quora to enhance your understanding. : https://www.quora.com/What-is-the-difference-between-supervised-and-unsupervised-learning-algorithms ?

ML:Linear Regression with One Variable

Model Representation

Recall that in *regression problems* , we are taking input variables and trying to fit the output onto a *continuous* expected result function.

Linear regression with one variable is also known as "univariate linear regression."

Univariate linear regression is used when you want to predict a **single output** value y from a **single input** value x. We're doing **supervised learning** here, so that means we already have an idea about what the input/output cause and effect should be.

The Hypothesis Function

Our hypothesis function has the general form:

$\hat{y} = h_\theta(x) = \theta_0 + \theta_1 x$

Note that this is like the equation of a straight line. We give to $h_\theta(x)$ values for θ_0 and θ_1 to get our estimated output $\hat{y}$. In other words, we are trying to create a function called h_θ that is trying to map our input data (the x's) to our output data (the y's).

Example:

Suppose we have the following set of training data:

input x	output y
0	4
1	7
2	7
3	8

Now we can make a random guess about our h_θ function: $\theta_0 = 2$ and $\theta_1 = 2$. The hypothesis function becomes $h_\theta(x) = 2 + 2x$.

So for input of 1 to our hypothesis, y will be 4. This is off by 3. Note that we will be trying out various values of θ_0 and θ_1 to try to find values which provide the best possible "fit" or the most representative "straight line" through the data points mapped on the x-y plane.

Cost Function

We can measure the accuracy of our hypothesis function by using a **cost function** . This takes an average (actually a fancier version of an average) of all the results of the hypothesis with inputs from x's compared to 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 $\left(\frac{1}{2m}\right)$ 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.

Now we are able to concretely measure the accuracy of our predictor function against the correct results we have so that we can predict new results we don't have.

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 straight line (defined by $h_\theta(x)$) which passes through this scattered set of data. 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. In the best case, 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.

ML: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 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). This can be kind of confusing; we are moving up to a higher level of abstraction. We are not graphing x and y itself, but the parameter range of our hypothesis function and the cost resulting from selecting particular set of parameters.

We put θ_0 on the x axis and θ_1 on the y axis, with the cost function on the vertical z axis. The points on our graph will be the result of the cost function using our hypothesis with those specific theta parameters.

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 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, and the size of each step is determined by the parameter α, which is called the learning rate.

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.

Intuitively, this could be thought of as:

repeat until convergence:

$\theta_j := \theta_j - \alpha$[Slope of tangent aka derivative in j dimension][Slope of tangent aka derivative in j dimension]

Gradient Descent for Linear Regression

When specifically applied to the case of linear regression, a new form of the gradient descent equation can be derived. We can substitute our actual cost function and our actual hypothesis function and modify the equation to (the derivation of the formulas are out of the scope of this course, but a really great one can be found here):

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 θ_1 and x_i, y_i are values of the given training set (data).

Note that we have separated out the two cases for θ_j into separate equations for θ_0 and θ_1; and that for θ_1 we are multiplying x_i at the end due to the derivative.

The 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.

Gradient Descent for Linear Regression: visual worked example

Some may find the following video (https://www.youtube.com/watch?v=WnqQrPNYz5Q) useful as it visualizes the improvement of the hypothesis as the error function reduces.

ML:Linear Algebra Review

Khan Academy has excellent Linear Algebra Tutorials (https://www.khanacademy.org/#linear-algebra)

Matrices and Vectors

Matrices are 2-dimensional arrays:

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

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:

$$\begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix}$$

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

Addition and Scalar Multiplication

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

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

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

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

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

Matrix-Vector Multiplication

We map the column of the vector onto each row of the matrix, multiplying each element and summing the result.

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

The result is a **vector** . The vector must be the **second** term of the multiplication. The number of **columns** of the matrix must equal the number of **rows** of the vector.

An **m x n matrix** multiplied by an **n x 1 vector** results in an **m x 1 vector** .

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.

Matrix Multiplication Properties

- Not commutative. $A*B \neq B*A$
- 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 should match the other matrix's **columns** . When multiplying the identity matrix before some other matrix ($I*A$), the square identity matrix should match the other matrix's **rows** .

Inverse and Transpose

The **inverse** of a matrix A is denoted A^{-1}. Multiplying by the inverse results in the identity matrix.

A non square matrix does not have an inverse matrix. We can compute inverses of matrices in octave with the pinv(A) function [1] and in matlab with the inv(A) function. Matrices that don't have an inverse are *singular* or *degenerate* .

The **transposition** of a matrix is like rotating the matrix 90 ° in clockwise direction and then reversing it. We can compute transposition of matrices in matlab with the transpose(A) function or A':

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

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

In other words:

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

ML:Linear Regression with Multiple Variables

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.

$$
\begin{aligned}
x_j^{(i)} &= \text{value of feature } j \text{ in the } i^{th} \text{ training example} \\
x^{(i)} &= \text{the column vector of all the feature inputs of the } i^{th} \text{ training example} \\
m &= \text{the number of training examples} \\
n &= \left|x^{(i)}\right| \text{; (the number of features)}
\end{aligned}
$$

Now define the multivariable form of the hypothesis function as follows, accommodating these multiple features:

$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$

In order to develop intuition about this function, we can think about θ_0 as the basic price of a house, θ_1 as the price per square meter, θ_2 as the price per floor, etc. x_1 will be the number of square meters in the house, x_2 the number of floors, etc.

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

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

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

Remark: Note that for convenience reasons in this course Mr. Ng assumes $x_0^{(i)} = 1$ for $(i \in 1, \dots, m)$

[**Note** : So that we can do matrix operations with theta and x, we will set $x_0^{(i)}$ = 1, for all values of i. This makes the two vectors 'theta' and $x_{(i)}$ match each other element-wise (that is, have the same number of elements: n+1).]

The training examples are stored in X row-wise, like such:

$$
X = \begin{bmatrix} x_0^{(1)} & x_1^{(1)} \\ x_0^{(2)} & x_1^{(2)} \\ x_0^{(3)} & x_1^{(3)} \end{bmatrix}, \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}
$$

You can calculate the hypothesis as a column vector of size (m x 1) with:

$h_\theta(X) = X\theta$

For the rest of these notes, and other lecture notes, X will represent a matrix of training examples $x_{(i)}$ stored row-wise.

Cost function

For the parameter vector θ (of type $\mathbb{R}^{n+1}$ or in $\mathbb{R}^{(n+1)\times 1}$, the cost function is:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

The vectorized version is:

$$J(\theta) = \frac{1}{2m}(X\theta - \vec{y})^T(X\theta - \vec{y})$$

Where $\vec{y}$ denotes the vector of all y values.

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:

$$\begin{aligned}
&\text{repeat until convergence: } \{ \\
&\theta_0 := \theta_0 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_0^{(i)} \\
&\theta_1 := \theta_1 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_1^{(i)} \\
&\theta_2 := \theta_2 - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_2^{(i)} \\
&\cdots \\
&\}
\end{aligned}$$

In other words:

$$\begin{aligned}
&\text{repeat until convergence: } \{ \\
&\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad \text{for j} := 0..\text{n} \\
&\}
\end{aligned}$$

Matrix Notation

The Gradient Descent rule can be expressed as:

$\theta := \theta - \alpha\nabla J(\theta)$

Where $\nabla J(\theta)$ is a column vector of the form:

$$\nabla J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix}$$

The j-th component of the gradient is the summation of the product of two terms:

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

Sometimes, the summation of the product of two terms can be expressed as the product of two vectors.

Here, $x_j^{(i)}$, for i = 1,...,m, represents the m elements of the j-th column, $\overrightarrow{x_j}$, of the training set X.

The other term $\left(h_\theta(x^{(i)}) - y^{(i)} \right)$ is the vector of the deviations between the predictions $h_\theta(x^{(i)})$ and the true values $y^{(i)}$. Re-writing $\frac{\partial J(\theta)}{\partial \theta_j}$, we have:

$$\begin{aligned} \frac{\partial J(\theta)}{\partial \theta_j} &= \frac{1}{m} \overrightarrow{x_j}^T (X\theta - \vec{y}) \\ \\ \nabla J(\theta) &= \frac{1}{m} X^T (X\theta - \vec{y}) \end{aligned}$$

Finally, the matrix notation (vectorized) of the Gradient Descent rule is:

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

Feature Normalization

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:

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

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

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

Example: x_i is housing prices with range of 100 to 2000, with a mean value of 1000. Then, $x_i := \frac{price - 1000}{1900}$.

Quiz question #1 on Feature Normalization (Week 2, Linear Regression with Multiple Variables)

Your answer should be rounded to exactly two decimal places. Use a '.' for the decimal point, not a ','. The tricky part of this question is figuring out which feature of which training example you are asked to normalize. Note that the mobile app doesn't allow entering a negative number (Jan 2016), so you will need to use a browser to submit this quiz if your solution requires a negative number.

Gradient Descent Tips

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

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

It has been proven that if learning rate α is sufficiently small, then J(θ) will decrease on every iteration. Andrew Ng recommends decreasing α by multiples of 3.

Features and Polynomial Regression

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

We can **combine** multiple features into one. For example, we can combine x_1 and x_2 into a new feature x_3 by taking $x_1 \cdot x_2$.

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 x_1, 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: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 \sqrt{x_1}$

Note that at 2:52 and through 6:22 in the "Features and Polynomial Regression" video, the curve that Prof Ng discusses about "doesn't ever come back down" is in reference to the hypothesis function that uses the sqrt() function (shown by the solid purple line), not the one that uses $size^2$ (shown with the dotted blue line). The quadratic form of the hypothesis function would have the shape shown with the blue dotted line if θ_2 was negative.

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

The "Normal Equation" is a method of finding the optimum theta **without iteration.**

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

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

Mathematical proof of the Normal equation requires knowledge of linear algebra and is fairly involved, so you do not need to worry about the details.

Proofs are available at these links for those who are interested:

https://en.wikipedia.org/wiki/Linear_least_squares_(mathematics)

http://eli.thegreenplace.net/2014/derivation-of-the-normal-equation-for-linear-regression

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 $X^T X$
Works well when n is large	Slow if n is very large

With the normal equation, computing the inversion has complexity $\mathcal{O}(n^3)$. 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.'

$X^T X$ may be **noninvertible** . The common causes are:

- 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.

ML:Octave Tutorial

Basic Operations

```
%% Change Octave prompt
PS1('>> ');
%% Change working directory in windows example:
cd 'c:/path/to/desired/directory name'
%% Note that it uses normal slashes and does not use escape characters
for the empty spaces.

%% elementary operations
5+6
3-2
5*8
1/2
2^6
1 == 2 % false
1 ~= 2 % true.  note, not "!="
1 && 0
1 || 0
xor(1,0)
```

```
%% variable assignment
a = 3; % semicolon suppresses output
b = 'hi';
c = 3>=1;

% Displaying them:
a = pi
disp(a)
disp(sprintf('2 decimals: %0.2f', a))
disp(sprintf('6 decimals: %0.6f', a))
format long
a
format short
a


%%  vectors and matrices
A = [1 2; 3 4; 5 6]

v = [1 2 3]
v = [1; 2; 3]
v = 1:0.1:2   % from 1 to 2, with stepsize of 0.1. Useful for plot axes
v = 1:6       % from 1 to 6, assumes stepsize of 1 (row vector)

C = 2*ones(2,3) % same as C = [2 2 2; 2 2 2]
w = ones(1,3)   % 1x3 vector of ones
w = zeros(1,3)
w = rand(1,3) % drawn from a uniform distribution
w = randn(1,3)% drawn from a normal distribution (mean=0, var=1)
w = -6 + sqrt(10)*(randn(1,10000));  % (mean = -6, var = 10) - note:
add the semicolon
hist(w)    % plot histogram using 10 bins (default)
hist(w,50) % plot histogram using 50 bins
% note: if hist() crashes, try "graphics_toolkit('gnu_plot')"

I = eye(4)   % 4x4 identity matrix

% help function
help eye
help rand
help help
```

Moving Data Around

Data files used in this section : featuresX.dat , priceY.dat

```
%% dimensions
sz = size(A) % 1x2 matrix: [(number of rows) (number of columns)]
size(A,1) % number of rows
size(A,2) % number of cols
length(v) % size of longest dimension


%% loading data
pwd   % show current directory (current path)
cd 'C:\Users\ang\Octave files'  % change directory
ls    % list files in current directory
load q1y.dat   % alternatively, load('q1y.dat')
load q1x.dat
who   % list variables in workspace
whos  % list variables in workspace (detailed view)
clear q1y      % clear command without any args clears all vars
v = q1x(1:10); % first 10 elements of q1x (counts down the columns)
```

```
save hello.mat v;  % save variable v into file hello.mat
save hello.txt v -ascii; % save as ascii
% fopen, fread, fprintf, fscanf also work  [[not needed in class]]

%% indexing
A(3,2)  % indexing is (row,col)
A(2,:)  % get the 2nd row.
        % ":" means every element along that dimension
A(:,2)  % get the 2nd col
A([1 3],:) % print all  the elements of rows 1 and 3

A(:,2) = [10; 11; 12]     % change second column
A = [A, [100; 101; 102]]; % append column vec
A(:) % Select all elements as a column vector.

% Putting data together
A = [1 2; 3 4; 5 6]
B = [11 12; 13 14; 15 16] % same dims as A
C = [A B]  % concatenating A and B matrices side by side
C = [A, B] % concatenating A and B matrices side by side
C = [A; B] % Concatenating A and B top and bottom
```

Computing on Data

```
%% initialize variables
A = [1 2;3 4;5 6]
B = [11 12;13 14;15 16]
C = [1 1;2 2]
v = [1;2;3]

%% matrix operations
A * C  % matrix multiplication
A .* B % element-wise multiplication
% A .* C  or A * B gives error - wrong dimensions
A .^ 2 % element-wise square of each element in A
1./v   % element-wise reciprocal
log(v)  % functions like this operate element-wise on vecs or matrices
exp(v)
abs(v)

-v  % -1*v

v + ones(length(v), 1)
% v + 1  % same

A'  % matrix transpose

%% misc useful functions

% max  (or min)
a = [1 15 2 0.5]
val = max(a)
[val,ind] = max(a) % val -  maximum element of the vector a and index -
index value where maximum occur
val = max(A) % if A is matrix, returns max from each column

% compare values in a matrix & find
a < 3 % checks which values in a are less than 3
find(a < 3) % gives location of elements less than 3
A = magic(3) % generates a magic matrix - not much used in ML
algorithms
[r,c] = find(A>=7)  % row, column indices for values matching
comparison

% sum, prod
sum(a)
```

```
prod(a)
floor(a) % or ceil(a)
max(rand(3),rand(3))
max(A,[],1) -  maximum along columns(defaults to columns - max(A,[]))
max(A,[],2) - maximum along rows
A = magic(9)
sum(A,1)
sum(A,2)
sum(sum( A .* eye(9) ))
sum(sum( A .* flipud(eye(9)) ))


% Matrix inverse (pseudo-inverse)
pinv(A)        % inv(A'*A)*A'
```

Plotting Data

```
%% plotting
t = [0:0.01:0.98];
y1 = sin(2*pi*4*t);
plot(t,y1);
y2 = cos(2*pi*4*t);
hold on;  % "hold off" to turn off
plot(t,y2,'r');
xlabel('time');
ylabel('value');
legend('sin','cos');
title('my plot');
print -dpng 'myPlot.png'
close;           % or,  "close all" to close all figs
figure(1); plot(t, y1);
figure(2); plot(t, y2);
figure(2), clf;  % can specify the figure number
subplot(1,2,1);  % Divide plot into 1x2 grid, access 1st element
plot(t,y1);
subplot(1,2,2);  % Divide plot into 1x2 grid, access 2nd element
plot(t,y2);
axis([0.5 1 -1 1]);  % change axis scale

%% display a matrix (or image)
figure;
imagesc(magic(15)), colorbar, colormap gray;
% comma-chaining function calls.
a=1,b=2,c=3
a=1;b=2;c=3;
```

Control statements: for, while, if statements

```
v = zeros(10,1);
for i=1:10,
    v(i) = 2^i;
end;
% Can also use "break" and "continue" inside for and while loops to
control execution.

i = 1;
while i <= 5,
  v(i) = 100;
  i = i+1;
```

```
end

i = 1;
while true,
  v(i) = 999;
  i = i+1;
  if i == 6,
    break;
  end;
end

if v(1)==1,
  disp('The value is one!');
elseif v(1)==2,
  disp('The value is two!');
else
  disp('The value is not one or two!');
end
```

Functions

To create a function, type the function code in a text editor (e.g. gedit or notepad), and save the file as "functionName.m"

Example function:

```
function y = squareThisNumber(x)

y = x^2;
```

To call the function in Octave, do either:

1) Navigate to the directory of the functionName.m file and call the function:

```
    % Navigate to directory:
    cd /path/to/function

    % Call the function:
    functionName(args)
```

2) Add the directory of the function to the load path and save it: **You should not use addpath/savepath for any of the assignments in this course. Instead use 'cd' to change the current working directory. Watch the video on submitting assignments in week 2 for instructions.**

```
    % To add the path for the current session of Octave:
    addpath('/path/to/function/')

    % To remember the path for future sessions of Octave, after
executing addpath above, also do:
    savepath
```

Octave's functions can return more than one value:

```
    function [y1, y2] = squareandCubeThisNo(x)
    y1 = x^2
    y2 = x^3
```

Call the above function this way:

```
    [a,b] = squareandCubeThisNo(x)
```

Vectorization

Vectorization is the process of taking code that relies on **loops** and converting it into **matrix operations** . It is more efficient, more elegant, and more concise.

As an example, let's compute our prediction from a hypothesis. Theta is the vector of fields for the hypothesis and x is a vector of variables.

With loops:

```
prediction = 0.0;
for j = 1:n+1,
  prediction += theta(j) * x(j);
end;
```

With vectorization:

```
prediction = theta' * x;
```

If you recall the definition multiplying vectors, you'll see that this one operation does the element-wise multiplication and overall sum in a very concise notation.

Working on and Submitting Programming Exercises

1. Download and extract the assignment's zip file.
2. Edit the proper file 'a.m', where a is the name of the exercise you're working on.
3. Run octave and cd to the assignment's extracted directory
4. Run the 'submit' function and enter the assignment number, your email, and a password (found on the top of the "Programming Exercises" page on coursera)

Video Lecture Table of Contents

Basic Operations

```
0:00    Introduction
3:15    Elementary and Logical operations
5:12    Variables
7:38    Matrices
8:30    Vectors
11:53   Histograms
```

```
12:44   Identity matrices
13:14   Help command
```

Moving Data Around

```
0:24    The size command
1:39    The length command
2:18    File system commands
2:25    File handling
4:50    Who, whos, and clear
6:50    Saving data
8:35    Manipulating data
12:10   Unrolling a matrix
12:35   Examples
14:50   Summary
```

Computing on Data

```
0:00    Matrix operations
0:57    Element-wise operations
4:28    Min and max
5:10    Element-wise comparisons
5:43    The find command
6:00    Various commands and operations
```

Plotting data

```
0:00    Introduction
0:54    Basic plotting
2:04    Superimposing plots and colors
3:15    Saving a plot to an image
4:19    Clearing a plot and multiple figures
4:59    Subplots
6:15    The axis command
6:39    Color square plots
8:35    Wrapping up
```

Control statements

```
0:10    For loops
1:33    While loops
3:35    If statements
4:54    Functions
6:15    Search paths
7:40    Multiple return values
8:59    Cost function example (machine learning)
12:24   Summary
```

Vectorization

```
0:00    Why vectorize?
1:30    Example
4:22    C++ example
5:40    Vectorization applied to gradient descent
9:45    Python
```

ML:Logistic Regression

Now we are switching from regression problems to **classification problems** . Don't be confused by the name "Logistic Regression"; it is named that way for historical reasons and is actually an approach to classification problems, not regression problems.

Binary Classification

Instead of our output vector y being a continuous range of values, it will only be 0 or 1.

y∈{0,1}

Where 0 is usually taken as the "negative class" and 1 as the "positive class", but you are free to assign any representation to it.

We're only doing two classes for now, called a "Binary Classification Problem."

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. This method doesn't work well because classification is not actually a linear function.

Hypothesis Representation

Our hypothesis should satisfy:

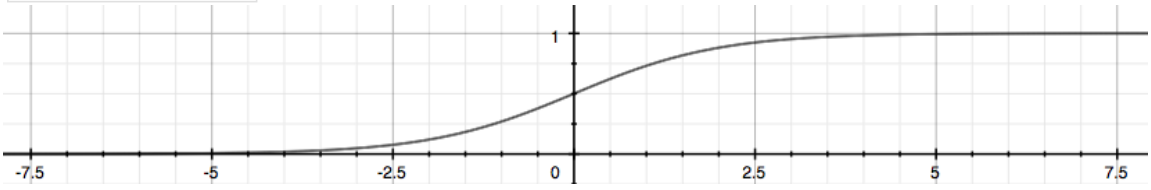
$0 \leq h_\theta(x) \leq 1$

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 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. Try playing with interactive plot of sigmoid function : (https://www.desmos.com/calculator/bgontvxotm).

We start with our old hypothesis (linear regression), except that we want to restrict the range to 0 and 1. This is accomplished by plugging $\theta^T x$ into the Logistic Function.

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

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

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%).

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.-

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

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

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

From these statements we can now say:

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

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

Example :

$$\theta = \begin{bmatrix} 5 \\ -1 \\ 0 \end{bmatrix}$$
$$y = 1\ if\ 5 + (-1)x_1 + 0x_2 \geq 0$$
$$5 - x_1 \geq 0$$
$$-x_1 \geq -5$$
$$x_1 \leq 5$$

In this case, our decision boundary is a straight vertical line placed on the graph where $x_1 = 5$, and everything to the left of that denotes y = 1, while everything to the right denotes y = 0.

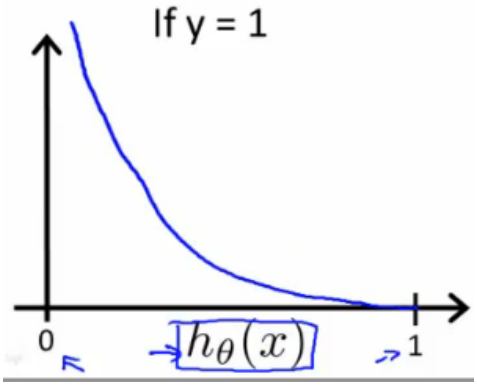
Again, the input to the sigmoid function g(z) (e.g. $\theta^T X$) doesn't need to be linear, and could be a function that describes a circle (e.g. $z = \theta_0 + \theta_1 x_1^2 + \theta_2 x_2^2$) or any shape to fit our data.

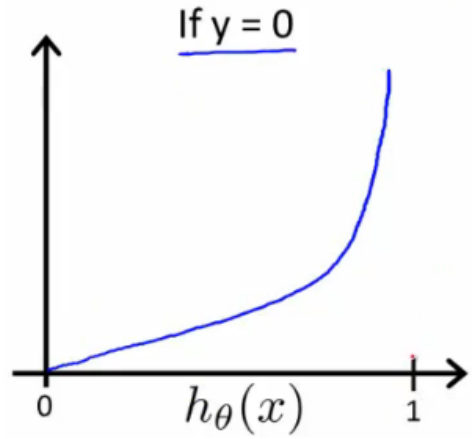
Cost Function

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:

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





The more our hypothesis is off from y, the larger the cost function output. If our hypothesis is equal to y, then our cost is 0:

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

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:

$$\text{Cost}(h_\theta(x), y) = -y\ \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x))$$

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 $-y \log(h_\theta(x))$ will be zero and will not affect the result.

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

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

A vectorized implementation is:

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

Gradient Descent

Remember that the general form of gradient descent is:

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

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

$$
\begin{aligned}
&Repeat\ \{ \\
&\theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \\
&\}
\end{aligned}
$$

Notice that this algorithm is identical to the one we used in linear regression. We still have to simultaneously update all values in theta.

A vectorized implementation is:

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

Partial derivative of J(θ)

First calculate derivative of sigmoid function (it will be useful while finding partial derivative of J(θ)):

$$
\begin{aligned}
\sigma(x)' &= \left(\frac{1}{1+e^{-x}}\right)' = \frac{-(1+e^{-x})'}{(1+e^{-x})^2} = \frac{-1' - (e^{-x})'}{(1+e^{-x})^2} = \frac{0 - (-x)'(e^{-x})}{(1+e^{-x})^2} = \frac{-(-1)(e^{-x})}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{e^{-x}}{1+e^{-x}}\right) = \sigma(x)\left(\frac{+1-1+e^{-x}}{1+e^{-x}}\right) = \sigma(x)\left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right) = \sigma(x)(1-\sigma(x))
\end{aligned}
$$

Now we are ready to find out resulting partial derivative:

$$
\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{-1}{m} \sum_{i=1}^{m} \left[y^{(i)} log(h_\theta(x^{(i)})) + (1-y^{(i)}) log(1-h_\theta(x^{(i)})) \right] \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \frac{\partial}{\partial \theta_j} log(h_\theta(x^{(i)})) + (1-y^{(i)}) \frac{\partial}{\partial \theta_j} log(1-h_\theta(x^{(i)})) \right] \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left[\frac{y^{(i)} \frac{\partial}{\partial \theta_j} h_\theta(x^{(i)})}{h_\theta(x^{(i)})} + \frac{(1-y^{(i)}) \frac{\partial}{\partial \theta_j} (1-h_\theta(x^{(i)}))}{1-h_\theta(x^{(i)})} \right] \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left[\frac{y^{(i)} \frac{\partial}{\partial \theta_j} \sigma(\theta^T x^{(i)})}{h_\theta(x^{(i)})} + \frac{(1-y^{(i)}) \frac{\partial}{\partial \theta_j} (1-\sigma(\theta^T x^{(i)}))}{1-h_\theta(x^{(i)})} \right] \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left[\frac{y^{(i)} \sigma(\theta^T x^{(i)})(1-\sigma(\theta^T x^{(i)})) \frac{\partial}{\partial \theta_j} \theta^T x^{(i)}}{h_\theta(x^{(i)})} + \frac{-(1-y^{(i)}) \sigma(\theta^T x^{(i)})(1-\sigma(\theta^T x^{(i)})) \frac{\partial}{\partial \theta_j} \theta^T x^{(i)}}{1-h_\theta(x^{(i)})} \right] \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left[\frac{y^{(i)} h_\theta(x^{(i)})(1-h_\theta(x^{(i)})) \frac{\partial}{\partial \theta_j} \theta^T x^{(i)}}{h_\theta(x^{(i)})} - \frac{(1-y^{(i)}) h_\theta(x^{(i)})(1-h_\theta(x^{(i)})) \frac{\partial}{\partial \theta_j} \theta^T x^{(i)}}{1-h_\theta(x^{(i)})} \right] \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)}(1-h_\theta(x^{(i)})) x_j^{(i)} - (1-y^{(i)}) h_\theta(x^{(i)}) x_j^{(i)} \right] \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)}(1-h_\theta(x^{(i)})) - (1-y^{(i)}) h_\theta(x^{(i)}) \right] x_j^{(i)} \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} - y^{(i)} h_\theta(x^{(i)}) - h_\theta(x^{(i)}) + y^{(i)} h_\theta(x^{(i)}) \right] x_j^{(i)} \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} - h_\theta(x^{(i)}) \right] x_j^{(i)} \\
&= \frac{1}{m} \sum_{i=1}^{m} \left[h_\theta(x^{(i)}) - y^{(i)} \right] x_j^{(i)}
\end{aligned}
$$

The vectorized version;

$\nabla J(\theta) = \frac{1}{m} \cdot X^T \cdot (g(X \cdot \theta) - \vec{y})$

Advanced Optimization

"Conjugate gradient", "BFGS", and "L-BFGS" are more sophisticated, faster ways to optimize θ that can be used instead of gradient descent. A. Ng suggests not to write these more sophisticated algorithms yourself (unless you are an expert in numerical computing) but use the libraries instead, as they're already tested and highly optimized. Octave provides them.

We first need to provide a function that evaluates the following two functions for a given input value θ:

$J(\theta)$
$\frac{\partial}{\partial \theta_j} J(\theta)$

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

```
function [jVal, gradient] = costFunction(theta)
  jVal = [...code to compute J(theta)...];
  gradient = [...code to compute derivative of J(theta)...];
end
```

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

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

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

Multiclass Classification: One-vs-all

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

In this case 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.

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

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.

ML:Regularization

The Problem of Overfitting

Regularization is designed to address the problem of overfitting.

High bias or underfitting 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. eg. if we take
$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$ then we are making an initial assumption that a linear model will fit the training data well and will be able to generalize but that may not be the case.

At the other extreme, overfitting or high variance is caused by a hypothesis function that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot

of unnecessary curves and angles unrelated to the data.

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

1) Reduce the number of features:

a) Manually select which features to keep.

b) Use a model selection algorithm (studied later in the course).

2) Regularization

Keep all the features, but reduce the parameters θ_j.

Regularization works well when we have a lot of slightly useful features.

Cost Function

If we have overfitting from our hypothesis function, we can reduce the weight that some of the terms in our function carry by increasing their cost.

Say we wanted to make the following function more quadratic:

$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

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.

We could also regularize all of our theta parameters in a single summation:

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

The λ, or lambda, is the **regularization parameter** . It determines how much the costs of our theta parameters are inflated. You can visualize the effect of regularization in this interactive plot : https://www.desmos.com/calculator/1hexc8ntqp

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.

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.

$$\begin{aligned}
&\text{Repeat } \{ \\
&\quad \theta_0 := \theta_0 - \alpha \ \frac{1}{m} \ \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)} \\
&\quad \theta_j := \theta_j - \alpha \ \left[\left(\frac{1}{m} \ \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \right] \qquad\qquad j \in \{1, 2 ... n\} \\
&\}
\end{aligned}$$

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:

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

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

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 λ.

Recall that if m ≤ n, then $X^T X$ is non-invertible. However, when we add the term λ·L, then $X^T X$ + λ·L becomes invertible.

Regularized Logistic Regression

We can regularize logistic regression in a similar way that we regularize linear regression. Let's start with the cost function.

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:

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

Note Well: The second sum, $\sum_{j=1}^{n}\theta_j^2$ **means to explicitly exclude** the bias term, θ_0. I.e. the θ vector is indexed from 0 to n (holding n+1 values, θ_0 through θ_n), and this sum explicitly skips θ_0, by running from 1 to n, skipping 0.

Gradient Descent

Just like with linear regression, we will want to **separately** update θ_0 and the rest of the parameters because we do not want to regularize θ_0.

$$\begin{aligned}
&\text{Repeat } \{ \\
&\quad \theta_0 := \theta_0 - \alpha\ \frac{1}{m}\ \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)} \\
&\quad \theta_j := \theta_j - \alpha\ \left[\left(\frac{1}{m}\ \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j\right] \qquad\qquad j \in \{1, 2 \ldots n\} \\
&\}
\end{aligned}$$

This is identical to the gradient descent function presented for linear regression.

Initial Ones Feature Vector

Constant Feature

As it turns out it is crucial to add a constant feature to your pool of features before starting any training of your machine. Normally that feature is just a set of ones for all your training examples.

Concretely, if X is your feature matrix then X_0 is a vector with ones.

Below are some insights to explain the reason for this constant feature. The first part draws some analogies from electrical engineering concept, the second looks at understanding the ones vector by using a simple machine learning example.

Electrical Engineering

From electrical engineering, in particular signal processing, this can be explained as DC and AC.

The initial feature vector X without the constant term captures the dynamics of your model. That means those features particularly record changes in your output y - in other words changing some feature X_i where $i \neq 0$ will have a change on the output y. AC is normally made out of many components or harmonics; hence we also have many features (yet we have one DC term).

The constant feature represents the DC component. In control engineering this can also be the steady state.

Interestingly removing the DC term is easily done by differentiating your signal - or simply taking a difference between consecutive points of a discrete signal (it should be noted that at this point the analogy is implying time-based signals - so this will also make sense for machine learning application with a time basis - e.g. forecasting stock exchange trends).

Another interesting note: if you were to play and AC+DC signal as well as an AC only signal where both AC components are the same then they would sound exactly the same. That is because we only hear changes in signals and Δ(AC+DC)=Δ(AC).

Housing price example

Suppose you design a machine which predicts the price of a house based on some features. In this case what does the ones vector help with?

Let's assume a simple model which has features that are directly proportional to the expected price i.e. if feature Xi increases so the expected price y will also increase. So as an example we could have two features: namely the size of the house in [m2], and the number of rooms.

When you train your machine you will start by prepending a ones vector X_0. You may then find after training that the weight for your initial feature of ones is some value θ0. As it turns, when applying your hypothesis function $h_\theta(X)$ - in the case of the initial feature you will just be multiplying by a constant (most probably θ0 if you not applying any other functions such as sigmoids). This constant (let's say it's θ_0 for argument's sake) is the DC term. It is a constant that doesn't change.

But what does it mean for this example? Well, let's suppose that someone knows that you have a working model for housing prices. It turns out that for this example, if they ask you how much money they can expect if they sell the house you can say that they need at least θ0 dollars (or rands) before you even use your learning machine. As with the above analogy, your constant θ0 is somewhat of a steady state where all your inputs are zeros. Concretely, this is the price of a house with no rooms which takes up no space.

However this explanation has some holes because if you have some features which decrease the price e.g. age, then the DC term may not be an absolute minimum of the price. This is because the age may make the price go even lower.

Theoretically if you were to train a machine without a ones vector $f_{AC}(X)$, it's output may not match the output of a machine which had a ones vector $f_{DC}(X)$. However, $f_{AC}(X)$ may have exactly the same trend as $f_{DC}(X)$ i.e. if you were to plot both machine's output you would find that they may look exactly the same except that it seems one output has just been shifted (by a constant). With reference to the housing price problem: suppose you make predictions on two houses $house_A$ and $house_B$ using both machines. It turns out while the outputs from the two machines would different, the difference between houseA and houseB's predictions according to both machines could be exactly the same. Realistically, that means a machine trained without the ones vector f_AC could actually be very useful if you have just one benchmark point. This is because you can find out the missing constant by simply taking a difference between the machine's prediction an actual price - then when making predictions you simply add that constant to what even output you get. That is: if $house_{benchmark}$ is your benchmark then the DC component is simply $price(house_{benchmark}) - f_{AC}(features(house_{benchmark}))$

A more simple and crude way of putting it is that the DC component of your model represents the inherent bias of the model. The other features then cause tension in order to move away from that bias position.

Kholofelo Moyaba

A simpler approach

A "bias" feature is simply a way to move the "best fit" learned vector to better fit the data. For example, consider a learning problem with a single feature X_1. The formula without the X_0 feature is just $theta_1 * X_1 = y$. This is graphed as a line that always passes through the origin, with slope y/theta. The x_0 term allows the line to pass through a different point on the y axis. This will almost always give a better fit. Not all best fit lines go through the origin (0,0) right?

Joe Cotton

ML:Neural Networks: Representation

Non-linear Hypotheses

Performing linear regression with a complex set of data with many features is very unwieldy. Say you wanted to create a hypothesis from three (3) features that included all the quadratic terms:

$$g(\theta_0 + \theta_1 x_1^2 + \theta_2 x_1 x_2 + \theta_3 x_1 x_3 + \theta_4 x_2^2 + \theta_5 x_2 x_3 + \theta_6 x_3^2)$$

That gives us 6 features. The exact way to calculate how many features for all polynomial terms is the combination function with repetition: http://www.mathsisfun.com/combinatorics/combinations-permutations.html $\frac{(n+r-1)!}{r!(n-1)!}$. In this case we are taking all two-element combinations of three features: $\frac{(3+2-1)!}{(2!\cdot(3-1)!)} = \frac{4!}{4} = 6$. (**Note** : you do not have to know these formulas, I just found it helpful for understanding).

For 100 features, if we wanted to make them quadratic we would get $\frac{(100+2-1)!}{(2\cdot(100-1)!)} = 5050$ resulting new features.

We can approximate the growth of the number of new features we get with all quadratic terms with $\mathcal{O}(n^2/2)$. And if you wanted to include all cubic terms in your hypothesis, the features would grow asymptotically at $\mathcal{O}(n^3)$. These are very steep growths, so as the number of our features increase, the number of quadratic or cubic features increase very rapidly and becomes quickly impractical.

Example: let our training set be a collection of 50 x 50 pixel black-and-white photographs, and our goal will be to classify which ones are photos of cars. Our feature set size is then n = 2500 if we compare every pair of pixels.

Now let's say we need to make a quadratic hypothesis function. With quadratic features, our growth is $\mathcal{O}(n^2/2)$. So our total features will be about $2500^2/2 = 3125000$, which is very impractical.

Neural networks offers an alternate way to perform machine learning when we have complex hypotheses with many features.

Neurons and the Brain

Neural networks are limited imitations of how our own brains work. They've had a big recent resurgence because of advances in computer hardware.

There is evidence that the brain uses only one "learning algorithm" for all its different functions. Scientists have tried cutting (in an animal brain) the connection between the ears and the auditory cortex and rewiring the optical nerve with the auditory cortex to find that the auditory cortex literally learns to see.

This principle is called "neuroplasticity" and has many examples and experimental evidence.

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 input (**dendrites**) as electrical input (called "spikes") that are channeled to outputs (**axons**).

In our model, our dendrites are like the input features $x_1 \cdots x_n$, and the output is the result of our hypothesis function:

In this model our x0 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: $\frac{1}{1+e^{-\theta^T x}}$. In neural networks however we sometimes call it a sigmoid (logistic) **activation** function.

Our "theta" parameters are sometimes instead called "weights" in the neural networks model.

Visually, a simplistic representation looks like:

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \rightarrow [\quad] \rightarrow h_\theta(x)$$

Our input nodes (layer 1) go into another node (layer 2), and are output as the hypothesis function.

The first layer is called the "input layer" and the final layer the "output layer," which gives the final value computed on the hypothesis.

We can have intermediate layers of nodes between the input and output layers called the "hidden layer."

We label these intermediate or "hidden" layer nodes $a_0^2 \cdots a_n^2$ and call them "activation units."

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

If we had one hidden layer, it would look visually something 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.

Example: 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 $s_{j+1} \times (s_j + 1) = 4 \times 3$.

Model Representation II

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

$$
\begin{aligned}
a_1^{(2)} &= g(z_1^{(2)}) \\
a_2^{(2)} &= g(z_2^{(2)}) \\
a_3^{(2)} &= g(z_3^{(2)})
\end{aligned}
$$

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

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

The vector representation of x and z^j is:

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

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

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

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

Now we can get a vector of our activation nodes for layer j as follows:

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

Where our function g can be applied element-wise to our vector $z^{(j)}$.

We can then add a bias unit (equal to 1) to layer j after we have computed $a^{(j)}$. This will be element $a_0^{(j)}$ and will be equal to 1.

To compute our final hypothesis, let's first compute another z vector:

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

We get this final z vector by multiplying the next theta matrix after $\Theta^{(j-1)}$ with the values of all the activation nodes we just got.

This last theta matrix $\Theta^{(j)}$ will have only **one row** so that our result is a single number.

We then get our final result with:

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

Notice that in this **last step** , between layer j and layer j+1, we are doing **exactly the same thing** as we did in logistic regression.

Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypotheses.

Examples and Intuitions I

A simple example of applying neural networks is by predicting x_1 AND x_2, which is the logical 'and' operator and is only true if both x_1 and x_2 are 1.

The graph of our functions will look like:

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

Remember that x_0 is our bias variable and is always 1.

Let's set our first theta matrix as:

$$\Theta^{(1)} = [-30 \quad 20 \quad 20]$$

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

$$h_\Theta(x) = g(-30 + 20x_1 + 20x_2)$$

$$\begin{array}{l} x_1 = 0 \ \ and \ \ x_2 = 0 \ \ then \ \ g(-30) \approx 0 \\ x_1 = 0 \ \ and \ \ x_2 = 1 \ \ then \ \ g(-10) \approx 0 \\ x_1 = 1 \ \ and \ \ x_2 = 0 \ \ then \ \ g(-10) \approx 0 \\ x_1 = 1 \ \ and \ \ x_2 = 1 \ \ then \ \ g(10) \approx 1 \end{array}$$

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.

Examples and Intuitions II

The $\Theta^(1)$ matrices for AND, NOR, and OR are:

$$\begin{array}{l} AND: \\ \quad \Theta^{(1)} = [-30 \quad 20 \quad 20] \\ NOR: \\ \quad \Theta^{(1)} = [10 \quad -20 \quad -20] \\ OR: \\ \quad \Theta^{(1)} = [-10 \quad 20 \quad 20] \end{array}$$

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

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

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

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

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

$$\Theta^{(2)} = [-10 \quad 20 \quad 20]$$

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

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

And there we have the XNOR operator using two hidden layers!

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 final resulting classes:

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

Our final layer of nodes, when multiplied by its theta matrix, will result in another vector, on which we will apply the g() logistic function to get a vector of hypothesis values.

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 $h_\Theta(x)_3$.

We can define our set of resulting classes as y:

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

Our final value of our hypothesis for a set of inputs will be one of the elements in y.

ML:Neural Networks: Learning

Cost Function

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

a) L= total number of layers in the network

b) s_l = number of units (not counting bias unit) in layer l

c) 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 k^{th} 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:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[y_k^{(i)}\log((h_\Theta(x^{(i)}))_k) + (1-y_k^{(i)})\log(1-(h_\Theta(x^{(i)}))_k)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(\Theta_{j,i}^{(l)})^2$$

We have added a few nested summations to account for our multiple output nodes. In the first part of the equation, between 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; and
- the triple sum simply adds up the squares of all the individual Θs in the entire network.
- the i in the triple sum does **not** refer to training example i

Backpropagation Algorithm

"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression.

Our goal is to compute:

$\min_\Theta J(\Theta)$

That is, we want to minimize our cost function J using an optimal set of parameters in theta.

In this section we'll look at the equations we use to compute the partial derivative of J(Θ):

$$\frac{\partial}{\partial\Theta_{i,j}^{(l)}}J(\Theta)$$

In back propagation we're going to compute for every node:

$\delta_j^{(l)}$ = "error" of node j in layer l

Recall that $a_j^{(l)}$ is activation node j in layer l.

For the **last layer** , we can compute the vector of delta values with:

$\delta^{(L)} = a^{(L)} - y$

Where L is our total number of layers and $a^{(L)}$ is the vector of outputs of the activation units for the last layer. So our "error values" for the last layer are simply the differences of our actual results in the last layer and the correct outputs in y.

To get the delta values of the layers before the last layer, we can use an equation that steps us back from right to left:

$$\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) \, .* \, g'(z^{(l)})$$

The delta values of layer l are calculated by multiplying the delta values in the next layer with the theta matrix of layer l. We then element-wise multiply that with a function called g', or g-prime, which is the derivative of the activation function g evaluated with the input values given by z(l).

The g-prime derivative terms can also be written out as:

$$g'(u) = g(u) \, .* \, (1 - g(u))$$

The full back propagation equation for the inner nodes is then:

$$\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) \, .* \, a^{(l)} \, .* \, (1 - a^{(l)})$$

A. Ng states that the derivation and proofs are complicated and involved, but you can still implement the above equations to do back propagation without knowing the details.

We can compute our partial derivative terms by multiplying our activation values and our error values for each training example t:

$$\frac{\partial J(\Theta)}{\partial \Theta_{i,j}^{(l)}} = \frac{1}{m} \sum_{t=1}^{m} a_j^{(t)(l)} \delta_i^{(t)(l+1)}$$

This however ignores regularization, which we'll deal with later.

Note: δ^{l+1} and a^{l+1} are vectors with s_{l+1} elements. Similarly, $a^{(l)}$ is a vector with s_l elements. Multiplying them produces a matrix that is s_{l+1} by s_l which is the same dimension as $\Theta^{(l)}$. That is, the process produces a gradient term for every element in $\Theta^{(l)}$. (Actually, $\Theta^{(l)}$ has s_l + 1 column, so the dimensionality is not exactly the same).

We can now take all these equations and put them together into a backpropagation algorithm:

Back propagation Algorithm

Given training set $\{(x^{(1)}, y^{(1)}) \cdots (x^{(m)}, y^{(m)})\}$

- Set $\Delta_{i,j}^{(l)} := 0$ for all (l,i,j)

For training example t =1 to m:

- Set $a^{(1)} := x^{(t)}$
- Perform forward propagation to compute $a^{(l)}$ for l=2,3,…,L
- Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$
- Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) \, .* \, a^{(l)} \, .* \, (1 - a^{(l)})$
- $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ or with vectorization, $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$
- $D_{i,j}^{(l)} := \frac{1}{m} \left(\Delta_{i,j}^{(l)} + \lambda \Theta_{i,j}^{(l)} \right)$ If j≠0 NOTE: Typo in lecture slide omits outside parentheses. This version is correct.
- $D_{i,j}^{(l)} := \frac{1}{m} \Delta_{i,j}^{(l)}$ If j=0

The capital-delta matrix is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative.

The actual proof is quite involved, but, the $D_{i,j}^{(l)}$ terms are the partial derivatives and the results we are looking for:

$$D_{i,j}^{(l)} = \frac{\partial J(\Theta)}{\partial \Theta_{i,j}^{(l)}}.$$

Backpropagation Intuition

The cost function is:

$$J(\theta) = -\frac{1}{m} \sum_{t=1}^{m} \sum_{k=1}^{K} \left[y_k^{(t)} \ \log(h_\theta(x^{(t)}))_k + (1 - y_k^{(t)}) \ \log(1 - h_\theta(x^{(t)})_k) \right] + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l+1} (\theta_{j,i}^{(l)})^2$$

If we consider simple non-multiclass classification (k = 1) and disregard regularization, the cost is computed with:

$$cost(t) = y^{(t)} \log(h_\theta(x^{(t)})) + (1 - y^{(t)}) \log(1 - h_\theta(x^{(t)}))$$

More intuitively you can think of that equation roughly as:

$$cost(t) \approx (h_\theta(x^{(t)}) - y^{(t)})^2$$

Intuitively, $\delta_j^{(l)}$ is the "error" for $a_j^{(l)}$ (unit j in layer l)

More formally, the delta values are actually the derivative of the cost function:

$$\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} cost(t)$$

Recall that our derivative is the slope of a line tangent to the cost function, so the steeper the slope the more incorrect we are.

Note: In lecture, sometimes i is used to index a training example. Sometimes it is used to index a unit in a layer. In the Back Propagation Algorithm described here, t is used to index a training example rather than overloading the use of i.

Implementation Note: Unrolling Parameters

With neural networks, we are working with sets of matrices:

$$\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}, \dots$$
$$D^{(1)}, D^{(2)}, D^{(3)}, \dots$$

In order to use optimizing functions such as "fminunc()", we will want to "unroll" all the elements and put them into one long vector:

```
thetaVector = [ Theta1(:); Theta2(:); Theta3(:); ]
deltaVector = [ D1(:); D2(:); D3(:) ]
```

If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11, then we can get back our original matrices from the "unrolled" versions as follows:

```
Theta1 = reshape(thetaVector(1:110),10,11)
Theta2 = reshape(thetaVector(111:220),10,11)
Theta3 = reshape(thetaVector(221:231),1,11)
```

NOTE: The lecture slides show an example neural network with 3 layers. However, *3* theta matrices are defined: Theta1, Theta2, Theta3. There should be only 2 theta matrices: Theta1 (10 x 11), Theta2 (1 x 11).

Gradient Checking

Gradient checking will assure that our backpropagation works as intended.

We can approximate the derivative of our cost function with:

$$\frac{\partial}{\partial \Theta} J(\Theta) \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}$$

With multiple theta matrices, we can approximate the derivative **with respect to** Θ_j as follows:

$$\frac{\partial}{\partial \Theta_j} J(\Theta) \approx \frac{J(\Theta_1, \dots, \Theta_j + \epsilon, \dots, \Theta_n) - J(\Theta_1, \dots, \Theta_j - \epsilon, \dots, \Theta_n)}{2\epsilon}$$

A good small value for ϵ (epsilon), guarantees the math above to become true. If the value be much smaller, may we will end up with numerical problems. The professor Andrew usually uses the value $\epsilon = 10^{-4}$.

We are only adding or subtracting epsilon to the $Theta_j$ matrix. In octave we can do it as follows:

```
epsilon = 1e-4;
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) += epsilon;
  thetaMinus = theta;
  thetaMinus(i) -= epsilon;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
end;
```

We then want to check that gradApprox ≈ deltaVector.

Once you've verified **once** that your backpropagation algorithm is correct, then you don't need to compute gradApprox again. The code to compute gradApprox is very slow.

Random Initialization

Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly.

Instead we can randomly initialize our weights:

Initialize each $\Theta_{ij}^{(l)}$ to a random value between$[-\epsilon, \epsilon]$:

$$\epsilon = \frac{\sqrt{6}}{\sqrt{\text{Loutput} + \text{Linput}}}$$

$$\Theta^{(l)} = 2\epsilon \, \text{rand}(\text{Loutput}, \text{Linput} + 1) - \epsilon$$

```
If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11.

Theta1 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta2 = rand(10,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
Theta3 = rand(1,11) * (2 * INIT_EPSILON) - INIT_EPSILON;
```

rand(x,y) will initialize a matrix of random real numbers between 0 and 1. (Note: this epsilon is unrelated to the epsilon from Gradient Checking)

Why use this method? This paper may be useful: https://web.stanford.edu/class/ee373b/nninitialization.pdf

Putting it Together

First, pick a network architecture; choose the layout of your neural network, including how many hidden units in each layer and how many layers total.

- Number of input units = dimension of features $x^{(i)}$
- Number of output units = number of classes
- Number of hidden units per layer = usually more the better (must balance with cost of computation as it increases with more hidden units)
- Defaults: 1 hidden layer. If more than 1 hidden layer, then the same number of units in every hidden layer.

Training a Neural Network

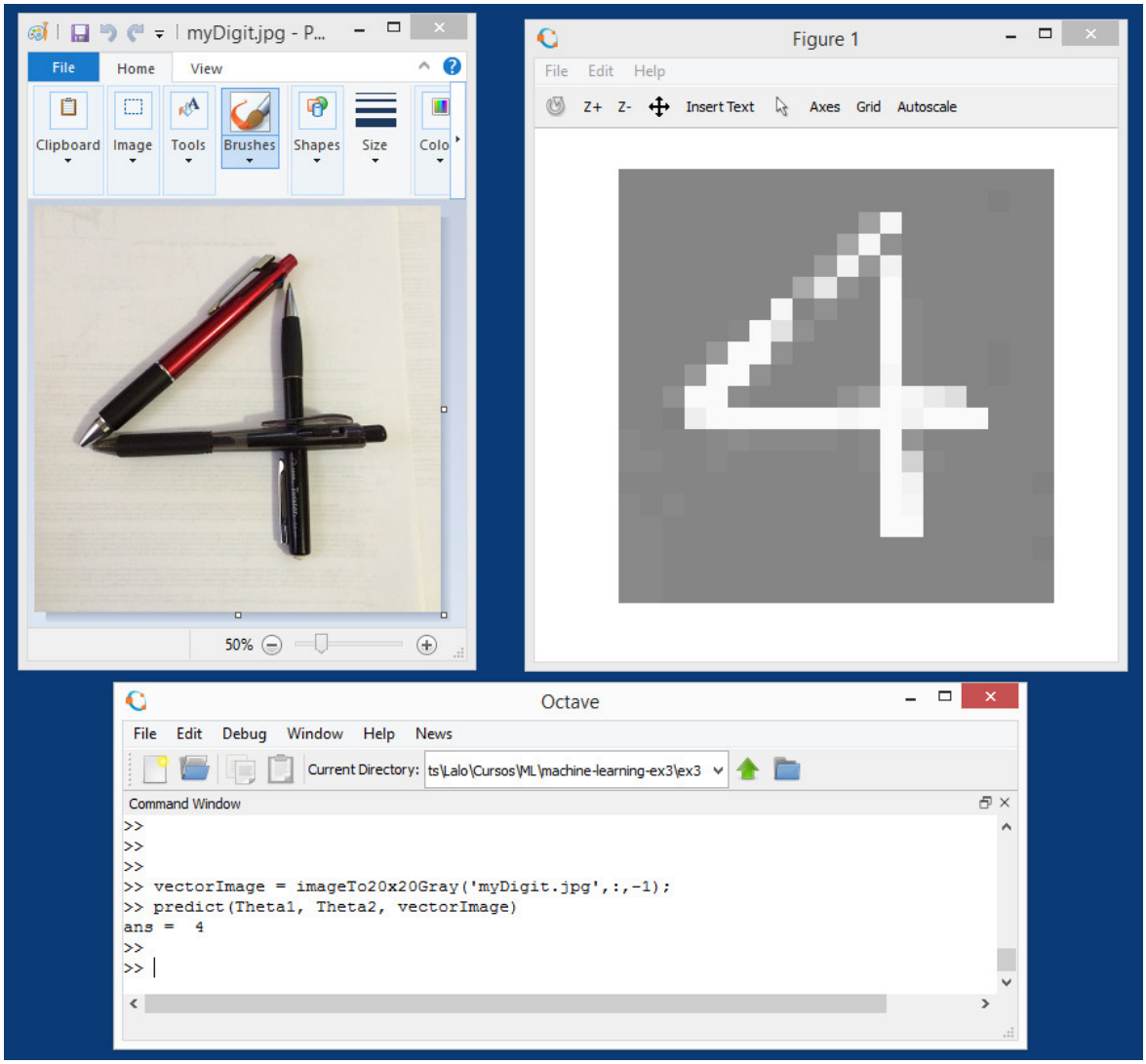
1. Randomly initialize the weights
2. Implement forward propagation to get $h_\theta(x^{(i)})$
3. Implement the cost function
4. Implement backpropagation to compute partial derivatives
5. Use gradient checking to confirm that your backpropagation works. Then disable gradient checking.
6. Use gradient descent or a built-in optimization function to minimize the cost function with the weights in theta.

When we perform forward and back propagation, we loop on every training example:

```
for i = 1:m,
   Perform forward propagation and backpropagation using example (x(i),y(i))
   (Get activations a(l) and delta terms d(l) for l = 2,...,L
```

Bonus: Tutorial on How to classify your own images of digits

This tutorial will guide you on how to use the classifier provided in exercise 3 to classify you own images like this:



It will also explain how the images are converted thru several formats to be processed and displayed.

Introduction

The classifier provided expects 20 x 20 pixels black and white images converted in a row vector of 400 real numbers like this

```
[ 0.14532, 0.12876, ...]
```

Each pixel is represented by a real number between -1.0 to 1.0, meaning -1.0 equal black and 1.0 equal white (any number in between is a shade of gray, and number 0.0 is exactly the middle gray).

.jpg and color RGB images

The most common image format that can be read by Octave is .jpg using function that outputs a three-dimensional matrix of integer numbers from 0 to 255, representing the height x width x 3 integers as indexes of a color map for each pixel (explaining color maps is beyond scope).

```
Image3DmatrixRGB = imread("myOwnPhoto.jpg");
```

Convert to Black & White

A common way to convert color images to black & white, is to convert them to a YIQ standard and keep only the Y component that represents the luma information (black & white). I and Q represent the chrominance information (color).Octave has a function **rgb2ntsc()** that outputs a similar three-dimensional matrix but of real numbers from -1.0 to 1.0, representing the height x width x 3 (Y luma, I in-phase, Q quadrature) intensity for each pixel.

```
Image3DmatrixYIQ = rgb2ntsc(MyImageRGB);
```

To obtain the Black & White component just discard the I and Q matrices. This leaves a two-dimensional matrix of real numbers from -1.0 to 1.0 representing the height x width pixels black & white values.

```
Image2DmatrixBW = Image3DmatrixYIQ(:,:,1);
```

Cropping to square image

It is useful to crop the original image to be as square as possible. The way to crop a matrix is by selecting an area inside the original B&W image and copy it to a new matrix. This is done by selecting the rows and columns that define the area. In other words, it is copying a rectangular subset of the matrix like this:

```
croppedImage = Image2DmatrixBW(origen1:size1, origin2:size2);
```

Cropping does not have to be all the way to a square. **It could be cropping just a percentage of the way to a square** so you can leave more of the image intact. The next step of scaling will take care of streaching the image to fit a square.

Scaling to 20 x 20 pixels

The classifier provided was trained with 20 x 20 pixels images so we need to scale our photos to meet. It may cause distortion depending on the height and width ratio of the cropped original photo. There are many ways to scale a photo but we are going to use the simplest one. We lay a scaled grid of 20 x 20 over the original photo and take a sample pixel on the center of each grid. To lay a scaled grid, we compute two vectors of 20 indexes each evenly spaced on the original size of the image. One for the height and one for the width of the image. For example, in an image of 320 x 200 pixels will produce to vectors like

```
[9    25    41    57    73 ... 313] % 20 indexes
```

```
[6    16    26    36    46 ... 196] % 20 indexes
```

Copy the value of each pixel located by the grid of these indexes to a new matrix. Ending up with a matrix of 20 x 20 real numbers.

Black & White to Gray & White

The classifier provided was trained with images of white digits over gray background. Specifically, the 20 x 20 matrix of real numbers ONLY range from 0.0 to 1.0 instead of the complete black & white range of -1.0 to 1.0, this means that we have to normalize our photos to a range 0.0 to 1.0 for this classifier to work. But also, we invert the black and white colors because is easier to "draw" black over white on our photos and we need to get white digits. So in short, we **invert black and white** and **stretch black to gray** .

Rotation of image

Some times our photos are automatically rotated like in our celular phones. The classifier provided can not recognize rotated images so we may need to rotate it back sometimes. This can be done with an Octave function **rot90()** like this.

```
ImageAligned = rot90(Image, rotationStep);
```

Where rotationStep is an integer: -1 mean rotate 90 degrees CCW and 1 mean rotate 90 degrees CW.

Approach

1. The approach is to have a function that converts our photo to the format the classifier is expecting. As if it was just a sample from the training data set.

2. Use the classifier to predict the digit in the converted image.

Code step by step

Define the function name, the output variable and three parameters, one for the filename of our photo, one optional cropping percentage (if not provided will default to zero, meaning no cropping) and the last optional rotation of the image (if not provided will default to cero, meaning no rotation).

```
function vectorImage = imageTo20x20Gray(fileName, cropPercentage=0, rotStep=0)
```

Read the file as a RGB image and convert it to Black & White 2D matrix (see the introduction).

```
% Read as RGB image
Image3DmatrixRGB = imread(fileName);
% Convert to NTSC image (YIQ)
Image3DmatrixYIQ = rgb2ntsc(Image3DmatrixRGB );
% Convert to grays keeping only luminance (Y)
%        ...and discard chrominance (IQ)
Image2DmatrixBW  = Image3DmatrixYIQ(:,:,1);
```

Establish the final size of the cropped image.

```
% Get the size of your image
oldSize = size(Image2DmatrixBW);
% Obtain crop size toward centered square (cropDelta)
% ...will be zero for the already minimum dimension
% ...and if the cropPercentage is zero,
```

```
% ...both dimensions are zero
% ...meaning that the original image will go intact to croppedImage
cropDelta = floor((oldSize - min(oldSize)) .* (cropPercentage/100));
% Compute the desired final pixel size for the original image
finalSize = oldSize - cropDelta;
```

Obtain the origin and amount of the columns and rows to be copied to the cropped image.

```
% Compute each dimension origin for croping
cropOrigin = floor(cropDelta / 2) + 1;
% Compute each dimension copying size
copySize = cropOrigin + finalSize - 1;
% Copy just the desired cropped image from the original B&W image
croppedImage = Image2DmatrixBW( ...
                    cropOrigin(1):copySize(1), cropOrigin(2):copySize(2));
```

Compute the scale and compute back the new size. This last step is extra. It is computed back so the code keeps general for future modification of the classifier size. For example: if changed from 20 x 20 pixels to 30 x 30. Then the we only need to change the line of code where the scale is computed.

```
% Resolution scale factors: [rows cols]
scale = [20 20] ./ finalSize;
% Compute back the new image size (extra step to keep code general)
newSize = max(floor(scale .* finalSize),1);
```

Compute two sets of 20 indexes evenly spaced. One over the original height and one over the original width of the image.

```
% Compute a re-sampled set of indices:
rowIndex = min(round(((1:newSize(1))-0.5)./scale(1)+0.5), finalSize(1));
colIndex = min(round(((1:newSize(2))-0.5)./scale(2)+0.5), finalSize(2));
```

Copy just the indexed values from old image to get new image of 20 x 20 real numbers. This is called "sampling" because it copies just a sample pixel indexed by a grid. All the sample pixels make the new image.

```
% Copy just the indexed values from old image to get new image
newImage = croppedImage(rowIndex,colIndex,:);
```

Rotate the matrix using the **rot90()** function with the rotStep parameter: -1 is CCW, 0 is no rotate, 1 is CW.

```
% Rotate if needed: -1 is CCW, 0 is no rotate, 1 is CW
newAlignedImage = rot90(newImage, rotStep);
```

Invert black and white because it is easier to draw black digits over white background in our photos but the classifier needs white digits.

```
% Invert black and white
invertedImage = - newAlignedImage;
```

Find the min and max gray values in the image and compute the total value range in preparation for normalization.

```
% Find min and max grays values in the image
maxValue = max(invertedImage(:));
minValue = min(invertedImage(:));
% Compute the value range of actual grays
delta = maxValue - minValue;
```

Do normalization so all values end up between 0.0 and 1.0 because this particular classifier do not perform well with negative numbers.

```
% Normalize grays between 0 and 1
normImage = (invertedImage - minValue) / delta;
```

Add some contrast to the image. The multiplication factor is the contrast control, you can increase it if desired to obtain sharper contrast (contrast only between gray and white, black was already removed in normalization).

```
% Add contrast. Multiplication factor is contrast control.
contrastedImage = sigmoid((normImage -0.5) * 5);
```

Show the image specifying the black & white range [-1 1] to avoid automatic ranging using the image range values of gray to white. Showing the photo with different range, does not affect the values in the output matrix, so do not affect the classifier. It is only as a visual feedback for the user.

```
% Show image as seen by the classifier
imshow(contrastedImage, [-1, 1] );
```

Finally, output the matrix as a unrolled vector to be compatible with the classifier.

```
% Output the matrix as a unrolled vector
vectorImage = reshape(normImage, 1, newSize(1) * newSize(2));
```

End function.

```
end;
```

Usage samples

Single photo

- Photo file in myDigit.jpg
- Cropping 60% of the way to square photo
- No rotationvectorImage = imageTo20x20Gray('myDigit.jpg',60); predict(Theta1, Theta2, vectorImage)
- Photo file in myDigit.jpg
- No cropping
- CCW rotationvectorImage = imageTo20x20Gray('myDigit.jpg',:,-1); predict(Theta1, Theta2, vectorImage)

Multiple photos

- Photo files in myFirstDigit.jpg, mySecondDigit.jpg
- First crop to square and second 25% of the way to square photo
- First no rotation and second CW rotationvectorImage(1,:) = imageTo20x20Gray('myFirstDigit.jpg',100); vectorImage(2,:) = imageTo20x20Gray('mySecondDigit.jpg',25,1); predict(Theta1, Theta2, vectorImage)

Tips

- JPG photos of black numbers over white background
- Preferred square photos but not required
- Rotate as needed because the classifier can only work with vertical digits
- Leave background space around digit. Al least 2 pixels when seen at 20 x 20 resolution. This means that the classifier only really works in a 16 x 16 area.
- Play changing the contrast multipier to 10 (or more).

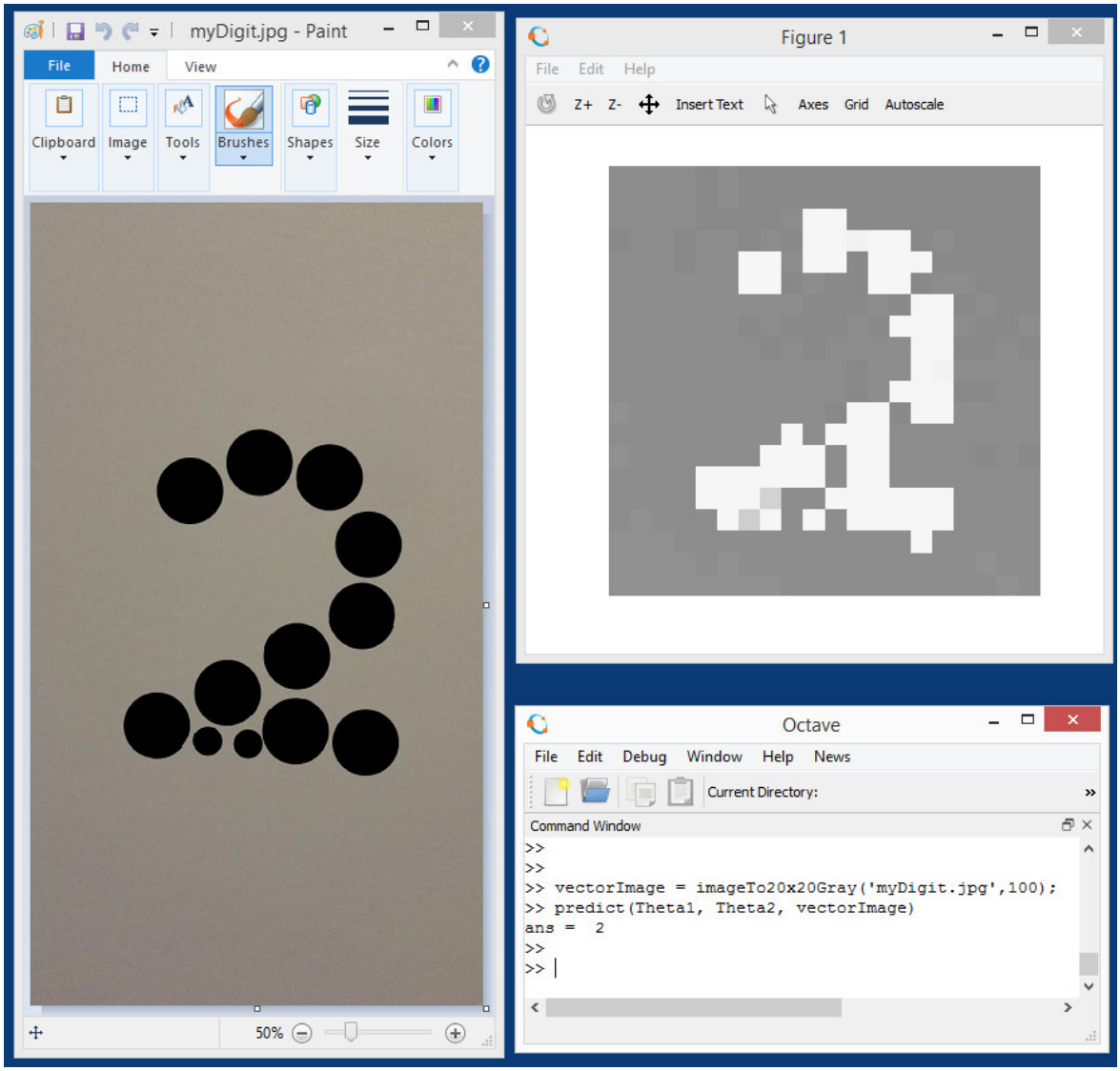
Complete code (just copy and paste)

```
function vectorImage = imageTo20x20Gray(fileName, cropPercentage=0, rotStep=0)
%IMAGETO20X20GRAY display reduced image and converts for digit classification
%
% Sample usage:
%       imageTo20x20Gray('myDigit.jpg', 100, -1);
%
%       First parameter: Image file name
%             Could be bigger than 20 x 20 px, it will
%             be resized to 20 x 20. Better if used with
%             square images but not required.
%
%       Second parameter: cropPercentage (any number between 0 and 100)
%             0  0% will be cropped (optional, no needed for square images)
%            50  50% of available croping will be cropped
%           100  crop all the way to square image (for rectangular images)
%
%       Third parameter: rotStep
%            -1  rotate image 90 degrees CCW
%             0  do not rotate (optional)
%             1  rotate image 90 degrees CW
%
% (Thanks to Edwin Frühwirth for parts of this code)
% Read as RGB image
Image3DmatrixRGB = imread(fileName);
% Convert to NTSC image (YIQ)
Image3DmatrixYIQ = rgb2ntsc(Image3DmatrixRGB );
% Convert to grays keeping only luminance (Y) and discard chrominance (IQ)
Image2DmatrixBW  = Image3DmatrixYIQ(:,:,1);
% Get the size of your image
oldSize = size(Image2DmatrixBW);
% Obtain crop size toward centered square (cropDelta)
% ...will be zero for the already minimum dimension
% ...and if the cropPercentage is zero,
% ...both dimensions are zero
% ...meaning that the original image will go intact to croppedImage
cropDelta = floor((oldSize - min(oldSize)) .* (cropPercentage/100));
```

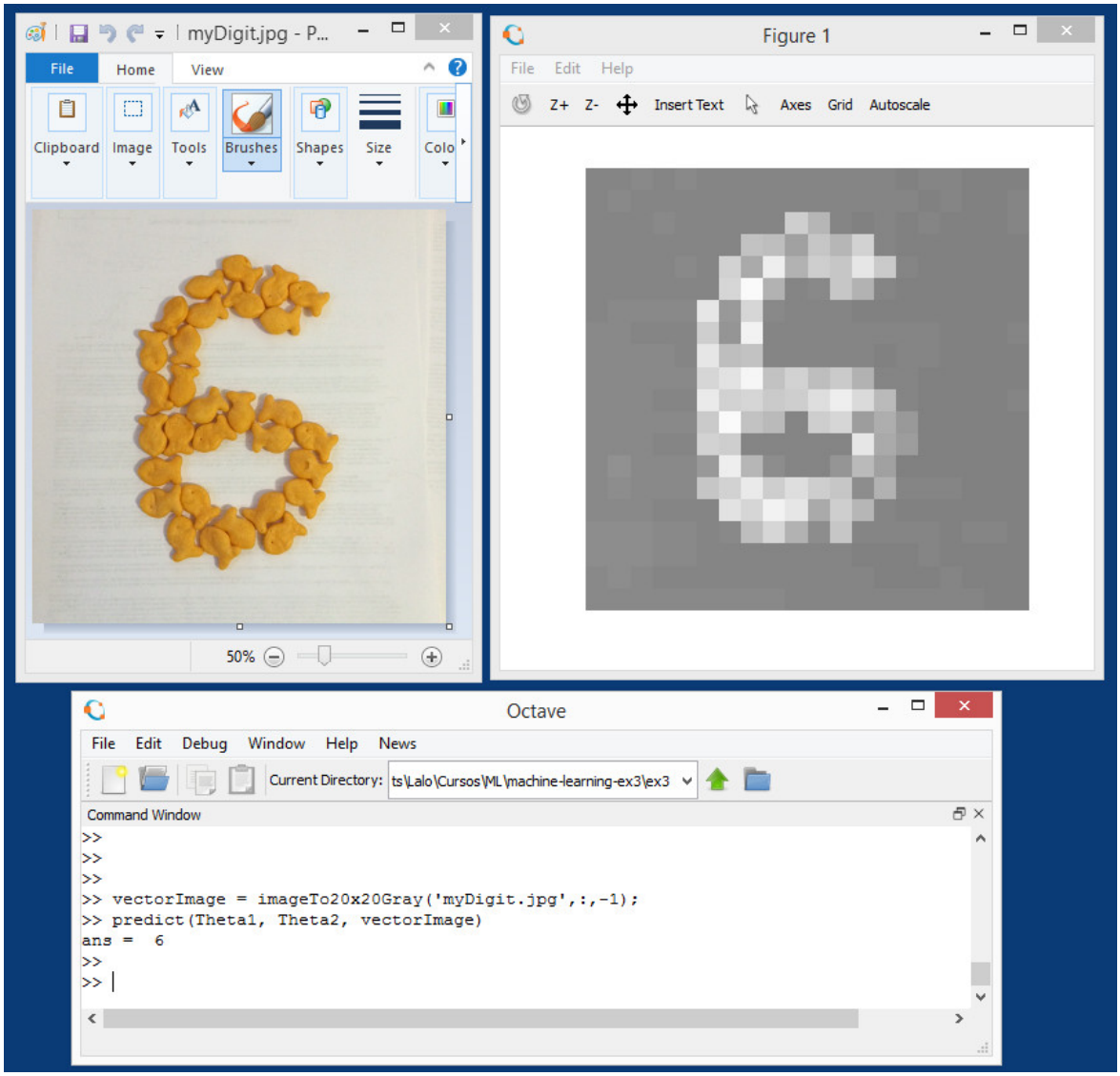
```
% Compute the desired final pixel size for the original image
finalSize = oldSize - cropDelta;
% Compute each dimension origin for croping
cropOrigin = floor(cropDelta / 2) + 1;
% Compute each dimension copying size
copySize = cropOrigin + finalSize - 1;
% Copy just the desired cropped image from the original B&W image
croppedImage = Image2DmatrixBW( ...
                    cropOrigin(1):copySize(1), cropOrigin(2):copySize(2));
% Resolution scale factors: [rows cols]
scale = [20 20] ./ finalSize;
% Compute back the new image size (extra step to keep code general)
newSize = max(floor(scale .* finalSize),1);
% Compute a re-sampled set of indices:
rowIndex = min(round(((1:newSize(1))-0.5)./scale(1)+0.5), finalSize(1));
colIndex = min(round(((1:newSize(2))-0.5)./scale(2)+0.5), finalSize(2));
% Copy just the indexed values from old image to get new image
newImage = croppedImage(rowIndex,colIndex,:);
% Rotate if needed: -1 is CCW, 0 is no rotate, 1 is CW
newAlignedImage = rot90(newImage, rotStep);
% Invert black and white
invertedImage = - newAlignedImage;
% Find min and max grays values in the image
maxValue = max(invertedImage(:));
minValue = min(invertedImage(:));
% Compute the value range of actual grays
delta = maxValue - minValue;
% Normalize grays between 0 and 1
normImage = (invertedImage - minValue) / delta;
% Add contrast. Multiplication factor is contrast control.
contrastedImage = sigmoid((normImage -0.5) * 5);
% Show image as seen by the classifier
imshow(contrastedImage, [-1, 1] );
% Output the matrix as a unrolled vector
vectorImage = reshape(contrastedImage, 1, newSize(1)*newSize(2));
end
```

Photo Gallery

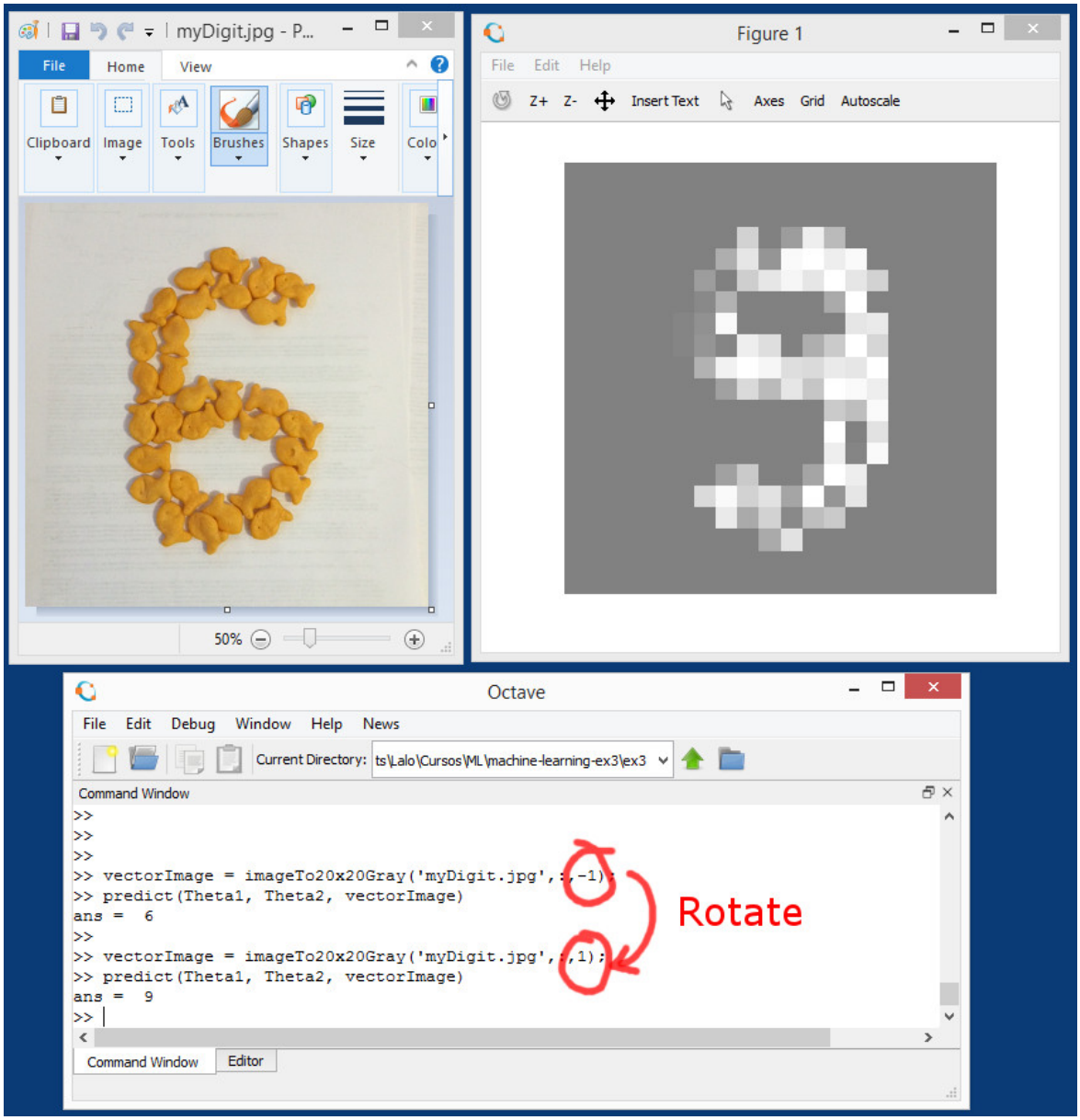
Digit 2



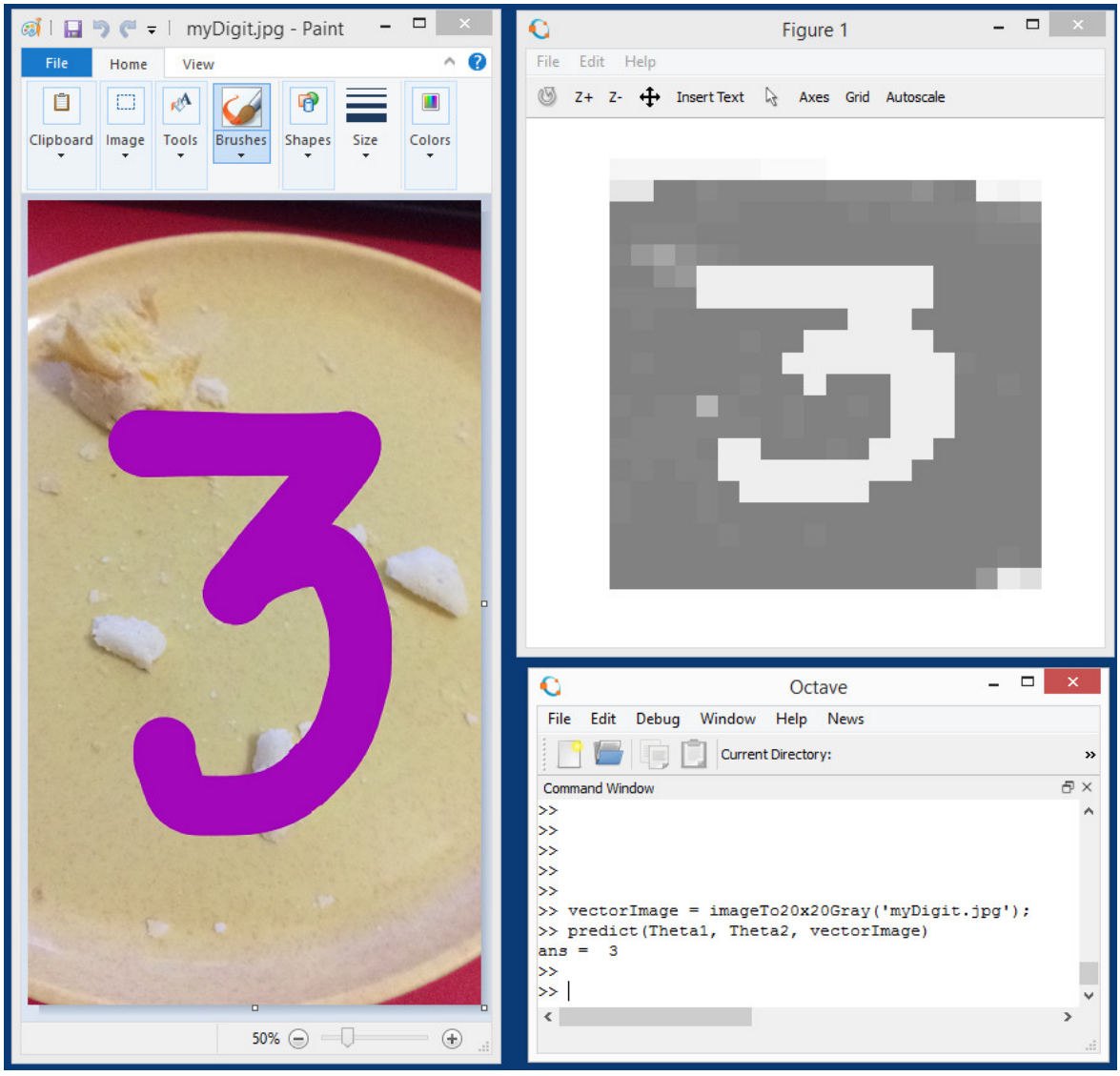
Digit 6



Digit 6 inverted is digit 9. This is the same photo of a six but rotated. Also, changed the contrast multiplier from 5 to 20. You can note that the gray background is smoother.


myDigit.jpg - P...
File
Home
View
Clipboard
Image
Tools
Brushes
Shapes
Size
Colo
50%
Figure 1
File Edit Help
Z+ Z- Insert Text Axes Grid Autoscale
Octave
File Edit Debug Window Help News
Current Directory: ts\Lalo\Cursos\ML\machine-learning-ex3\ex3
Command Window
>>
>>
>>
>> vectorImage = imageTo20x20Gray('myDigit.jpg',:,-1);
>> predict(Theta1, Theta2, vectorImage)
ans = 6
>>
>> vectorImage = imageTo20x20Gray('myDigit.jpg',:,1);
>> predict(Theta1, Theta2, vectorImage)
ans = 9
>>
Rotate
Command Window
Editor

Digit 3



Explanation of Derivatives Used in Backpropagation

- We know that for a logistic regression classifier (which is what all of the output neurons in a neural network are), we use the cost function, $J(\theta) = -ylog(h_\theta(x)) - (1-y)log(1-h_\theta(x))$, and apply this over the K output neurons, and for all m examples.
- The equation to compute the partial derivatives of the theta terms in the output neurons:
- $\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial \theta^{(L-1)}}$
- And the equation to compute partial derivatives of the theta terms in the [last] hidden layer neurons (layer L-1):
- $\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L-1)}}{\partial \theta^{(L-2)}}$
- Clearly they share some pieces in common, so a delta term ($\delta^{(L)}$) can be used for the common pieces between the output layer and the hidden layer immediately before it (with the possibility that there could be many hidden layers if we wanted):
- $\delta^{(L)} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}}$
- And we can go ahead and use another delta term ($\delta^{(L-1)}$) for the pieces that would be shared by the final hidden layer and a hidden layer before that, if we had one. Regardless, this delta term will still serve to make the math and implementation more concise.
- $\delta^{(L-1)} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}$
- $\delta^{(L-1)} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}$
- With these delta terms, our equations become:
- $\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial \theta^{(L-1)}}$

- $\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = \delta^{(L-1)} \frac{\partial z^{(L-1)}}{\partial \theta^{(L-2)}}$
- Now, time to evaluate these derivatives:
- Let's start with the output layer:
- $\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial \theta^{(L-1)}}$
- Using $\delta^{(L)} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}}$, we need to evaluate both partial derivatives.
- Given $J(\theta) = -ylog(a^{(L)}) - (1-y)log(1-a^{(L)})$, where $a^{(L)} = h_\theta(x))$, the partial derivative is:
- $\frac{\partial J(\theta)}{\partial a^{(L)}} = \frac{1-y}{1-a^{(L)}} - \frac{y}{a^{(L)}}$
- And given a=g(z), where $g = \frac{1}{1+e^{-z}}$, the partial derivative is:
- $\frac{\partial a^{(L)}}{\partial z^{(L)}} = a^{(L)}(1-a^{(L)})$
- So, let's substitute these in for $\delta^{(L)}$:

$\delta^{(L)} = \frac{\partial J(\theta)}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}}$

$\delta^{(L)} = (\frac{1-y}{1-a^{(L)}} - \frac{y}{a^{(L)}})(a^{(L)}(1-a^{(L)}))$

$\delta^{(L)} = a^{(L)} - y$

- So, for a 3-layer network (L=3),

$\delta^{(3)} = a^{(3)} - y$

- Note that this is the correct equation, as given in our notes.
- Now, given z=θ∗input, and in layer L the input is $a^{(L-1)}$, the partial derivative is:

$\frac{\partial z^{(L)}}{\partial \theta^{(L-1)}} = a^{(L-1)}$

- **Put it together for the output layer** :

$\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial \theta^{(L-1)}}$

$\frac{\partial J(\theta)}{\partial \theta^{(L-1)}} = (a^{(L)} - y)(a^{(L-1)})$

- Let's continue on for the hidden layer (let's assume we only have 1 hidden layer):

$\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = \delta^{(L-1)} \frac{\partial z^{(L-1)}}{\partial \theta^{(L-2)}}$

- Let's figure out $\delta(L-1)$.
- Once again, given z=θ∗input, the partial derivative is:

$\frac{\partial z^{(L)}}{\partial a^{(L-1)}} = \theta^{(L-1)}$

- And: $\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} = a^{(L-1)}(1-a^{(L-1)})$
- So, let's substitute these in for $\delta^{(L-1)}$:

$\delta^{(L-1)} = \delta^{(L)} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}$

$\delta^{(L-1)} = \delta^{(L)} (\theta^{(L-1)})(a^{(L-1)}(1-a^{(L-1)}))$

$\delta^{(L-1)} = \delta^{(L)} \theta^{(L-1)} a^{(L-1)}(1-a^{(L-1)})$

- So, for a 3-layer network,

$\delta^{(2)} = \delta^{(3)} \theta^{(2)} a^{(2)}(1-a^{(2)})$

- **Put it together for the [last] hidden layer** :

$\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = \delta^{(L-1)} \frac{\partial z^{(L-1)}}{\partial \theta^{(L-2)}}$

$\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = (\delta^{(L)} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}})(a^{(L-2)})$

$\frac{\partial J(\theta)}{\partial \theta^{(L-2)}} = ((a^{(L)} - y)(\theta^{(L-1)})(a^{(L-1)}(1 - a^{(L-1)})))(a^{(L-2)})$

NN for linear systems

Introduction

The NN we created for classification can easily be modified to have a linear output. First solve the 4th programming exercise. You can create a new function script, nnCostFunctionLinear.m, with the following characteristics

- There is only one output node, so you do not need the 'num_labels' parameter.
- Since there is one linear output, you do not need to convert y into a logical matrix.
- You still need a non-linear function in the hidden layer.
- The non-linear function is often the tanh() function - it has an output range from -1 to +1, and its gradient is easily implemented. Let g(z)=tanh(z).
- The gradient of tanh is $g'(z) = 1 - g(z)^2$. Use this in backpropagation in place of the sigmoid gradient.
- Remove the sigmoid function from the output layer (i.e. calculate a3 without using a sigmoid function), since we want a linear output.
- Cost computation: Use the linear cost function for J (from ex1 and ex5) for the unregularized portion. For the regularized portion, use the same method as ex4.
- Where reshape() is used to form the Theta matrices, replace 'num_labels' with '1'.

You still need to randomly initialize the Theta values, just as with any NN. You will want to experiment with different epsilon values. You will also need to create a predictLinear() function, using the tanh() function in the hidden layer, and a linear output.

Testing your linear NN

Here is a test case for your nnCostFunctionLinear()

```
% inputs
nn_params = [31 16 15 -29 -13 -8 -7 13 54 -17 -11 -9 16]'/ 10;
il = 1;
hl = 4;
X = [1; 2; 3];
y = [1; 4; 9];
lambda = 0.01;

% command
[j g] = nnCostFunctionLinear(nn_params, il, hl, X, y, lambda)

% results
j =  0.020815
g =
    -0.0131002
    -0.0110085
    -0.0070569
     0.0189212
    -0.0189639
    -0.0192539
    -0.0102291
     0.0344732
     0.0024947
     0.0080624
     0.0021964
     0.0031675
    -0.0064244
```

Now create a script that uses the 'ex5data1.mat' from ex5, but without creating the polynomial terms. With 8 units in the hidden layer and MaxIter set to 200, you should be able to get a final cost value of 0.3 to 0.4. The results will vary a bit due to the random Theta initialization. If you plot the training set and the predicted values for the training set (using your predictLinear() function), you should have a good match.

Deriving the Sigmoid Gradient Function

We let the sigmoid function be $\sigma(x) = \frac{1}{1+e^{-x}}$

Deriving the equation above yields to $(\frac{1}{1+e^{-x}})^2 \frac{d}{ds} \frac{1}{1+e^{-x}}$

Which is equal to $(\frac{1}{1+e^{-x}})^2 e^{-x}(-1)$

$(\frac{1}{1+e^{-x}})(\frac{1}{1+e^{-x}})(-e^{-x})$

$(\frac{1}{1+e^{-x}})(\frac{-e^{-x}}{1+e^{-x}})$

$\sigma(x)(1-\sigma(x))$

Additional Resources for Backpropagation

- Very thorough conceptual [example] (https://web.archive.org/web/20150317210621/https://www4.rgu.ac.uk/files/chapter3%20-%20bp.pdf)
- Short derivation of the backpropagation algorithm: http://pandamatak.com/people/anand/771/html/node37.html
- Stanford University Deep Learning notes: http://ufldl.stanford.edu/wiki/index.php/Backpropagation_Algorithm
- Very thorough explanation and proof: http://neuralnetworksanddeeplearning.com/chap2.html

ML:Advice for Applying Machine Learning

Deciding What to Try Next

Errors in your predictions can be troubleshooted by:

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

Don't just pick one of these avenues at random. We'll explore diagnostic techniques for choosing one of the above solutions in the following sections.

Evaluating a Hypothesis

A hypothesis may have low error for the training examples but still be inaccurate (because of overfitting).

With a given dataset of training examples, we can split up the data into two sets: a **training set** and a **test set** .

The new procedure using these two sets is then:

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

The test set error

1. For linear regression: $J_{test}(\Theta) = \frac{1}{2m_{test}} \sum_{i=1}^{m_{test}} (h_\Theta(x_{test}^{(i)}) - y_{test}^{(i)})^2$
2. For classification ~ Misclassification error (aka 0/1 misclassification error):

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

This gives us a binary 0 or 1 error result based on a misclassification.

The average test error for the test set is

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

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.
- The error of your hypothesis as measured on the data set with which you trained the parameters will be lower than any other data set.

In order to choose the model of your hypothesis, you can test each degree of polynomial and look at the error result.

Without the Validation Set (note: this is a bad method - do not use it)

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 test set.
3. Estimate the generalization error also using the test set with $J_{test}(\Theta^{(d)})$, (d = theta from polynomial with lower error);

In this case, we have trained one variable, d, or the degree of the polynomial, using the test set. This will cause our error value to be greater for any other set of data.

Use of the CV set

To solve this, we can introduce a third set, the **Cross Validation Set** , to serve as an intermediate set that we can train d with. Then our test set will give us an accurate, non-optimistic error.

One example 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.

With the Validation Set (note: this method presumes we do not also use the CV set for regularization)

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.

(Mentor note: be aware that using the CV set to select 'd' means that we cannot also use it for the validation curve process of setting the lambda value).

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. We need to find a golden mean between these two.

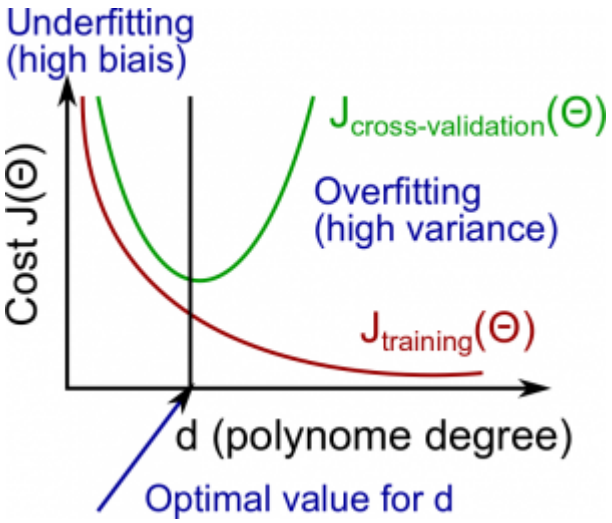
The training error will tend to **decrease** as we increase the degree d of the polynomial.

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 represented in the figure below:



Regularization and Bias/Variance

Instead of looking at the degree d contributing to bias/variance, now we will look at the regularization parameter λ.

- Large λ: High bias (underfitting)
- Intermediate λ: just right

- Small λ: High variance (overfitting)

A large lambda heavily penalizes all the Θ parameters, which greatly simplifies the line of our resulting function, so causes underfitting.

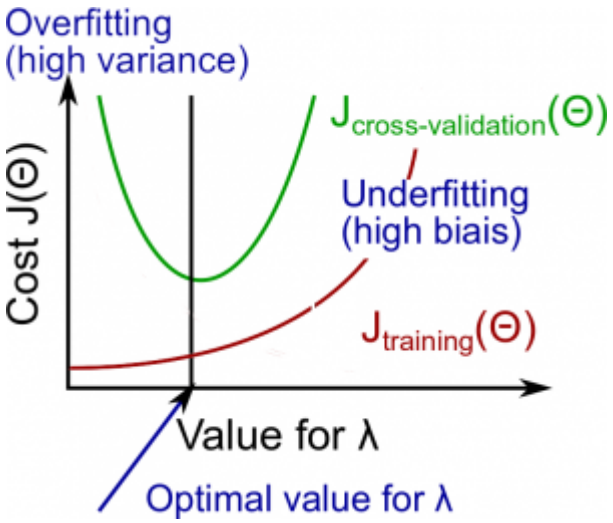
The relationship of λ to the training set and the variance set is as follows:

Low λ : $J_{train}(\Theta)$ is low and $J_{CV}(\Theta)$ is high (high variance/overfitting).

Intermediate λ : $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ are somewhat low and $J_{train}(\Theta) \approx J_{CV}(\Theta)$.

Large λ : both $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ will be high (underfitting /high bias)

The figure below illustrates the relationship between lambda and the hypothesis:



In order to choose the model and the regularization λ, we need:

1. Create a list of lambdas (i.e. λ∈{0,0.01,0.02,0.04,0.08,0.16,0.32,0.64,1.28,2.56,5.12,10.24});

2. Create a set of models with different degrees or any other variants.

3. Iterate through the λs and for each λ go through all the models to learn some Θ.

4. Compute the cross validation error using the learned Θ (computed with λ) on the $J_{CV}(\Theta)$ 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 3 examples will easily have 0 errors because we can always find a quadratic curve that exactly touches 3 points.

- 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.

With 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** .

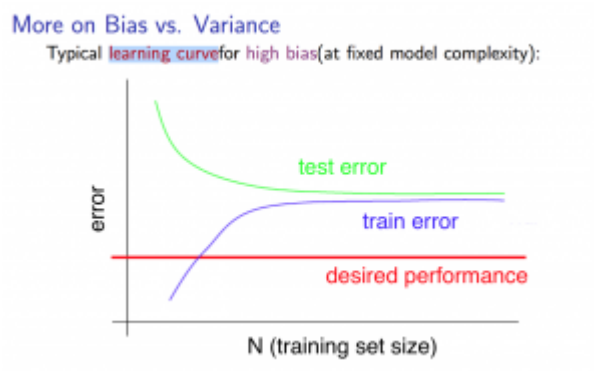
For high variance, we have the following relationships in terms of the training set size:

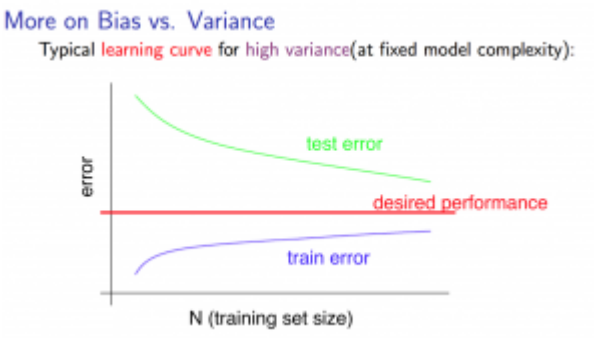
With high variance

Low training set size : $J_{train}(\Theta)$ will be low and $J_{CV}(\Theta)$ will 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.

Model Selection:

Choosing M the order of polynomials.

How can we tell which parameters Θ to leave in the model (known as "model selection")?

There are several ways to solve this problem:

- Get more data (very difficult).
- Choose the model which best fits the data without overfitting (very difficult).
- Reduce the opportunity for overfitting through regularization.

Bias: approximation error (Difference between expected value and optimal value)

- High Bias = UnderFitting (BU)
- $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ both will be high and $J_{train}(\Theta) \approx J_{CV}(\Theta)$

Variance: estimation error due to finite data

- High Variance = OverFitting (VO)
- $J_{train}(\Theta)$ is low and $J_{CV}(\Theta) \gg J_{train}(\Theta)$

Intuition for the bias-variance trade-off:

- Complex model => sensitive to data => much affected by changes in X => high variance, low bias.
- Simple model => more rigid => does not change as much with changes in X => low variance, high bias.

One of the most important goals in learning: finding a model that is just right in the bias-variance trade-off.

Regularization Effects:

- Small values of λ allow model to become finely tuned to noise leading to large variance => overfitting.
- Large values of λ pull weight parameters to zero leading to large bias => underfitting.

Model Complexity Effects:

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

A typical rule of thumb when running diagnostics is:

- More training examples fixes high variance but not high bias.
- Fewer features fixes high variance but not high bias.
- Additional features fixes high bias but not high variance.
- The addition of polynomial and interaction features fixes high bias but not high variance.
- When using gradient descent, decreasing lambda can fix high bias and increasing lambda can fix high variance (lambda is the regularization parameter).
- When using neural networks, small neural networks are more prone to under-fitting and big neural networks are prone to over-fitting. Cross-validation of network size is a way to choose alternatives.

ML:Machine Learning System Design

Prioritizing What to Work On

Different ways we can approach a machine learning problem:

- Collect lots of data (for example "honeypot" project but doesn't always work)

- Develop sophisticated features (for example: using email header data in spam emails)
- Develop algorithms to process your input in different ways (recognizing misspellings in spam).

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

Error Analysis

The recommended approach to solving machine learning problems is:

- Start with a simple algorithm, implement it quickly, and test it early.
- Plot learning curves to decide if more data, more features, etc. will help
- Error analysis: manually examine the errors on examples in the cross validation set and try to spot a trend.

It's important to get error results as a single, numerical value. Otherwise it is difficult to assess your algorithm's performance.

You may need to process your input before it is useful. For example, if your input is a set of words, you may want to treat the same word with different forms (fail/failing/failed) as one word, so must use "stemming software" to recognize them all as one.

Error Metrics for Skewed Classes

It is sometimes difficult to tell whether a reduction in error is actually an improvement of the algorithm.

- For example: In predicting a cancer diagnoses where 0.5% of the examples have cancer, we find our learning algorithm has a 1% error. However, if we were to simply classify every single example as a 0, then our error would reduce to 0.5% even though we did not improve the algorithm.

This usually happens with **skewed classes** ; that is, when our class is very rare in the entire data set.

Or to say it another way, when we have lot more examples from one class than from the other class.

For this we can use **Precision/Recall** .

- Predicted: 1, Actual: 1 --- True positive
- Predicted: 0, Actual: 0 --- True negative
- Predicted: 0, Actual, 1 --- False negative
- Predicted: 1, Actual: 0 --- False positive

Precision : of all patients we predicted where y=1, what fraction actually has cancer?

$$\frac{\text{True Positives}}{\text{Total number of predicted positives}} = \frac{\text{True Positives}}{\text{True Positives} + \text{False positives}}$$

Recall : Of all the patients that actually have cancer, what fraction did we correctly detect as having cancer?

$$\frac{\text{True Positives}}{\text{Total number of actual positives}} = \frac{\text{True Positives}}{\text{True Positives} + \text{False negatives}}$$

These two metrics give us a better sense of how our classifier is doing. We want both precision and recall to be high.

In the example at the beginning of the section, if we classify all patients as 0, then our **recall** will be $\frac{0}{0+f} = 0$, so despite having a lower error percentage, we can quickly see it has worse recall.

Accuracy = $\frac{true positive + true negative}{total population}$

Note 1: if an algorithm predicts only negatives like it does in one of exercises, the precision is not defined, it is impossible to divide by 0. F1 score will not be defined too.

Trading Off Precision and Recall

We might want a **confident** prediction of two classes using logistic regression. One way is to increase our threshold:

- Predict 1 if: $h_\theta(x) \geq 0.7$
- Predict 0 if: $h_\theta(x) < 0.7$

This way, we only predict cancer if the patient has a 70% chance.

Doing this, we will have **higher precision** but **lower recall** (refer to the definitions in the previous section).

In the opposite example, we can lower our threshold:

- Predict 1 if: $h_\theta(x) \geq 0.3$
- Predict 0 if: $h_\theta(x) < 0.3$

That way, we get a very **safe** prediction. This will cause **higher recall** but **lower precision** .

The greater the threshold, the greater the precision and the lower the recall.

The lower the threshold, the greater the recall and the lower the precision.

In order to turn these two metrics into one single number, we can take the **F value** .

One way is to take the **average** :

$$\frac{P+R}{2}$$

This does not work well. If we predict all y=0 then that will bring the average up despite having 0 recall. If we predict all examples as y=1, then the very high recall will bring up the average despite having 0 precision.

A better way is to compute the **F Score** (or F1 score):

$$\text{F Score} = 2\frac{PR}{P+R}$$

In order for the F Score to be large, both precision and recall must be large.

We want to train precision and recall on the **cross validation set** so as not to bias our test set.

Data for Machine Learning

How much data should we train on?

In certain cases, an "inferior algorithm," if given enough data, can outperform a superior algorithm with less data.

We must choose our features to have **enough** information. A useful test is: Given input x, would a human expert be able to confidently predict y?

Rationale for large data : if we have a **low bias** algorithm (many features or hidden units making a very complex function), then the larger the training set we use, the less we will have overfitting (and the more accurate the algorithm will be on the test set).

Quiz instructions

When the quiz instructions tell you to enter a value to "two decimal digits", what it really means is "two significant digits". So, just for example, the value 0.0123 should be entered as "0.012", not "0.01".

References:

- https://class.coursera.org/ml/lecture/index
- http://www.cedar.buffalo.edu/~srihari/CSE555/Chap9.Part2.pdf
- http://blog.stephenpurpura.com/post/13052575854/managing-bias-variance-tradeoff-in-machine-learning
- http://www.cedar.buffalo.edu/~srihari/CSE574/Chap3/Bias-Variance.pdf

Optimization Objective

The **Support Vector Machine** (SVM) is yet another type of *supervised* machine learning algorithm. It is sometimes cleaner and more powerful.

Recall that in logistic regression, we use the following rules:

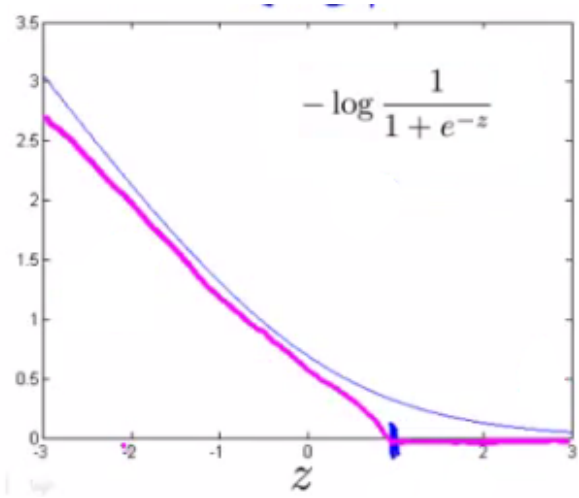
if y=1, then $h_\theta(x) \approx 1$ and $\Theta^T x \gg 0$

if y=0, then $h_\theta(x) \approx 0$ and $\Theta^T x \ll 0$

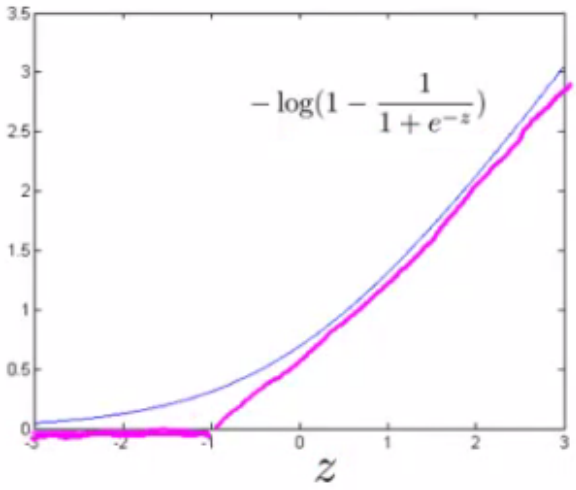
Recall the cost function for (unregularized) logistic regression:

$$\begin{aligned} J(\theta) &= \frac{1}{m}\sum_{i=1}^{m} -y^{(i)}\log(h_\theta(x^{(i)})) - (1-y^{(i)})\log(1-h_\theta(x^{(i)})) \\ &= \frac{1}{m}\sum_{i=1}^{m} -y^{(i)}\log\left(\frac{1}{1+e^{-\theta^T x^{(i)}}}\right) - (1-y^{(i)})\log\left(1-\frac{1}{1+e^{-\theta^T x^{(i)}}}\right) \end{aligned}$$

To make a support vector machine, we will modify the first term of the cost function $-\log(h_\theta(x)) = -\log\left(\frac{1}{1+e^{-\theta^T x}}\right)$ so that when $\theta^T x$ (from now on, we shall refer to this as z) is **greater than** 1, it outputs 0. Furthermore, for values of z less than 1, we shall use a straight decreasing line instead of the sigmoid curve.(In the literature, this is called a hinge loss (https://en.wikipedia.org/wiki/Hinge_loss) function.)



Similarly, we modify the second term of the cost function $-\log(1-h_{\theta(x)}) = -\log\left(1-\frac{1}{1+e^{-\theta^T x}}\right)$ so that when z is **less than** -1, it outputs 0. We also modify it so that for values of z greater than -1, we use a straight increasing line instead of the sigmoid curve.



We shall denote these as $\text{cost}_1(z)$ and $\text{cost}_0(z)$ (respectively, note that $\text{cost}_1(z)$ is the cost for classifying when y=1, and $\text{cost}_0(z)$ is the cost for classifying when y=0), and we may define them as follows (where k is an arbitrary constant defining the magnitude of the slope of the line):

$z = \theta^T x$

$\text{cost}_0(z) = \max(0, k(1+z))$

$\text{cost}_1(z) = \max(0, k(1-z))$

Recall the full cost function from (regularized) logistic regression:

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

Note that the negative sign has been distributed into the sum in the above equation.

We may transform this into the cost function for support vector machines by substituting $\text{cost}_0(z)$ and $\text{cost}_1(z)$:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} y^{(i)}\ \text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)})\ \text{cost}_0(\theta^T x^{(i)}) + \frac{\lambda}{2m}\sum_{j=1}^{n} \Theta_j^2$$

We can optimize this a bit by multiplying this by m (thus removing the m factor in the denominators). Note that this does not affect our optimization, since we're simply multiplying our cost function by a positive constant (for example, minimizing $(u-5)^2 + 1$ gives us 5; multiplying it by 10 to make it $10(u-5)^2 + 10$ still gives us 5 when minimized).

$$J(\theta) = \sum_{i=1}^{m} y^{(i)}\ \text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)})\ \text{cost}_0(\theta^T x^{(i)}) + \frac{\lambda}{2}\sum_{j=1}^{n} \Theta_j^2$$

Furthermore, convention dictates that we regularize using a factor C, instead of λ, like so:

$$J(\theta) = C\sum_{i=1}^{m} y^{(i)}\ \text{cost}_1(\theta^T x^{(i)}) + (1-y^{(i)})\ \text{cost}_0(\theta^T x^{(i)}) + \frac{1}{2}\sum_{j=1}^{n} \Theta_j^2$$

This is equivalent to multiplying the equation by $C = \frac{1}{\lambda}$, and thus results in the same values when optimized. Now, when we wish to regularize more (that is, reduce

overfitting), we *decrease* C, and when we wish to regularize less (that is, reduce underfitting), we *increase* C.

Finally, note that the hypothesis of the Support Vector Machine is *not* interpreted as the probability of y being 1 or 0 (as it is for the hypothesis of logistic regression). Instead, it outputs either 1 or 0. (In technical terms, it is a discriminant function.)

$$h_\theta(x) = \begin{cases} 1 & \text{if } \Theta^T x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Large Margin Intuition

A useful way to think about Support Vector Machines is to think of them as *Large Margin Classifiers* .

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

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

Now when we set our constant C to a very **large** value (e.g. 100,000), our optimizing function will constrain Θ such that the equation A (the summation of the cost of each example) equals 0. We impose the following constraints on Θ:

$\Theta^T x \geq 1$ if y=1 and $\Theta^T x \leq -1$ if y=0.

If C is very large, we must choose Θ parameters such that:

$\sum_{i=1}^m y^{(i)} \text{cost}_1(\Theta^T x) + (1 - y^{(i)}) \text{cost}_0(\Theta^T x) = 0$

This reduces our cost function to:

$$\begin{aligned} J(\theta) &= C \cdot 0 + \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2 \\ &= \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2 \end{aligned}$$

Recall the decision boundary from logistic regression (the line separating the positive and negative examples). In SVMs, the decision boundary has the special property that it is **as far away as possible** from both the positive and the negative examples.

The distance of the decision boundary to the nearest example is called the **margin** . Since SVMs maximize this margin, it is often called a *Large Margin Classifier* .

The SVM will separate the negative and positive examples by a **large margin** .

This large margin is only achieved when **C is very large** .

Data is **linearly separable** when a **straight line** can separate the positive and negative examples.

If we have **outlier** examples that we don't want to affect the decision boundary, then we can **reduce** C.

Increasing and decreasing C is similar to respectively decreasing and increasing λ, and can simplify our decision boundary.

Mathematics Behind Large Margin Classification (Optional)

Vector Inner Product

Say we have two vectors, u and v:

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

The **length of vector v** is denoted $||v||$, and it describes the line on a graph from origin (0,0) to (v_1, v_2).

The length of vector v can be calculated with $\sqrt{v_1^2 + v_2^2}$by the Pythagorean theorem.

The **projection** of vector v onto vector u is found by taking a right angle from u to the end of v, creating a right triangle.

- p= length of projection of v onto the vector u.
- $u^T v = p \cdot ||u||$

Note that $u^T v = ||u|| \cdot ||v|| \cos\theta$ where θ is the angle between u and v. Also, $p = ||v|| \cos\theta$. If you substitute p for $||v|| \cos\theta$, you get $u^T v = p \cdot ||u||$.

So the product $u^T v$ is equal to the length of the projection times the length of vector u.

In our example, since u and v are vectors of the same length, $u^T v = v^T u$.

$u^T v = v^T u = p \cdot ||u|| = u_1 v_1 + u_2 v_2$

If the **angle** between the lines for v and u is **greater than 90 degrees** , then the projection p will be **negative** .

$$\begin{aligned} &\min_{\Theta} \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2 \\ &= \frac{1}{2}(\Theta_1^2 + \Theta_2^2 + \cdots + \Theta_n^2) \\ &= \frac{1}{2}(\sqrt{\Theta_1^2 + \Theta_2^2 + \cdots + \Theta_n^2})^2 \\ &= \frac{1}{2}||\Theta||^2 \end{aligned}$$

We can use the same rules to rewrite $\Theta^T x^{(i)}$:

$\Theta^T x^{(i)} = p^{(i)} \cdot ||\Theta|| = \Theta_1 x_1^{(i)} + \Theta_2 x_2^{(i)} + \cdots + \Theta_n x_n^{(i)}$

So we now have a new **optimization objective** by substituting $p^{(i)} \cdot ||\Theta||$ in for $\Theta^T x^{(i)}$:

If y=1, we want $p^{(i)} \cdot ||\Theta|| \geq 1$

If y=0, we want $p^{(i)} \cdot ||\Theta|| \leq -1$

The reason this causes a "large margin" is because: the vector for Θ is perpendicular to the decision boundary. In order for our optimization objective (above) to hold true, we need the absolute value of our projections $p^{(i)}$ to be as large as possible.

If $\Theta_0 = 0$, then all our decision boundaries will intersect (0,0). If $\Theta_0 \neq 0$, the support vector machine will still find a large margin for the decision boundary.

Kernels I

Kernels allow us to make complex, non-linear classifiers using Support Vector Machines.

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

To do this, we find the "similarity" of x and some landmark $l^{(i)}$:

$$f_i = similarity(x, l^{(i)}) = \exp(-\frac{||x - l^{(i)}||^2}{2\sigma^2})$$

This "similarity" function is called a **Gaussian Kernel** . It is a specific example of a kernel.

The similarity function can also be written as follows:

$$f_i = similarity(x, l^{(i)}) = \exp(-\frac{\sum_{j=1}^{n}(x_j - l_j^{(i)})^2}{2\sigma^2})$$

There are a couple properties of the similarity function:

If $x \approx l^{(i)}$, then $f_i = \exp(-\frac{\approx 0^2}{2\sigma^2}) \approx 1$

If x is far from $l^{(i)}$, then $f_i = \exp(-\frac{(large\ number)^2}{2\sigma^2}) \approx 0$

In other words, if x and the landmark are close, then the similarity will be close to 1, and if x and the landmark are far away from each other, the similarity will be close to 0.

Each landmark gives us the features in our hypothesis:

$$\begin{aligned} l^{(1)} &\rightarrow f_1 \\ l^{(2)} &\rightarrow f_2 \\ l^{(3)} &\rightarrow f_3 \\ &\dots \\ h_\Theta(x) &= \Theta_1 f_1 + \Theta_2 f_2 + \Theta_3 f_3 + \dots \end{aligned}$$

σ^2 is a parameter of the Gaussian Kernel, and it can be modified to increase or decrease the **drop-off** of our feature f_i. Combined with looking at the values inside Θ, we can choose these landmarks to get the general shape of the decision boundary.

Kernels II

One way to get the landmarks is to put them in the **exact same** locations as all the training examples. This gives us m landmarks, with one landmark per training example.

Given example x:

$f_1 = similarity(x, l^{(1)})$, $f_2 = similarity(x, l^{(2)})$, $f_3 = similarity(x, l^{(3)})$, and so on.

This gives us a "feature vector," $f_{(i)}$ of all our features for example $x_{(i)}$. We may also set $f_0 = 1$ to correspond with Θ_0. Thus given training example $x_{(i)}$:

$$x^{(i)} \rightarrow \begin{bmatrix} f_1^{(i)} = similarity(x^{(i)}, l^{(1)}) \\ f_2^{(i)} = similarity(x^{(i)}, l^{(2)}) \\ \vdots \\ f_m^{(i)} = similarity(x^{(i)}, l^{(m)}) \end{bmatrix}$$

Now to get the parameters Θ we can use the SVM minimization algorithm but with $f^{(i)}$ substituted in for $x^{(i)}$:

$\min_\Theta C \sum_{i=1}^{m} y^{(i)} \text{cost}_1(\Theta^T f^{(i)}) + (1 - y^{(i)}) \text{cost}_0(\theta^T f^{(i)}) + \frac{1}{2} \sum_{j=1}^{n} \Theta_j^2$

Using kernels to generate f(i) is not exclusive to SVMs and may also be applied to logistic regression. However, because of computational optimizations on SVMs, kernels combined with SVMs is much faster than with other algorithms, so kernels are almost always found combined only with SVMs.

Choosing SVM Parameters

Choosing C (recall that $C = \frac{1}{\lambda}$

- If C is large, then we get higher variance/lower bias
- If C is small, then we get lower variance/higher bias

The other parameter we must choose is σ^2 from the Gaussian Kernel function:

With a large σ^2, the features fi vary more smoothly, causing higher bias and lower variance.

With a small σ^2, the features fi vary less smoothly, causing lower bias and higher variance.

Using An SVM

There are lots of good SVM libraries already written. A. Ng often uses 'liblinear' and 'libsvm'. In practical application, you should use one of these libraries rather than rewrite the functions.

In practical application, the choices you do need to make are:

- Choice of parameter C
- Choice of kernel (similarity function)

- No kernel ("linear" kernel) -- gives standard linear classifier
- Choose when n is large and when m is small
- Gaussian Kernel (above) -- need to choose σ^2
- Choose when n is small and m is large

The library may ask you to provide the kernel function.

Note: do perform feature scaling before using the Gaussian Kernel.

Note: not all similarity functions are valid kernels. They must satisfy "Mercer's Theorem" which guarantees that the SVM package's optimizations run correctly and do not diverge.

You want to train C and the parameters for the kernel function using the training and cross-validation datasets.

Multi-class Classification

Many SVM libraries have multi-class classification built-in.

You can use the *one-vs-all* method just like we did for logistic regression, where $y \in 1, 2, 3, \ldots, K$ with $\Theta^{(1)}, \Theta^{(2)}, \ldots, \Theta(K)$. We pick class i with the largest $(\Theta^{(i)})^T x$.

Logistic Regression vs. SVMs

If n is large (relative to m), then use logistic regression, or SVM without a kernel (the "linear kernel")

If n is small and m is intermediate, then use SVM with a Gaussian Kernel

If n is small and m is large, then manually create/add more features, then use logistic regression or SVM without a kernel.

In the first case, we don't have enough examples to need a complicated polynomial hypothesis. In the second example, we have enough examples that we may need a complex non-linear hypothesis. In the last case, we want to increase our features so that logistic regression becomes applicable.

Note : a neural network is likely to work well for any of these situations, but may be slower to train.

Additional references

- "An Idiot's Guide to Support Vector Machines": http://web.mit.edu/6.034/wwwbob/svm-notes-long-08.pdf

ML:Clustering

Unsupervised Learning: Introduction

Unsupervised learning is contrasted from supervised learning because it uses an **unlabeled** training set rather than a labeled one.

In other words, we don't have the vector y of expected results, we only have a dataset of features where we can find structure.

Clustering is good for:

- Market segmentation
- Social network analysis
- Organizing computer clusters
- Astronomical data analysis

K-Means Algorithm

The K-Means Algorithm is the most popular and widely used algorithm for automatically grouping data into coherent subsets.

1. Randomly initialize two points in the dataset called the *cluster centroids* .
2. Cluster assignment: assign all examples into one of two groups based on which cluster centroid the example is closest to.
3. Move centroid: compute the averages for all the points inside each of the two cluster centroid groups, then move the cluster centroid points to those averages.
4. Re-run (2) and (3) until we have found our clusters.

Our main variables are:

- K (number of clusters)
- Training set $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$
- Where $x^{(i)} \in \mathbb{R}^n$

Note that we **will not use** the x0=1 convention.

The algorithm:

```
Randomly initialize K cluster centroids mu(1), mu(2), ..., mu(K)
Repeat:
   for i = 1 to m:
      c(i):= index (from 1 to K) of cluster centroid closest to x(i)
```

```
    for k = 1 to K:
      mu(k):= average (mean) of points assigned to cluster k
```

The **first for-loop** is the 'Cluster Assignment' step. We make a vector *c* where *c(i)* represents the centroid assigned to example *x(i)* .

We can write the operation of the Cluster Assignment step more mathematically as follows:

$c^{(i)} = argmin_k \, ||x^{(i)} - \mu_k||^2$

That is, each $c^{(i)}$ contains the index of the centroid that has minimal distance to $x^{(i)}$.

By convention, we square the right-hand-side, which makes the function we are trying to minimize more sharply increasing. It is mostly just a convention. But a convention that helps reduce the computation load because the Euclidean distance requires a square root but it is canceled.

Without the square:

$$||x^{(i)} - \mu_k|| = || \quad \sqrt{(x_1^i - \mu_{1(k)})^2 + (x_2^i - \mu_{2(k)})^2 + (x_3^i - \mu_{3(k)})^2 + \dots} \quad ||$$

With the square:

$$||x^{(i)} - \mu_k||^2 = || \quad (x_1^i - \mu_{1(k)})^2 + (x_2^i - \mu_{2(k)})^2 + (x_3^i - \mu_{3(k)})^2 + \dots \quad ||$$

...so the square convention serves two purposes, minimize more sharply and less computation.

The **second for-loop** is the 'Move Centroid' step where we move each centroid to the average of its group.

More formally, the equation for this loop is as follows:

$$\mu_k = \frac{1}{n}[x^{(k_1)} + x^{(k_2)} + \cdots + x^{(k_n)}] \in \mathbb{R}^n$$

Where each of $x^{(k_1)}, x^{(k_2)}, \dots, x^{(k_n)}$ are the training examples assigned to group $m\mu_k$.

If you have a cluster centroid with **0 points** assigned to it, you can randomly **re-initialize** that centroid to a new point. You can also simply **eliminate** that cluster group.

After a number of iterations the algorithm will **converge** , where new iterations do not affect the clusters.

Note on non-separated clusters: some datasets have no real inner separation or natural structure. K-means can still evenly segment your data into K subsets, so can still be useful in this case.

Optimization Objective

Recall some of the parameters we used in our algorithm:

- $c^{(i)}$ = index of cluster (1,2,...,K) to which example x(i) is currently assigned
- μ_k= cluster centroid k (μk∈$\mathbb{R}$n)
- $\mu_{c^{(i)}}$ = cluster centroid of cluster to which example x(i) has been assigned

Using these variables we can define our **cost function** :

$$J(c^{(i)},\ldots,c^{(m)},\mu_1,\ldots,\mu_K) = \frac{1}{m}\sum_{i=1}^{m}||x^{(i)}-\mu_{c^{(i)}}||^2$$

Our **optimization objective** is to minimize all our parameters using the above cost function:

$min_{c,\mu}\ J(c,\mu)$

That is, we are finding all the values in sets c, representing all our clusters, and μ, representing all our centroids, that will minimize **the average of the distances** of every training example to its corresponding cluster centroid.

The above cost function is often called the **distortion** of the training examples.

In the **cluster assignment step** , our goal is to:

Minimize J(...) with $c^{(1)},\ldots,c^{(m)}$ (holding $\mu_1,\ldots,\mu_K$ fixed)

In the **move centroid** step, our goal is to:

Minimize J(...) with $\mu_1,\ldots,\mu_K$

With k-means, it is **not possible for the cost function to sometimes increase** . It should always descend.

Random Initialization

There's one particular recommended method for randomly initializing your cluster centroids.

1. Have K<m. That is, make sure the number of your clusters is less than the number of your training examples.
2. Randomly pick K training examples. (Not mentioned in the lecture, but also be sure the selected examples are unique).
3. Set $\mu_1,\ldots,\mu_K$ equal to these K examples.

K-means **can get stuck in local optima** . To decrease the chance of this happening, you can run the algorithm on many different random initializations. In cases where K<10 it is strongly recommended to run a loop of random initializations.

```
for i = 1 to 100:
   randomly initialize k-means
   run k-means to get 'c' and 'm'
   compute the cost function (distortion) J(c,m)
pick the clustering that gave us the lowest cost
```

Choosing the Number of Clusters

Choosing K can be quite arbitrary and ambiguous.

The elbow method : plot the cost J and the number of clusters K. The cost function should reduce as we increase the number of clusters, and then flatten out. Choose K at the point where the cost function starts to flatten out.

However, fairly often, the curve is **very gradual** , so there's no clear elbow.

Note: J will **always** decrease as K is increased. The one exception is if k-means gets stuck at a bad local optimum.

Another way to choose K is to observe how well k-means performs on a **downstream purpose** . In other words, you choose K that proves to be most useful for some goal you're trying to achieve from using these clusters.

Bonus: Discussion of the drawbacks of K-Means

This links to a discussion that shows various situations in which K-means gives totally correct but unexpected results:
http://stats.stackexchange.com/questions/133656/how-to-understand-the-drawbacks-of-k-means

ML:Dimensionality Reduction

Motivation I: Data Compression

- We may want to reduce the dimension of our features if we have a lot of redundant data.
- To do this, we find two highly correlated features, plot them, and make a new line that seems to describe both features accurately. We place all the new features on this single line.

Doing dimensionality reduction will reduce the total data we have to store in computer memory and will speed up our learning algorithm.

Note: in dimensionality reduction, we are reducing our features rather than our number of examples. Our variable m will stay the same size; n, the number of features each example from $x^{(1)}$ to $x^{(m)}$ carries, will be reduced.

Motivation II: Visualization

It is not easy to visualize data that is more than three dimensions. We can reduce the dimensions of our data to 3 or less in order to plot it.

We need to find new features, z_1, z_2(and perhaps z_3) that can effectively **summarize** all the other features.

Example: hundreds of features related to a country's economic system may all be combined into one feature that you call "Economic Activity."

Principal Component Analysis Problem Formulation

The most popular dimensionality reduction algorithm is *Principal Component Analysis* (PCA)

Problem formulation

Given two features, x_1 and x_2, we want to find a single line that effectively describes both features at once. We then map our old features onto this new line to get a new single feature.

The same can be done with three features, where we map them to a plane.

The **goal of PCA** is to **reduce** the average of all the distances of every feature to the projection line. This is the **projection error** .

Reduce from 2d to 1d: find a direction (a vector $u^{(1)} \in \mathbb{R}^n$) onto which to project the data so as to minimize the projection error.

The more general case is as follows:

Reduce from n-dimension to k-dimension: Find k vectors $u^{(1)}, u^{(2)}, \ldots, u^{(k)}$ onto which to project the data so as to minimize the projection error.

If we are converting from 3d to 2d, we will project our data onto two directions (a plane), so k will be 2.

PCA is not linear regression

- In linear regression, we are minimizing the **squared error** from every point to our predictor line. These are vertical distances.
- In PCA, we are minimizing the **shortest distance** , or shortest *orthogonal* distances, to our data points.

More generally, in linear regression we are taking all our examples in x and applying the parameters in Θ to predict y.

In PCA, we are taking a number of features $x_1, x_2, \ldots, x_n$, and finding a closest common dataset among them. We aren't trying to predict any result and we aren't applying any theta weights to the features.

Principal Component Analysis Algorithm

Before we can apply PCA, there is a data pre-processing step we must perform:

Data preprocessing

- Given training set: x(1),x(2),…,x(m)
- Preprocess (feature scaling/mean normalization):

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

- Replace each $x_j^{(i)}$ with $x_j^{(i)} - \mu_j$
- If different features on different scales (e.g., x_1 = size of house, x_2 = number of bedrooms), scale features to have comparable range of values.

Above, we first subtract the mean of each feature from the original feature. Then we scale all the features $x_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{s_j}$

We can define specifically what it means to reduce from 2d to 1d data as follows:

$\Sigma = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)})(x^{(i)})^T$

The z values are all real numbers and are the projections of our features onto $u^{(1)}$.

So, PCA has two tasks: figure out $u^{(1)}, \ldots, u^{(k)}$ and also to find $z_1, z_2, \ldots, z_m$.

The mathematical proof for the following procedure is complicated and beyond the scope of this course.

1. Compute "covariance matrix"

$\Sigma = \frac{1}{m} \sum_{i=1}^{m} (x^{(i)})(x^{(i)})^T$

This can be vectorized in Octave as:

```
Sigma = (1/m) * X' * X;
```

We denote the covariance matrix with a capital sigma (which happens to be the same symbol for summation, confusingly---they represent entirely different things).

Note that $x^{(i)}$ is an n×1 vector, $(x^{(i)})^T$ is an 1×n vector and X is a m×n matrix (row-wise stored examples). The product of those will be an n×n matrix, which are the dimensions of Σ.

2. Compute "eigenvectors" of covariance matrix Σ

```
[U,S,V] = svd(Sigma);
```

svd() is the 'singular value decomposition', a built-in Octave function.

What we actually want out of svd() is the 'U' matrix of the Sigma covariance matrix: $U \in \mathbb{R}^{n \times n}$. U contains $u^{(1)}, \ldots, u^{(n)}$, which is exactly what we want.

3. Take the first k columns of the U matrix and compute z

We'll assign the first k columns of U to a variable called 'Ureduce'. This will be an n×k matrix. We compute z with:

$z^{(i)} = Ureduce^T \cdot x^{(i)}$

$UreduceZ^T$ will have dimensions k×n while x(i) will have dimensions n×1. The product $Ureduce^T \cdot x^{(i)}$ will have dimensions k×1.

To summarize, the whole algorithm in octave is roughly:

```
Sigma = (1/m) * X' * X; % compute the covariance matrix
[U,S,V] = svd(Sigma);   % compute our projected directions
Ureduce = U(:,1:k);     % take the first k directions
Z = X * Ureduce;        % compute the projected data points
```

Reconstruction from Compressed Representation

If we use PCA to compress our data, how can we uncompress our data, or go back to our original number of features?

To go from 1-dimension back to 2d we do: $z \in \mathbb{R} \rightarrow x \in \mathbb{R}^2$.

We can do this with the equation: $x_{approx}^{(1)} = U_{reduce} \cdot z^{(1)}$.

Note that we can only get approximations of our original data.

Note: It turns out that the U matrix has the special property that it is a Unitary Matrix. One of the special properties of a Unitary Matrix is:

$U^{-1} = U^*$ where the "*" means "conjugate transpose".

Since we are dealing with real numbers here, this is equivalent to:

$U^{-1} = U^T$ So we could compute the inverse and use that, but it would be a waste of energy and compute cycles.

Choosing the Number of Principal Components

How do we choose k, also called the *number of principal components* ? Recall that k is the dimension we are reducing to.

One way to choose k is by using the following formula:

- Given the average squared projection error: $\frac{1}{m}\sum_{i=1}^{m}||x^{(i)} - x_{approx}^{(i)}||^2$
- Also given the total variation in the data: $\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}||^2$
- Choose k to be the smallest value such that:
  $$\frac{\frac{1}{m}\sum_{i=1}^{m}||x^{(i)} - x_{approx}^{(i)}||^2}{\frac{1}{m}\sum_{i=1}^{m}||x^{(i)}||^2} \leq 0.01$$

In other words, the squared projection error divided by the total variation should be less than one percent, so that **99% of the variance is retained** .

Algorithm for choosing k

1. Try PCA with k=1,2,…
2. Compute U_{reduce}, z, x
3. Check the formula given above that 99% of the variance is retained. If not, go to step one and increase k.

This procedure would actually be horribly inefficient. In Octave, we will call svd:

```
[U,S,V] = svd(Sigma)
```

Which gives us a matrix S. We can actually check for 99% of retained variance using the S matrix as follows:

$$\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{n} S_{ii}} \geq 0.99$$

Advice for Applying PCA

The most common use of PCA is to speed up supervised learning.

Given a training set with a large number of features (e.g. $x^{(1)}, \ldots, x^{(m)} \in \mathbb{R}^{10000}$) we can use PCA to reduce the number of features in each example of the training set (e.g. $z^{(1)}, \ldots, z^{(m)} \in \mathbb{R}^{1000}$).

Note that we should define the PCA reduction from $x^{(i)}$ to $z^{(i)}$ only on the training set and not on the cross-validation or test sets. You can apply the mapping z(i) to your cross-validation and test sets after it is defined on the training set.

Applications

- Compressions

Reduce space of data

Speed up algorithm

- Visualization of data

Choose k = 2 or k = 3

Bad use of PC A: trying to prevent overfitting. We might think that reducing the features with PCA would be an effective way to address overfitting. It might work, but is not recommended because it does not consider the values of our results y. Using just regularization will be at least as effective.

Don't assume you need to do PCA. **Try your full machine learning algorithm without PCA first.** Then use PCA if you find that you need it.

ML:Anomaly Detection

Problem Motivation

Just like in other learning problems, we are given a dataset $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$.

We are then given a new example, x_{test}, and we want to know whether this new example is abnormal/anomalous.

We define a "model" p(x) that tells us the probability the example is not anomalous. We also use a threshold ϵ (epsilon) as a dividing line so we can say which examples are anomalous and which are not.

A very common application of anomaly detection is detecting fraud:

- $x^{(i)} =$ features of user i's activities
- Model p(x) from the data.
- Identify unusual users by checking which have p(x)<ϵ.

If our anomaly detector is flagging **too many** anomalous examples, then we need to **decrease** our threshold ϵ

Gaussian Distribution

The Gaussian Distribution is a familiar bell-shaped curve that can be described by a function $\mathcal{N}(\mu, \sigma^2)$

Let x∈$\mathbb{R}$. If the probability distribution of x is Gaussian with mean μ, variance σ^2, then:

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

The little ~ or 'tilde' can be read as "distributed as."

The Gaussian Distribution is parameterized by a mean and a variance.

Mu, or μ, describes the center of the curve, called the mean. The width of the curve is described by sigma, or σ, called the standard deviation.

The full function is as follows:

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

We can estimate the parameter μ from a given dataset by simply taking the average of all the examples:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

We can estimate the other parameter, σ^2, with our familiar squared error formula:

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} (x^{(i)} - \mu)^2$$

Algorithm

Given a training set of examples, $\{x^{(1)}, \ldots, x^{(m)}\}$ where each example is a vector, $x \in \mathbb{R}^n$.

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

In statistics, this is called an "independence assumption" on the values of the features inside training example x.

More compactly, the above expression can be written as follows:

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

The algorithm

Choose features x_i that you think might be indicative of anomalous examples.

Fit parameters $\mu_1, \ldots, \mu_n, \sigma_1^2, \ldots, \sigma_n^2$

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

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

Given a new example x, compute p(x):

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

Anomaly if p(x)<ϵ

A vectorized version of the calculation for μ is $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$. You can vectorize σ^2 similarly.

Developing and Evaluating an Anomaly Detection System

To evaluate our learning algorithm, we take some labeled data, categorized into anomalous and non-anomalous examples (y = 0 if normal, y = 1 if anomalous).

Among that data, take a large proportion of **good** , non-anomalous data for the training set on which to train p(x).

Then, take a smaller proportion of mixed anomalous and non-anomalous examples (you will usually have many more non-anomalous examples) for your cross-validation and test sets.

For example, we may have a set where 0.2% of the data is anomalous. We take 60% of those examples, all of which are good (y=0) for the training set. We then take 20% of the examples for the cross-validation set (with 0.1% of the anomalous examples) and another 20% from the test set (with another 0.1% of the anomalous).

In other words, we split the data 60/20/20 training/CV/test and then split the anomalous examples 50/50 between the CV and test sets.

Algorithm evaluation:

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

On a cross validation/test example x, predict:

If p(x) < ϵ (**anomaly**), then y=1

If p(x) ≥ ϵ (**normal**), then y=0

Possible evaluation metrics (see "Machine Learning System Design" section):

- True positive, false positive, false negative, true negative.
- Precision/recall
- F_1 score

Note that we use the cross-validation set to choose parameter ϵ

Anomaly Detection vs. Supervised Learning

When do we use anomaly detection and when do we use supervised learning?

Use anomaly detection when...

- We have a very small number of positive examples (y=1 ... 0-20 examples is common) and a large number of negative (y=0) examples.
- We have many different "types" of anomalies and it is 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.

Use supervised learning when...

- We have a large number of both positive and negative examples. In other words, the training set is more evenly divided into classes.
- We have enough positive examples for the algorithm to get a sense of what new positives examples look like. The future positive examples are likely similar to the ones in the training set.

Choosing What Features to Use

The features will greatly affect how well your anomaly detection algorithm works.

We can check that our features are **gaussian** by plotting a histogram of our data and checking for the bell-shaped curve.

Some **transforms** we can try on an example feature x that does not have the bell-shaped curve are:

- log(x)
- log(x+1)
- log(x+c) for some constant
- $\sqrt{x}$
- $x^{1/3}$

We can play with each of these to try and achieve the gaussian shape in our data.

There is an **error analysis procedure** for anomaly detection that is very similar to the one in supervised learning.

Our goal is for p(x) to be large for normal examples and small for anomalous examples.

One common problem is when p(x) is similar for both types of examples. In this case, you need to examine the anomalous examples that are giving high probability in detail and try to figure out new features that will better distinguish the data.

In general, choose features that might take on unusually large or small values in the event of an anomaly.

Multivariate Gaussian Distribution (Optional)

The multivariate gaussian distribution is an extension of anomaly detection and may (or may not) catch more anomalies.

Instead of modeling $p(x_1), p(x_2), \ldots$ separately, we will model p(x) all in one go. Our parameters will be: $\mu \in \mathbb{R}^n$ and $\Sigma \in \mathbb{R}^{n \times n}$

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2} \left|\Sigma\right|^{1/2}} exp(-1/2(x-\mu)^T \Sigma^{-1}(x-\mu))$$

The important effect is that we can model oblong gaussian contours, allowing us to better fit data that might not fit into the normal circular contours.

Varying Σ changes the shape, width, and orientation of the contours. Changing μ will move the center of the distribution.

Check also:

- The Multivariate Gaussian Distribution http://cs229.stanford.edu/section/gaussians.pdf Chuong B. Do, October 10, 2008.

Anomaly Detection using the Multivariate Gaussian Distribution (Optional)

When doing anomaly detection with multivariate gaussian distribution, we compute μ and Σ normally. We then compute p(x) using the new formula in the previous section and flag an anomaly if p(x) < ϵ.

The original model for p(x) corresponds to a multivariate Gaussian where the contours of $p(x; \mu, \Sigma)$ are axis-aligned.

The multivariate Gaussian model can automatically capture correlations between different features of x.

However, the original model maintains some advantages: it is computationally cheaper (no matrix to invert, which is costly for large number of features) and it performs well even with small training set size (in multivariate Gaussian model, it should be greater than the number of features for Σ to be invertible).

ML:Recommender Systems

Problem Formulation

Recommendation is currently a very popular application of machine learning.

Say we are trying to recommend movies to customers. We can use the following definitions

- n_u = number of users
- n_m = number of movies
- $r(i,j) = 1$ if user j has rated movie i
- $y(i,j)$ = rating given by user j to movie i (defined only if r(i,j)=1)

Content Based Recommendations

We can introduce two features, x_1 and x_2 which represents how much romance or how much action a movie may have (on a scale of 0−1).

One approach is that we could do linear regression for every single user. 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.

- $\theta^{(j)}$ = parameter vector for user j
- $x^{(i)}$ = feature vector for movie i

For user j, movie i, predicted rating: $(\theta^{(j)})^T (x^{(i)})$

- $m^{(j)}$ = number of movies rated by user j

To learn $\theta^{(j)}$, we do the following

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

This is our familiar linear regression. The base of the first summation is choosing all i such that $r(i,j) = 1$.

To get the parameters for all our users, we do the following:

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

We can apply our linear regression gradient descent update using the above cost function.

The only real difference is that we **eliminate the constant** $\frac{1}{m}$.

Collaborative Filtering

It can be very difficult to find features such as "amount of romance" or "amount of action" in a movie. To figure this out, we can use *feature finders* .

We can let the users tell us how much they like the different genres, providing their parameter vector immediately for us.

To infer the features from given parameters, we use the squared error function with regularization over all the users:

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

You can also **randomly guess** the values for theta to guess the features repeatedly. You will actually converge to a good set of features.

Collaborative Filtering Algorithm

To speed things up, we can simultaneously minimize our features and our parameters:

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

It looks very complicated, but we've only combined the cost function for theta and the cost function for x.

Because the algorithm can learn them itself, the bias units where x0=1 have been removed, therefore x∈$\mathbb{R}$n and θ∈$\mathbb{R}$n.

These are the steps in the algorithm:

1. Initialize $x^{(i)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)}$ to small random values. This serves to break symmetry and ensures that the algorithm learns features $x^{(i)}, \ldots, x^{(n_m)}$ that are different from each other.
2. Minimize $J(x^{(i)}, \ldots, x^{(n_m)}, \theta^{(1)}, \ldots, \theta^{(n_u)})$ 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 $\theta^T x$.

Vectorization: Low Rank Matrix Factorization

Given matrices X (each row containing features of a particular movie) and Θ (each row containing the weights for those features for a given user), then the full matrix Y of all predicted ratings of all movies by all users is given simply by: $Y = X\Theta^T$.

Predicting how similar two movies i and j are can be done using the distance between their respective feature vectors x. Specifically, we are looking for a small value of $||x^{(i)} - x^{(j)}||$.

Implementation Detail: Mean Normalization

If the ranking system for movies is used from the previous lectures, then new users (who have watched no movies), will be assigned new movies incorrectly. Specifically, they will be assigned θ with all components equal to zero due to the minimization of the regularization term. That is, we assume that the new user will rank all movies 0, which does not seem intuitively correct.

We rectify this problem by normalizing the data relative to the mean. First, we use a matrix Y to store the data from previous ratings, where the ith row of Y is the ratings for the ith movie and the jth column corresponds to the ratings for the jth user.

We can now define a vector

$\mu = [\mu_1, \mu_2, \ldots, \mu_{n_m}]$

such that

$\mu_i = \frac{\sum_{j:r(i,j)=1} Y_{i,j}}{\sum_j r(i,j)}$

Which is effectively the mean of the previous ratings for the ith movie (where only movies that have been watched by users are counted). We now can normalize the data by subtracting u, the mean rating, from the actual ratings for each user (column in matrix Y):

As an example, consider the following matrix Y and mean ratings μ:

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 \\ 4 & ? & ? & 0 \\ 0 & 0 & 5 & 4 \\ 0 & 0 & 5 & 0 \end{bmatrix}, \quad \mu = \begin{bmatrix} 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{bmatrix}$$

The resulting Y′ vector is:

$$Y' = \begin{bmatrix} 2.5 & 2.5 & -2.5 & -2.5 \\ 2 & ? & ? & -2 \\ -.2.25 & -2.25 & 3.75 & 1.25 \\ -1.25 & -1.25 & 3.75 & -1.25 \end{bmatrix}$$

Now we must slightly modify the linear regression prediction to include the mean normalization term:

$(\theta^{(j)})^T x^{(i)} + \mu_i$

Now, for a new user, the initial predicted values will be equal to the μ term instead of simply being initialized to zero, which is more accurate.

Learning with Large Datasets

We mainly benefit from a very large dataset when our algorithm has high variance when m is small. Recall that if our algorithm has high bias, more data will not have any benefit.

Datasets can often approach such sizes as m = 100,000,000. In this case, our gradient descent step will have to make a summation over all one hundred million examples. We will want to try to avoid this -- the approaches for doing so are described below.

Stochastic Gradient Descent

Stochastic gradient descent is an alternative to classic (or batch) gradient descent and is more efficient and scalable to large data sets.

Stochastic gradient descent is written out in a different but similar way:

$$cost(\theta, (x^{(i)}, y^{(i)})) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$

The only difference in the above cost function is the elimination of the m constant within $\frac{1}{2}$.

$$J_{train}(\theta) = \frac{1}{m}\sum_{i=1}^{m} cost(\theta, (x^{(i)}, y^{(i)}))$$

J_{train} is now just the average of the cost applied to all of our training examples.

The algorithm is as follows

1. Randomly 'shuffle' the dataset
2. For $i = 1 \dots m$

$$\Theta_j := \Theta_j - \alpha(h_\Theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$$

This algorithm will only try to fit one training example at a time. This way we can make progress in gradient descent without having to scan all m training examples first. Stochastic gradient descent will be unlikely to converge at the global minimum and will instead wander around it randomly, but usually yields a result that is close enough. Stochastic gradient descent will usually take 1-10 passes through your data set to get near the global minimum.

Mini-Batch Gradient Descent

Mini-batch gradient descent can sometimes be even faster than stochastic gradient descent. Instead of using all m examples as in batch gradient descent, and instead of using only 1 example as in stochastic gradient descent, we will use some in-between number of examples b.

Typical values for b range from 2-100 or so.

For example, with b=10 and m=1000:

Repeat:

For $i = 1, 11, 21, 31, \ldots, 991$

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

We're simply summing over ten examples at a time. The advantage of computing more than one example at a time is that we can use vectorized implementations over the b examples.

Stochastic Gradient Descent Convergence

How do we choose the learning rate α for stochastic gradient descent? Also, how do we debug stochastic gradient descent to make sure it is getting as close as possible to the global optimum?

One strategy is to plot the average cost of the hypothesis applied to every 1000 or so training examples. We can compute and save these costs during the gradient descent iterations.

With a smaller learning rate, it is **possible** that you may get a slightly better solution with stochastic gradient descent. That is because stochastic gradient descent will oscillate and jump around the global minimum, and it will make smaller random jumps with a smaller learning rate.

If you increase the number of examples you average over to plot the performance of your algorithm, the plot's line will become smoother.

With a very small number of examples for the average, the line will be too noisy and it will be difficult to find the trend.

One strategy for trying to actually converge at the global minimum is to **slowly decrease α over time** . For example $\alpha = \frac{const1}{iterationNumber + const2}$

However, this is not often done because people don't want to have to fiddle with even more parameters.

Online Learning

With a continuous stream of users to a website, we can run an endless loop that gets (x,y), where we collect some user actions for the features in x to predict some behavior y.

You can update θ for each individual (x,y) pair as you collect them. This way, you can adapt to new pools of users, since you are continuously updating theta.

Map Reduce and Data Parallelism

We can divide up batch gradient descent and dispatch the cost function for a subset of the data to many different machines so that we can train our algorithm in parallel.

You can split your training set into z subsets corresponding to the number of machines you have. On each of those machines calculate $\sum_{i=p}^{q}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)}$, where we've split the data starting at p and ending at q.

MapReduce will take all these dispatched (or 'mapped') jobs and 'reduce' them by calculating:

$$\Theta_j := \Theta_j - \alpha\frac{1}{z}(temp_j^{(1)} + temp_j^{(2)} + \cdots + temp_j^{(z)})$$

For all $j = 0, \ldots, n$.

This is simply taking the computed cost from all the machines, calculating their average, multiplying by the learning rate, and updating theta.

Your learning algorithm is MapReduceable if it can be *expressed as computing sums of functions over the training set* . Linear regression and logistic regression are easily parallelizable.

For neural networks, you can compute forward propagation and back propagation on subsets of your data on many machines. Those machines can report their derivatives back to a 'master' server that will combine them.