Machine Learning

Stanford University

Week 1

1.1 Introduction

What is Machine Learning?

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

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

Example: playing checkers.

E = the experience of playing many games of checkers

T = the task of playing checkers.

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

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

Supervised learning and Unsupervised learning.

Supervised Learning

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

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

Example 1:

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

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

Example 2:

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

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

Unsupervised Learning

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

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

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

Example:

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

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

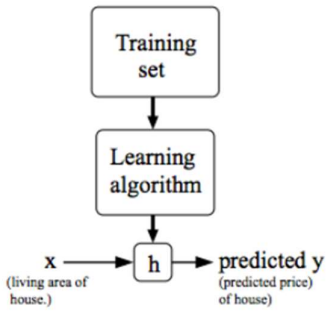
1.2 Linear Regression with One Variable

Model and Cost Function

Model Representation

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

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



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

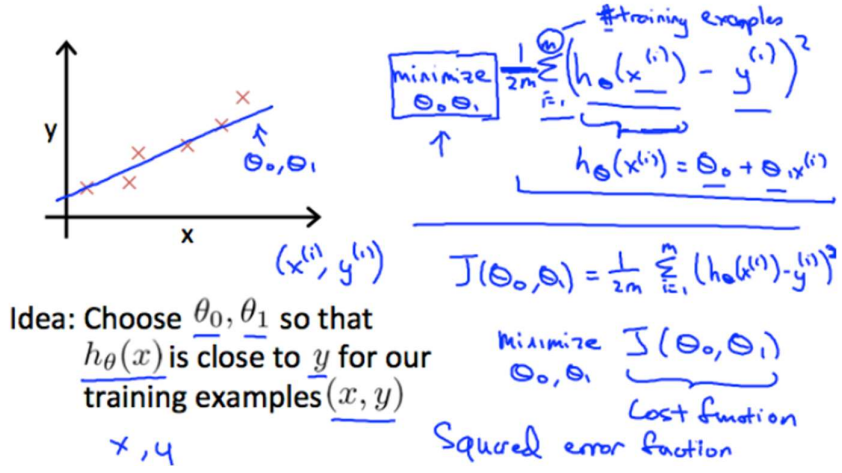
Cost Function

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

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

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.

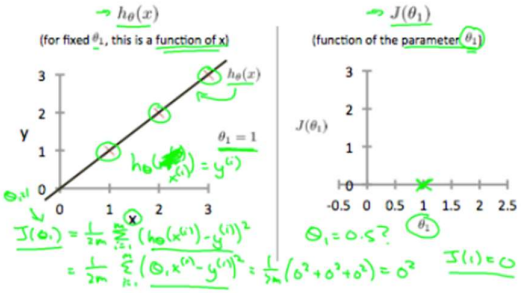
This function is otherwise called the "Squared error function", or "Mean squared error". The mean is halved $\left(\frac{1}{2}\right)$ as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the $\frac{1}{2}$ term. The following image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



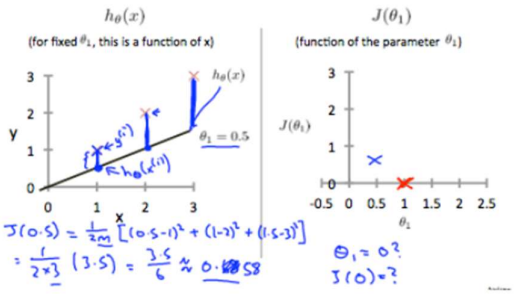
Cost Function: Intuition I

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

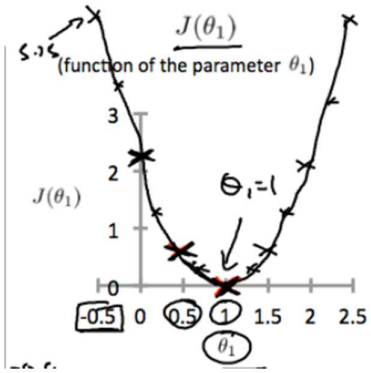
Our objective is to get the best possible line. The best possible line will be such so that the average squared vertical distances of the scattered points from the line will be the least. Ideally, the line should pass through all the points of our training data set. In such a case, the value of $J(\theta_0,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



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



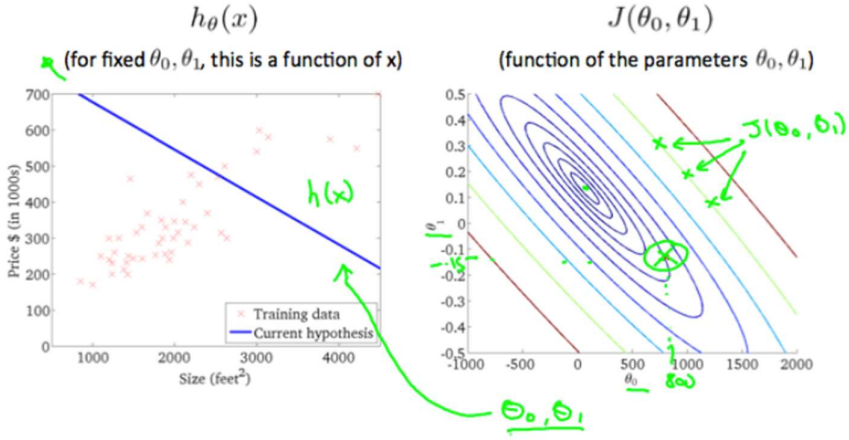
This increases our cost function to 0.58. Plotting several other points yields to the following graph:



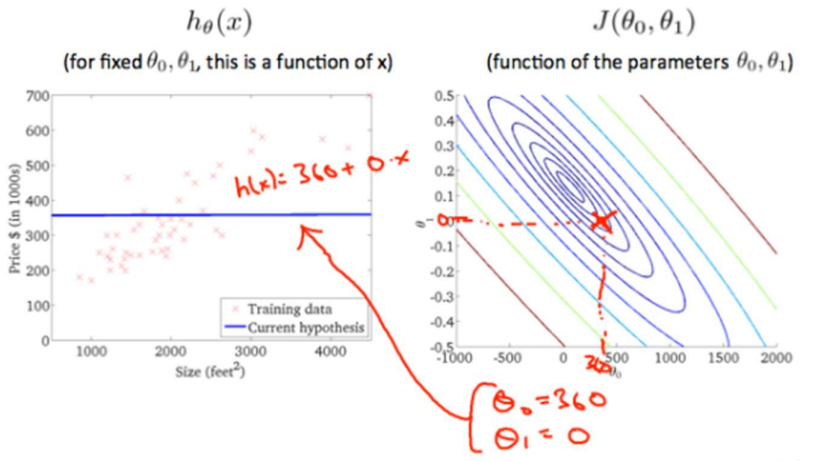
Thus as a goal, we should try to minimize the cost function. In this case, $\theta_1 = 1$ is our global minimum.

Cost Function: Intuition II

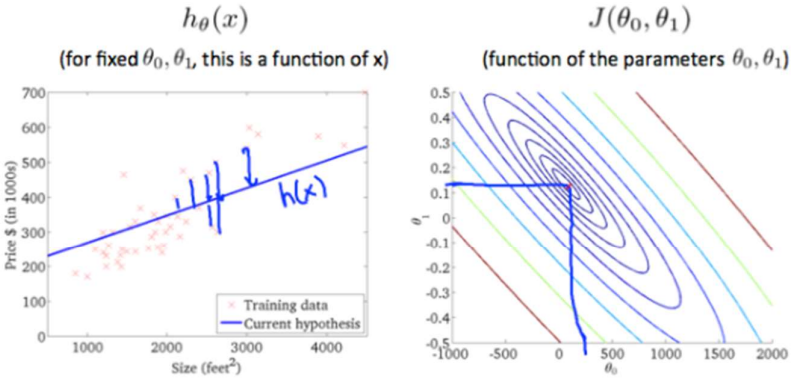
A contour plot is a graph that contains many contour lines. A contour line of a two variable function has a constant value at all points of the same line. An example of such a graph is the one to the right below.



Taking any color and going along the 'circle', one would expect to get the same value of the cost function. For example, the three green points found on the green line above have the same value for $J(\theta_0, \theta_1)$ and as a result, they are found along the same line. The circled x displays the value of the cost function for the graph on the left when θ_0 = 800 and θ_1 = -0.15. Taking another h(x) and plotting its contour plot, one gets the following graphs:



When θ_0 = 360 and θ_1 = 0, the value of $J(\theta_0, \theta_1)$ in the contour plot gets closer to the center thus reducing the cost function error. Now giving our hypothesis function a slightly positive slope results in a better fit of the data.



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

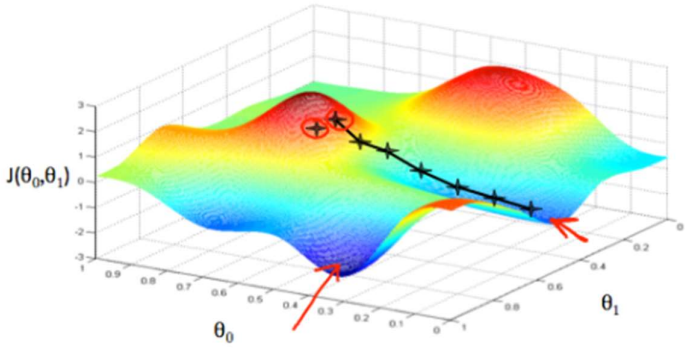
Parameter Learning

Gradient Descent

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

Imagine that we graph our hypothesis function based on its fields θ_0 and θ_1 (actually we are graphing the cost function as a function of the parameter estimates). We are not graphing x and y itself, but the parameter range of our hypothesis function and the cost resulting from selecting a particular set of parameters.

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



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

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

For example, the distance between each 'star' in the graph above represents a step determined by our parameter α. A smaller α would result in a smaller step and a larger α results in a larger step. The direction in which the step is taken is determined by the partial derivative of $J(\theta_0, \theta_1)$. Depending on where one starts on the graph, one could end up at different points. The image above shows us two different starting points that end up in two different places.

The gradient descent algorithm is:

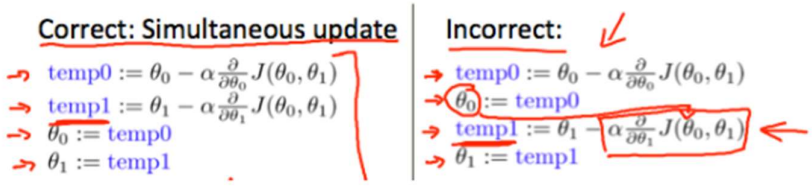
repeat until convergence:

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

where

j=0,1 represents the feature index number.

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



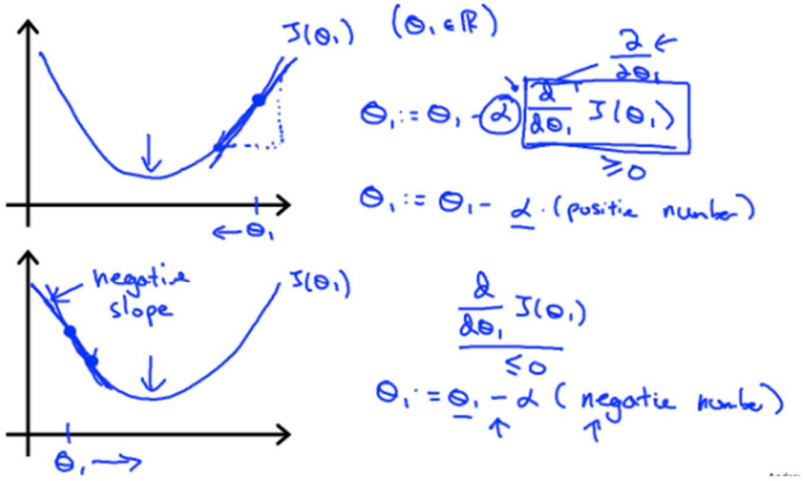
Gradient Descent Intuition

In this video we explored the scenario where we used one parameter θ_1 and plotted its cost function to implement a gradient descent. Our formula for a single parameter was :

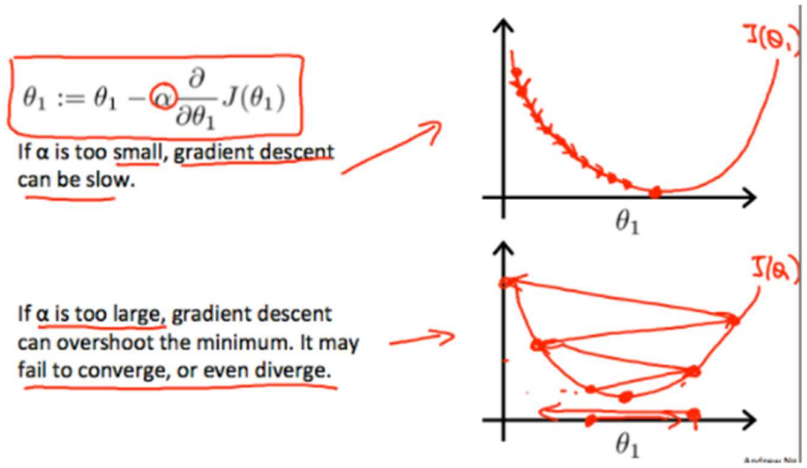
Repeat until convergence:

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

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



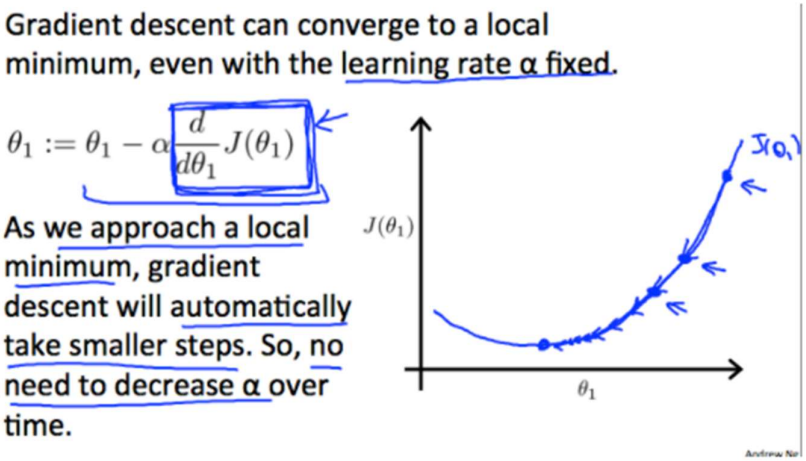
On a side note, we should adjust our parameter α to ensure that the gradient descent algorithm converges in a reasonable time. Failure to converge or too much time to obtain the minimum value imply that our step size is wrong.



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

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

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



Gradient Descent for Linear Regression

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

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

repeat until convergence: {

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

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

}

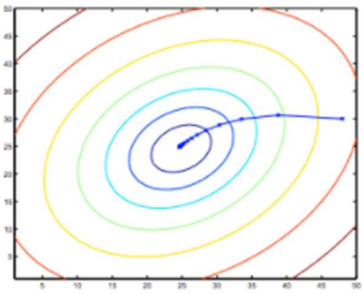
where m is the size of the training set, θ_0 a constant that will be changing simultaneously with θ_1 and x_i , y_i are values of the given training set (data).

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 following is a derivation of $\frac{\partial}{\partial \theta_j} J(\theta)$ for a single example :

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

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

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



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

Linear Algebra Review

Matrices and Vectors

Matrices are 2-dimensional arrays:

$$\begin{bmatrix} a & b & cd & e \\ fg & h & ij & k \\ l \end{bmatrix}$$

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:

$$\begin{bmatrix} wxyz \end{bmatrix}$$

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.

Run the cell below to get familiar with the commands in Octave/Matlab. Feel free to create matrices and vectors and try out different things.

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

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

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

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

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

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

Run

Reset

Addition and Scalar Multiplication

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

$$\begin{bmatrix} a & bc & d \end{bmatrix} + \begin{bmatrix} w & xy & z \end{bmatrix} = \begin{bmatrix} a+w & b+xc+y & d+z \end{bmatrix}$$

Subtracting Matrices:

$$\begin{bmatrix} a & bc & d \end{bmatrix} - \begin{bmatrix} w & xy & z \end{bmatrix} = \begin{bmatrix} a-w & b-xc-y & d-z \end{bmatrix}$$

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

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

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

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

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

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

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

% Initialize constant s
s = 2

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

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

% See how scalar multiplication works
mult_As = A * s

% Divide A by s
div_As = A / s

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

```

Run

Reset

Matrix Vector Multiplication

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

$$\begin{bmatrix} a & bc \\ de & f \end{bmatrix} * \begin{bmatrix} xy \end{bmatrix} = \begin{bmatrix} a*x+b*yc*x+d*ye*x+f*y \end{bmatrix}$$

The result is a **vector**. 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**.

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

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

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

% Multiply A * v
Av = A * v

```

Run

Reset

Matrix-Matrix Multiplication

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

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

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

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

For example:

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

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

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

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

Run

Reset

Matrix Multiplication Properties

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

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

$$\begin{bmatrix} 1 & 0 & 00 & 1 & 00 & 0 & 1 \end{bmatrix}$$

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

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

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

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

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

% How about A*I ?
AI = A*I

% Compute A*B
AB = A*B

% Is it equal to B*A?
BA = B*A

% Note that IA = AI but AB != BA
```

Run

Reset

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 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':

$$A = \begin{bmatrix} a & bc \\ de & f \end{bmatrix}$$

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

In other words:

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

```
% Initialize matrix A
A = [1,2,0;0,5,6;7,0,9]

% Transpose A
A_trans = A'

% Take the inverse of A
A_inv = inv(A)

% What is A^(-1)*A?
A_invA = inv(A)*A
```

Run

Reset

Week 2

2.1 Linear Regression with Multiple Variables

Multivariate Linear Regression

Multiple Features

Note: [7:25 - θ^T is a 1 by (n+1) matrix and not an (n+1) by 1 matrix]

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

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

$$\begin{aligned} x_j^{(i)} &= \text{value of feature } j \text{ in the } i^{th} \text{ training example} \\ x^{(i)} &= \text{the input (features) of the } i^{th} \text{ training example} \\ m &= \text{the number of training examples} \\ n &= \text{the number of features} \end{aligned}$$

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

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

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

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

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

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 we assume $x_0^{(i)} = 1$ for $(i \in 1, \dots, m)$. This allows us to do matrix operations with theta and x. Hence making the two vectors 'θ' and $x^{(i)}$ match each other element-wise (that is, have the same number of elements: n+1).]

Gradient Descent for Multiple Variables

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

repeat until convergence: {

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

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

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

$\dots$

}

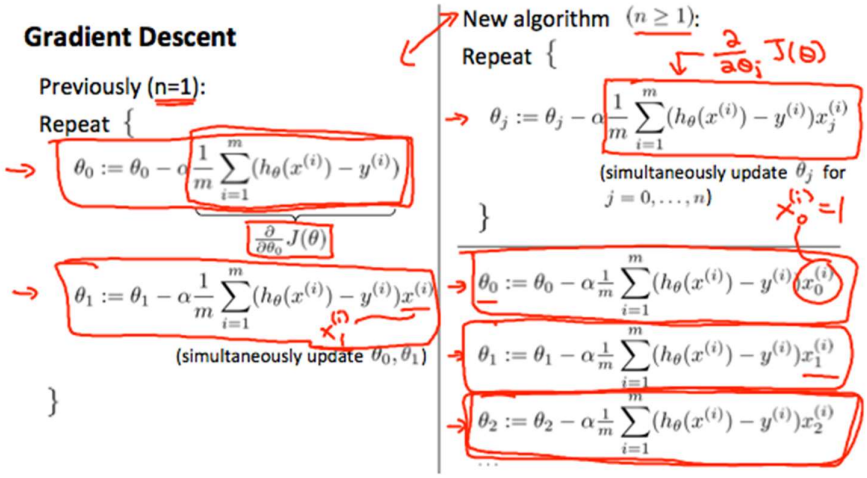
In other words:

repeat until convergence: {

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

}

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



Gradient Descent in Practice I – Feature Scaling

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

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

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

$-1 \leq x_{(i)} \leq 1$

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.

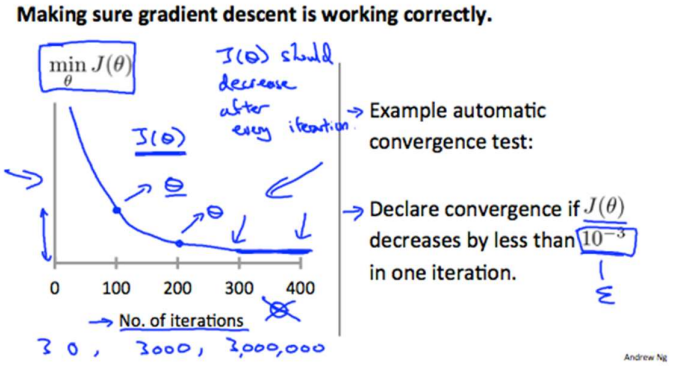
For example, if x_i represents housing prices with a range of 100 to 2000 and a mean value of 1000, then, $x_i := \frac{price - 1000}{1900}$.

Gradient Descent in Practice II – Learning Rate

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

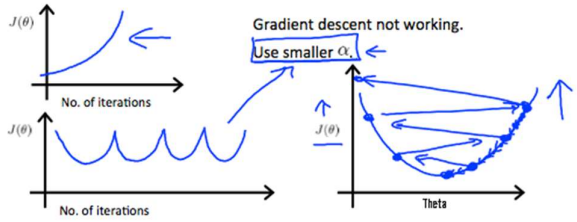
Debugging gradient descent. Make a plot with *number of iterations* on the x-axis. Now plot the cost function, J(θ) over the number of iterations of gradient descent. If J(θ) ever increases, then you probably need to decrease α.

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



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

Making sure gradient descent is working correctly.



- For sufficiently small α, $J(\theta)$ should decrease on every iteration.
- But if α is too small, gradient descent can be slow to converge.

To summarize:

If α is too small: slow convergence.

If α is too large: may not decrease on every iteration and thus may not converge.

Features and Polynomial Regression

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

We can **combine** multiple features into one. For example, we can combine x_1 and x_2 into a new feature 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: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 \sqrt{x_1}$

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

Computing Parameters Analytically

Normal Equation

Note: [8:00 to 8:44 - The design matrix X (in the bottom right side of the slide) given in the example should have elements x with subscript 1 and superscripts varying from 1 to m because for all m training sets there are only 2 features x_0 and x_1. 12:56 - The X matrix is m by (n+1) and NOT n by n.]

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

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

Examples: $m = 4$.

x_0	Size (feet²) x_1	Number of bedrooms x_2	Number of floors x_3	Age of home (years) x_4	Price ($1000) y
1	2104	5	1	45	460
1	1416	3	2	40	232
1	1534	3	2	30	315
1	852	2	1	36	178

$$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix} \qquad y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

m x (n+1)

m-dimensional vector

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

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

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

Gradient Descent	Normal Equation
Need to choose alpha	No need to choose alpha
Needs many iterations	No need to iterate
O (kn^2)	O (n^3), need to calculate inverse of $X^T X$
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 Non-invertibility

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

If $X^T X$ is **noninvertible,** the common causes might be having :

- Redundant features, where two features are very closely related (i.e. 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.

Machine Learning: Programming Exercise 1

Linear Regression

In this exercise, you will implement linear regression and get to see it work on data.

Files needed for this exercise

- `ex1.mlx` - MATLAB Live Script that steps you through the exercise
- ex1data1.txt - Dataset for linear regression with one variable
- ex1data2.txt - Dataset for linear regression with multiple variables
- `submit.m` - Submission script that sends your solutions to our servers
- *`warmUpExercise.m` - Simple example function in MATLAB
- *`plotData.m` - Function to display the dataset
- *`computeCost.m` - Function to compute the cost of linear regression
- *`gradientDescent.m` - Function to run gradient descent
- **`computeCostMulti.m` - Cost function for multiple variables
- **`gradientDescentMulti.m` - Gradient descent for multiple variables
- **`featureNormalize.m` - Function to normalize features
- **`normalEqn.m` - Function to compute the normal equations

indicates files you will need to complete

****indicates optional exercises***

Clear existing variables and confirm that your Current Folder is set correctly

Click into this section, then click the 'Run Section' button above. This will execute the `clear` command to clear existing variables and the `dir` command to list the files in your Current Folder. The output should contain all of the files listed above and the 'lib' folder. If it does not, right-click the 'ex1' folder and select 'Open' before proceding or see the instructions in `README.mlx` for more details.

```
pwd
```

```
ans = '/MATLAB Drive/machine-learning-ex/ex1'
```

```
clear
dir
```

```
.                       computeCostMulti.m      ex1data1.txt            featureNormalize.m
gradientDescentMulti.m  plotData.m              warmUpExercise.m
..                      ex1.mlx                 ex1data2.txt            figure5.pdf
lib                     submit.m
computeCost.m           ex1_companion.mlx       ex5                     gradientDescent.m
normalEqn.m             token.mat
```

Before you begin

The workflow for completing and submitting the programming exercises in MATLAB Online differs from the original course instructions. Before beginning this exercise, make sure you have read through the instructions in `README.mlx` which is included with the programming exercise files. `README.mlx` also contains solutions to the many common issues you may encounter while completing and submitting the exercises in MATLAB Online. Make sure you are following instructions in `README.mlx` and have checked for an existing solution before seeking help on the discussion forums.

Table of Contents

Linear Regression
  Files needed for this exercise
    Clear existing variables and confirm that your Current Folder is set correctly
  Before you begin
1. A simple MATLAB function
  1.1 Submitting Solutions
2. Linear regression with one variable
  2.1 Plotting the data
  2.2 Gradient Descent
    2.2.1 Update Equations
    2.2.2 Implementation
    2.2.3 Computing the cost
    2.2.4 Gradient descent
  2.3 Debugging
  2.4 Visualizing
Optional Exercises:
3. Linear regression with multiple variables
  3.1 Feature Normalization
    Add the bias term
  3.2 Gradient Descent
    3.2.1 Optional (ungraded) exercise: Selecting learning rates
  3.3 Normal Equations
Submission and Grading

1. A simple MATLAB function

The first part of this script gives you practice with MATLAB syntax and the homework submission process. In the file `warmUpExercise.m`, you will find the outline of a MATLAB function. Modify it to return a 5 x 5 identity matrix by filling in the following code:

```
A = eye(5);
```

When you are finished, save `warmUpExercise.m`, then run the code contained in this section to call `warmUpExercise()`.

5x5 Identity Matrix:

```
warmUpExercise()
```

```
ans = 5×5
     1     0     0     0     0
     0     1     0     0     0
```

```
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

You should see output similar to the following:

```
ans =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

You can toggle between right-hand-side output and in-line output for printing results and displaying figures inside a Live Script by selecting the appropriate box in the upper right of the Live Editor window.

1.1 Submitting Solutions

After completing a part of the exercise, you can submit your solutions for that section by running the section of code below, which calls the `submit.m` script. Your score for each section will then be displayed as output. **Enter your login and your unique submission token *inside the command window* when prompted. For future submissions of this exercise, you will only be asked to confirm your credentials.** Your submission token for each exercise is found in the corresponding homework assignment course page. New tokens can be generated if you are experiencing issues with your current token. You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration.

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter your login and token when prompted.*

2. Linear regression with one variable

In this part of this exercise, you will implement linear regression with one variable to predict profits for a food truck. Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet. The chain already has trucks in various cities and you have data for profits and populations from the cities. You would like to use this data to help you select which city to expand to next.

The file `ex1data1.txt` contains the dataset for our linear regression problem. The first column is the population of a city and the second column is the profit of a food truck in that city. A negative value for profit indicates a loss. This script has already been set up to load this data for you.

2.1 Plotting the data

Before starting on any task, it is often useful to understand the data by visualizing it. For this dataset, you can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population). Many other problems that you will encounter in real life are multi-dimensional and can't be plotted on a 2-d plot.

Run the code below to load the dataset from the data file into the variables X and y:

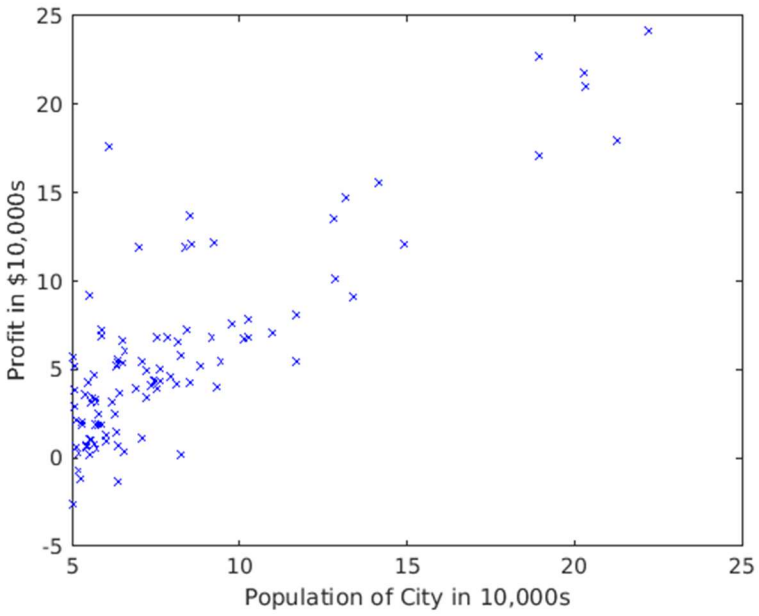
```
data = load('ex1data1.txt'); % read comma separated data
X = data(:, 1); y = data(:, 2);
```

Your job is to complete **`plotData.m`** to draw the plot; modify the file and fill in the following code:

```
plot(x, y, 'rx', 'MarkerSize', 10); % Plot the data
ylabel('Profit in $10,000s'); % Set the y-axis label
xlabel('Population of City in 10,000s'); % Set the x-axis label
```

Once you are finished, save `plotData.m`, and execute the code in this section which will call `plotData`.

```
plotData(X,y)
```



The resulting plot should appear as in Figure 1 below:

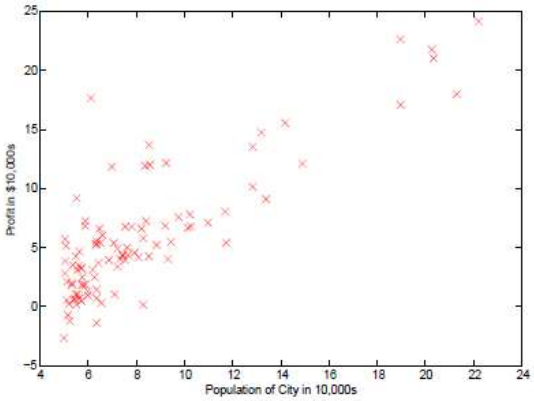


Figure 1: Scatter plot of training data

To learn more about the `plot` command, you can run the command **`help plot`** at the command prompt, type `plot()` inside the MATLAB Live Editor and click on the "(?)" tooltip, or you can search the MATLAB documentation for "plot". Note that to change the markers to red x's in the plot, we used the option: **`'rx'`** together with the **`plot`** command, i.e.,

```
plot(..,[your options here],..,'rx');
```

2.2 Gradient Descent

In this section, you will fit the linear regression parameters to our dataset using gradient descent.

2.2.1 Update Equations

The objective of linear regression is to minimize the cost function

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

where the hypothesis $h_\theta(x)$ is given by the linear model

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

Recall that the parameters of your model are the θ values. These are the values you will adjust to minimize cost $J(\theta)$. One way to do this is to use the batch gradient descent algorithm. In batch gradient descent, each iteration performs the update

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \qquad (\text{simultaneously update } \theta_j \text{ for all } j)$$

With each step of gradient descent, your parameters j come closer to the optimal values that will achieve the lowest cost $J(\theta)$.

Implementation Note: We store each example as a row in the the `X` matrix in MATLAB. To take into account the intercept term (θ_0), we add an additional first column to **`X`** and set it to all ones. This allows us to treat θ_0 as simply another 'feature'.

2.2.2 Implementation

In this script, we have already set up the data for linear regression. In the following lines, we add another dimension to our data to accommodate the θ_0 intercept term. Run the code below to initialize the parameters to 0 and the learning rate `alpha` to 0.01.

```
m = length(X); % number of training examples
X = [ones(m,1),data(:,1)]; % Add a column of ones to x
theta = zeros(2, 1); % initialize fitting parameters
iterations = 1500;
alpha = 0.01;
```

2.2.3 Computing the cost $J(\theta)$

As you perform gradient descent to minimize the cost function $J(\theta)$, it is helpful to monitor the convergence by computing the cost. In this section, you will implement a function to calculate $J(\theta)$ so you can check the convergence of your gradient descent implementation.

Your next task is to complete the code in the file `computeCost.m`, which is a function that computes $J(\theta)$. As you are doing this, remember that the variables `X` and `y` are not scalar values, but matrices whose rows represent the examples from the training set.

Once you have completed the function definition, run this section. The code below will call `computeCost` once using θ initialized to zeros, and you will see the cost printed to the screen. You should expect to see a cost of 32.07 for the first output below:

```
% Compute and display initial cost with theta all zeros
computeCost(X, y, theta)
```

```
ans = 32.0727
```

Next we call `computeCost` again, this time with non-zero `theta` values as an additional test. You should expect to see an output of 54.24 below:

```
% Compute and display initial cost with non-zero theta
computeCost(X, y,[-1; 2])
```

```
ans = 54.2425
```

If the outputs above match the expected values, you can submit your solution for assessment. If the outputs do not match or you receive an error, check your cost function for mistakes, then rerun this section once you have addressed them.

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

2.2.4 Gradient descent

Next, you will implement gradient descent in the file `gradientDescent.m`. The loop structure has been written for you, and you only need to supply the updates to θ within each iteration.

As you program, make sure you understand what you are trying to optimize and what is being updated. Keep in mind that the cost $J(\theta)$ is parameterized by the vector θ, not X and y. That is, we minimize the value of $J(\theta)$ by changing the values of the vector θ, not by changing X or y. Refer to the equations given earlier and to the video lectures if you are uncertain.

A good way to verify that gradient descent is working correctly is to look at the value of `J` and check that it is decreasing with each step. The starter code for `gradientDescent.m` calls `computeCost` on every iteration and prints the cost. Assuming you have implemented gradient descent and `computeCost` correctly, your value of $J(\theta)$ should never increase, and should converge to a steady value by the end of the algorithm.

After you are finished, run this execute this section. The code below will use your final parameters to plot the linear fit. The result should look something like Figure 2 below:

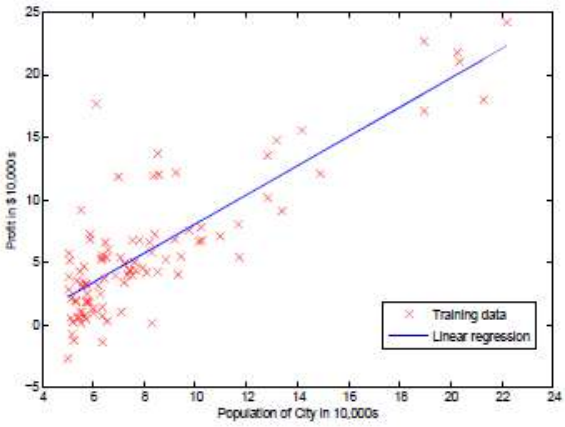


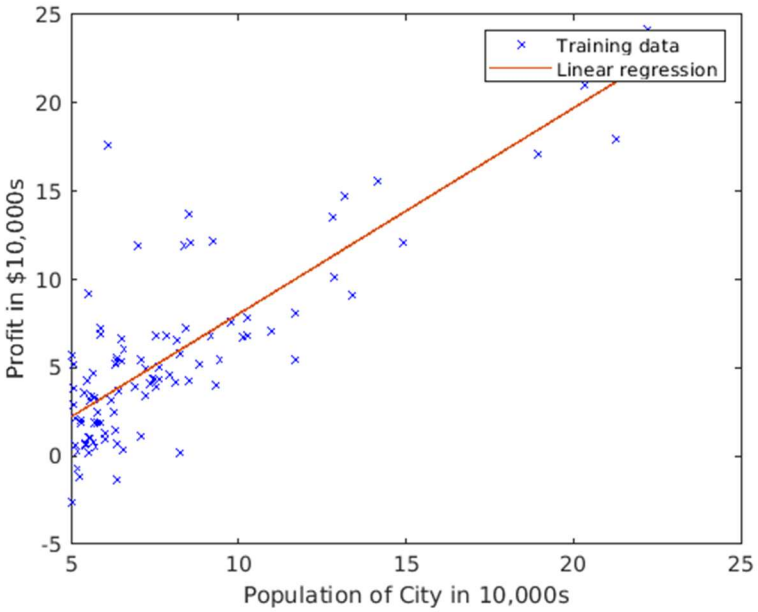
Figure 2: Training data with linear regression fit

Your final values for θ will also be used to make predictions on profits in areas of 35,000 and 70,000 people.

```
% Run gradient descent:
% Compute theta
theta = gradientDescent(X, y, theta, alpha, iterations);

% Print theta to screen
% Display gradient descent's result
fprintf('Theta computed from gradient descent:\n%f,\n%f',theta(1),theta(2))
```

```
Theta computed from gradient descent:
-3.630291,
1.166362
```

```
% Plot the linear fit
hold on; % keep previous plot visible
plot(X(:,2), X*theta, '-')
legend('Training data', 'Linear regression')
hold off % don't overlay any more plots on this figure
```




```
% Predict values for population sizes of 35,000 and 70,000
predict1 = [1, 3.5] *theta;
fprintf('For population = 35,000, we predict a profit of %f\n',
predict1*10000);
```

```
For population = 35,000, we predict a profit of 4519.767868
```

```
predict2 = [1, 7] * theta;
fprintf('For population = 70,000, we predict a profit of %f\n',
predict2*10000);
```

```
For population = 70,000, we predict a profit of 45342.450129
```

Note the way that the lines above use matrix multiplication, rather than explicit summation or looping, to calculate the predictions. This is an example of *code vectorization* in MATLAB.

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

2.3 Debugging

Here are some things to keep in mind as you implement gradient descent:

- MATLAB array indices start from one, not zero. If you're storing θ_0 and θ_1 in a vector called **`theta`**, the values will be **`theta(1)`** and **`theta(2)`**.
- If you are seeing many errors at runtime, inspect your matrix operations to make sure that you're adding and multiplying matrices of compatible dimensions. Printing the dimensions of variables with the `size` command will help you debug.

- By default, MATLAB interprets math operators to be matrix operators. This is a common source of size incompatibility errors. If you don't want matrix multiplication, you need to add the "dot" notation to specify this to MATLAB. For example, `A*B` does a matrix multiply, while `A.*B` does an element-wise multiplication.

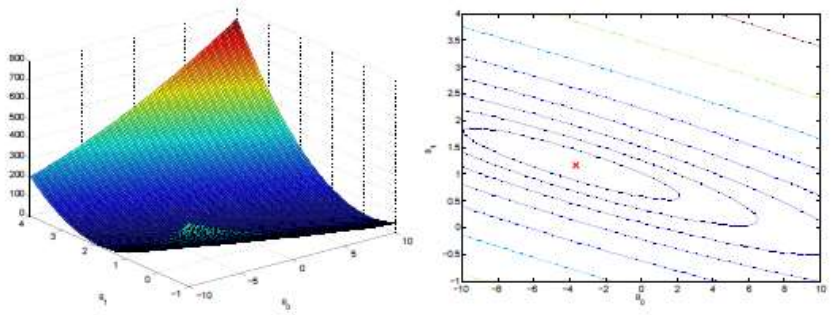
2.4 Visualizing $J(\theta)$

To understand the cost function $J(\theta)$ better, you will now plot the cost over a 2-dimensional grid of θ_0 and θ_1 values. You will not need to code anything new for this part, but you should understand how the code you have written already is creating these images.

In the next step, there is code set up to calculate $J(\theta)$ over a grid of values using the `computeCost` function that you wrote.

```
% Visualizing J(theta_0, theta_1):
% Grid over which we will calculate J
theta0_vals = linspace(-10, 10, 100);
theta1_vals = linspace(-1, 4, 100);

% initialize J_vals to a matrix of 0's
J_vals = zeros(length(theta0_vals), length(theta1_vals));

% Fill out J_vals
for i = 1:length(theta0_vals)
    for j = 1:length(theta1_vals)
       t = [theta0_vals(i); theta1_vals(j)];
       J_vals(i,j) = computeCost(X, y, t);
    end
end
```

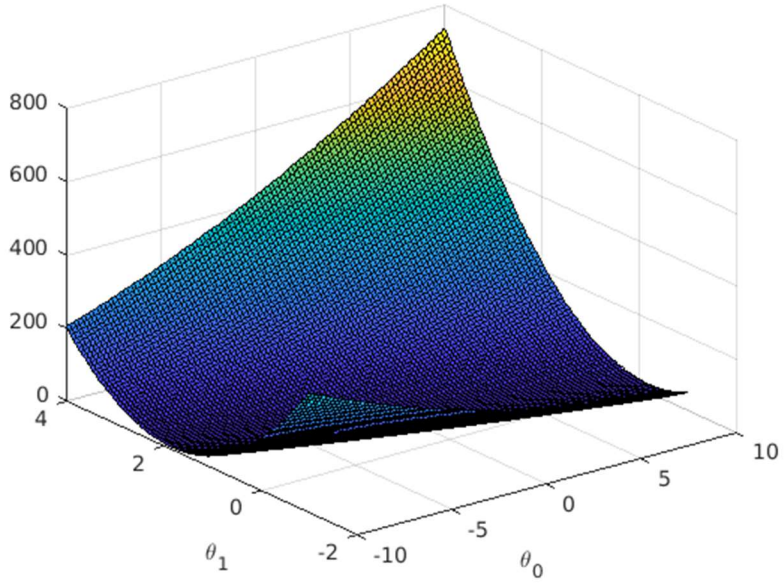
After the code above is executed, you will have a 2-D array of $J(\theta)$ values. The code below will then use these values to produce surface and contour plots of $J(\theta)$ using the **`surf`** and **`contour`** commands. Run the code in this section now. The resulting plots should look something like the figure below.



(a) Surface (b) Contour, showing minimum

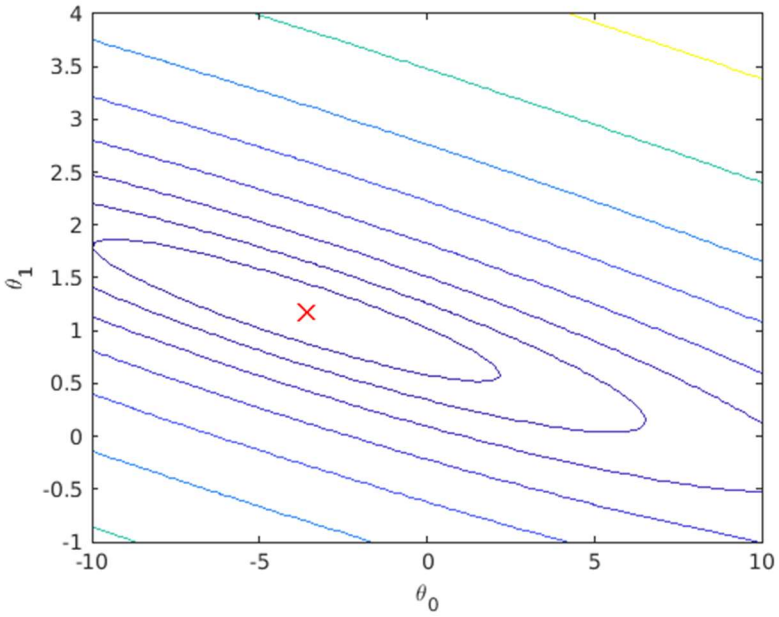
Figure 3: Cost function $J(\theta)$

```
% Because of the way meshgrids work in the surf command, we need to
% transpose J_vals before calling surf, or else the axes will be flipped
J_vals = J_vals';

% Surface plot
figure;
surf(theta0_vals, theta1_vals, J_vals)
xlabel('\theta_0'); ylabel('\theta_1');
```




```
% Contour plot
figure;
% Plot J_vals as 15 contours spaced logarithmically between 0.01 and 100
contour(theta0_vals, theta1_vals, J_vals, logspace(-2, 3, 20))
```

```
xlabel('\theta_0'); ylabel('\theta_1');
hold on;
plot(theta(1), theta(2), 'rx', 'MarkerSize', 10, 'LineWidth', 2);
hold off;
```



The purpose of these graphs is to show you that how $J(\theta)$ varies with changes in θ_0 and θ_1. The cost function $J(\theta)$ is bowl-shaped and has a global mininum. (This is easier to see in the contour plot than in the 3D surface plot). This minimum is the optimal point for θ_0 and θ_1, and each step of gradient descent moves closer to this point.

Optional Exercises:

If you have successfully completed the material above, congratulations! You now understand linear regression and should able to start using it on your own datasets. For the rest of this programming exercise, we have included the following optional exercises. These exercises will help you gain a deeper understanding of the material, and if you are able to do so, we encourage you to complete them as well.

3. Linear regression with multiple variables

In this part, you will implement linear regression with multiple variables to predict the prices of houses. Suppose you are selling your house and you want to know what a good market price would be. One way to do this is to first collect information on recent houses sold and make a model of housing prices.

The file `ex1data2.txt` contains a training set of housing prices in Portland, Oregon. The first column is the size of the house (in square feet), the second column is the number of bedrooms, and the third column is the price of the house. Run this section now to preview the data.

```
% Load Data
data = load('ex1data2.txt');
X = data(:, 1:2);
y = data(:, 3);
m = length(y);

% Print out some data points
% First 10 examples from the dataset
fprintf(' x = [%.0f %.0f], y = %.0f \n', [X(1:10,:) y(1:10,:)]');
```

```
 x = [2104 3], y = 399900
 x = [1600 3], y = 329900
 x = [2400 3], y = 369000
 x = [1416 2], y = 232000
 x = [3000 4], y = 539900
 x = [1985 4], y = 299900
 x = [1534 3], y = 314900
 x = [1427 3], y = 198999
 x = [1380 3], y = 212000
 x = [1494 3], y = 242500
```

The remainder of this script has been set up to help you step through this exercise.

3.1 Feature Normalization

This section of the script will start by loading and displaying some values from this dataset. By looking at the values, note that house sizes are about 1000 times the number of bedrooms. When features differ by orders of magnitude, first performing feature scaling can make gradient descent converge much more quickly.

Your task here is to complete the code in `featureNormalize.m` to:

- Subtract the mean value of each feature from the dataset.
- After subtracting the mean, additionally scale (divide) the feature values by their respective "standard deviations".

The standard deviation is a way of measuring how much variation there is in the range of values of a particular feature (most data points will lie within ± 2 standard deviations of the mean); this is an alternative to taking the range of values (*max* - *min*). In MATLAB, you can use the `std` function to compute the standard deviation. For example, inside `featureNormalize.m`, the quantity `X(:,1)` contains all the values of x_1 (house sizes) in the training set, so `std(X(:,1))` computes the standard deviation of the house sizes. At the time that `featureNormalize.m` is called, the extra column of 1's corresponding to $x_0 = 1$ has not yet been added to `X` (see the code below for details).

You will do this for all the features and your code should work with datasets of all sizes (any number of features / examples). Note that each column of the matrix **`X`** corresponds to one feature. When you are finished with `featureNormailize.m`, run this section to normailze the features of the housing dataset.

```
% Scale features and set them to zero mean
[X, mu, sigma] = featureNormalize(X);
mu
```

```
mu = 1×2
10^3 ×
    2.0007    0.0032
```

```
sigma
```

```
sigma = 1×2
  794.7024    0.7610
```

Implementation Note: When normalizing the features, it is important to store the values used for normalization - the mean value and the standard deviation used for the computations. After learning the parameters from the model, we often want to predict the prices of houses we have not seen before. Given a new x value (living room area and number of bedrooms), we must first normalize x using the mean and standard deviation that we had previously computed from the training set.

Add the bias term

Now that we have normailzed the features, we again add a column of ones corresponding to θ_0 to the data matrix `X`.

```
% Add intercept term to X
X = [ones(m, 1) X];
```

3.2 Gradient Descent

Previously, you implemented gradient descent on a univariate regression problem. The only difference now is that there is one more feature in the matrix `X`. The hypothesis function and the batch gradient descent update rule remain unchanged.

You should complete the code in `computeCostMulti.m` and `gradientDescentMulti.m` to implement the cost function and gradient descent for linear regression *with multiple variables*. If your code in the previous part (single variable) already supports multiple variables, you can use it here too.

Make sure your code supports any number of features and is well-vectorized. You can use the command `size(X,2)` to find out how many features are present in the dataset.

We have provided you with the following starter code below that runs gradient descent with a particular learning rate (`alpha`). Your task is to first make sure that your functions `computeCost` and `gradientDescent` already work with this starter code and support multiple variables.

Implementation Note: In the multivariate case, the cost function can also be written in the following vectorized form:

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

where

$$X = \begin{bmatrix} - (x^{(1)})^T - \\ - (x^{(2)})^T - \\ \vdots \\ - (x^{(m)})^T - \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$$

The vectorized version is efficient when you're working with numerical computing tools like MATLAB. If you are an expert with matrix operations, you can prove to yourself that the two forms are equivalent.

```
% Run gradient descent
% Choose some alpha value
alpha = 0.1;
num_iters = 400;

% Init Theta and Run Gradient Descent
theta = zeros(3, 1);
[theta, ~] = gradientDescentMulti(X, y, theta, alpha, num_iters);

% Display gradient descent's result
fprintf('Theta computed from gradient
descent:\n%f\n%f\n%f',theta(1),theta(2),theta(3))
```

```
Theta computed from gradient descent:
340412.659574
110631.048958
-6649.472950
```

Finally, you should complete and run the code below to predict the price of a 1650 sq-ft, 3 br house using the value of `theta` obtained above.

Hint: At prediction, make sure you do the same feature normalization. Recall that the first column of `X` is all ones. Thus, it does not need to be normalized.

```
% Estimate the price of a 1650 sq-ft, 3 br house
% ====================== YOUR CODE HERE ======================

price = X*theta; % Enter your price formula here

% ============================================================

fprintf('Predicted price of a 1650 sq-ft, 3 br house (using gradient
descent):\n $%f', price);
```

```
Predicted price of a 1650 sq-ft, 3 br house (using gradient descent):
$230854.294079
```

3.2.1 Optional (ungraded) exercise: Selecting learning rates

In this part of the exercise, you will get to try out dierent learning rates for the dataset and find a learning rate that converges quickly. You can change the learning rate by modifying the code below and changing the part of the code that sets the learning rate.

The code below will call your `gradientDescent` function and run gradient descent for about 50 iterations at the chosen learning rate. The function should also return the history of $J(\theta)$ values in a vector `J`. After the last iteration, the code plots the `J` values against the number of the iterations. If you picked a learning rate within a good range, your plot should look similar Figure 4 below.

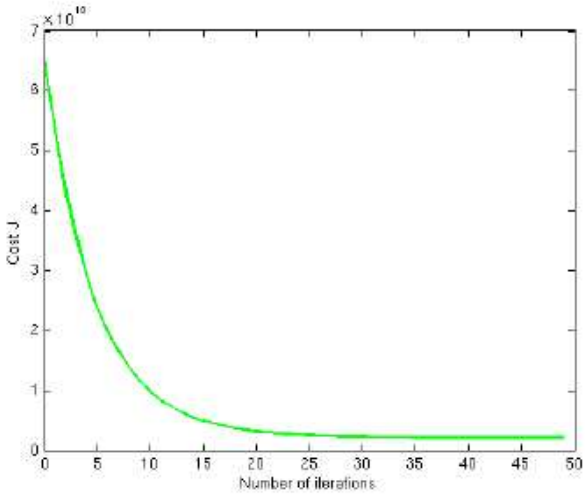


Figure 4: Convergence of gradient descent with an appropriate learning rate

If your graph looks very different, especially if your value of $J(\theta)$ increases or even blows up, use the control to adjust your learning rate and try again. We recommend trying values of the learning rate on a log-scale, at multiplicative steps of about 3 times the previous value (i.e., 0.3, 0.1, 0.03, 0.01 and so on). You may also want to adjust the number of iterations you are running if that will help you see the overall trend in the curve.

Implementation Note: If your learning rate is too large, $J(\theta)$ can diverge and 'blow up', resulting in values which are too large for computer calculations. In these situations, MATLAB will tend to return `NaNs`. `NaN` stands for 'not a number' and is often caused by undefined operations that involve $\pm\infty$.

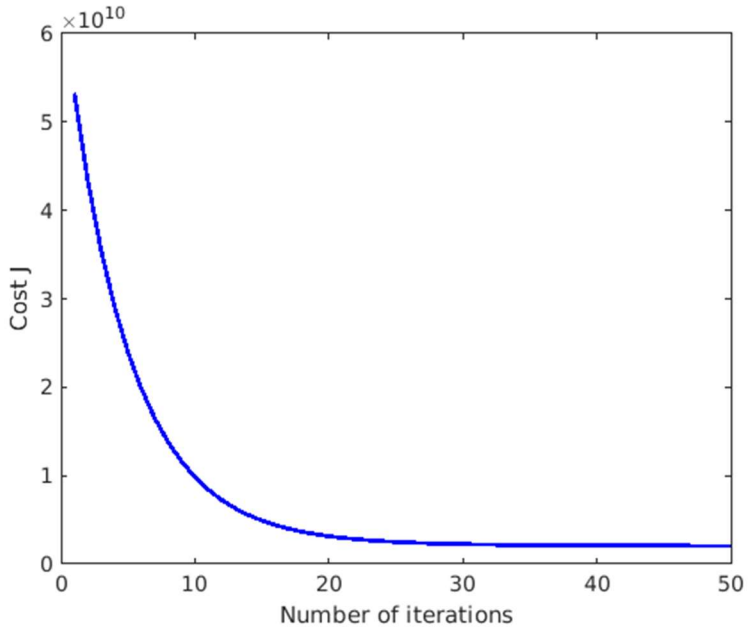
MATLAB Tip: To compare how different learning learning rates affect convergence, it's helpful to plot `J` for several learning rates on the same figure. In MATLAB, this can be done by performing gradient descent multiple times with a `hold on` command between plots. Make sure to use the `hold off` command when you are done plotting in that figure. Concretely, if you've tried three different values of `alpha` (you should probably try more values than this) and stored the costs in `J1`, `J2` and `J3`, you can use the following commands to plot them on the same figure:

```
plot(1:50, J1(1:50), 'b');
hold on
plot(1:50, J2(1:50), 'r');
plot(1:50, J3(1:50), 'k');
```

```
hold off
```

The final arguments `'b'`,`'r'`, and `'k'` specify different colors for the plots. If desired, you can use this technique and adapt the code below to plot multiple convergence histories in the same plot.

```
% Run gradient descent:
% Choose some alpha value
alpha = 0.1;
num_iters = 50;

% Init Theta and Run Gradient Descent
theta = zeros(3, 1);
[~, J_history] = gradientDescentMulti(X, y, theta, alpha, num_iters);

% Plot the convergence graph
plot(1:num_iters, J_history, '-b', 'LineWidth', 2);
xlabel('Number of iterations');
ylabel('Cost J');
```



Notice the changes in the convergence curves as the learning rate changes. With a small learning rate, you should find that gradient descent takes a very long time to converge to the optimal value. Conversely, with a large learning rate, gradient descent might not converge or might even diverge!

Using the best learning rate that you found, run the section of code below, which will run gradient descent until convergence to find the final values of θ. Next, use this value of θ to predict the price of a house with 1650 square feet and 3 bedrooms. You will use value later to check your implementation of the normal equations. Don't forget to normalize your features when you make this prediction!

```
% Run gradient descent
% Replace the value of alpha below best alpha value you found above
alpha = 0.1;
num_iters = 50;

% Init Theta and Run Gradient Descent
theta = zeros(3, 1);
[theta, ~] = gradientDescentMulti(X, y, theta, alpha, num_iters);

% Display gradient descent's result
fprintf('Theta computed from gradient
descent:\n%f\n%f\n%f',theta(1),theta(2),theta(3))
```

```
Theta computed from gradient descent:
338658.249249
104127.515597
-172.205334
```

```
% Estimate the price of a 1650 sq-ft, 3 br house. You can use the same
% code you entered ealier to predict the price
% ====================== YOUR CODE HERE ======================

price = X*theta; % Enter your price formula here

% ============================================================

fprintf('Predicted price of a 1650 sq-ft, 3 br house (using gradient
descent):\n $%f', price);
```

```
Predicted price of a 1650 sq-ft, 3 br house (using gradient descent):
$234178.987928
```

3.3 Normal Equations

In the lecture videos, you learned that the closed-form solution to linear regression is

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

Using this formula does not require any feature scaling, and you will get an exact solution in one calculation: there is no "loop until convergence" like in gradient descent.

Complete the code in `normalEqn.m` to use the formula above to calculate θ, then run the code in this section. Remember that while you don't need to scale your features, we still need to add a column of 1's to the `X` matrix to have an intercept term (θ_0) . Note that the code below will add the column of 1's to `X` for you.

```
% Solve with normal equations:
% Load Data
data = csvread('ex1data2.txt');
X = data(:, 1:2);
y = data(:, 3);
m = length(y);

% Add intercept term to X
X = [ones(m, 1) X];

% Calculate the parameters from the normal equation
theta = normalEqn(X, y);

% Display normal equation's result
fprintf('Theta computed from the normal equations:\n%f\n%f\n%f',
theta(1),theta(2),theta(3));
```

```
Theta computed from the normal equations:
89597.909544
139.210674
-8738.019113
```

Optional (ungraded) exercise: Now, once you have found θ using this method, use it to make a price prediction for a 1650-square-foot house with 3 bedrooms. You should find that gives the same predicted price as the value you obtained using the model fit with gradient descent (in Section 3.2.1).

```
% Estimate the price of a 1650 sq-ft, 3 br house.
% ====================== YOUR CODE HERE ======================

price = [1, 1650, 3]*theta; % Enter your price forumla here

% ============================================================

fprintf('Predicted price of a 1650 sq-ft, 3 br house (using normal
equations):\n $%f', price);
```

```
Predicted price of a 1650 sq-ft, 3 br house (using normal equations):
 $293081.464335
```

computeCost.m :

```
function J = computeCost(X, y, theta)
  %COMPUTECOST Compute cost for linear regression
  %   J = COMPUTECOST(X, y, theta) computes the cost of using theta as the
  %   parameter for linear regression to fit the data points in X and y

  % Initialize some useful values
  m = length(y); % number of training examples

  % You need to return the following variables correctly
  J = 0;

  % ====================== YOUR CODE HERE ======================
  % Instructions: Compute the cost of a particular choice of theta
  %               You should set J to the cost.

  %%%%%%%%%%%%% CORRECT %%%%%%%%%
  % h = X*theta;
  % temp = 0;
  % for i=1:m
  %   temp = temp + (h(i) - y(i))^2;
  % end
  % J = (1/(2*m)) * temp;
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  %%%%%%%%%%%%% CORRECT: Vectorized Implementation %%%%%%%%%
  J = (1/(2*m))*sum(((X*theta)-y).^2);
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % =========================================================================
end
```

gradientDescent.m :

```
function [theta, J_history] = gradientDescent(X, y, theta, alpha, num_iters)
  %GRADIENTDESCENT Performs gradient descent to learn theta
  %   theta = GRADIENTDESCENT(X, y, theta, alpha, num_iters) updates theta by
  %   taking num_iters gradient steps with learning rate alpha

  % Initialize some useful values
  m = length(y); % number of training examples
  J_history = zeros(num_iters, 1);

  for iter = 1:num_iters

   % ====================== YOUR CODE HERE ======================
   % Instructions: Perform a single gradient step on the parameter vector
   %               theta.
   %
   % Hint: While debugging, it can be useful to print out the values
   %       of the cost function (computeCost) and gradient here.

   %%%%%%%%% CORRECT %%%%%%%
   %error = (X * theta) - y;
   %temp0 = theta(1) - ((alpha/m) * sum(error .* X(:,1)));
   %temp1 = theta(2) - ((alpha/m) * sum(error .* X(:,2)));
   %theta = [temp0; temp1];
   %%%%%%%%%%%%%%%%%%%%%%%%%

   %%%%%%%%% CORRECT %%%%%%%
   %error = (X * theta) - y;
   %temp0 = theta(1) - ((alpha/m) * X(:,1)'*error);
   %temp1 = theta(2) - ((alpha/m) * X(:,2)'*error);
   %theta = [temp0; temp1];
   %%%%%%%%%%%%%%%%%%%%%%%%%

   %%%%%%%%% CORRECT %%%%%%%
   error = (X * theta) - y;
   theta = theta - ((alpha/m) * X'*error);
   %%%%%%%%%%%%%%%%%%%%%%%%%

   % ============================================================

   % Save the cost J in every iteration
   J_history(iter) = computeCost(X, y, theta);

  end
end
```

computeCostMulti.m :

```
function J = computeCostMulti(X, y, theta)
  %COMPUTECOSTMULTI Compute cost for linear regression with multiple variables
  %   J = COMPUTECOSTMULTI(X, y, theta) computes the cost of using theta as the
  %   parameter for linear regression to fit the data points in X and y

  % Initialize some useful values
  m = length(y); % number of training examples

  % You need to return the following variables correctly
  J = 0;

  % ====================== YOUR CODE HERE ======================
  % Instructions: Compute the cost of a particular choice of theta
  %               You should set J to the cost.

  J = (1/(2*m))*(sum(((X*theta)-y).^2));

  % =========================================================================

end
```

gradientDescentMulti.m :

```
function [theta, J_history] = gradientDescentMulti(X, y, theta, alpha,
num_iters)
  %GRADIENTDESCENTMULTI Performs gradient descent to learn theta
  %   theta = GRADIENTDESCENTMULTI(x, y, theta, alpha, num_iters) updates theta
by
  %   taking num_iters gradient steps with learning rate alpha

  % Initialize some useful values
  m = length(y); % number of training examples
  J_history = zeros(num_iters, 1);

  for iter = 1:num_iters

   % ====================== YOUR CODE HERE ======================
   % Instructions: Perform a single gradient step on the parameter vector
   %               theta.
   %
   % Hint: While debugging, it can be useful to print out the values
   %       of the cost function (computeCostMulti) and gradient here.
   %

   %%%%%%%% CORRECT %%%%%%%%%%
   error = (X * theta) - y;
   theta = theta - ((alpha/m) * X'*error);
   %%%%%%%%%%%%%%%%%%%%%%%%%%%

   % ============================================================

   % Save the cost J in every iteration
   J_history(iter) = computeCostMulti(X, y, theta);

  end
end
```

Week 3

3.1 Logistic Regression

Classification and Representation

Classification

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

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

Hypothesis Representation

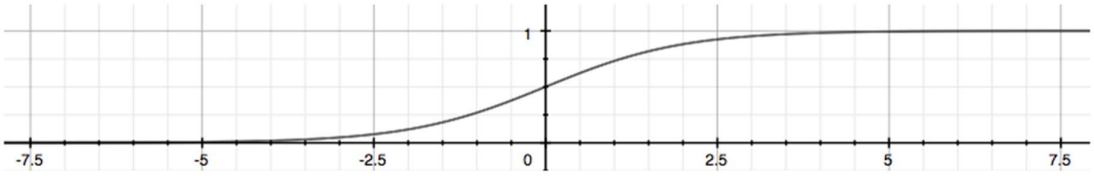
We could approach the classification problem ignoring the fact that y is discrete-valued, and use our old linear regression algorithm to try to predict y given x. However, it is easy to construct examples where this method performs very poorly. Intuitively, it also doesn't make sense for $h_\theta(x)$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

Our new form uses the "Sigmoid Function," also called the "Logistic Function":

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

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

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



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

$h_\theta(x)$ will give us the **probability** that our output is 1. For example, $h_\theta(x) = 0.7$ gives us a probability of 70% that our output is 1. Our probability that our prediction is 0 is just the complement of our probability that it is 1 (e.g. if probability that it is 1 is 70%, then the probability that it is 0 is 30%).

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

Decision Boundary

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

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

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

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

Remember.

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

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.

Logistic Regression Model

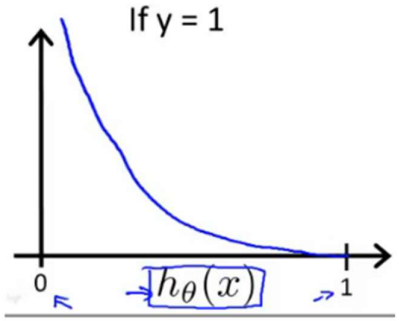
Cost Function

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

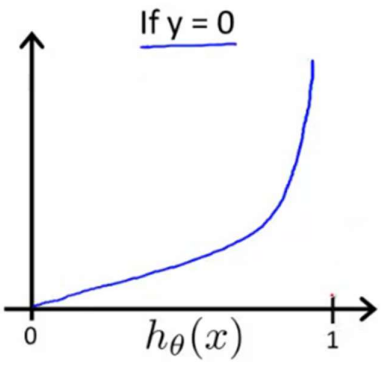
Instead, our cost function for logistic regression looks like:

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

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



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



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

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

Note: [6:53 - the gradient descent equation should have a 1/m factor]

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:

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

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:

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

Advanced Optimization

Note: [7:35 - '100' should be 100 instead. The value provided should be an integer and not a character string.]

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

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

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

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

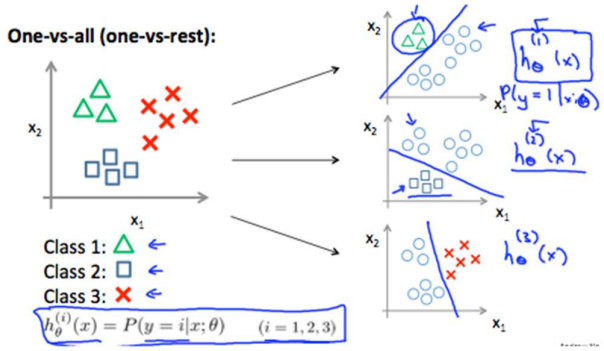
Now we will approach the classification of data when we have more than two categories. Instead of y = {0,1} we will expand our definition so that y = {0,1...n}.

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

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

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.

The following image shows how one could classify 3 classes:



To summarize:

Train a logistic regression classifier $h_\theta(x)$ for each class to predict the probability that y = i.

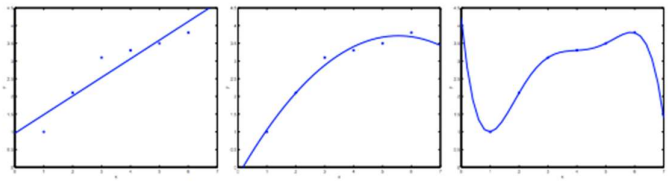
To make a prediction on a new x, pick the class that maximizes $h_\theta(x)$

3.2 Regularization

Solving the Problem of Overfitting

The Problem of Overfitting

Consider the problem of predicting y from x ∈ R. The leftmost figure below shows the result of fitting a $y = \theta_0 + \theta_1 x$ to a dataset. We see that the data doesn't really lie on straight line, and so the fit is not very good.



Instead, if we had added an extra feature x^2 , and fit $y = \theta_0 + \theta_1 x + \theta_2 x^2$, then we obtain a slightly better fit to the data (See middle figure). Naively, it might seem that the more features we add, the better. However, there is also a danger in adding too many features: The rightmost figure is the result of fitting a 5^{th} order polynomial $y = \sum_{j=0}^{5} \theta_j x^j$. We see that even though the fitted curve passes through the data perfectly, we would not expect this to be a very good predictor of, say, housing prices (y) for different living areas (x). Without formally defining what these terms mean, we'll say the figure on the left shows an instance of **underfitting**—in which the data clearly shows structure not captured by the model—and the figure on the right is an example of **overfitting**.

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

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

1) Reduce the number of features:

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

2) Regularization

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

Cost function

Note: [5:18 - There is a typo. It should be $\sum_{j=1}^{n} \theta_j^2$ instead of $\sum_{i=1}^{n} \theta_j^2$]

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

Say we wanted to make the following function more quadratic:

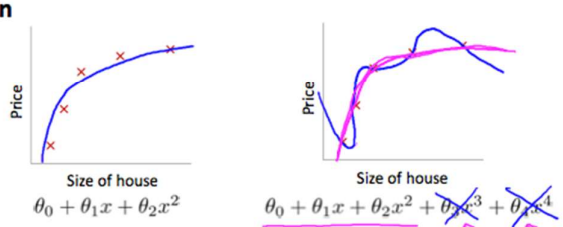
$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

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

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

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

Intuition



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

Suppose we penalize and make θ_3, θ_4 really small.

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

$\theta_3 \approx 0$ $\quad$ $\theta_4 \approx 0$

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

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

The λ, or lambda, is the **regularization parameter**. It determines how much the costs of our theta parameters are inflated.

Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If lambda is chosen to be too large, it may smooth out the function too much and cause underfitting. Hence, what would happen if $\lambda = 0$ or is too small ?

Regularized Linear Regression

Note: [8:43 - It is said that X is non-invertible if m $\leq$ n. The correct statement should be that X is non-invertible if m < n, and may be non-invertible if m = n.

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.

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

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

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

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

Normal Equation

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

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

$$\theta = \left(X^T X + \lambda \cdot L \right)^{-1} X^T y$$
$$\text{where } L = \begin{bmatrix} 0 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \end{bmatrix}$$

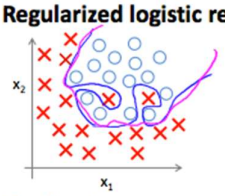
L is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else. It should have dimension (n+1)×(n+1). Intuitively, this is the identity matrix (though we are not including x_0), multiplied with a single real number λ.

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.

Regularised Logistic Regression

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

Regularized logistic regression.



$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^2 x_2 + \theta_4 x_1^2 x_2^2 + \theta_5 x_1^2 x_2^3 + \dots)$$

Cost function:

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

$\theta_1, \theta_2, \dots \theta_n$

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:

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

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

Gradient descent

Repeat {

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

$$\theta_j := \theta_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j\right] \quad (j = 1, 2, 3, \dots, n)$$

$\theta_1 \dots \theta_n$

}

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

$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$

Machine Learning: Programming Exercise 2

Logistic Regression

In this exercise, you will implement logistic regression and apply it to two different datasets.

Files needed for this exercise

- `ex2.mlx` - MATLAB Live Script that steps you through the exercise
- ex2data1.txt - Training set for the first half of the exercise
- ex2data2.txt - Training set for the second half of the exercise
- `submit.m` - Submission script that sends your solutions to our servers
- `mapFeature.m` - Function to generate polynomial features
- `plotDecisionBoundary.m` - Function to plot classifier's decision boundary
- *`plotData.m` - Function to plot 2D classification data
- *`sigmoid.m` - Sigmoid function
- *`costFunction.m` - Logistic regression cost function
- *`predict.m` - Logistic regression prediction function
- *`costFunctionReg.m` - Regularized logistic regression cost function

indicates files you will need to complete

Clear existing variables and confirm that your Current Folder is set correctly

Click into this section, then click the 'Run Section' button above. This will execute the `clear` command to clear existing variables and the `dir` command to list the files in your Current Folder. The output should contain all of the files listed above and the 'lib' folder. If it does not, right-click the 'ex2' folder and select 'Open' before proceding or see the instructions in `README.mlx` for more details.

```
clear
dir
```

```
.                       costFunctionReg.m       ex2data1.txt            mapFeature.m
predict.m               token.mat
..                      ex2.mlx                 ex2data2.txt            plotData.m
sigmoid.m
costFunction.m          ex2_companion.mlx       lib
plotDecisionBoundary.m  submit.m
```

Before you begin

The workflow for completing and submitting the programming exercises in MATLAB Online differs from the original course instructions. Before beginning this exercise, make sure you have read through the instructions in `README.mlx` which is included with the programming exercise files. `README` also contains solutions to the many common issues you may encounter while completing and submitting the exercises in MATLAB Online. Make sure you are following instructions in `README` and have checked for an existing solution before seeking help on the discussion forums.

Table of Contents

Logistic Regression
  Files needed for this exercise
    Clear existing variables and confirm that your Current Folder is set correctly
  Before you begin
1. Logistic Regression
  1.1 Visualizing the data
  1.2 Implementation
    1.2.1 Warmup exercise: sigmoid function
    1.2.2 Cost function and gradient
    Initialize the data
    Compute the gradient
    1.2.3 Learning parameters using fminunc
    1.2.4 Evaluating logistic regression
2. Regularized logistic regression
  2.1 Visualizing the data
  2.2 Feature mapping
  2.3 Cost function and gradient
    2.3.1 Learning parameters using fminunc
  2.4 Plotting the decision boundary
  2.5 Optional (ungraded) exercises
Submission and Grading

1. Logistic Regression

In this part of the exercise, you will build a logistic regression model to predict whether a student gets admitted into a university. Suppose that you are the administrator of a university department and you want to determine each applicant's chance of admission based on their results on two exams. You have historical data from previous applicants that you can use as a training set for logistic regression. For each training example, you have the applicant's scores on two exams and the admissions decision.

Your task is to build a classification model that estimates an applicant's probability of admission based the scores from those two exams. `ex2.mlx` will guide you through the exercise. To begin, run the code below to load the data into MATLAB.

```
% Load Data
% The first two columns contain the exam scores and the third column contains
the label.
data = load('ex2data1.txt');
X = data(:, [1, 2]);
y = data(:, 3);
```

1.1 Visualizing the data

Before starting to implement any learning algorithm, it is always good to visualize the data if possible. The code below will load the data and display it on a 2-dimensional plot by calling the function `plotData`. You will now complete the code in `plotData` so that it displays a figure like Figure 1, where the axes are the two exam scores, and the positive and negative examples are shown with different markers.

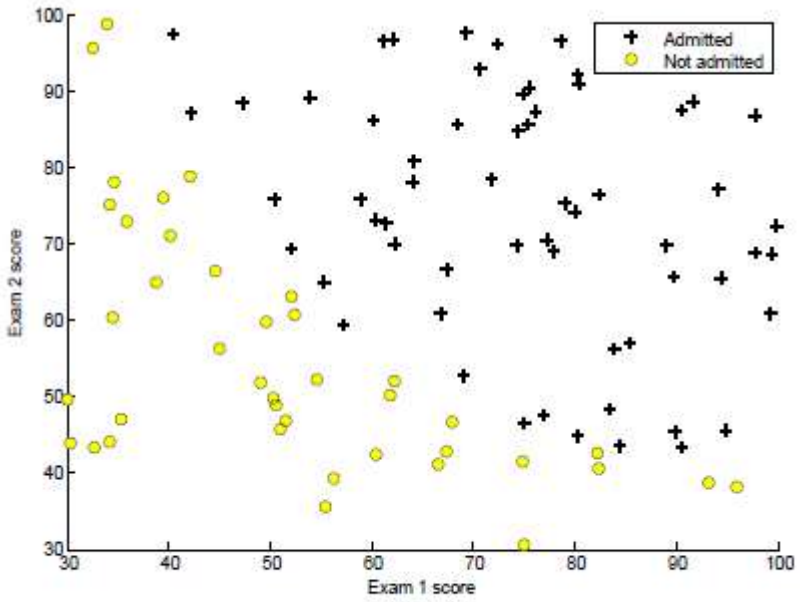


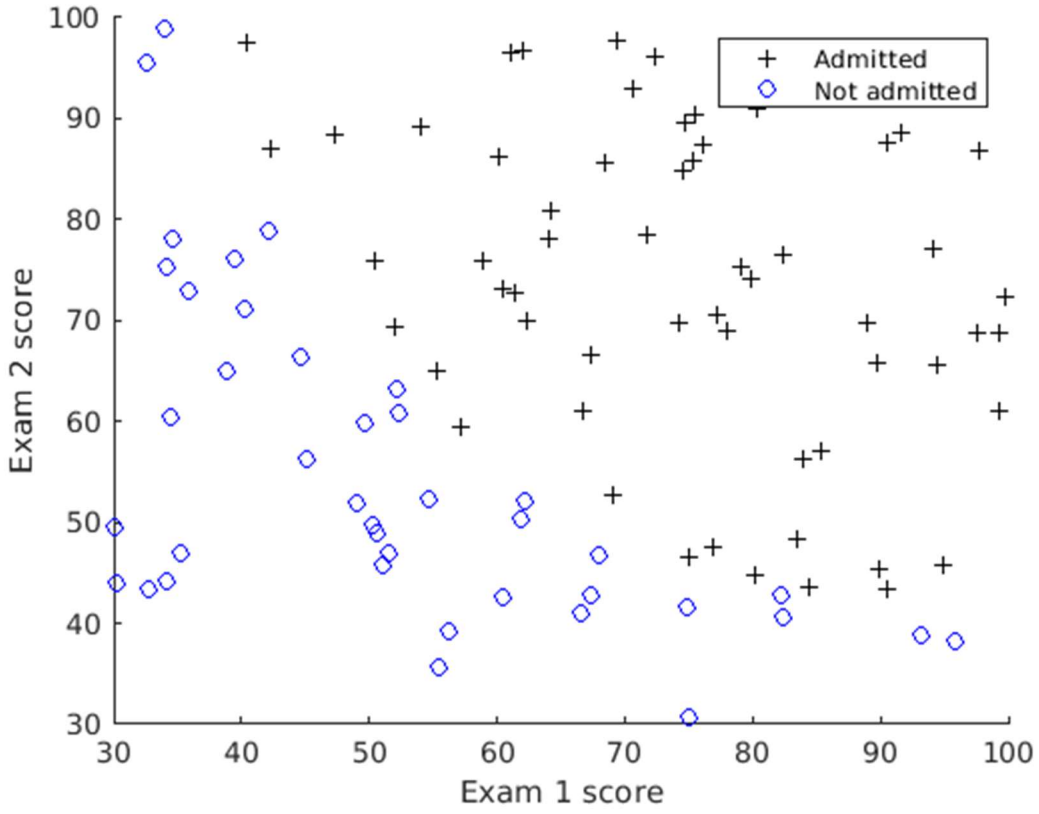
Figure 1: Scatter plot of training data

To help you get more familiar with plotting, we have left `plotData.m` empty so you can try to implement it yourself. However, this is an optional (ungraded) exercise. We also provide our implementation below so you can copy it or refer to it. If you choose to copy our example, make sure you learn what each of its commands is doing by consulting the MATLAB documentation.

```
      % Find Indices of Positive and Negative Examples
      pos = find(y==1); neg = find(y == 0);
      % Plot Examples
      plot(X(pos, 1), X(pos, 2), 'k+','LineWidth', 2, 'MarkerSize', 7);
      plot(X(neg, 1), X(neg, 2), 'ko', 'MarkerFaceColor',
'y','MarkerSize', 7);
```

Once you have added your own or the above code to `plotData.m`, run the code in this section to call the `plotData` function.

```
 % Plot the data with + indicating (y = 1) examples and o indicating (y = 0)
examples.
 plotData(X, y);
```




```
% Labels and Legend
% xlabel('Exam 1 score')
% ylabel('Exam 2 score')

% Specified in plot order
% legend('Admitted', 'Not admitted')
```

1.2 Implementation

1.2.1 Warmup exercise: sigmoid function

Before you start with the actual cost function, recall that the logistic regression hypothesis is defined as:

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

where function g is the sigmoid function. The sigmoid function is defined as:

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

Your first step is to implement this function in `sigmoid.m` so it can be called by the rest of your program. When you are finished, try testing a few values by calling `sigmoid(x)` in the code section below. For large positive values of `x`, the sigmoid should be close to 1, while for large negative

values, the sigmoid should be close to 0. Evaluating `sigmoid(0)` should give you exactly 0.5. **Your code should also work with vectors and matrices**. For a matrix, your function should perform the sigmoid function on every element.

```
% Provide input values to the sigmoid function below and run to check your
implementation
sigmoid(0)
```

```
ans = 0.5000
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

1.2.2 Cost function and gradient

Now you will implement the cost function and gradient for logistic regression. Complete the code in `costFunction.m` to return the cost and gradient. Recall that the cost function in logistic regression is

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

and the gradient of the cost is a vector of the same length as θ where the jth element (for $j = 0, 1, \ldots, n$) is defined as follows:

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

Note that while this gradient looks identical to the linear regression gradient, the formula is actually different because linear and logistic regression have different definitions of $h_\theta(x)$. Once you are done, run the code sections below to set up your data and call your `costFunction` using the initial parameters of θ. You should see that the cost is about 0.693 and gradients of about -0.1000, -12.0092, and -11.2628.

Initialize the data

```
%  Setup the data matrix appropriately
[m, n] = size(X);

% Add intercept term to X
X = [ones(m, 1) X];

% Initialize the fitting parameters
initial_theta = zeros(n + 1, 1);
```

Compute the gradient

```
% Compute and display the initial cost and gradient
[cost, grad] = costFunction(initial_theta, X, y);
```

```
fprintf('Cost at initial theta (zeros): %f\n', cost);
```

```
Cost at initial theta (zeros): 0.693147
```

```
disp('Gradient at initial theta (zeros):');
```

```
Gradient at initial theta (zeros):
```

```
disp(grad);
```

```
   -0.1000
  -12.0092
  -11.2628
```

1.2.3 Learning parameters using `fminunc`

In the previous assignment, you found the optimal parameters of a linear regression model by implementing gradent descent. You wrote a cost function and calculated its gradient, then took a gradient descent step accordingly. This time, instead of taking gradient descent steps, you will use a MATLAB built-in function called `fminunc`.

MATLAB's `fminunc` is an optimization solver that finds the minimum of an unconstrained* function. For logistic regression, you want to optimize the cost function $J(\theta)$ with parameters. Concretely, you are going to use `fminunc` to find the best parameters θ for the logistic regression cost function, given a fixed dataset (of X and y values). You will pass to `fminunc` the following inputs:

- The initial values of the parameters we are trying to optimize.
- A function that, when given the training set and a particular θ computes the logistic regression cost and gradient with respect to θ for the dataset (X, y)

**Constraints in optimization often refer to constraints on the parameters, for example, constraints that bound the possible values θ can take (e.g. $\theta < 1$). Logistic regression does not have such constraints since θ is allowed to take any real value.*

We already have code written below to call `fminunc` with the correct arguments:

- We first define the options to be used with `fminunc`. Specically, we set the `GradObj` option to on, which tells `fminunc` that our function returns both the cost and the gradient. This allows `fminunc` to use the gradient when minimizing the function.
- Furthermore, we set the `MaxIter` option to 400, so that `fminunc` will run for at most 400 steps before it terminates.
- To specify the actual function we are minimizing, we use a 'short-hand' for specifying functions with: `@(t)(costFunction(t,X,y))`. This creates a function, with argument `t`, which calls your `costFunction`. This allows us to wrap the `costFunction` for use with `fminunc`.
- If you have completed the `costFunction` correctly, `fminunc` will converge on the right optimization parameters and return the final values of the cost and θ. Notice that by using

`fminunc`, you did not have to write any loops yourself, or set a learning rate like you did for gradient descent. This is all done by `fminunc`: you only needed to provide a function calculating the cost and the gradient.

- Once `fminunc` completes, the remaining code will call your `costFunction` function using the optimal parameters of θ. You should see that the cost is about 0.203. This final θ value will then be used to plot the decision boundary on the training data, resulting in a figure similar to Figure 2. We also encourage you to look at the code in `plotDecisionBoundary.m` to see how to plot such a boundary using the θ values.

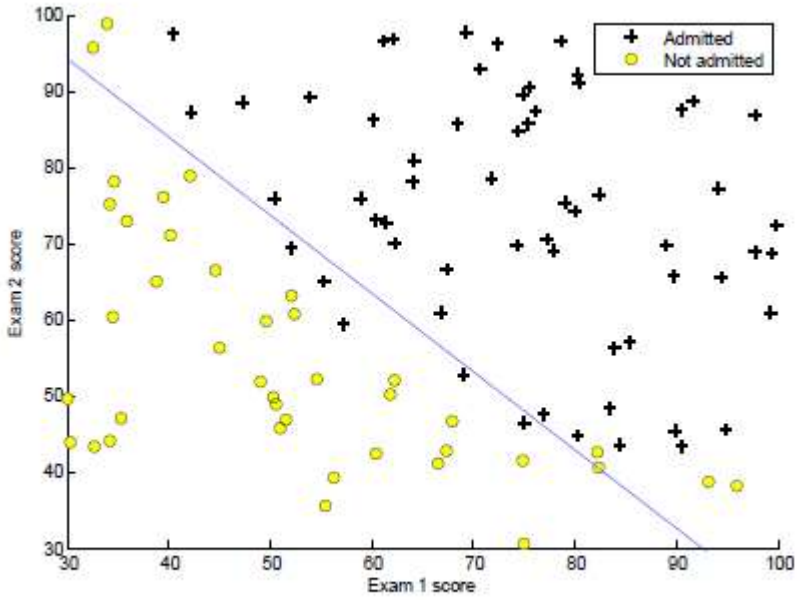


Figure 2: Training data with decision boundary

Run the code below and verify the results.

```
% Set options for fminunc
options = optimoptions(@fminunc,'Algorithm','Quasi-Newton','GradObj', 'on',
'MaxIter', 400);

% Run fminunc to obtain the optimal theta
% This function will return theta and the cost
[theta, cost] = fminunc(@(t)(costFunction(t, X, y)), initial_theta, options);
```

```
Local minimum found.

Optimization completed because the size of the gradient is less than
the value of the optimality tolerance.

<stopping criteria details>
```

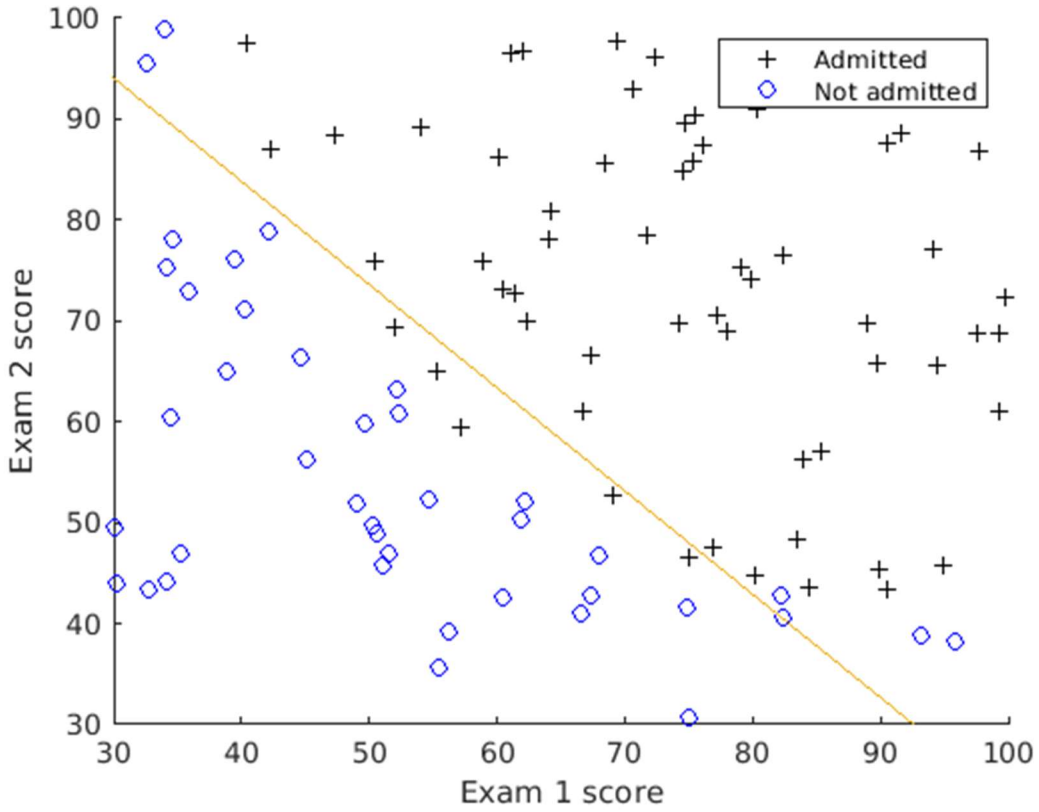
```
% Print theta
fprintf('Cost at theta found by fminunc: %f\n', cost);
```

```
Cost at theta found by fminunc: 0.203498
```

```
disp('theta:');disp(theta);
```

```
theta:
  -25.1613
    0.2062
    0.2015
```

```
% Plot Boundary
plotDecisionBoundary(theta, X, y);
% Add some labels
hold on;
% Labels and Legend
xlabel('Exam 1 score')
ylabel('Exam 2 score')
% Specified in plot order
legend('Admitted', 'Not admitted')
hold off;
```



1.2.4 Evaluating logistic regression

After learning the parameters, you can use the model to predict whether a particular student will be admitted. For a student with an Exam 1 score of 45 and an Exam 2 score of 85, you should expect to see an admission probability of 0.776. Another way to evaluate the quality of the parameters we have found is to see how well the learned model predicts on our training set. In this part, your task is to complete the code in `predict.m`. The predict function will produce '1' or '0' predictions given a dataset and a learned parameter vector θ.

After you have completed the code in `predict.m`, the code below will proceed to report the training accuracy of your classifier by computing the percentage of examples it got correct.

```
 %  Predict probability for a student with score 45 on exam 1  and score 85 on
exam 2
 prob = sigmoid([1 45 85] * theta);
 fprintf('For a student with scores 45 and 85, we predict an admission
probability of %f\n\n', prob);
```

```
For a student with scores 45 and 85, we predict an admission probability of 0.776291
```

```
 % Compute accuracy on our training set
 p = predict(theta, X);
 fprintf('Train Accuracy: %f\n', mean(double(p == y)) * 100);
```

```
Train Accuracy: 89.000000
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

2. Regularized logistic regression

In this part of the exercise, you will implement regularized logistic regression to predict whether microchips from a fabrication plant passes quality assurance (QA). During QA, each microchip goes through various tests to ensure it is functioning correctly. Suppose you are the product manager of the factory and you have the test results for some microchips on two different tests. From these two tests, you would like to determine whether the microchips should be accepted or rejected. To help you make the decision, you have a dataset of test results on past microchips, from which you can build a logistic regression model.

2.1 Visualizing the data

Similar to the previous parts of this exercise, `plotData` is used in the code below to generate a figure like Figure 3, where the axes are the two test scores, and the positive ($y = 1,$ accepted) and negative ($y = 0,$ rejected) examples are shown with different markers.

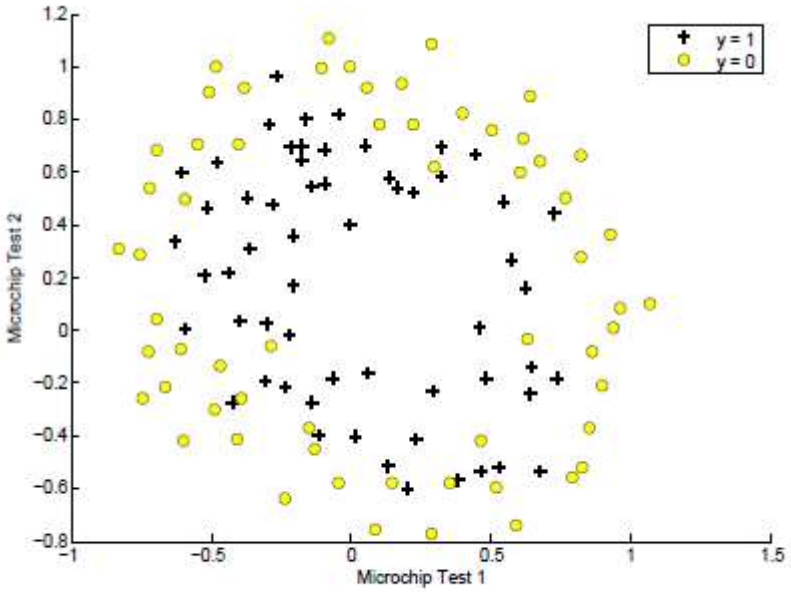


Figure 3: Plot of training data

Run the code below and confirm you plot matches Figure 3.

```
%  The first two columns contains the X values and the third column
%  contains the label (y).
data = load('ex2data2.txt');
X = data(:, [1, 2]); y = data(:, 3);

plotData(X, y);
% Put some labels
hold on;
% Labels and Legend
xlabel('Microchip Test 1')
ylabel('Microchip Test 2')
% Specified in plot order
legend('y = 1', 'y = 0')
hold off;
```

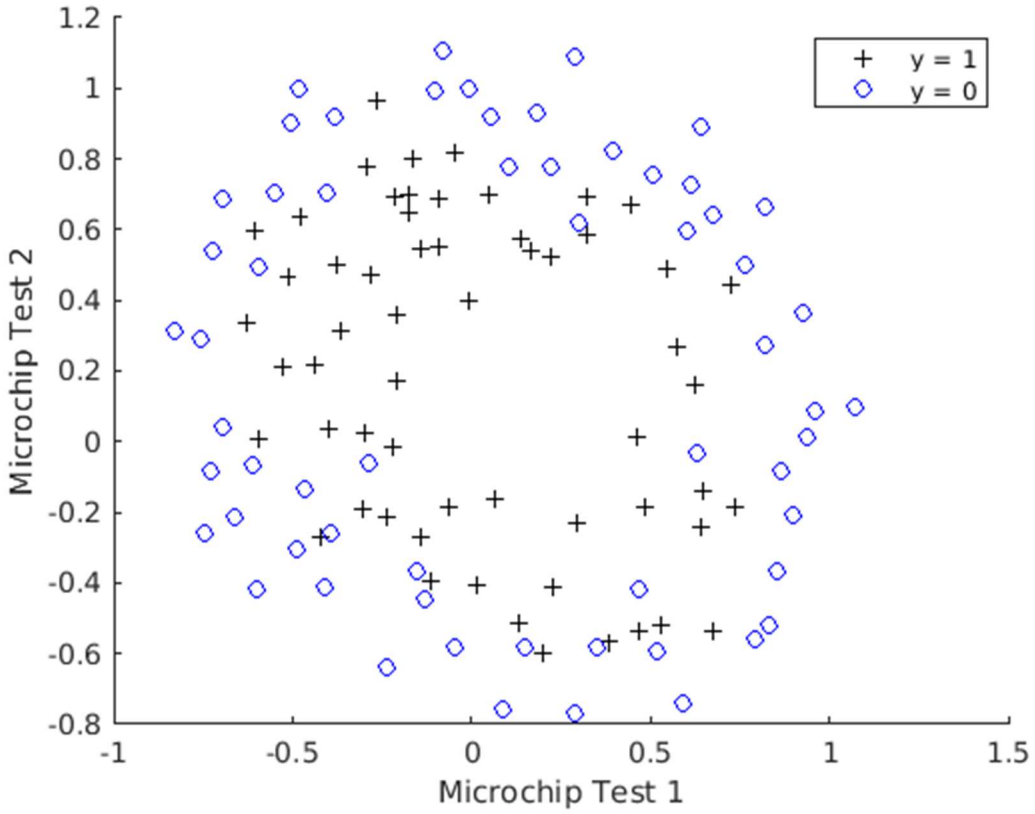


Figure 3 shows that our dataset cannot be separated into positive and negative examples by a straight-line through the plot. Therefore, a straightforward application of logistic regression will not perform well on this dataset since logistic regression will only be able to find a linear decision boundary.

2.2 Feature mapping

One way to fit the data better is to create more features from each data point. In the provided function `mapFeature.m`, we will map the features into all polynomial terms of x_1 and x_2 up to the sixth power.

$$\text{mapFeature}(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \\ x_1^3 \\ \vdots \\ x_1 x_2^5 \\ x_2^6 \end{bmatrix}$$

As a result of this mapping, our vector of two features (the scores on two QA tests) has been transformed into a 28-dimensional vector. A logistic regression classifier trained on this higher-dimension feature vector will have a more complex decision boundary and will appear nonlinear when drawn in our 2-dimensional plot.

Run the code below to map the features.

```
% Add Polynomial Features
% Note that mapFeature also adds a column of ones for us, so the intercept term
is handled
X = mapFeature(X(:,1), X(:,2));
```

While the feature mapping allows us to build a more expressive classifier, it also more susceptible to overfitting. In the next parts of the exercise, you will implement regularized logistic regression to fit the data and also see for yourself how regularization can help combat the overfitting problem.

2.3 Cost function and gradient

Now you will implement code to compute the cost function and gradient for regularized logistic regression. Complete the code in `costFunctionReg.m` to return the cost and gradient. Recall that the regularized cost function in logistic regression is

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

Note that you should not regularize the parameter θ_0. In MATLAB, recall that indexing starts from 1, hence, you should not be regularizing the `theta(1)` parameter (which corresponds to θ_0) in the code. The gradient of the cost function is a vector where the $j\text{th}$ element is defined as follows:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \qquad \text{for } j = 0,$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \right) + \frac{\lambda}{m} \theta_j \qquad \text{for } j \geq 1,$$

Once you are done, run the code below to call your `costFunctionReg` function using the initial value of θ (initialized to all zeros). You should see that the cost is about 0.693.

```
% Initialize fitting parameters
initial_theta = zeros(size(X, 2), 1);

% Set regularization parameter lambda to 1
lambda = 1;

% Compute and display initial cost and gradient for regularized logistic
regression
cost = costFunctionReg(initial_theta, X, y, lambda);
fprintf('Cost at initial theta (zeros): %f\n', cost);
```

```
Cost at initial theta (zeros): 0.693147
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

2.3.1 Learning parameters using `fminunc`

Similar to the previous parts, the next step is to use `fminunc` to learn the optimal parameters. If you have completed the cost and gradient for regularized logistic regression (`costFunctionReg.m`) correctly, you should be able to run the code in the following sections to learn the parameters using `fminunc` for multiple values of λ.

2.4 Plotting the decision boundary

To help you visualize the model learned by this classifier, we have provided the function `plotDecisionBoundary.m` which plots the (nonlinear) decision boundary that separates the positive and negative examples. In `plotDecisionBoundary.m`, we plot the nonlinear decision boundary by computing the classifier's predictions on an evenly spaced grid and then drew a contour plot of where the predictions change from $y = 0$ to $y = 1$. After learning the parameters, the code in the next section will plot a decision boundary similar to Figure 4.

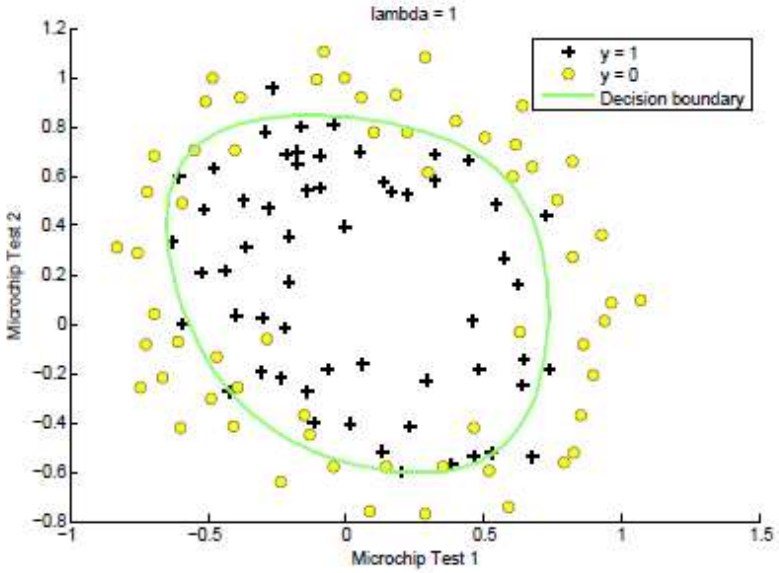


Figure 4: Training data with decision boundary ($\lambda = 1$)

2.5 Optional (ungraded) exercises

In this part of the exercise, you will get to try out different regularization parameters for the dataset to understand how regularization prevents overfitting. Notice the changes in the decision boundary as you vary λ. With a small λ, you should find that the classifier gets almost every training example correct, but draws a very complicated boundary, thus overfitting the data (Figure 5).

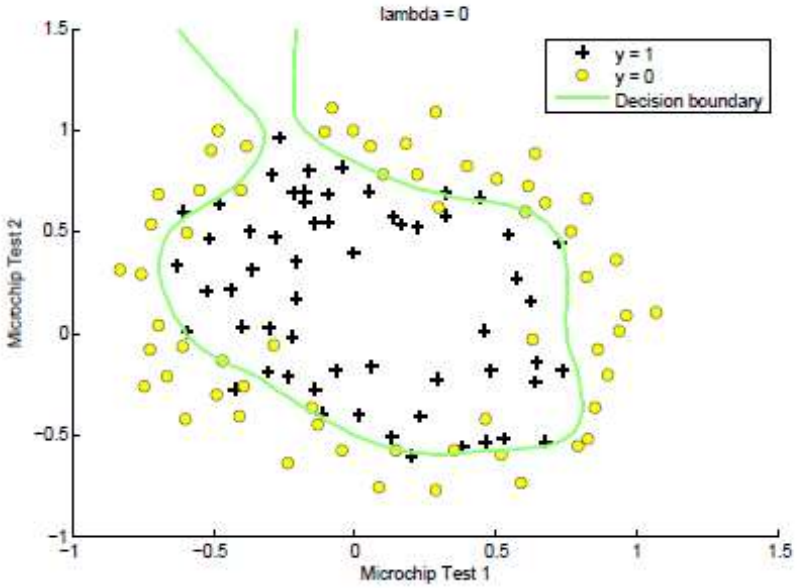


Figure 5: No regularization (Overfitting) ($\lambda = 0$)

This is not a good decision boundary: for example, it predicts that a point at $x = (0.25, 1.5)$ is accepted $(y = 1)$, which seems to be an incorrect decision given the training set. With a larger λ, you should see a plot that shows an simpler decision boundary which still separates the positives and negatives fairly well. However, if λ is set to too high a value, you will not get a good fit and the decision boundary will not follow the data so well, thus underfitting the data (Figure 6).

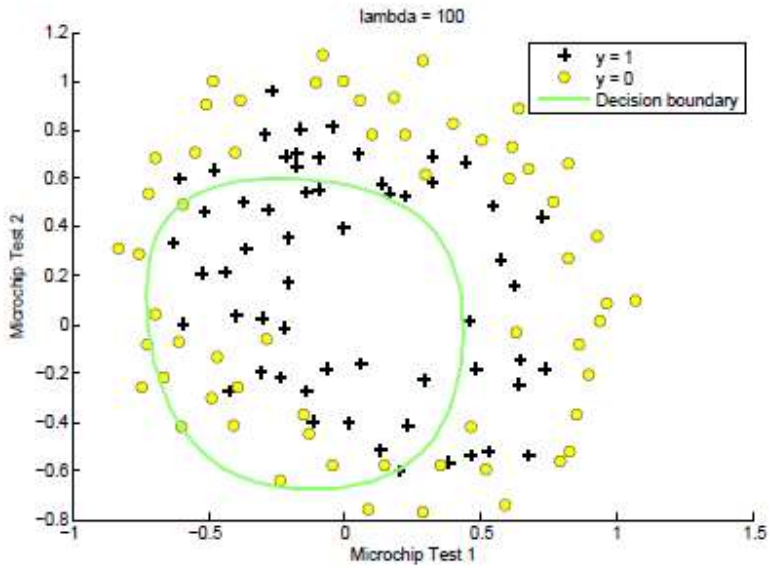


Figure 6: Too much regularization (Underfitting) ($\lambda = 100$)

Use the control below to try the following values of λ: 0, 1, 10, 100.

- How does the decision boundary change when you vary λ?
- How does the training set accuracy vary?

```
% Initialize fitting parameters
initial_theta = zeros(size(X, 2), 1);

lambda = 0;
% Set Options
options = optimoptions(@fminunc,'Algorithm','Quasi-Newton','GradObj', 'on',
'MaxIter', 1000);

% Optimize
[theta, J, exit_flag] = fminunc(@(t)(costFunctionReg(t, X, y, lambda)),
initial_theta, options);
% Plot Boundary
plotDecisionBoundary(theta, X, y);
hold on;
title(sprintf('lambda = %g', lambda))
```

```
% Labels and Legend
xlabel('Microchip Test 1')
ylabel('Microchip Test 2')

legend('y = 1', 'y = 0', 'Decision boundary')
hold off;

% Compute accuracy on our training set
p = predict(theta, X);

fprintf('Train Accuracy: %f\n', mean(double(p == y)) * 100);
```

You do not need to submit any solutions for these optional (ungraded) exercises.

Submission and Grading

After completing various parts of the assignment, be sure to use the `submit` function system to submit your solutions to our servers. The following is a breakdown of how each part of this exercise is scored.

Part	Submitted File	Points
Sigmoid Function	`sigmoid.m`	5 points
Compute cost for logistic regression	`costFunction.m`	30 points
Gradient for logistic regression	`costFunction.m`	30 points
Predict Function	`predict.m`	5 points
Compute cost for regularized LR	`costFunctionReg.m`	15 points
Gradient for regularized LR	`costFunctionReg.m`	15 points
Total Points		100 points

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration.

sigmoid.m :

```
function g = sigmoid(z)
  %SIGMOID Compute sigmoid function
  %   g = SIGMOID(z) computes the sigmoid of z.

  % You need to return the following variables correctly
  g = zeros(size(z));

  % ====================== YOUR CODE HERE ======================
  % Instructions: Compute the sigmoid of each value of z (z can be a matrix,
  %               vector or scalar).
  g = 1./(1+exp(-z));

  % =============================================================
end
```

costFunction.m :

```
function [J, grad] = costFunction(theta, X, y)
  %COSTFUNCTION Compute cost and gradient for logistic regression
  %   J = COSTFUNCTION(theta, X, y) computes the cost of using theta as the
  %   parameter for logistic regression and the gradient of the cost
  %   w.r.t. to the parameters.

  % Initialize some useful values
  m = length(y); % number of training examples

  % You need to return the following variables correctly
  J = 0;
  grad = zeros(size(theta));

  % ====================== YOUR CODE HERE ======================
  % Instructions: Compute the cost of a particular choice of theta.
  %               You should set J to the cost.
  %               Compute the partial derivatives and set grad to the partial
  %               derivatives of the cost w.r.t. each parameter in theta
  %
  % Note: grad should have the same dimensions as theta
  %
  %DIMENSIONS:
  %   theta = (n+1) x 1
  %   X     = m x (n+1)
  %   y     = m x 1
  %   grad  = (n+1) x 1
  %   J     = Scalar

  z = X * theta;      % m x 1
  h_x = sigmoid(z);   % m x 1

  J = (1/m)*sum((-y.*log(h_x))-((1-y).*log(1-h_x))); % scalar

  grad = (1/m)* (X'*(h_x-y));     % (n+1) x 1

  % =============================================================

end
```

predict.m :

```
function p = predict(theta, X)
  %PREDICT Predict whether the label is 0 or 1 using learned logistic
  %regression parameters theta
  %   p = PREDICT(theta, X) computes the predictions for X using a
  %   threshold at 0.5 (i.e., if sigmoid(theta'*x) >= 0.5, predict 1)

  m = size(X, 1); % Number of training examples

  % You need to return the following variables correctly
  p = zeros(m, 1);

  % ====================== YOUR CODE HERE ======================
  % Instructions: Complete the following code to make predictions using
  %               your learned logistic regression parameters.
  %               You should set p to a vector of 0's and 1's
  %
  % Dimentions:
  % X     =  m x (n+1)
  % theta = (n+1) x 1

  h_x = sigmoid(X*theta);
  p=(h_x>=0.5);

  %p = double(sigmoid(X * theta)>=0.5);
  % =========================================================================
end
```

costFunctionReg.m :

```
function [J, grad] = costFunctionReg(theta, X, y, lambda)
  %COSTFUNCTIONREG Compute cost and gradient for logistic regression with
regularization
  %   J = COSTFUNCTIONREG(theta, X, y, lambda) computes the cost of using
  %   theta as the parameter for regularized logistic regression and the
  %   gradient of the cost w.r.t. to the parameters.

  % Initialize some useful values
  m = length(y); % number of training examples

  % You need to return the following variables correctly
  J = 0;
  grad = zeros(size(theta));

  % ====================== YOUR CODE HERE ======================
  % Instructions: Compute the cost of a particular choice of theta.
  %               You should set J to the cost.
  %               Compute the partial derivatives and set grad to the partial
  %               derivatives of the cost w.r.t. each parameter in theta

  %DIMENSIONS:
  %   theta = (n+1) x 1
  %   X     = m x (n+1)
  %   y     = m x 1
  %   grad  = (n+1) x 1
  %   J     = Scalar

  z = X * theta;      % m x 1
  h_x = sigmoid(z);  % m x 1

  reg_term = (lambda/(2*m)) * sum(theta(2:end).^2);

  J = (1/m)*sum((-y.*log(h_x))-((1-y).*log(1-h_x))) + reg_term; % scalar

  grad(1) = (1/m)* (X(:,1)'*(h_x-y));                                    % 1 x 1
  grad(2:end) = (1/m)* (X(:,2:end)'*(h_x-y))+(lambda/m)*theta(2:end);  % n x 1

  % =============================================================
end
```

Week 4

4.1 Neural Networks: Representation

Neural Networks

Model Representation I

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

Visually, a simplistic representation looks like:

$$[x_0 x_1 x_2] \rightarrow [\ ] \rightarrow h_\theta(x)$$

Our input nodes (layer 1), also known as the "input layer", go into another node (layer 2), which finally outputs the hypothesis function, known as the "output layer".

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

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

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

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

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

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

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

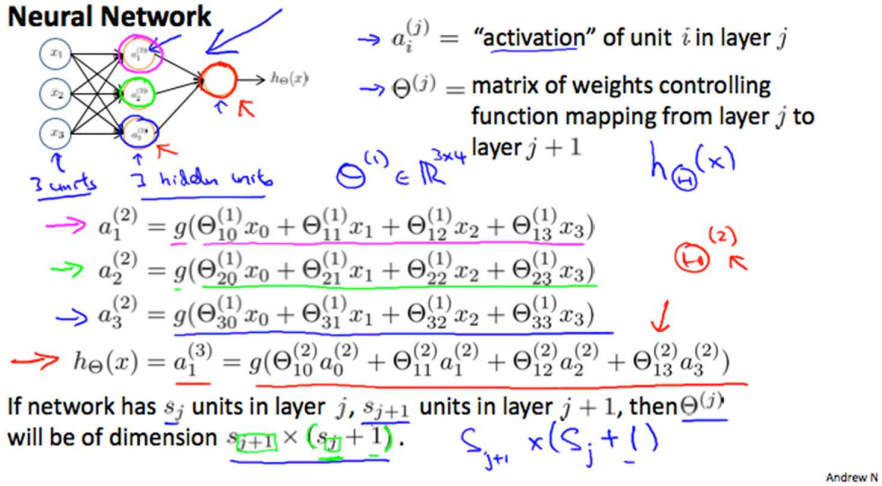
This is saying that we compute our activation nodes by using a 3×4 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied by yet another parameter matrix $\Theta^{(2)}$ containing the weights for our second layer of nodes.

Each layer gets its own matrix of weights, $\Theta^{(j)}$.

The dimensions of these matrices of weights is determined as follows:

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

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



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

Model Representation II

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

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

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 by 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:

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

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** which is multiplied by one column $a^{(j)}$ so that our result is a single number. We then get our final result with:

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

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

Applications

Examples and 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:

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

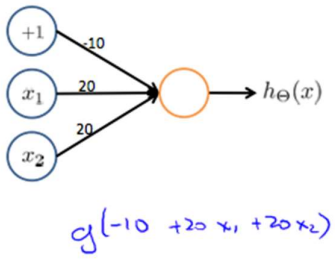
This will cause the output of our hypothesis to only be positive if both x_1 and x_2 are 1. In other words:

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

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

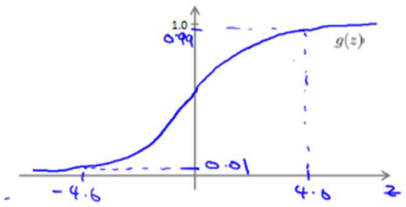
So we have constructed one of the fundamental operations in computers by using a small neural network rather than using an actual AND gate. Neural networks can also be used to simulate all the other logical gates. The following is an example of the logical operator 'OR', meaning either x_1 is true or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

Example: OR function



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

Where g(z) is the following:



Examples and Intuitions II

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

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

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).

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

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

$$
\Theta^{(1)} = [-30 \quad 20 \quad 2010 \quad -20 \quad -20]
$$

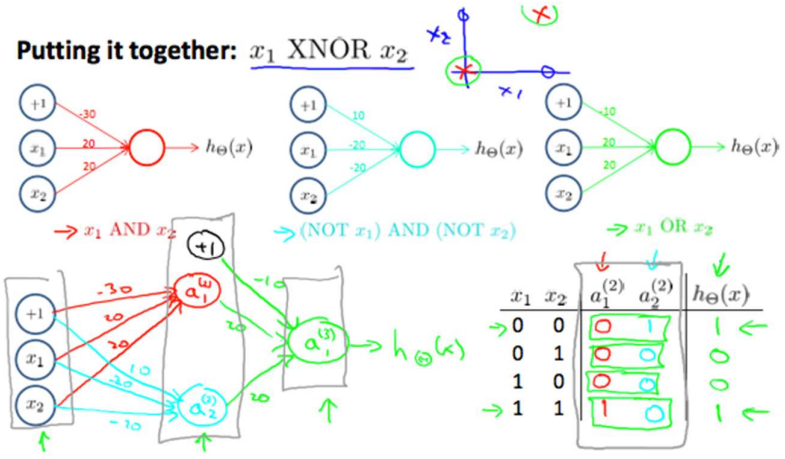
For the transition between the second and third layer, we'll use a $\Theta^{(2)}$ matrix that uses the value for OR:

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

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

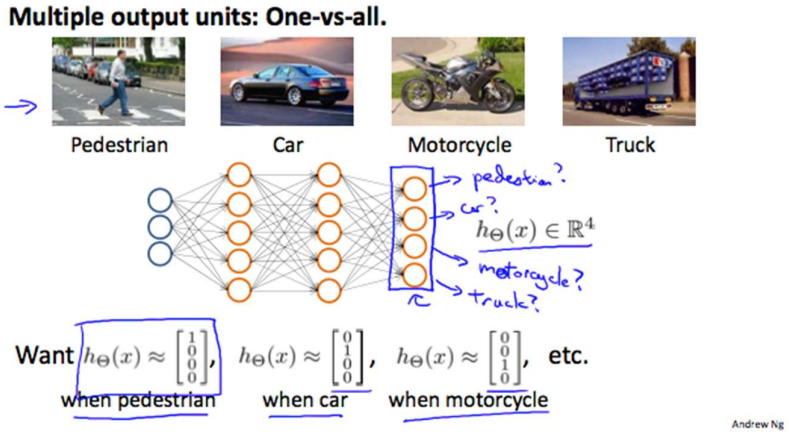
$$
\begin{aligned}
a^{(2)} &= g(\Theta^{(1)} \cdot x) \\
a^{(3)} &= g(\Theta^{(2)} \cdot a^{(2)}) \\
h_\Theta(x) &= a^{(3)}
\end{aligned}
$$

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



Multiclass Classification

To classify data into multiple classes, we let our hypothesis function return a vector of values. Say we wanted to classify our data into one of four categories. We will use the following example to see how this classification is done. This algorithm takes as input an image and classifies it accordingly:



We can define our set of resulting classes as y:

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

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

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

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

$$h_\Theta(x) = [0010]$$

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

Machine Learning: Programming Exercise 3

Multi-class Classification and Neural Networks

In this exercise, you will implement one-vs-all logistic regression and neural networks to recognize hand-written digits.

Files needed for this exercise

- `ex3.mlx` - MATLAB Live Script that steps you through the exercise
- `ex3data1.mat` - Training set of hand-written digits
- `ex3weights.mat` - Initial weights for the neural network exercise
- `submit.m` - Submission script that sends your solutions to our servers
- `displayData.m` - Function to help visualize the dataset
- `fmincg.m` - Function minimization routine (similar to `fminunc`)
- `sigmoid.m` - Sigmoid function
- `*lrCostFunction.m` - Logistic regression cost function
- `*oneVsAll.m` - Train a one-vs-all multi-class classifier
- `*predictOneVsAll.m` - Predict using a one-vs-all multi-class classifier
- `*predict.m` - Neural network prediction function

indicates files you will need to complete

Clear existing variables and confirm that your Current Folder is set correctly

Click into this section, then click the 'Run Section' button above. This will execute the `clear` command to clear existing variables and the `dir` command to list the files in your Current Folder. The output should contain all of the files listed above and the 'lib' folder. If it does not, right-click the 'ex3' folder and select 'Open' before proceding or see the instructions in `README.mlx` for more details.

```
clear
dir
```

```
.                 displayData.m      ex3_companion.mat  ex3data1.mat       fmincg.m
lrCostFunction.m  predict.m          sigmoid.m          token.mat
..                ex3.mlx            ex3_companion.mlx  ex3weights.mat     lib
oneVsAll.m        predictOneVsAll.m  submit.m
```

Before you begin

The workflow for completing and submitting the programming exercises in MATLAB Online differs from the original course instructions. Before beginning this exercise, make sure you have read through the instructions in `README.mlx` which is included with the programming exercise files. `README` also contains solutions to the many common issues you may encounter while completing and submitting the exercises in MATLAB Online. Make sure you are following instructions in `README` and have checked for an existing solution before seeking help on the discussion forums.

Table of Contents

Multi-class Classification and Neural Networks
  Files needed for this exercise
    Clear existing variables and confirm that your Current Folder is set correctly
  Before you begin
1. Multi-class Classification
  1.1 Dataset
1.2 Visualizing the data
  1.3 Vectorizing logistic regression
    1.3.1 Vectorizing the cost function
    1.3.2 Vectorizing the gradient
    1.3.3 Vectorizing regularized logistic regression
  1.4 One-vs-all classication
    1.4.1 One-vs-all prediction
2. Neural Networks
  2.1 Model representation
  2.2 Feedforward propagation and prediction
Submission and Grading

1. Multi-class Classification

For this exercise, you will use logistic regression and neural networks to recognize handwritten digits (from 0 to 9). Automated handwritten digit recognition is widely used today - from recognizing zip codes (postal codes) on mail envelopes to recognizing amounts written on bank checks. This exercise will show you how the methods you've learned can be used for this classication task. In the first part of the exercise, you will extend your previous implemention of logistic regression and apply it to one-vs-all classification.

1.1 Dataset

You are given a data set in `ex3data1.mat` that contains 5000 training examples of handwritten digits*. The .mat format means that that the data has been saved in a native MATLAB matrix format, instead of a text (ASCII) format like a csv-file. These matrices can be read directly into your program by using the load command. After loading, matrices of the correct dimensions and values will appear in your program's memory. The matrix will already be named, so you do not need to assign names to them.

**This is a subset of the MNIST handwritten digit dataset*

Run the code below to load the data.

```
% Load saved matrices from file
load('ex3data1.mat');
% The matrices X and y will now be in your MATLAB environment
```

There are 5000 training examples in `ex3data1.mat`, where each training example is a 20 pixel by 20 pixel grayscale image of the digit. Each pixel is represented by a floating point number indicating the grayscale intensity at that location. The 20 by 20 grid of pixels is 'unrolled' into a 400-dimensional vector. Each of these training examples becomes a single row in our data matrix `X`. This gives us a 5000 by 400 matrix `X` where every row is a training example for a handwritten digit image.

$$X = \begin{bmatrix} -(x^{(1)})^T - \\ -(x^{(2)})^T - \\ \vdots \\ -(x^{(m)})^T - \end{bmatrix}$$

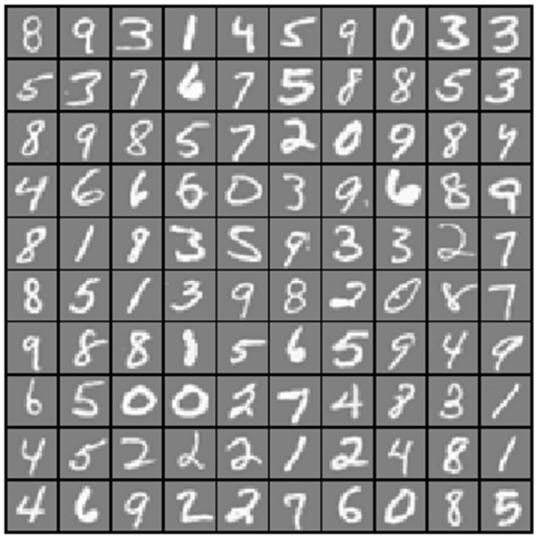
The second part of the training set is a 5000-dimensional vector `y` that contains labels for the training set. To make things more compatible with MATLAB indexing, where there is no zero index, we have mapped the digit zero to the value ten. Therefore, a '0' digit is labeled as '10', while the digits '1' to '9' are labeled as '1' to '9' in their natural order.

1.2 Visualizing the data

You will begin by visualizing a subset of the training set. The code below randomly selects selects 100 rows from `X` and passes those rows to the `displayData` function. This function maps each row to a 20 pixel by 20 pixel grayscale image and displays the images together. We have provided the `displayData` function, and you are encouraged to examine the code to see how it works. After you run this step, you should see an image like Figure 1.

Figure 1: Examples from the dataset

```
m = size(X, 1);
% Randomly select 100 data points to display
rand_indices = randperm(m);
sel = X(rand_indices(1:100), :);
displayData(sel);
```



1.3 Vectorizing logistic regression

You will be using multiple one-vs-all logistic regression models to build a multi-class classifier. Since there are 10 classes, you will need to train 10 separate logistic regression classifiers. To make this training efficient, it is important to ensure that your code is well vectorized. In this section, you will implement a vectorized version of logistic regression that does not employ any `for` loops. You can use your code in the last exercise as a starting point for this exercise.

1.3.1 Vectorizing the cost function

We will begin by writing a vectorized version of the cost function. Recall that in (unregularized) logistic regression, the cost function is

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

To compute each element in the summation, we have to compute $h_\theta(x^{(i)})$ for every example i, where $h_\theta(x^{(i)}) = g(\theta^T x^{(i)})$ and $g(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function. It turns out that we can compute this quickly for all our examples by using matrix multiplication. Let us define X and θ as

$$X\theta = \begin{bmatrix} -(x^{(1)})^T\theta - \\ -(x^{(2)})^T\theta - \\ \vdots \\ -(x^{(m)})^T\theta - \end{bmatrix} = \begin{bmatrix} -\theta^T(x^{(1)}) - \\ -\theta^T(x^{(2)}) - \\ \vdots \\ -\theta^T(x^{(m)}) - \end{bmatrix}$$

In the last equality, we used the fact that $a^T b = b^T a$ if a and b are vectors. This allows us to compute the products $\theta^T x^{(i)}$ for all our examples i in one line of code.

Your job is to write the unregularized cost function in the file `lrCostFunction.m` Your implementation should use the strategy we presented above to calculate $\theta^T x^{(i)}$. You should also use a vectorized approach for the rest of the cost function. A fully vectorized version of `lrCostFunction.m` should not contain any loops. (Hint: You might want to use the element-wise multiplication operation (`.*`) and the sum operation `sum` when writing this function)

1.3.2 Vectorizing the gradient

Recall that the gradient of the (unregularized) logistic regression cost is a vector where the jth element is defined as

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

To vectorize this operation over the dataset, we start by writing out all the partial derivatives explicitly for all θ_j,

$$\begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \frac{\partial J(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix} = \frac{1}{m}\begin{bmatrix} \sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_0^{(i)} \\ \sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_1^{(i)} \\ \sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_2^{(i)} \\ \vdots \\ \sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_n^{(i)} \end{bmatrix} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x^{(i)} = \frac{1}{m}X^T(h_\theta(x) - y) \quad (1)$$

where

$$h_\theta(x) - y = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}$$

Note that $x^{(i)}$ is a vector, while $(h_\theta(x^{(i)}) - y^{(i)})$ is a scalar (single number). To understand the last step of the derivation, let $\beta_i = (h_\theta(x^{(i)}) - y^{(i)})$ and observe that:

$$\sum_i \beta_i x^{(i)} = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}\begin{bmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_m \end{bmatrix} = X^T\beta,$$

The expression above allows us to compute all the partial derivatives without any loops. If you are comfortable with linear algebra, we encourage you to work through the matrix multiplications above to convince yourself that the vectorized version does the same computations. You should now

implement Equation (1) to compute the correct vectorized gradient. Once you are done, complete the function `lrCostFunction.m` by implementing the gradient.

Debugging Tip: Vectorizing code can sometimes be tricky. One common strategy for debugging is to print out the sizes of the matrices you are working with using the size function. For example, given a data matrix X of size 100 x 20 (100 examples, 20 features) and θ, a vector with dimensions 20 x 1, you can observe that $X\theta$ is a valid multiplication operation, while θX is not. Furthermore, if you have a non-vectorized version of your code, you can compare the output of your vectorized code and non-vectorized code to make sure that they produce the same outputs.

1.3.3 Vectorizing regularized logistic regression

After you have implemented vectorization for logistic regression, you will now add regularization to the cost function. Recall that for regularized logistic regression, the cost function is defined as

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

Note that you should not be regularizing θ_0 which is used for the bias term. Correspondingly, the partial derivative of regularized logistic regression cost for θ_j is defined as

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \qquad \text{for } j = 0,$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j \qquad \text{for } j \geq 1$$

Now modify your code in `lrCostFunction` to account for regularization. Once again, you should not put any loops into your code. When you are finished, run the code below to test your vectorized implementation and compare to expected outputs:

```
theta_t = [-2; -1; 1; 2];
X_t = [ones(5,1) reshape(1:15,5,3)/10];
y_t = ([1;0;1;0;1] >= 0.5);
lambda_t = 3;
[J, grad] = lrCostFunction(theta_t, X_t, y_t, lambda_t);

fprintf('Cost: %f | Expected cost: 2.534819\n',J);
```

```
Cost: 2.534819 | Expected cost: 2.534819
```

```
fprintf('Gradients:\n'); fprintf('%f\n',grad);
```

```
Gradients:
0.146561
-0.548558
0.724722
```

```
1.398003
```

```
fprintf('Expected gradients:\n 0.146561\n -0.548558\n 0.724722\n 1.398003');
```

```
Expected gradients:
 0.146561
 -0.548558
 0.724722
 1.398003
```

MATLAB Tip: When implementing the vectorization for regularized logistic regression, you might often want to only sum and update certain elements of θ. In MATLAB, you can index into the matrices to access and update only certain elements. For example, `A(:,3:5)= B(:, 1:3)` will replace columns 3 to 5 of `A` with the columns 1 to 3 from `B`. One special keyword you can use in indexing is the `end` keyword in indexing. This allows us to select columns (or rows) until the end of the matrix. For example, `A(:, 2:end)` will only return elements from the 2nd to last column of `A`. Thus, you could use this together with the `sum` and `.^` operations to compute the sum of only the elements you are interested in (e.g. `sum(z(2:end).^2)`). In the starter code, `lrCostFunction.m`, we have also provided hints on yet another possible method computing the regularized gradient.

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

1.4 One-vs-all classication

In this part of the exercise, you will implement one-vs-all classification by training multiple regularized logistic regression classifiers, one for each of the K classes in our dataset (Figure 1). In the handwritten digits dataset, $K = 10$, but your code should work for any value of K.

You should now complete the code in `oneVsAll.m` to train one classifier for each class. In particular, your code should return all the classifier parameters in a matrix $\Theta \in \mathbb{R}^{K \times (N+1)}$, where each row of Θ corresponds to the learned logistic regression parameters for one class. You can do this with a `for` loop from 1 to `K`, training each classifier independently.

Note that the `y` argument to this function is a vector of labels from 1 to 10, where we have mapped the digit '0' to the label 10 (to avoid confusions with indexing). When training the classifier for class $k \in \{1, \ldots, K\}$, you will want a m-dimensional vector of labels y, where $y_j \in \{0, 1\}$ indicates whether the j-th training instance belongs to class k $(y_j = 1)$, or if it belongs to a different class $(y_j = 0)$. You may find logical arrays helpful for this task.

MATLAB Tip: Logical arrays in MATLAB are arrays which contain binary (0 or 1) elements. In MATLAB, evaluating the expression `a == b` for a vector `a` (of size $m \times 1$) and scalar `b` will return a vector of the same size as `a` with ones at positions where the elements of `a` are equal to `b` and zeroes where they are different. To see how this works for yourself, run the following code:

```
a = 1:10; % Create a and b
b = 3;
disp(a == b) % You should try different values of b here
```

```
    0   0   1   0   0   0   0   0   0   0
```

Furthermore, you will be using `fmincg` for this exercise (instead of `fminunc`). `fmincg` works similarly to `fminunc`, but is more more efficient for dealing with a large number of parameters. After you have correctly completed the code for `oneVsAll.m`, run the code below to use your `oneVsAll` function to train a multi-class classifier.

```
num_labels = 10; % 10 labels, from 1 to 10
lambda = 0.1;
[all_theta] = oneVsAll(X, y, num_labels, lambda);
```

```
Iteration     1 | Cost: 2.802128e-01
. . .
Iteration    50 | Cost: 1.003244e-02
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

1.4.1 One-vs-all prediction

After training your one-vs-all classifier, you can now use it to predict the digit contained in a given image. For each input, you should compute the 'probability' that it belongs to each class using the trained logistic regression classifiers. Your one-vs-all prediction function will pick the class for which the corresponding logistic regression classifier outputs the highest probability and return the class label $(1, 2, ..., \text{or } K)$ as the prediction for the input example.

You should now complete the code in `predictOneVsAll.m` to use the one-vs-all classifier to make predictions. Once you are done, run the code below to call your `predictOneVsAll` function using the learned value of Θ. You should see that the training set accuracy is about 94.9% (i.e., it classifies 94.9% of the examples in the training set correctly).

```
pred = predictOneVsAll(all_theta, X);
fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == y)) * 100);
```

```
Training Set Accuracy: 94.880000
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

2. Neural Networks

In the previous part of this exercise, you implemented multi-class logistic regression to recognize handwritten digits. However, logistic regression cannot form more complex hypotheses as it is only a linear classier. (You could add more features such as polynomial features to logistic regression, but that can be very expensive to train.*)* In this part of the exercise, you will implement a neural network to recognize handwritten digits using the same training set as before. The neural network will be able to represent complex models that form non-linear hypotheses.

For this week, you will be using parameters from a neural network that we have already trained. Your goal is to implement the feedforward propagation algorithm to use our weights for prediction. In next week's exercise, you will write the backpropagation algorithm for learning the neural network parameters.

2.1 Model representation

Our neural network is shown in Figure 2. It has 3 layers- an input layer, a hidden layer and an output layer. Recall that our inputs are pixel values of digit images. Since the images are of size 20 x 20, this gives us 400 input layer units (excluding the extra bias unit which always outputs +1). As before, the training data will be loaded into the variables `X` and `y`.

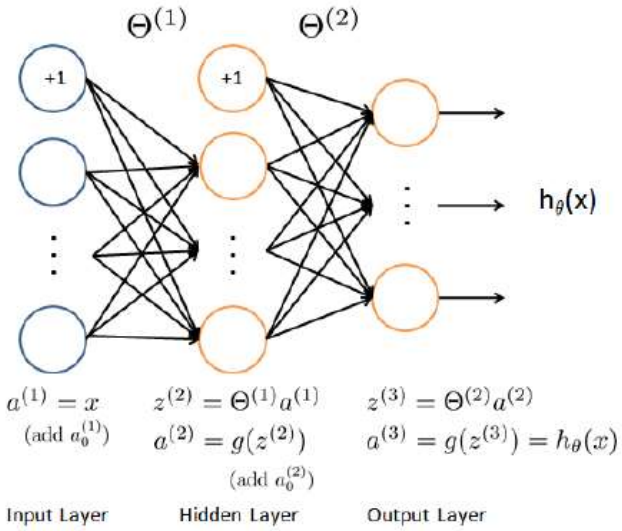
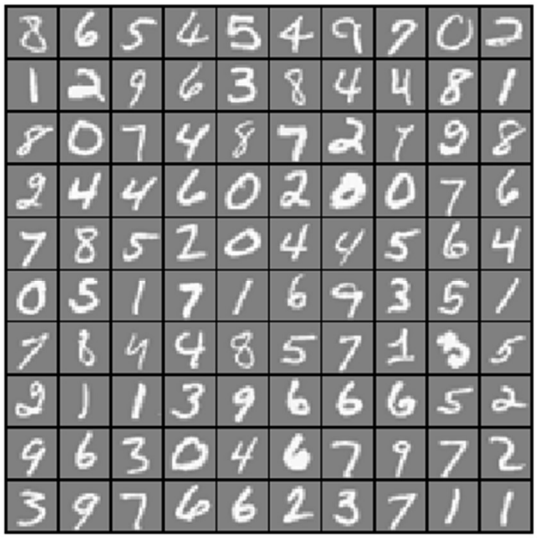


Figure 2: Neural network model.

You have been provided with a set of network parameters $(\Theta^{(1)}, \Theta^{(2)})$ already trained by us. These are stored in `ex3weights.mat` and are loaded into `Theta1` and `Theta2` by running the code below. The parameters have dimensions that are sized for a neural network with 25 units in the second layer and 10 output units (corresponding to the 10 digit classes).

```
load('ex3data1.mat');
m = size(X, 1);

% Randomly select 100 data points to display
sel = randperm(size(X, 1));
sel = sel(1:100);
displayData(X(sel, :));
```



```
% Load saved matrices from file
load('ex3weights.mat');
% Theta1 has size 25 x 401
% Theta2 has size 10 x 26
```

2.2 Feedforward propagation and prediction

Now you will implement feedforward propagation for the neural network. You will need to complete the code in `predict.m` to return the neural network's prediction. You should implement the feedforward computation that computes $h_\theta(x^{(i)})$ for every example i and returns the associated predictions. Similar to the one-vs-all classication strategy, the prediction from the neural network will be the label that has the largest output $(h_\theta(x))_k$.

Implementation Note: The matrix `X` contains the examples in rows. When you complete the code in `predict.m`, you will need to add the column of 1's to the matrix. The matrices `Theta1` and `Theta2` contain the parameters for each unit in rows. Specically, the first row of `Theta1` corresponds to the first hidden unit in the second layer. In MATLAB, when you compute $z^{(2)} = \Theta^{(1)}a^{(1)}$, be sure that you index (and if necessary, transpose) `X` correctly so that you get $a^{(1)}$ as a column vector.

Once you are done, run the code below to call your predict function using the loaded set of parameters for `Theta1` and `Theta2`. You should see that the accuracy is about 97.5%.

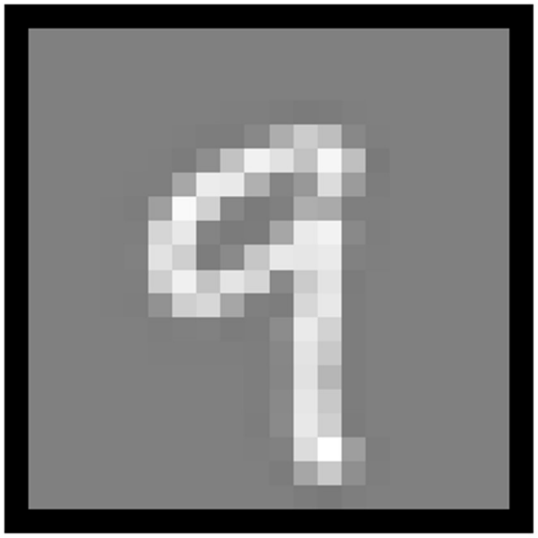
```
pred = predict(Theta1, Theta2, X);
fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == y)) * 100);
```

The code below will displaying images from the training set one at a time, while the console prints out the predicted label for the displayed image. Rerun to repeat with another image.

```
%  Randomly permute examples
rp = randi(m);
% Predict
pred = predict(Theta1, Theta2, X(rp,:));
fprintf('\nNeural Network Prediction: %d (digit %d)\n', pred, mod(pred, 10));
```

```
Neural Network Prediction: 9 (digit 9)
```

```
% Display
displayData(X(rp, :));
```



You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

lrCostFunction.m :

```
function [J, grad] = lrCostFunction(theta, X, y, lambda)
  %LRCOSTFUNCTION Compute cost and gradient for logistic regression with
  %regularization
  %   J = LRCOSTFUNCTION(theta, X, y, lambda) computes the cost of using
  %   theta as the parameter for regularized logistic regression and the
  %   gradient of the cost w.r.t. to the parameters.

  % Initialize some useful values
  m = length(y); % number of training examples

  % You need to return the following variables correctly
  J = 0;
  grad = zeros(size(theta));

  % ====================== YOUR CODE HERE ======================
  % Instructions: Compute the cost of a particular choice of theta.
  %               You should set J to the cost.
  %               Compute the partial derivatives and set grad to the partial
  %               derivatives of the cost w.r.t. each parameter in theta

  %DIMENSIONS:
  %   theta = (n+1) x 1
  %   X     = m x (n+1)
  %   y     = m x 1
  %   grad  = (n+1) x 1
  %   J     = Scalar

  z   = X * theta;   % m x 1
  h_x = sigmoid(z);  % m x 1

  reg_term = (lambda/(2*m)) * sum(theta(2:end).^2);

  J = (1/m)*sum((-y.*log(h_x))-((1-y).*log(1-h_x))) + reg_term; % scalar

  grad(1) = (1/m) * (X(:,1)'*(h_x-y));                                    % 1 x 1
  grad(2:end) = (1/m) * (X(:,2:end)'*(h_x-y)) + (lambda/m)*theta(2:end);  % n x 1

  % =============================================================

  grad = grad(:);
end
```

oneVsAll.m :

```
function [all_theta] = oneVsAll(X, y, num_labels, lambda)
  %ONEVSALL trains multiple logistic regression classifiers and returns all
  %the classifiers in a matrix all_theta, where the i-th row of all_theta
  %corresponds to the classifier for label i
  %   [all_theta] = ONEVSALL(X, y, num_labels, lambda) trains num_labels
  %   logistic regression classifiers and returns each of these classifiers
  %   in a matrix all_theta, where the i-th row of all_theta corresponds
  %   to the classifier for label i

  % num_labels = No. of output classifier (Here, it is 10)

  % Some useful variables
  m = size(X, 1);        % No. of Training Samples == No. of Images : (Here, 5000)
  n = size(X, 2);        % No. of features == No. of pixels in each Image : (Here,
400)

  % You need to return the following variables correctly
  all_theta = zeros(num_labels, n + 1);
  %DIMENSIONS: num_labels x (input_layer_size+1) == num_labels x
(no_of_features+1) == 10 x 401

  %DIMENSIONS: X = m x input_layer_size
  %Here, 1 row in X represents 1 training Image of pixel 20x20

  % Add ones to the X data matrix
  X = [ones(m, 1) X];   %DIMENSIONS: X = m x (input_layer_size+1) = m x
(no_of_features+1)

  % ====================== YOUR CODE HERE ======================
  % Instructions: You should complete the following code to train num_labels
  %               logistic regression classifiers with regularization
  %               parameter lambda.
  %
  % Hint: theta(:) will return a column vector.
  %
  % Hint: You can use y == c to obtain a vector of 1's and 0's that tell you
  %       whether the ground truth is true/false for this class.
  %
  % Note: For this assignment, we recommend using fmincg to optimize the cost
  %       function. It is okay to use a for-loop (for c = 1:num_labels) to
  %       loop over the different classes.
  %
  %       fmincg works similarly to fminunc, but is more efficient when we
  %       are dealing with large number of parameters.
  %
  % Example Code for fmincg:
  %
  %     % Set Initial theta
  %     initial_theta = zeros(n + 1, 1);
  %
  %     % Set options for fminunc
  %     options = optimset('GradObj', 'on', 'MaxIter', 50);
```

```
  %
  %     % Run fmincg to obtain the optimal theta
  %     % This function will return theta and the cost
  %     [theta] = ...
  %         fmincg (@(t)(lrCostFunction(t, X, (y == c), lambda)), ...
  %                 initial_theta, options);
  %

  initial_theta = zeros(n+1, 1);
  options = optimset('GradObj', 'on', 'MaxIter', 50);

  for c=1:num_labels
  all_theta(c,:) = ...
           fmincg (@(t)(lrCostFunction(t, X, (y == c), lambda)), ...
                   initial_theta, options);
  end

  % =========================================================================
end
```

predictOneVsAll.m :

```
function p = predictOneVsAll(all_theta, X)
  %PREDICT Predict the label for a trained one-vs-all classifier. The labels
  %are in the range 1..K, where K = size(all_theta, 1).
  %  p = PREDICTONEVSALL(all_theta, X) will return a vector of predictions
  %  for each example in the matrix X. Note that X contains the examples in
  %  rows. all_theta is a matrix where the i-th row is a trained logistic
  %  regression theta vector for the i-th class. You should set p to a vector
  %  of values from 1..K (e.g., p = [1; 3; 1; 2] predicts classes 1, 3, 1, 2
  %  for 4 examples)

  m = size(X, 1);     % No. of Input Examples to Predict (Each row = 1 Example)
  num_labels = size(all_theta, 1); %No. of Ouput Classifier

  % You need to return the following variables correctly
  p = zeros(size(X, 1), 1);    % No_of_Input_Examples x 1 == m x 1

  % Add ones to the X data matrix
  X = [ones(m, 1) X];

  % ====================== YOUR CODE HERE ======================
  % Instructions: Complete the following code to make predictions using
  %               your learned logistic regression parameters (one-vs-all).
  %               You should set p to a vector of predictions (from 1 to
  %               num_labels).
  %
  % Hint: This code can be done all vectorized using the max function.
  %       In particular, the max function can also return the index of the
  %       max element, for more information see 'help max'. If your examples
  %       are in rows, then, you can use max(A, [], 2) to obtain the max
  %       for each row.
  %
  % num_labels = No. of output classifier (Here, it is 10)
  % DIMENSIONS:
  % all_theta = 10 x 401 = num_labels x (input_layer_size+1) == num_labels x
(no_of_features+1)

  prob_mat = X * all_theta';     % 5000 x 10 == no_of_input_image x num_labels
  [prob, p] = max(prob_mat,[],2); % m  x 1
  %returns maximum element in each row  == max. probability and its index for each
input image
  %p: predicted output (index)
  %prob: probability of predicted output

  % =========================================================================
end
```

predict.m :

```
function p = predict(Theta1, Theta2, X)
  %PREDICT Predict the label of an input given a trained neural network
  %   p = PREDICT(Theta1, Theta2, X) outputs the predicted label of X given the
  %   trained weights of a neural network (Theta1, Theta2)

  % Useful values
  m = size(X, 1);
  num_labels = size(Theta2, 1);

  % You need to return the following variables correctly
  p = zeros(size(X, 1), 1);  % m x 1

  % ====================== YOUR CODE HERE ======================
  % Instructions: Complete the following code to make predictions using
  %               your learned neural network. You should set p to a
  %               vector containing labels between 1 to num_labels.
  %
  % Hint: The max function might come in useful. In particular, the max
  %       function can also return the index of the max element, for more
  %       information see 'help max'. If your examples are in rows, then, you
  %       can use max(A, [], 2) to obtain the max for each row.
  %
  %DIMENSIONS:
  % theta1 = 25 x 401
  % theta2 = 10 x 26

  % layer1 (input)  = 400 nodes + 1bias
  % layer2 (hidden) = 25 nodes + 1bias
  % layer3 (output) = 10 nodes
  %
  % theta dimensions = S_(j+1) x ((S_j)+1)
  % theta1 = 25 x 401
  % theta2 = 10 x 26

  % theta1:
  %     1st row indicates: theta corresponding to all nodes from layer1 connecting
to for 1st node of layer2
  %     2nd row indicates: theta corresponding to all nodes from layer1 connecting
to for 2nd node of layer2
  %     and
  %     1st Column indicates: theta corresponding to node1 from layer1 to all
nodes in layer2
  %     2nd Column indicates: theta corresponding to node2 from layer1 to all
nodes in layer2
  %
  % theta2:
  %     1st row indicates: theta corresponding to all nodes from layer2 connecting
to for 1st node of layer3
  %     2nd row indicates: theta corresponding to all nodes from layer2 connecting
to for 2nd node of layer3
  %     and
```

```
  %     1st Column indicates: theta corresponding to node1 from layer2 to all
nodes in layer3
  %     2nd Column indicates: theta corresponding to node2 from layer2 to all
nodes in layer3

  a1 = [ones(m,1) X]; % 5000 x 401 == no_of_input_images x no_of_features % Adding
1 in X
  %No. of rows = no. of input images
  %No. of Column = No. of features in each image

  z2 = a1 * Theta1';  % 5000 x 25
  a2 = sigmoid(z2);   % 5000 x 25

  a2 =  [ones(size(a2,1),1) a2];  % 5000 x 26

  z3 = a2 * Theta2';  % 5000 x 10
  a3 = sigmoid(z3);  % 5000 x 10

  [prob, p] = max(a3,[],2);
  %returns maximum element in each row  == max. probability and its index for each
input image
  %p: predicted output (index)
  %prob: probability of predicted output

  % =========================================================================
end
```

Week 5

5.1 Neural Networks: Learning

Cost Function and Backpropagation

Cost Function

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

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

Recall that in neural networks, we may have many output nodes. We denote $h_\Theta(x)_k$ as being a hypothesis that results in the 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, before the square brackets, we have an additional nested summation that loops through the number of output nodes.

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

Note:

- the double sum simply adds up the logistic regression costs calculated for each cell in the output layer
- the triple sum simply adds up the squares of all the individual Θs in the entire network.
- the i in the triple sum does **not** refer to training example i

Backpropagation Algorithm

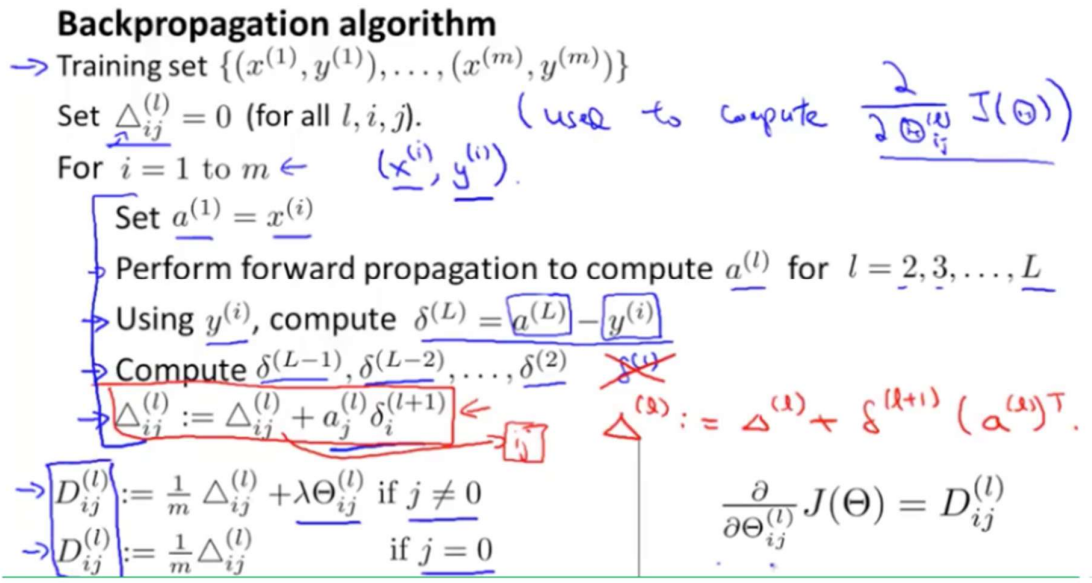
"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute:

$\min_\Theta J(\Theta)$

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):

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

To do so, we use the following 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), (hence you end up having a matrix full of zeros)

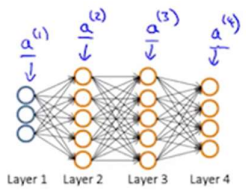
For training example t =1 to m:

1. Set $a^{(1)} := x^{(t)}$
2. Perform forward propagation to compute $a^{(l)}$ for l=2,3,...,L

Gradient computation

Given one training example (x, y):
Forward propagation:

$a^{(1)} = x$
$z^{(2)} = \Theta^{(1)} a^{(1)}$
$a^{(2)} = g(z^{(2)})$ (add $a_0^{(2)}$)
$z^{(3)} = \Theta^{(2)} a^{(2)}$
$a^{(3)} = g(z^{(3)})$ (add $a_0^{(3)}$)
$z^{(4)} = \Theta^{(3)} a^{(3)}$
$a^{(4)} = h_\Theta(x) = g(z^{(4)})$



3. Using $y^{(t)}$, compute $\delta^{(L)} = a^{(L)} - y^{(t)}$

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:

4. Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$ using $\delta^{(l)} = ((\Theta^{(l)})^T \delta^{(l+1)}) \, .* \, a^{(l)} \, .* \, (1 - a^{(l)})$

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:

$$g'(z^{(l)}) = a^{(l)} \, .* \, (1 - a^{(l)})$$

5. $\Delta_{i,j}^{(l)} := \Delta_{i,j}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ or with vectorization, $\Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$

Hence we update our new Δ matrix.

- $D_{i,j}^{(l)} := \frac{1}{m} \left(\Delta_{i,j}^{(l)} + \lambda \Theta_{i,j}^{(l)} \right)$, if j≠0.
- $D_{i,j}^{(l)} := \frac{1}{m} \Delta_{i,j}^{(l)}$ If j=0

The capital-delta matrix D is used as an "accumulator" to add up our values as we go along and eventually compute our partial derivative. Thus we get $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$

Backpropagation Intuition

Note: [4:39, the last term for the calculation for z_1^3 (three-color handwritten formula) should be a_2^2 instead of a_1^2. 6:08 - the equation for cost(i) is incorrect. The first term is missing parentheses for the log() function, and the second term should be $(1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))$. 8:50 - $\delta^{(4)} = y - a^{(4)}$ is incorrect and should be $\delta^{(4)} = a^{(4)} - y$.]

Recall that the cost function for a neural network is:

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

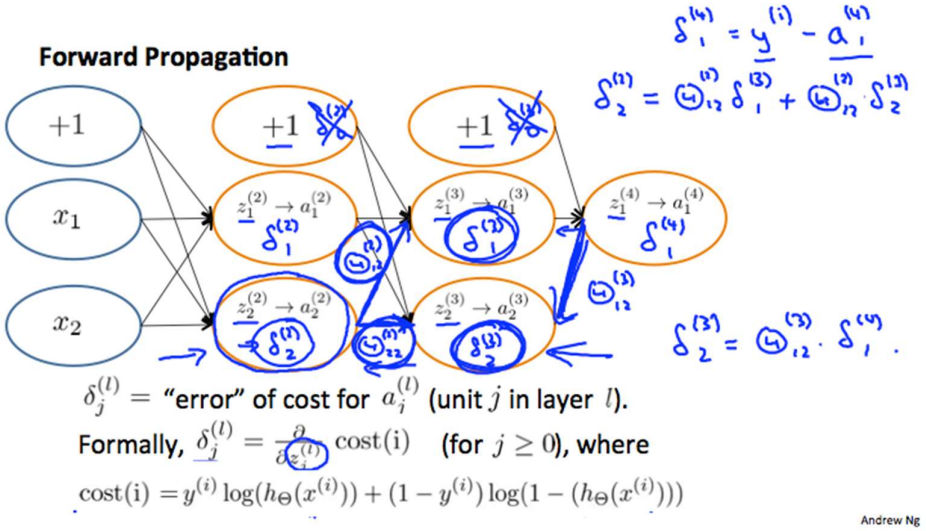
If we consider simple non-multiclass classification (k = 1) and disregard regularization, the cost is computed with:

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

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. Let us consider the following neural network below and see how we could calculate some $\delta_j^{(l)}$:



In the image above, to calculate $\delta_2^{(2)}$, we multiply the weights $\Theta_{12}^{(2)}$ and $\Theta_{22}^{(2)}$ by their respective δ values found to the right of each edge. So we get $\delta_2^{(2)} = \Theta_{12}^{(2)} * \delta_1^{(3)} + \Theta_{22}^{(2)} * \delta_2^{(3)}$. To calculate every single possible $\delta_j^{(l)}$, we could start from the right of our diagram. We can think of our edges as our Θ_{ij}. Going from right to left, to calculate the value of $\delta_j^{(l)}$, you can just take the over all sum of each weight times the δ it is coming from. Hence, another example would be $\delta_2^{(3)} = \Theta_{12}^{(3)} * \delta_1^{(4)}$.

Backpropagation in Practice

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)

```

To summarize:

Learning Algorithm

-> Have initial parameters $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

-> Unroll to get `initialTheta` to pass to

-> `fminunc(@costFunction, initialTheta, options)`

`function [jval, gradientVec] = costFunction(thetaVec)`

From `thetaVec`, get $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$.

Use forward prop/back prop to compute $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\Theta)$.

Unroll $D^{(1)}, D^{(2)}, D^{(3)}$ to get `gradientVec`.

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 small value for ϵ (epsilon) such as $\epsilon = 10^{-4}$, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems.

Hence, we are only adding or subtracting epsilon to the Θ_j matrix. In octave we can do it as follows:

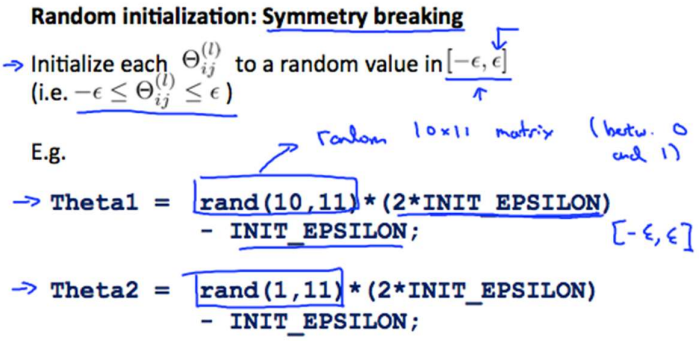
```
epsilon = 1e-4;
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) += epsilon;
  thetaMinus = theta;
  thetaMinus(i) -= epsilon;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
end;

```

We previously saw how to calculate the deltaVector. So once we compute our gradApprox vector, we can check that gradApprox ≈ deltaVector.

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

Random Initialisation

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 for our Θ matrices using the following method:



Hence, we initialize each $\Theta_{ij}^{(l)}$ to a random value between $[-\epsilon, \epsilon]$. Using the above formula guarantees that we get the desired bound. The same procedure applies to all the Θ's. Below is some working code you could use to experiment.

```
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) is just a function in octave that will initialize a matrix of random real numbers between 0 and 1.

(Note: the epsilon used above is unrelated to the epsilon from Gradient Checking)

Putting it Together

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

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

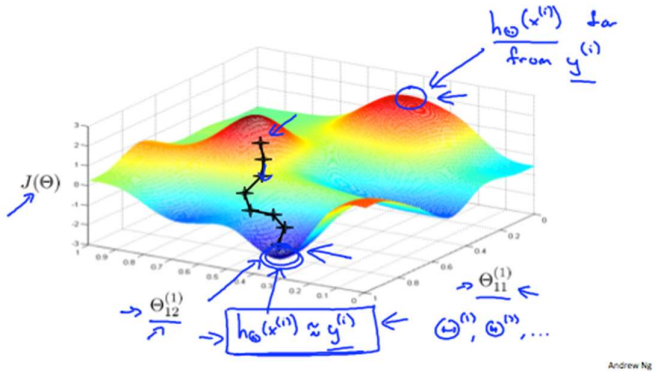
Training a Neural Network

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

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
```

The following image gives us an intuition of what is happening as we are implementing our neural network:



Ideally, you want $h_\Theta(x^{(i)}) \approx y^{(i)}$. This will minimize our cost function. However, keep in mind that $J(\Theta)$ is not convex and thus we can end up in a local minimum instead.

Machine Learning: Programming Exercise 4

Neural Networks Learning

In this exercise, you will implement the backpropagation algorithm for neural networks and apply it to the task of hand-written digit recognition.

Files needed for this exercise

- `ex4.mlx` - MATLAB Live Script that steps you through the exercise
- `ex4data1.mat` - Training set of hand-written digits
- `ex4weights.mat` - Neural network parameters for exercise 4
- `submit.m` - Submission script that sends your solutions to our servers
- `displayData.m` - Function to help visualize the dataset
- `fmincg.m` - Function minimization routine (similar to `fminunc`)
- `sigmoid.m` - Sigmoid function
- `computeNumericalGradient.m` - Numerically compute gradients
- `checkNNGradients.m` - Function to help check your gradients
- `debugInitializeWeights.m` - Function for initializing weights
- `predict.m` - Neural network prediction function
- *`sigmoidGradient.m` - Compute the gradient of the sigmoid function
- *`randInitializeWeights.m` - Randomly initialize weights
- *`nnCostFunction.m` - Neural network cost function

**** indicates files you will need to complete***

Clear existing variables and confirm that your Current Folder is set correctly

Click into this section, then click the 'Run Section' button above. This will execute the `clear` command to clear existing variables and the `dir` command to list the files in your Current Folder. The output should contain all of the files listed above and the 'lib' folder. If it does not, right-click the 'ex4' folder and select 'Open' before proceding or see the instructions in `README.mlx` for more details.

```
clear
dir
```

```
.                         computeNumericalGradient.m  ex4.mlx
ex4weights.mat            nnCostFunction.m            sigmoid.m
token.mat
..                        debugInitializeWeights.m    ex4_companion.mlx          fmincg.m
predict.m                 sigmoidGradient.m
checkNNGradients.m        displayData.m               ex4data1.mat               lib
randInitializeWeights.m   submit.m
```

Before you begin

The workflow for completing and submitting the programming exercises in MATLAB Online differs from the original course instructions. Before beginning this exercise, make sure you have read

through the instructions in `README.mlx` which is included with the programming exercise files. `README` also contains solutions to the many common issues you may encounter while completing and submitting the exercises in MATLAB Online. Make sure you are following instructions in `README` and have checked for an existing solution before seeking help on the discussion forums.

Table of Contents

Neural Networks Learning
    Files needed for this exercise
        Clear existing variables and confirm that your Current Folder is set correctly
    Before you begin
1. Neural Networks
    1.1 Visualizing the data
    1.2 Model representation
    1.3 Feedforward and cost function
    1.4 Regularized cost function
2. Backpropagation
    2.1 Sigmoid gradient
    2.2 Random initialization
    2.3 Backpropagation
    2.4 Gradient checking
    2.5 Regularized neural networks
    2.6 Learning parameters using fmincg
3. Visualizing the hidden layer
    3.1 Optional (ungraded) exercise
Submission and Grading

1. Neural Networks

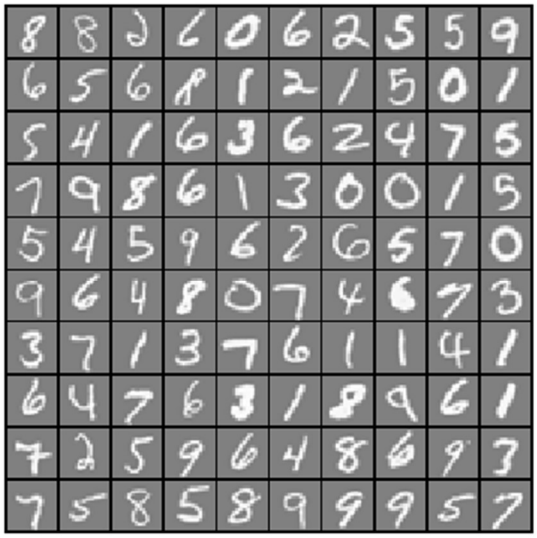
In the previous exercise, you implemented feedforward propagation for neural networks and used it to predict handwritten digits with the weights we provided. In this exercise, you will implement the backpropagation algorithm to learn the parameters for the neural network.

1.1 Visualizing the data

The code below will load the data and display it on a 2-dimensional plot (Figure 1) by calling the function `displayData`. This is the same dataset that you used in the previous exercise. Run the code below to load the training data into the variables `X` and `y`.

Figure 1: Examples from the dataset

```
load('ex4data1.mat');
m = size(X, 1);

% Randomly select 100 data points to display
sel = randperm(size(X, 1));
sel = sel(1:100);
displayData(X(sel, :));
```



There are 5000 training examples in `ex4data1.mat`, where each training example is a 20 pixel by 20 pixel grayscale image of the digit. Each pixel is represented by a floating point number indicating the grayscale intensity at that location. The 20 by 20 grid of pixels is 'unrolled' into a 400-dimensional vector. Each of these training examples becomes a single row in our data matrix `X`. This gives us a 5000 by 400 matrix `X` where every row is a training example for a handwritten digit image.

$$X = \begin{bmatrix} -(x^{(1)})^T - \\ -(x^{(2)})^T - \\ \vdots \\ -(x^{(m)})^T - \end{bmatrix}$$

The second part of the training set is a 5000-dimensional vector y that contains labels for the training set. To make things more compatible with MATLAB indexing, where there is no zero index, we have mapped the digit zero to the value ten. Therefore, a '0' digit is labeled as '10', while the digits '1' to '9' are labeled as '1' to '9' in their natural order.

1.2 Model representation

Our neural network is shown in Figure 2. It has 3 layers- an input layer, a hidden layer and an output layer. Recall that our inputs are pixel values of digit images. Since the images are of size 20 x 20, this gives us 400 input layer units (not counting the extra bias unit which always outputs +1).

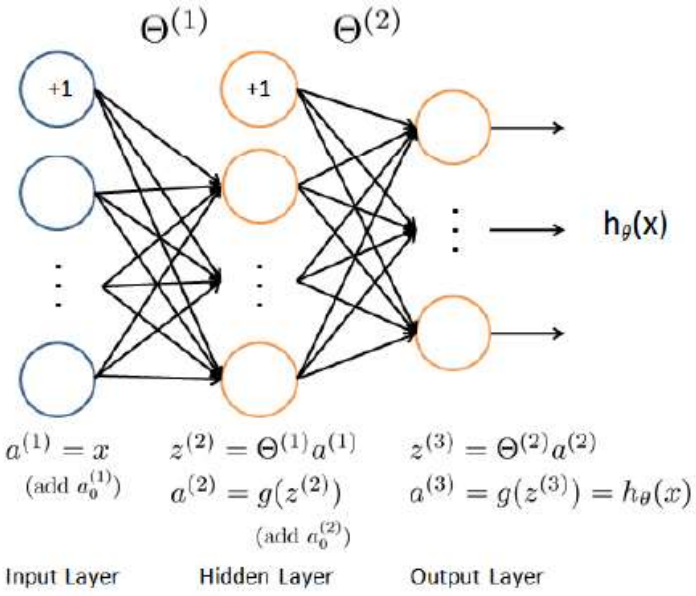


Figure 2: Neural network model.

You have been provided with a set of network parameters $(\Theta^{(1)}, \Theta^{(2)})$ already trained by us. These are stored in `ex4weights.mat`. Run the code below to load them into `Theta1` and `Theta2`. The parameters have dimensions that are sized for a neural network with 25 units in the second layer and 10 output units (corresponding to the 10 digit classes).

```
% Load the weights into variables Theta1 and Theta2
load('ex4weights.mat');
```

1.3 Feedforward and cost function

Now you will implement the cost function and gradient for the neural network. First, complete the code in `nnCostFunction.m` to return the cost. Recall that the cost function for the neural network (without regularization) is

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log\left((h_\theta(x^{(i)}))_k\right) - (1-y_k^{(i)})\log\left(1-(h_\theta(x^{(i)}))_k\right)\right],$$

where $h_\theta(x^{(i)})$ is computed as shown in the Figure 2 and $K = 10$ is the total number of possible labels. Note that $h_\theta(x^{(i)})_k = a_k^{(3)}$ is the activation (output value) of the k-th output unit. Also, recall that whereas the original labels (in the variable y) were $1, 2, \ldots, 10$, for the purpose of training a neural network, we need to recode the labels as vectors containing only values 0 or 1, so that

$$y = \begin{bmatrix}1\\0\\0\\\vdots\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\\vdots\\0\end{bmatrix}, \ldots \text{or} \begin{bmatrix}0\\0\\0\\\vdots\\1\end{bmatrix}$$

For example, if $x^{(i)}$ is an image of the digit 5, then the corresponding $y^{(i)}$ (that you should use with the cost function) should be a 10-dimensional vector with $y_5 = 1$, and the other elements equal to 0. You should implement the feedforward computation that computes $h_\theta(x^{(i)})$ for every example i and sum the cost over all examples. Your code should also work for a dataset of any size, with any number of labels (you can assume that there are always at least $K \geq 3$ labels).

Implementation Note: The matrix `X` contains the examples in rows (i.e., `X(i,:)'` is the `i`-th training example $x^{(i)}$, expressed as a n x 1 vector.) When you complete the code in `nnCostFunction.m`, you will need to add the column of 1's to the `X` matrix. The parameters for each unit in the neural network is represented in `Theta1` and `Theta2` as one row. Specifically, the first row of `Theta1` corresponds to the first hidden unit in the second layer. You can use a `for` loop over the examples to compute the cost. We suggest implementing the feedforward cost *without* regularization first so that it will be easier for you to debug. Later, you will get to implement the regularized cost.

Once you are done, run the code below to call your `nnCostFunction` using the loaded set of parameters for `Theta1` and `Theta2`. You should see that the cost is about 0.287629.

```
input_layer_size  = 400;  % 20x20 Input Images of Digits
hidden_layer_size = 25;   % 25 hidden units
```

```
 num_labels = 10;          % 10 labels, from 1 to 10 (note that we have mapped
"0" to label 10)

 % Unroll parameters
 nn_params = [Theta1(:) ; Theta2(:)];

 % Weight regularization parameter (we set this to 0 here).
 lambda = 0;

 J = nnCostFunction(nn_params, input_layer_size, hidden_layer_size, num_labels,
X, y, lambda);

 fprintf('Cost at parameters (loaded from ex4weights): %f', J);
```

```
Cost at parameters (loaded from ex4weights): 0.287629
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

1.4 Regularized cost function

The cost function for neural networks with regularization is given by

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log\left((h_\theta(x^{(i)})_k\right) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right]$$
$$+\frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}\left(\Theta_{j,k}^{(1)}\right)^2+\sum_{j=1}^{10}\sum_{k=1}^{25}\left(\Theta_{j,k}^{(2)}\right)^2\right]$$

You can assume that the neural network will only have 3 layers- an input layer, a hidden layer and an output layer. However, your code should work for any number of input units, hidden units and outputs units. While we have explicitly listed the indices above for $\Theta^{(1)}$ and $\Theta^{(2)}$ for clarity, do note that **your code should in general work with $\Theta^{(1)}$ and $\Theta^{(2)}$ of any size.**

Note that you should not be regularizing the terms that correspond to the bias. For the matrices Theta1 and Theta2, this corresponds to the first column of each matrix. You should now add regularization to your cost function. Notice that you can first compute the unregularized cost function J using your existing `nnCostFunction.m` and then later add the cost for the regularization terms. Once you are done, run the code below to call your `nnCostFunction` using the loaded set of parameters for `Theta1` and `Theta2`, and $\lambda = 1$. You should see that the cost is about 0.383770.

```
 % Weight regularization parameter (we set this to 1 here).
 lambda = 1;

 J = nnCostFunction(nn_params, input_layer_size, hidden_layer_size, num_labels,
X, y, lambda);
 fprintf('Cost at parameters (loaded from ex4weights): %f', J);
```

```
Cost at parameters (loaded from ex4weights): 0.383770
```

You should now submit your solutions. Enter/confirm your login and token in the command window when prompted.

2. Backpropagation

In this part of the exercise, you will implement the backpropagation algorithm to compute the gradient for the neural network cost function. You will need to complete the `nnCostFunction.m` so that it returns an appropriate value for `grad`. Once you have computed the gradient, you will be able to train the neural network by minimizing the cost function $J(\theta)$ using an advanced optimizer such as `fmincg`.

You will first implement the backpropagation algorithm to compute the gradients for the parameters for the (unregularized) neural network. After you have verified that your gradient computation for the unregularized case is correct, you will implement the gradient for the regularized neural network.

2.1 Sigmoid gradient

To help you get started with this part of the exercise, you will first implement the sigmoid gradient function. The gradient for the sigmoid function can be computed as

$$g'(z) = \frac{d}{dz} g(z) = g(z)(1 - g(z))$$

where

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

When you are done, try testing a few values by calling `sigmoidGradient(z)` below. For large values (both positive and negative) of `z`, the gradient should be close to 0. When `z = 0`, the gradient should be exactly 0.25. Your code should also work with vectors and matrices. For a matrix, your function should perform the sigmoid gradient function on every element.

```
% Call your sigmoidGradient function
sigmoidGradient(0)
```

```
ans = 0.2500
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

2.2 Random initialization

When training neural networks, it is important to randomly initialize the parameters for symmetry breaking. One effective strategy for random initialization is to randomly select values for $\Theta^{(l)}$ uniformly in the range $[-\epsilon_{int}, \epsilon_{init}]$. You should use $\epsilon_{init} = 0.12$*. This range of values ensures that the parameters are kept small and makes the learning more efficient.

Your job is to complete `randInitializeWeights.m` to initialize the weights for Θ; modify the file and fill in the following code:

```
% Randomly initialize the weights to small values
epsilon_init = 0.12;
W = rand(L out, 1 + L in) * 2 * epsilon_init - epsilon init;
```

When you are done, run the code below to call `randInitialWeights` and initialize the neural network parameters.

```
initial_Theta1 = randInitializeWeights(input_layer_size, hidden_layer_size);
initial_Theta2 = randInitializeWeights(hidden_layer_size, num_labels);

% Unroll parameters
initial_nn_params = [initial_Theta1(:) ; initial_Theta2(:)];
```

*One effective strategy for choosing ϵ_{init} is to base it on the number of units in the network. A good choice of ϵ_{init} is $\epsilon_{init} = \frac{\sqrt{6}}{\sqrt{L_{in} + L_{out}}}$, where $L_{in} = s_l$ and $L_{out} = s_l + 1$ are the number of units in the layers adjacent to $\Theta^{(l)}$.

You do not need to submit any code for this part of the exercise.

2.3 Backpropagation

Now, you will implement the backpropagation algorithm. Recall that the intuition behind the backpropagation algorithm is as follows. Given a training example $(x^{(t)}, y^{(t)})$, we will first run a 'forward pass' to compute all the activations throughout the network, including the output value of the hypothesis $h_\Theta(x)$. Then, for each node j in layer l, we would like to compute an 'error term' $\delta_j^{(l)}$ that measures how much that node was 'responsible' for any errors in our output.

For an output node, we can directly measure the difference between the network's activation and the true target value, and use that to define $\delta_j^{(3)}$ (since layer 3 is the output layer). For the hidden units, you will compute $\delta_j^{(l)}$ based on a weighted average of the error terms of the nodes in layer $(l+1)$.

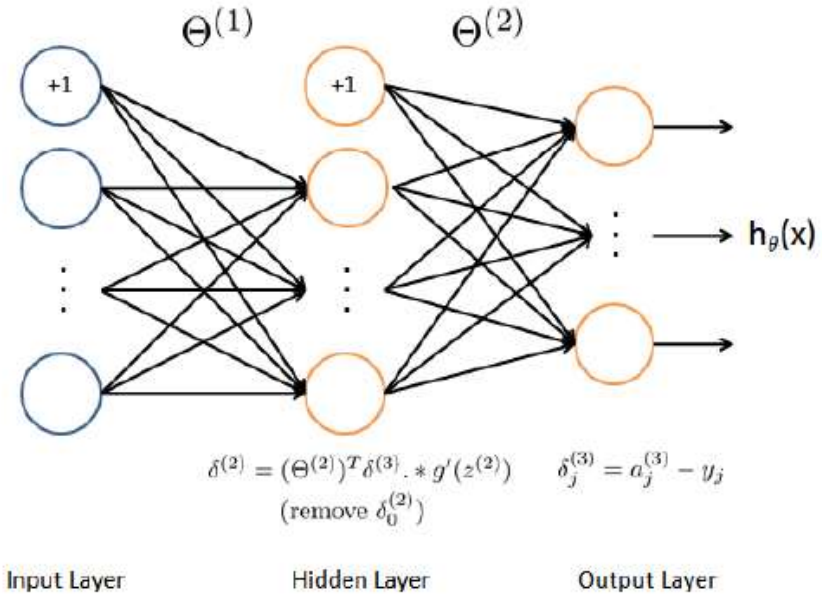


Figure 3: Backpropagation Updates.

In detail, here is the backpropagation algorithm (also depicted in Figure 3). You should implement steps 1 to 4 in a loop that processes one example at a time. Concretely, you should implement a `for` loop for `t = 1:m` and place steps 1-4 below inside the `for` loop, with the t^{th} iteration performing the calculation on the t^{th} training example $(x^{(t)}, y^{(t)})$. Step 5 will divide the accumulated gradients by m to obtain the gradients for the neural network cost function.

1. Set the input layer's values $(a^{(1)})$ to the t-th training example $x^{(t)}$. Perform a feedforward pass (Figure 2), computing the activations $(z^{(2)}, a^{(2)}, z^{(3)}, a^{(3)})$ for layers 2 and 3. Note that you need to add a $+1$ term to ensure that the vectors of activations for layers $a^{(1)}$ and $a^{(2)}$ also include the bias unit. In MATLAB, if `a_1` is a column vector, adding one corresponds to `a_1 = [1; a_1]`.
2. For each output unit k in layer 3 (the output layer), set $\delta_k^{(3)} = (a_k^{(3)} - y_k)$ where $y_k \in \{0, 1\}$ indicates whether the current training example belongs to class k $(y_k = 1)$, or if it belongs to a different class $(y_k = 0)$.You may find logical arrays helpful for this task (explained in the previous programming exercise).
3. For the hidden layer $l = 2$, set $\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$
4. Accumulate the gradient from this example using the following formula: $\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$. Note that you should skip or remove $\delta_0^{(2)}$. In MATLAB, removing $\delta_0^{(2)}$ corresponds to `delta_2 = delta_2(2:end)`.
5. Obtain the (unregularized) gradient for the neural network cost function by dividing the accumulated gradients by $\frac{1}{m}$: $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$

MATLAB Tip: You should implement the backpropagation algorithm only after you have successfully completed the feedforward and cost functions. While implementing the backpropagation

algorithm, it is often useful to use the size function to print out the sizes of the variables you are working with if you run into dimension mismatch errors ("nonconformant arguments") errors.

After you have implemented the backpropagation algorithm, the code in the next section will run gradient checking on your implementation. The gradient check will allow you to increase your confidence that your code is computing the gradients correctly.

2.4 Gradient checking

In your neural network, you are minimizing the cost function $J(\Theta)$. To perform gradient checking on your parameters, you can imagine 'unrolling' the parameters $\Theta^{(1)}, \Theta^{(2)}$ into a long vector θ. By doing so, you can think of the cost function being $J(\theta)$ instead and use the following gradient checking procedure.

Suppose you have a function $f_i(\theta)$ that purportedly computes $\frac{\partial}{\partial \theta_i} J(\theta)$; you'd like to check if f_i is outputting correct derivative values.

$$\text{Let } \theta^{i+} = \theta + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix} \text{ and } \theta^{i-} = \theta - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix}$$

So, $\theta^{(i+)}$ is the same as θ, except its i-th element has been incremented by ϵ. Similarly, $\theta^{(i-)}$is the corresponding vector with the i-th element decreased by ϵ. You can now numerically verify $f_i(\theta)$'s correctness by checking, for each i, that:

$$f_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\epsilon}$$

The degree to which these two values should approximate each other will depend on the details of J. But assuming $\epsilon = 10^{-4}$, you'll usually find that the left- and right-hand sides of the above will agree to at least 4 signicant digits (and often many more). We have implemented the function to compute the numerical gradient for you in `computeNumericalGradient.m`. While you are not required to modify the file, we highly encourage you to take a look at the code to understand how it works.

The code below will run the provided function `checkNNGradients.m` which will create a small neural network and dataset that will be used for checking your gradients. If your backpropagation implementation is correct, you should see a relative dierence that is less than 1e-9.

```
checkNNGradients;
```

```
    -0.0093   -0.0093
     0.0089    0.0089
    -0.0084   -0.0084
```

```
 0.0076    0.0076
-0.0067   -0.0067
-0.0000   -0.0000
 0.0000    0.0000
-0.0000   -0.0000
 0.0000    0.0000
-0.0000   -0.0000
-0.0002   -0.0002
 0.0002    0.0002
-0.0003   -0.0003
 0.0003    0.0003
-0.0004   -0.0004
-0.0001   -0.0001
 0.0001    0.0001
-0.0001   -0.0001
 0.0002    0.0002
-0.0002   -0.0002
 0.3145    0.3145
 0.1111    0.1111
 0.0974    0.0974
 0.1641    0.1641
 0.0576    0.0576
 0.0505    0.0505
 0.1646    0.1646
 0.0578    0.0578
 0.0508    0.0508
 0.1583    0.1583
 0.0559    0.0559
 0.0492    0.0492
 0.1511    0.1511
 0.0537    0.0537
 0.0471    0.0471
 0.1496    0.1496
 0.0532    0.0532
 0.0466    0.0466

The above two columns you get should be very similar.
(Left-Your Numerical Gradient, Right-Analytical Gradient)

If your backpropagation implementation is correct, then
the relative difference will be small (less than 1e-9).

Relative Difference: 2.2366e-11
```

Practical Tip: When performing gradient checking, it is much more efficient to use a small neural network with a relatively small number of input units and hidden units, thus having a relatively small number of parameters. Each dimension of θ requires two evaluations of the cost function and this can be expensive. In the function `checkNNGradients`, our code creates a small random model and dataset which is used with `computeNumericalGradient` for gradient checking. Furthermore, after you are condent that your gradient computations are correct, you should turn off gradient checking before running your learning algorithm.

Practical Tip: Gradient checking works for any function where you are computing the cost and the gradient. Concretely, you can use the same `computeNumericalGradient.m` function to check if your gradient implementations for the other exercises are correct too (e.g. logistic regression's cost function). Once your cost function passes the gradient check for the (unregularized) neural network cost function, you should submit the neural network gradient function (backpropagation).

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

2.5 Regularized neural networks

After you have successfully implemeted the backpropagation algorithm, you will add regularization to the gradient. To account for regularization, it turns out that you can add this as an additional term after computing the gradients using backpropagation. Specically, after you have computed $\Delta_{ij}^{(l)}$ using backpropagation, you should add regularization using

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} \text{ for } j = 0,$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \frac{\lambda}{m} \Theta_{ij}^{(l)} \text{ for } j \geq 1$$

Note that you should *not* be regularizing the first column of $\Theta^{(l)}$ which is used for the bias term. Furthermore, in the parameters $\Theta_{ij}^{(l)}$, i is indexed starting from 1, and j is indexed starting from 0. Thus,

$$\Theta^{(l)} = \begin{bmatrix} \Theta_{1,0}^{(l)} & \Theta_{1,1}^{(l)} & \cdots \\ \Theta_{2,0}^{(l)} & \Theta_{2,1}^{(l)} & \\ \vdots & & \ddots \end{bmatrix}$$

Somewhat confusingly, indexing in MATLAB starts from 1 (for both i and j), thus `Theta1(2, 1)` actually corresponds to $\Theta_{2,0}^{(l)}$ (i.e., the entry in the second row, first column of the matrix $\Theta^{(1)}$ shown above)

Now, modify your code that computes `grad` in `nnCostFunction` to account for regularization. After you are done, run the code below to run gradient checking on your implementation. If your code is correct, you should expect to see a relative difference that is less than 1e-9.

```
%  Check gradients by running checkNNGradients
lambda = 3;
checkNNGradients(lambda);
```

```
   -0.0093   -0.0093
    0.0089    0.0089
   -0.0084   -0.0084
    0.0076    0.0076
   -0.0067   -0.0067
```

```
   -0.0168   -0.0168
    0.0394    0.0394
    0.0593    0.0593
    0.0248    0.0248
   -0.0327   -0.0327
   -0.0602   -0.0602
   -0.0320   -0.0320
    0.0249    0.0249
    0.0598    0.0598
    0.0386    0.0386
   -0.0174   -0.0174
   -0.0576   -0.0576
   -0.0452   -0.0452
    0.0091    0.0091
    0.0546    0.0546
    0.3145    0.3145
    0.1111    0.1111
    0.0974    0.0974
    0.1187    0.1187
    0.0000    0.0000
    0.0337    0.0337
    0.2040    0.2040
    0.1171    0.1171
    0.0755    0.0755
    0.1257    0.1257
   -0.0041   -0.0041
    0.0170    0.0170
    0.1763    0.1763
    0.1131    0.1131
    0.0862    0.0862
    0.1323    0.1323
   -0.0045   -0.0045
    0.0015    0.0015

  The above two columns you get should be very similar.
  (Left-Your Numerical Gradient, Right-Analytical Gradient)

  If your backpropagation implementation is correct, then
  the relative difference will be small (less than 1e-9).

  Relative Difference: 2.17629e-11
```

```
 % Also output the costFunction debugging value
 % This value should be about 0.576051
 debug_J  = nnCostFunction(nn_params, input_layer_size, hidden_layer_size,
num_labels, X, y, lambda);
 fprintf('Cost at (fixed) debugging parameters (w/ lambda = 3): %f', debug_J);
```

```
  Cost at (fixed) debugging parameters (w/ lambda = 3): 0.576051
```

You should now submit your solutions. Enter **`submit`** *at the command prompt, then enter or confirm your login and token when prompted.*

2.6 Learning parameters using `fmincg`

After you have successfully implemented the neural network cost function and gradient computation, run the code below to use `fmincg` to learn a good set of parameters. After the training completes, the code will report the training accuracy of your classifier by computing the percentage of examples it got correct. If your implementation is correct, you should see a reported training accuracy of about 95.3% (this may vary by about 1% due to the random initialization). It is possible to get higher training accuracies by training the neural network for more iterations.

```
 options = optimset('MaxIter', 50);
 lambda = 1;

 % Create "short hand" for the cost function to be minimized
 costFunction = @(p) nnCostFunction(p, input_layer_size, hidden_layer_size,
num_labels, X, y, lambda);

 % Now, costFunction is a function that takes in only one argument (the
 % neural network parameters)
 [nn_params, ~] = fmincg(costFunction, initial_nn_params, options);
```

```
  Iteration     1 | Cost: 3.291172e+00
  . . .
  Iteration    50 | Cost: 2.728559e+00
```

```
 % Obtain Theta1 and Theta2 back from nn_params
 Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)),
hidden_layer_size, (input_layer_size + 1));
 Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size +
1))):end), num_labels, (hidden_layer_size + 1));

 pred = predict(Theta1, Theta2, X);
 fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == y)) * 100);
```

```
  Training Set Accuracy: 33.900000
```

3. Visualizing the hidden layer

One way to understand what your neural network is learning is to visualize what the representations captured by the hidden units. Informally, given a particular hidden unit, one way to visualize what it computes is to find an input x that will cause it to activate (that is, to have an activation value $(a_i^{(l)})$ close to 1). For the neural network you trained, notice that the i^{th} row of $\Theta^{(1)}$ is a 401-dimensional vector that represents the parameter for the i^{th} hidden unit. If we discard the bias term, we get a 400 dimensional vector that represents the weights from each input pixel to the hidden unit.

Thus, one way to visualize the 'representation' captured by the hidden unit is to reshape this 400 dimensional vector into a 20 x 20 image and display it.*

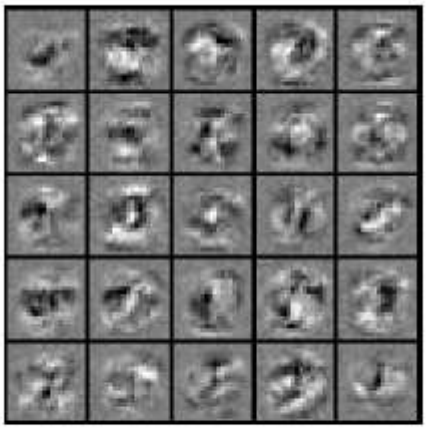
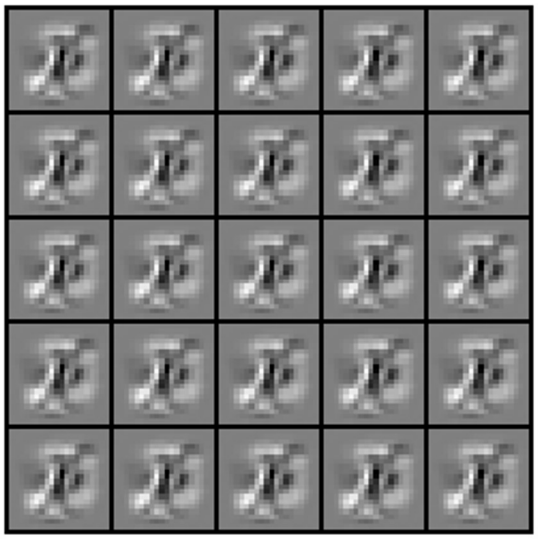


Figure 4: Visualization of Hidden Units.

The code below does this by using the `displayData` function and it will show you an image (similar to Figure 4) with 25 units, each corresponding to one hidden unit in the network. In your trained network, you should find that the hidden units corresponds roughly to detectors that look for strokes and other patterns in the input.

```
% Visualize Weights
displayData(Theta1(:, 2:end));
```



**It turns out that this is equivalent to finding the input that gives the highest activation for the hidden unit, given a 'norm' constraint on the input (i.e.,* $\|x\|_2 \leq 1$*).*

3.1 Optional (ungraded) exercise

In this part of the exercise, you will get to try out different learning settings for the neural network to see how the performance of the neural network varies with the regularization parameter and number of training steps (the `MaxIter` option when using `fmincg`). Neural networks are very powerful models that can form highly complex decision boundaries. Without regularization, it is possible for a neural network to 'overfit' a training set so that it obtains close to 100% accuracy on the training set but does not do as well on new examples that it has not seen before. You can set the regularization λ to a smaller value and the `MaxIter` parameter to a higher number of iterations to see this for youself. You will also be able to see for yourself the changes in the visualizations of the hidden units when you change the learning parameters λ and `MaxIter`.

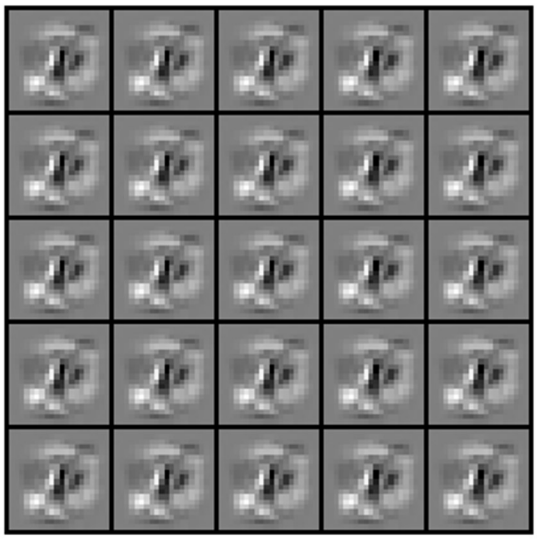
```
% Change lambda and MaxIter to see how it affects the result
lambda = 3;
MaxIter = 50;

options = optimset('MaxIter', MaxIter);

% Create "short hand" for the cost function to be minimized
costFunction = @(p) nnCostFunction(p,input_layer_size, hidden_layer_size,
num_labels, X, y, lambda);

% Now, costFunction is a function that takes in only one argument (the neural
network parameters)
[nn_params, ~] = fmincg(costFunction, initial_nn_params, options);
```

```
Iteration     1 | Cost: 3.292204e+00
. . .
Iteration    50 | Cost: 2.726447e+00
```

```
% Obtain Theta1 and Theta2 back from nn_params
Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)),
hidden_layer_size, (input_layer_size + 1));
Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size +
1))):end), num_labels, (hidden_layer_size + 1));

pred = predict(Theta1, Theta2, X);
fprintf('\nTraining Set Accuracy: %f\n', mean(double(pred == y)) * 100);
```

```
Training Set Accuracy: 32.160000
```

```
% Visualize Weights
displayData(Theta1(:, 2:end));
```



You do not need to submit any solutions for this optional (ungraded) exercise.

Submission and Grading

After completing various parts of the assignment, be sure to use the submit function system to submit your solutions to our servers. The following is a breakdown of how each part of this exercise is scored.

Part	Submitted File	Points
Feedforward and Cost Function	`nnCostFunction.m`	30 points
Regularized Cost Function	`nnCostFunction.m`	15 points
Sigmoid Gradient	`sigmoidGradient.m`	5 points
Neural Net Gradient Function (Backpropagation)	`nnCostFunction.m`	40 points
Regularized Gradient	`nnCostFunction.m`	10 points
Total Points		100 points

You are allowed to submit your solutions multiple times, and we will take only the highest score into consideration.

sigmoidGradient.m :

```
function g = sigmoidGradient(z)
  %SIGMOIDGRADIENT returns the gradient of the sigmoid function
  %evaluated at z
  %   g = SIGMOIDGRADIENT(z) computes the gradient of the sigmoid function
  %   evaluated at z. This should work regardless if z is a matrix or a
  %   vector. In particular, if z is a vector or matrix, you should return
  %   the gradient for each element.

  g = zeros(size(z));

  % ====================== YOUR CODE HERE ======================
  % Instructions: Compute the gradient of the sigmoid function evaluated at
  %               each value of z (z can be a matrix, vector or scalar).

  g = sigmoid(z).*(1-sigmoid(z));

  % =============================================================
end
```

randInitializeWeights.m :

```
function W = randInitializeWeights(L_in, L_out)
  %RANDINITIALIZEWEIGHTS Randomly initialize the weights of a layer with L_in
  %incoming connections and L_out outgoing connections
  %   W = RANDINITIALIZEWEIGHTS(L_in, L_out) randomly initializes the weights
  %   of a layer with L_in incoming connections and L_out outgoing
  %   connections.
  %
  %   Note that W should be set to a matrix of size(L_out, 1 + L_in) as
  %   the first column of W handles the "bias" terms
  %

  % You need to return the following variables correctly
  W = zeros(L_out, 1 + L_in);

  % ====================== YOUR CODE HERE ======================
  % Instructions: Initialize W randomly so that we break the symmetry while
  %               training the neural network.
  %
  % Note: The first column of W corresponds to the parameters for the bias unit
  %
  % epsilon_init = 0.12;

  epsilon_init = sqrt(6)/(sqrt(L_in)+sqrt(L_out));
  W = - epsilon_init + rand(L_out, 1 + L_in) * 2 * epsilon_init ;

  % =========================================================================
end
```

nnCostFunction.m :

```
function [J, grad] = nnCostFunction(nn_params, ...
      input_layer_size, ...
      hidden_layer_size, ...
      num_labels, ...
      X, y, lambda)
  %NNCOSTFUNCTION Implements the neural network cost function for a two layer
  %neural network which performs classification
  %   [J grad] = NNCOSTFUNCTON(nn_params, hidden_layer_size, num_labels, ...
  %   X, y, lambda) computes the cost and gradient of the neural network. The
  %   parameters for the neural network are "unrolled" into the vector
  %   nn_params and need to be converted back into the weight matrices.
  %
  %   The returned parameter grad should be a "unrolled" vector of the
  %   partial derivatives of the neural network.
  %

  % Reshape nn_params back into the parameters Theta1 and Theta2, the weight
matrices
  % for our 2 layer neural network
  % DIMENSIONS:
  % Theta1 = 25 x 401
  % Theta2 = 10 x 26

  Theta1 = reshape(nn_params(1:hidden_layer_size * (input_layer_size + 1)), ...
      hidden_layer_size, (input_layer_size + 1));

  Theta2 = reshape(nn_params((1 + (hidden_layer_size * (input_layer_size +
1))):end), ...
      num_labels, (hidden_layer_size + 1));

  % Setup some useful variables
  m = size(X, 1);

  % You need to return the following variables correctly
  J = 0;
  Theta1_grad = zeros(size(Theta1)); %25 x401
  Theta2_grad = zeros(size(Theta2)); %10 x 26

  % ====================== YOUR CODE HERE ======================
  % Instructions: You should complete the code by working through the
  %               following parts.
  %
  % Part 1: Feedforward the neural network and return the cost in the
  %         variable J. After implementing Part 1, you can verify that your
  %         cost function computation is correct by verifying the cost
  %         computed in ex4.m
  %
  % Part 2: Implement the backpropagation algorithm to compute the gradients
  %         Theta1_grad and Theta2_grad. You should return the partial derivatives
of
  %         the cost function with respect to Theta1 and Theta2 in Theta1_grad and
  %         Theta2_grad, respectively. After implementing Part 2, you can check
```

```
  %         that your implementation is correct by running checkNNGradients
  %
  %         Note: The vector y passed into the function is a vector of labels
  %               containing values from 1..K. You need to map this vector into a
  %               binary vector of 1's and 0's to be used with the neural network
  %               cost function.
  %
  %         Hint: We recommend implementing backpropagation using a for-loop
  %               over the training examples if you are implementing it for the
  %               first time.
  %
  % Part 3: Implement regularization with the cost function and gradients.
  %
  %         Hint: You can implement this around the code for
  %               backpropagation. That is, you can compute the gradients for
  %               the regularization separately and then add them to Theta1_grad
  %               and Theta2_grad from Part 2.
  %

  %%%%%%%%%%% Part 1: Calculating J w/o Regularization %%%%%%%%%%%%%%%

  X = [ones(m,1), X];  % Adding 1 as first column in X

  a1 = X; % 5000 x 401

  z2 = a1 * Theta1';  % m x hidden_layer_size == 5000 x 25
  a2 = sigmoid(z2); % m x hidden_layer_size == 5000 x 25
  a2 = [ones(size(a2,1),1), a2]; % Adding 1 as first column in z = (Adding bias
unit) % m x (hidden_layer_size + 1) == 5000 x 26

  z3 = a2 * Theta2';  % m x num_labels == 5000 x 10
  a3 = sigmoid(z3); % m x num_labels == 5000 x 10

  h_x = a3; % m x num_labels == 5000 x 10

  %Converting y into vector of 0's and 1's for multi-class classification

  %%%%% WORKING %%%%%
  % y_Vec = zeros(m,num_labels);
  % for i = 1:m
  %     y_Vec(i,y(i)) = 1;
  % end
  %%%%%%%%%%%%%%%%%%%

  y_Vec = (1:num_labels)==y; % m x num_labels == 5000 x 10

  %Costfunction Without regularization
  J = (1/m) * sum(sum((-y_Vec.*log(h_x))-((1-y_Vec).*log(1-h_x))));  %scalar


  %%%%%%%%%%% Part 2: Implementing Backpropogation for Theta_gra w/o
Regularization %%%%%%%%%%%%%

  %%%%%%% WORKING: Backpropogation using for loop %%%%%%%
  % for t=1:m
```

```
  %     % Here X is including 1 column at begining
  %
  %     % for layer-1
  %     a1 = X(t,:)'; % (n+1) x 1 == 401 x 1
  %
  %     % for layer-2
  %     z2 = Theta1 * a1;  % hidden_layer_size x 1 == 25 x 1
  %     a2 = [1; sigmoid(z2)]; % (hidden_layer_size+1) x 1 == 26 x 1
  %
  %     % for layer-3
  %     z3 = Theta2 * a2; % num_labels x 1 == 10 x 1
  %     a3 = sigmoid(z3); % num_labels x 1 == 10 x 1
  %
  %     yVector = (1:num_labels)'==y(t); % num_labels x 1 == 10 x 1
  %
  %     %calculating delta values
  %     delta3 = a3 - yVector; % num_labels x 1 == 10 x 1
  %
  %     delta2 = (Theta2' * delta3) .* [1; sigmoidGradient(z2)]; %
(hidden_layer_size+1) x 1 == 26 x 1
  %
  %     delta2 = delta2(2:end); % hidden_layer_size x 1 == 25 x 1 %Removing delta2
for bias node
  %
  %     % delta_1 is not calculated because we do not associate error with the
input
  %
  %     % CAPITAL delta update
  %     Theta1_grad = Theta1_grad + (delta2 * a1'); % 25 x 401
  %     Theta2_grad = Theta2_grad + (delta3 * a2'); % 10 x 26
  %
  % end
  %
  % Theta1_grad = (1/m) * Theta1_grad; % 25 x 401
  % Theta2_grad = (1/m) * Theta2_grad; % 10 x 26
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  %%%%%% WORKING: Backpropogation (Vectorized Implementation) %%%%%%%
  % Here X is including 1 column at begining
  A1 = X; % 5000 x 401

  Z2 = A1 * Theta1';  % m x hidden_layer_size == 5000 x 25
  A2 = sigmoid(Z2); % m x hidden_layer_size == 5000 x 25
  A2 = [ones(size(A2,1),1), A2]; % Adding 1 as first column in z = (Adding bias
unit) % m x (hidden_layer_size + 1) == 5000 x 26

  Z3 = A2 * Theta2';  % m x num_labels == 5000 x 10
  A3 = sigmoid(Z3); % m x num_labels == 5000 x 10

  % h_x = a3; % m x num_labels == 5000 x 10

  y_Vec = (1:num_labels)==y; % m x num_labels == 5000 x 10

  DELTA3 = A3 - y_Vec; % 5000 x 10
```

```
  DELTA2 = (DELTA3 * Theta2) .* [ones(size(Z2,1),1) sigmoidGradient(Z2)]; % 5000 x
26
  DELTA2 = DELTA2(:,2:end); % 5000 x 25 %Removing delta2 for bias node

  Theta1_grad = (1/m) * (DELTA2' * A1); % 25 x 401
  Theta2_grad = (1/m) * (DELTA3' * A2); % 10 x 26

  %%%%%%%%%%%% WORKING: DIRECT CALCULATION OF THETA GRADIENT WITH REGULARISATION
%%%%%%%%%%%
  % %Regularization term is later added in Part 3
  % Theta1_grad = (1/m) * Theta1_grad + (lambda/m) * [zeros(size(Theta1, 1), 1)
Theta1(:,2:end)]; % 25 x 401
  % Theta2_grad = (1/m) * Theta2_grad + (lambda/m) * [zeros(size(Theta2, 1), 1)
Theta2(:,2:end)]; % 10 x 26

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%


  %%%%%%%%%%%% Part 3: Adding Regularisation term in J and Theta_grad
%%%%%%%%%%%%%
  reg_term = (lambda/(2*m)) * (sum(sum(Theta1(:,2:end).^2)) +
sum(sum(Theta2(:,2:end).^2))); %scalar

  %Costfunction With regularization
  J = J + reg_term; %scalar

  %Calculating gradients for the regularization
  Theta1_grad_reg_term = (lambda/m) * [zeros(size(Theta1, 1), 1) Theta1(:,2:end)];
% 25 x 401
  Theta2_grad_reg_term = (lambda/m) * [zeros(size(Theta2, 1), 1) Theta2(:,2:end)];
% 10 x 26

  %Adding regularization term to earlier calculated Theta_grad
  Theta1_grad = Theta1_grad + Theta1_grad_reg_term;
  Theta2_grad = Theta2_grad + Theta2_grad_reg_term;

  % -------------------------------------------------------------

  % =========================================================================

  % Unroll gradients
  grad = [Theta1_grad(:) ; Theta2_grad(:)];

end
```

Week 6

6.1 Advice for Applying Machine Learning

Evaluating a Learning Algorithm

Evaluating a Hypothesis

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

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

We can move on to evaluate our new hypothesis.

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

The new procedure using these two sets is then:

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):

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

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. It could over fit and as a result your predictions on the test set would be poor. The error of your hypothesis as measured on the data set with which you trained the parameters will be lower than the error on any other data set.

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

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

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

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

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

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

Bias vs. Variance

Diagnosing Bias vs. Variance

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

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

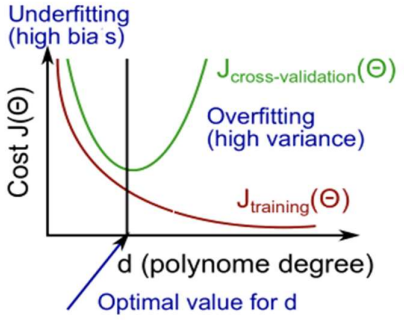
The training error will tend to **decrease** as we increase the degree d of the polynomial.

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

High bias (underfitting): both $J_{train}(\Theta)$ and $J_{CV}(\Theta)$ will be high. Also, $J_{CV}(\Theta) \approx J_{train}(\Theta)$.

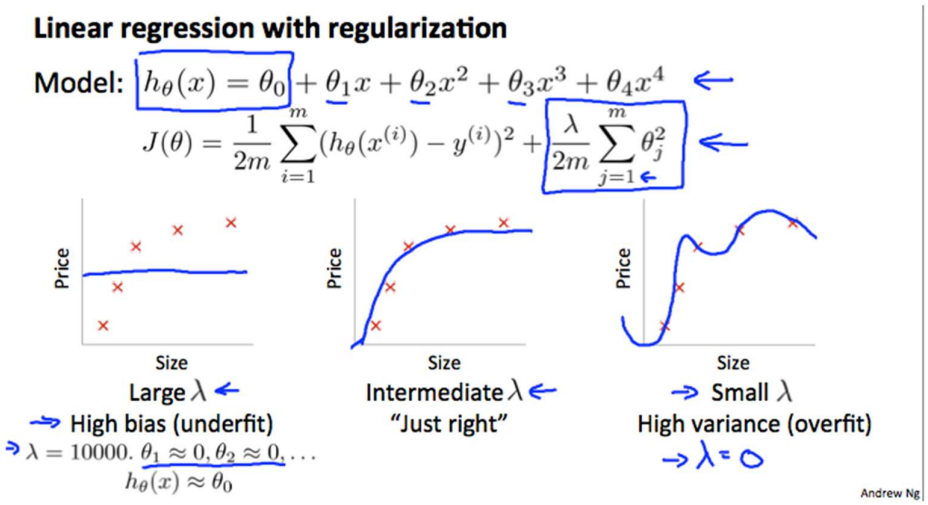
High variance (overfitting): $J_{train}(\Theta)$ will be low and $J_{CV}(\Theta)$ will be much greater than $J_{train}(\Theta)$.

The is summarized in the figure below:



Regularisation and Bias/Variance

Note: [The regularization term below and through out the video should be $\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$ and **NOT** $\frac{\lambda}{2m}\sum_{j=1}^{m}\theta_j^2$]



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

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

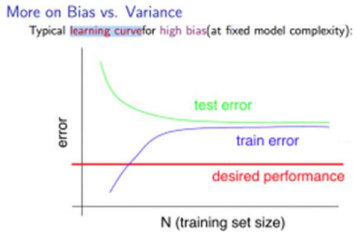
- As the training set gets larger, the error for a quadratic function increases.
- The error value will plateau out after a certain m, or training set size.

Experiencing high bias:

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

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

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

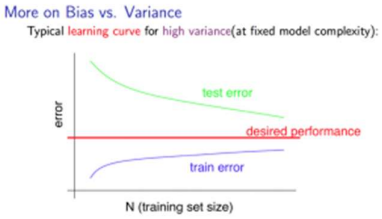


Experiencing high variance:

Low training set size: $J_{train}(\Theta)$ 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. You can then select the one that performs best.

Model Complexity Effects:

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

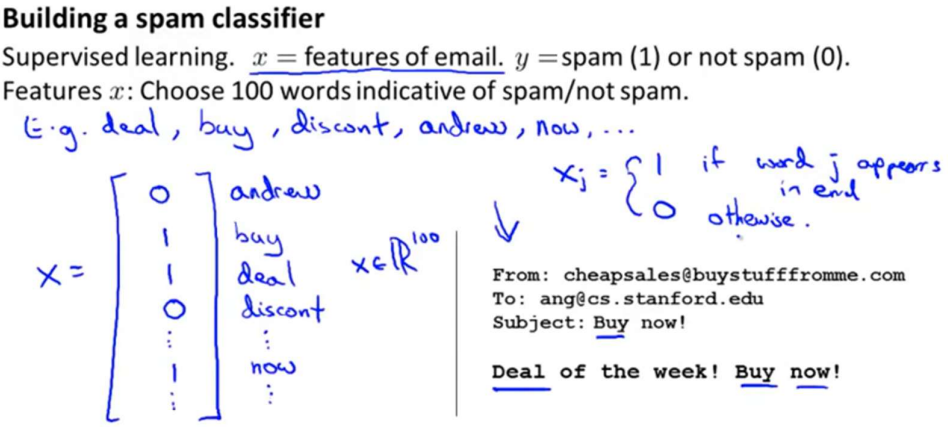
6.2 Machine Learning System Design

Building a Spam Classifier

Prioritizing What to Work On

System Design Example:

Given a data set of emails, we could construct a vector for each email. Each entry in this vector represents a word. The vector normally contains 10,000 to 50,000 entries gathered by finding the most frequently used words in our data set. If a word is to be found in the email, we would assign its respective entry a 1, else if it is not found, that entry would be a 0. Once we have all our x vectors ready, we train our algorithm and finally, we could use it to classify if an email is a spam or not.



So how could you spend your time to improve the accuracy of this classifier?

- Collect lots of data (for example "honeypot" project but doesn't always work)
- Develop sophisticated features (for example: using email header data in spam emails)
- Develop algorithms to process your input in different ways (recognizing misspellings in spam).

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

Error Analysis

The recommended approach to solving machine learning problems is to:

- Start with a simple algorithm, implement it quickly, and test it early on your cross validation data.
- Plot learning curves to decide if more data, more features, etc. are likely to help.
- Manually examine the errors on examples in the cross validation set and try to spot a trend where most of the errors were made.

For example, assume that we have 500 emails and our algorithm misclassifies a 100 of them. We could manually analyze the 100 emails and categorize them based on what type of emails they are. We could then try to come up with new cues and features that would help us classify these 100 emails correctly. Hence, if most of our misclassified emails are those which try to steal passwords, then we could find some features that are particular to those emails and add them to our model. We could also see how classifying each word according to its root changes our error rate:

The importance of numerical evaluation

Should discount/discounts/discounted/discounting be treated as the same word?
Can use "stemming" software (E.g. "Porter stemmer")
universe/university.
Error analysis may not be helpful for deciding if this is likely to improve performance. Only solution is to try it and see if it works.
Need numerical evaluation (e.g., cross validation error) of algorithm's performance with and without stemming.
Without stemming: 5% error With stemming: 3% error
Distinguish upper vs. lower case (Mom/mom): 3.2%

It is very important to get error results as a single, numerical value. Otherwise it is difficult to assess your algorithm's performance. For example if we use stemming, which is the process of treating the same word with different forms (fail/failing/failed) as one word (fail), and get a 3% error rate instead of 5%, then we should definitely add it to our model. However, if we try to distinguish between upper case and lower case letters and end up getting a 3.2% error rate instead of 3%, then we should avoid using this new feature. Hence, we should try new things, get a numerical value for our error rate, and based on our result decide whether we want to keep the new feature or not.