WEEK1

INTRODUCTON:

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.

Basic types of learning:

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.

Categories of supervised learning problems:

Regression: we are trying to predict results within a continuous output, meaning that we are trying to map input variables to some continuous function.

Classification: we are instead trying to predict results in a discrete output. In other words, we are trying to map input variables into discrete categories.

Examples:

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

Classification: Given a patient with a tumor, we have to predict whether the tumor is malignant or benign.

Unsupervised learning:

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

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

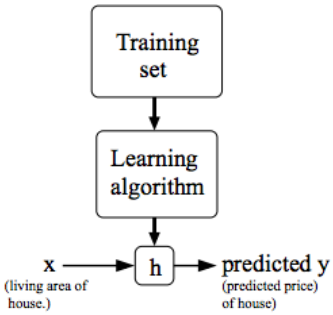
Note: With unsupervised learning there is no feedback based on the prediction results.

Examples:

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).

Proposition: 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 “” 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:



LINEAR REGRESSION WITH ONE VARIABLE

Cost Function (Mean squared error):

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.

/* So we have to minimize it to get better results, note that it is a parabola equation */

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

Gradient Descent:

/* Optimization algorithm, we iterate over graph until get the optimal value*/

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 θ1, θ2, θ3, …, θn . Updating a specific parameter prior to calculating another one on the jth iteration would yield to a wrong implementation.

Note:
If a(alpha) is too small, gradient descent can be slow
If a (alpha) is too large, gradient descent can overshoot the minimum

LINEAR ALGEBRA REVIEW:

Matrix: A matrix is a rectangular array of numbers written between square brackets (Technically, two dimensional array).
Dimension of a matrix: n*m where n is the number of rows and m is the number of columns.
Vector: A vector is a matrix that has only 1 column (special case of matrix n*1 matrix), and n is the **dimension of the vector**.

Addition and Scalar Multiplication:
Addition and subtraction are element-wise, so you simply add or subtract each corresponding element.
In scalar multiplication, we simply multiply every element by the scalar value.
In scalar division, we simply divide every element by the scalar value.

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

Matrix-Matrix Multiplication
We multiply two matrices by breaking it into several vector multiplications and concatenating the result.
An m x n matrix multiplied by an n x o matrix results in an m x o matrix.

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 (just like multiplying numbers by 1)

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'.

WEEK2

MATLAB AND OCTAVE INSTALLATION

LINEAR REGRESSION WITH MULTIPLE VARIABLES

also known as "**multivariate linear regression**"

New hypothesis:

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

where;

n is the number of features

$x_1, x_2, x_3, \ldots, x_n$ are the features

$\theta_1, \theta_2, \theta_3, \ldots, \theta_n$ are the parameters

$x_0=1$ (acceptance)

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

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

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: (update the parameters **simultaneously**)

repeat until convergence: {

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

}

Gradient Descent in Practice I –

```
/* Tricks for faster convergence of gradient descent */
```

Feature Scaling:

We can modify the ranges of our input variables so that they are all roughly the same. Ideally:

$-1 \leq x_i \leq 1$

Or:

$-0.5 \leq x_i \leq 0.5$ `/* it is not required, it is a kind of optimization */`

Mean Normalization:
By the formula:

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

Where:
μ_i = the average of all the values for feature (i)
/* S = max – min of the range */

Gradient Descent in Practice II –

Learning Rate:

Debugging gradient descent. Make a plot with number of iterations on the x-axis. Now plot the cost function, J(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
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:

/* linear model can not give accurate values some curves in the graph can improve the results */

For example, if our hypothesis function is $h\theta(x)=\theta0+\theta1x1$ then we can create additional features based on x1, to get the quadratic function $h\theta(x)=\theta0+\theta1x1+\theta2$ x1^2 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 x2 and x3 where x2=x1^2 and $x_3 = x_1^3$.

Important note: If you choose your features this way then feature scaling becomes very important.

Normal Equation

```
/* another algorithm to find the optimum (min error) θs,
more friendly :P */
/* I can say that finding θs by gradient descent is a
numerical analysis method and normal equation method is an
algebraic method */
```

The normal equation formula is: $\theta = (X^T X)^{-1} X^T y$

A comparison between two methods:

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

Issue: What if X^t * X is non invertible?
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" (will be explained later)

MATLAB/OCTAVE TUTORIAL

WEEK3

PART1: LOGISTIC REGRESSION

CLASSIFICATION AND REPRESENTATION:

/* to classify inputs into finite number of classes, if we have only two then it is a binary classification problem, for classification problems it in not a good idea to use linear regression because we can not fit a line for discrete output */

LOGISTIC REGRESSION:

/* Since Linear regression is not suitable for discrete output we will use this model */

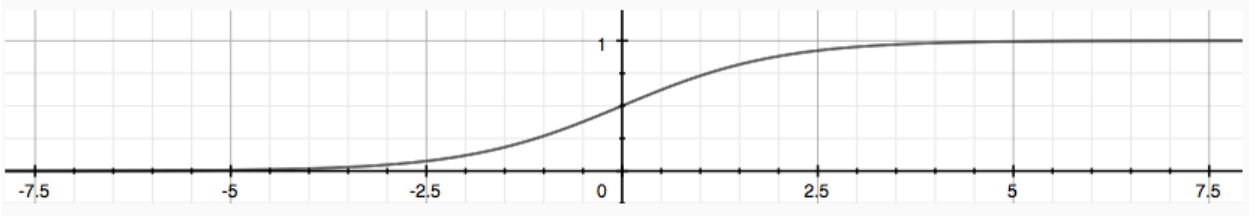
Logistic Function: (Also known as Sigmoid Function)

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

$$z = \theta^T x$$

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

And its graph



/* Outputs are varying on the interval [0,1], so, we can think it as a probability function where h(x) = P(y=1|x:theta), this is valid only for binary classification problems */

Decision Boundary:

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

/* the line that we notice after plotting the data which separates the classes it might be too complicated due to the dataset, as a simple example it can be a simple line, but for complicated statements by creating non linear (polynomial) relations on features we can define more complicated boundaries (non linear boundary) */

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. /* A function is said to be **CONVEX** if it has only one optima on its graph */

Cost function for logistic regression looks like:

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

Logistic regression cost function properties:

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

/* mathematically in the given condition, cost function must approach infinity, but e.g. y=0 and h(x)→1 means that when y=0 the hypothesis predicted truly, so cost function must be zero (there is no error), similarly for the other property */

Note that writing the cost function in this way guarantees that J(θ) is convex for logistic regression.

/* remember that J(θ) = 1/m*(Sigma(i=1 to m)(cost(h,y))) */

The comprehensive (simplified) form of linear regression cost function:

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

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:

ADVANCED OPTIMIZATION:

"Conjugate gradient", "BFGS", and "L-BFGS" are more sophisticated, faster ways

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

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

to optimize θ that can be used instead of gradient descent.

```
/* We can apply these algorithms using fminunc Matlab pre-
implemented function which takes (@costFunction,
initialTheta, options) as parameters. Using the pre-
implemented functions gives us more accurate results due to
the complexity of the algorithms */
```

Multiclass Classification:

Instead of y = {0,1} we will expand our definition so that y = {0,1...n}.

One-vs-all concept:

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 hθ (x)

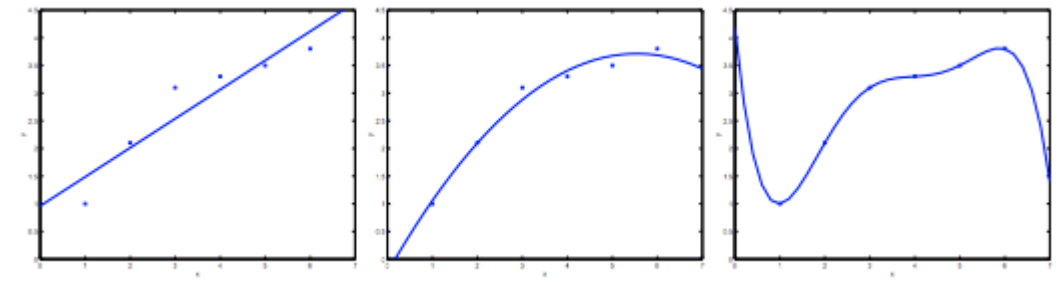
```
/* here we are converting the multiclass classification
problem into multi sub binary classification problem. If we
have k classes, we create k hypotheses and for every x we
calculate the value of each h, then we choose the greatest,
remember that we can think h in logistic regression as a
probability function */
```

REGULARIZATION:

Overfitting:

/* the problem when the algorithm fails to generalize */

This terminology is applied to both linear and logistic regression. There are two



High bias /* good but not ideal */

Good /* Can by applied fort he given dataset and unseen data*/

Overfit /* Ideal for given data set but fails to generalize for unseen data*/

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 Regularization:
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.
/* I think here we are allowing the cost function to be not optimal, small error is allowed here for more general model */

We could also regularize all of our theta parameters in a single summation as /* Our modified cost function */:

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

Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting.

Note: If lambda is chosen to be too large, it may smooth out the function too much and cause underfitting.

Regularized Linear Regression:

We will modify our gradient descent function to:

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

Normal Equation:

Also we can 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 x0), multiplied with a single real number λ.

This matrix is guaranteed to be invertible /* the proof is given in the video */

Regularized Logistic Regression:

Similarly, we can define cost function as:

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

Note:

The second sum means to explicitly exclude the bias term by running from 1 to n, skipping 0. Thus, when computing the equation, we should continuously update the two following equations:

Gradient descent

Repeat $\{$

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

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

WEEK4

NEURAL NETWORKS

Model Representation:

```
/* We will try to build something like human neurons, our
model will consist of input layer(x vector, our feature
vector), hidden layer(s)(where we will do computational
operations) and output layer(last named as h) */
```

In neural networks, we use the same logistic function (or sigmoid activation function) as in classification, and our "theta" parameters are sometimes called "weights".

Note: In this model our x0 input node is sometimes called the "**bias unit**" It is always equal to 1.

Visually, a simplistic representation of a neural network looks like:

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

In this example, we label hidden layer nodes a0^2⋯an^2 and call them "**activation units**". The values for each of the "activation" nodes is obtained as follows:

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

Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been multiplied parameter matrix $\Theta^{(2)}$ containing the weights for our second layer of nodes.

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

```
/* To pass from jth layer that has m units to (j+1) layer
that has n units the weight matrix dimension must be =
n*(m+1)  */
```

Vectorization:

The model can be expressed in the form:

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

W here:

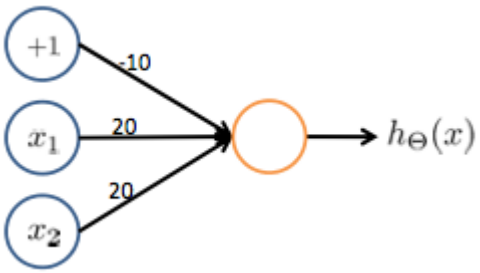
$$z^{(j+1)} = \Theta^{(j)} a^{(j)}$$

So:

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

Applications:

A simple example of applying neural networks is by predicting x_1 OR x_2, so we can construct the fundamental operations in computers by using a small neural network.



Multi-class classificaition:

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.

Sample y:

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

WEEK5

COST FUNCTION AND BACKPROPAGATION:

/* remember that we use neural network for classification problems (binary or multi-class) */

Essential variables:

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

Cost function formula:

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

/* #columns of current theta = #nodes of current layer (including bias unit)
#rows of current theta = #nodes of next layer (excluding bias unit) */

Backpropagation Algorithm:

A neural-network terminology for minimizing cost function.

Steps of the algorithm:

Given training set { (x^1, y^1), ……, (x^m, y^m) }

Set Δ^l_{ij} :=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 all l = 2,3,…,L

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 δ^{L-1} , δ^{L-2}, ……, δ^2 using :

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

5- $\Delta^{l}_{ij} := \Delta^{l}_{ij} + a^{l}_{j}\,\delta^{l+1}_{i}$ or with vectorization

$\Delta^{l} := \Delta^{l} + \delta^{l+1}(a^{l})^{T}$

Hence we update our new Δ matrix.

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

- $D^{(l)}_{i,j} := \frac{1}{m}\left(\Delta^{(l)}_{i,j} + \lambda\Theta^{(l)}_{i,j}\right)$, if j≠0.
- $D^{(l)}_{i,j} := \frac{1}{m}\Delta^{(l)}_{i,j}$ If j=0

$\frac{\partial}{\partial\Theta^{(l)}_{ij}} J(\Theta) = D^{(l)}_{ij}$

Thus we get:

Gradient Checking:

Gradient checking will assure that our backpropagation works as intended. We can approximate the derivative of our cost function with:

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

So once we compute our gradApprox vector, we can check that gradApprox ≈ deltaVector. `/* to check that gradApprox values are true */`

Note: 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 Initialization:

Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our Θ matrices using the following method:

Initialize each Θ^{l}_{ij} to a random value in $[-\varepsilon, \varepsilon]$

As a summary:

First, pick a network architecture.

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

WEEK6

EVALUATING A LEARNING ALGORITHM:

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 ~ 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:

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

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

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

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

1- Optimize the parameters in Θ using the training set for each polynomial degree.
2- Find the polynomial degree d with the least error using the cross validation set.
3- Estimate the generalization error using the test set with $J_{test}(\Theta^{(d)})$, (d = theta from polynomial with lower error);
This way, the degree of the polynomial d has not been trained using the test set.

Diagnosing Bias vs. Variance:

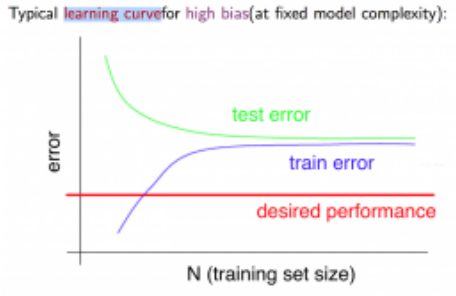
/* we can face two kinds of problems; underfitting(high bias) and overfitting(high variance)
Remember that d is the degree the polynomia
Increasing d ➔ decreasing $error_{train}$
Increasing d ➔ decreasing $error_{cv}$ up to a point then it will increase (because of the overfitting)
So to summarize:
High bias(underfitting) ~ High J_{train} and Jcv (Jcv ≈ J_{test})
High variance (overfitting) ~ Low J_{train} and high Jcv
*/

Regularization and Bias/Variance:

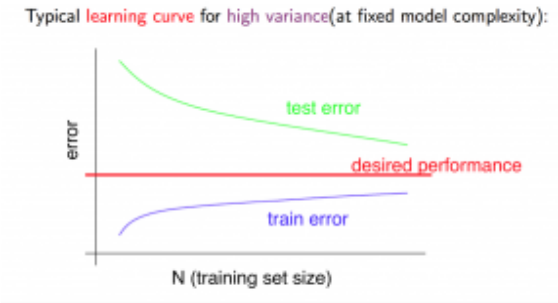
How do we choose our parameter λ to get it 'just right' ? /* with neither underfitting nor overfitting */
1- Create a list of lambdas (i.e. λ∈{0 , 0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, 1.28, 2.56, 5.12, 10.24});
2- Create a set of models with different degrees or any other variants.
3- Iterate through 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:



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 $Jtrain(\Theta) \approx JCV(\Theta)$.
/* More data can not improve the results */



Experiencing **HIGH VARIANCE**:
Low training set size: $J_{train}(\Theta)$ will be low and $J_{CV}(\Theta)$ will be high.
: $J_{train}(\Theta)$ 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.
/* More data can improve the results */

Machine Learning System Design:

Prioritizing What to Work On:

Tricks to improve the accuracy of a classifier (e. g. mail spam 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).

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.

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

Error Metrices:

Precision = true positives / #predicted positive

Recall = true positives / #actual positive

F1 Score = 2PR/(P+R)

```
/*
TP: True positive, FP: False positive, FN: False negative
Precision = TP/(TP+FP)
Recall = TP/(TP+FN)
Threshold increasing:
Lower recall, higher precision.
*/
```

WEEK 7

SUPPORT VECTOR MACHINE:

SVM Hypothesis:

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

h(x) = 1 if $\Theta^T X \geq 0$

0 otherwise

Note: C=1/ λ

SVM Parameters:

Large C: Lower bias, high variance

Small C: Higher bias, low variance

Large σ^2: Features f_i vary more smoothly, higher bias, lower variance

Small σ^2: Features f_i vary less smoothly, lower bias, higher variance

```
/* Use linear kernel when n is large and m is small
Use Gaussian kernel when n is small and m is large */
```

Note: Do perform feature scaling before using the Gaussian kernel.

Logistic regression vs. SVMs

If n is large (relative to m):

Use logistic regression, or SVM without a kernel (linear kernel)

If n is small, m is intermediate:

Use SVM with Gaussian kernel

If n is small, m is large:

Create/add more features, then use logistic regression or SVM without a kernel

Neural network likely to work well for most of these settings, but may be slower to train.