**Supervised Learning-**

* We are given data set and already know correct output
* We have idea of relationship b/w input and output

Regression SL- predict the result within continuous output, map output from continuous function.

Eg. predict age on basis of picture of a person

Eg. predict cost of house on basis of size, location, age etc

Classification SL- predict result in discrete output, map input to discrete categories.

Eg. predict tumour whether it is malign or benign on basis of its size etc

**UnSupervised Learning-**

* No idea what the output looks like.
* Clustering Data based on relationship among variables in data
* There is no feedback based on results

Eg.- cocktail party problem- where we separate audio from different sources.

eg. - Classify spam or email among different mails

eg. - group similar type of news in same news group

Model Representation-

Xi  - input variable

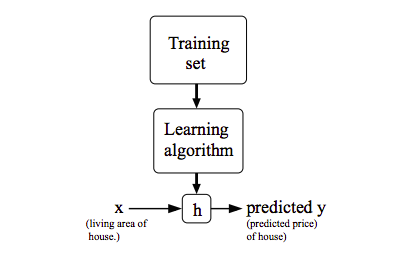
Yi - output variable

(xi ,yi) - training example

M - no of training example in training set

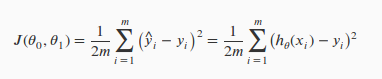
Out goal is given a training set, to learn a function h: x-> y so h(x) is a good predictor for y.

h is called hypothesis

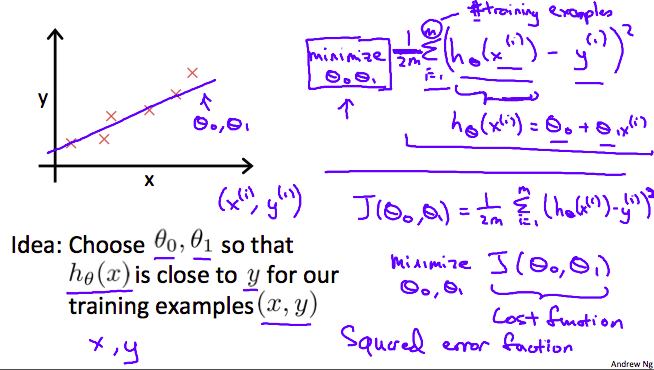


**Cost Function**

We measure accuracy of our hypothesis by cost function.it takes average of difference between hypothesis value to the actual value.



It is also called squared error function or mean squared error



Our goal is to minimize cost function.

**Gradient Descent Algorithm-**

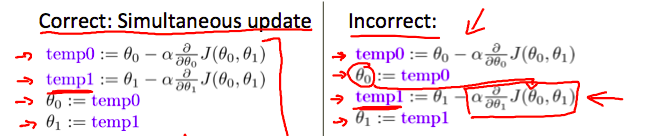
Repeat until convergence {

Screenshot from 2016-12-09 20-39-56.png

(where j=0,1 represents the index number)

}

:= denotes simultaneous update.

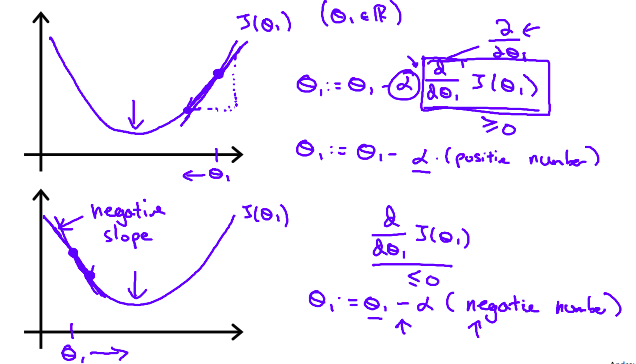


Intuition of gradient descent algo-

When slope is +ve, +ve so will decrease.

When slope is -ve, will increase.

When slope is 0, means minimum point is reached so no change will occur in values.



If is too small, gradient descent is too slow.

If is too large then it may overshoot the minimum. It may fail to converge, or even diverge.

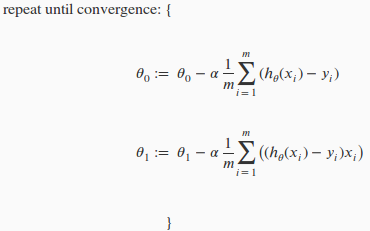
Gradient descent may converge to local minimum, even if learning rate is fixed.

When slope is higher higher, so will increase or decrease at faster rate.

When slope is less will increase or decrease at slower rate.

**Gradient Descent applied to linear regression algorithm.**

We can substitute actual cost function and our action hypothesis function and modify the equation to :



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.**

In linear regression there is only one local minima which is global minima. So gradient descent algo always converges to global minima.

**MultiVariate Linear Regression**

- value of jth feature in ith training example.

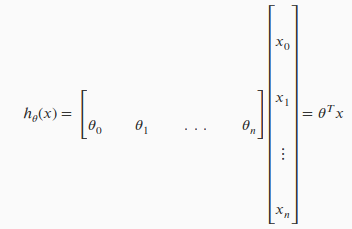
m -number of training examples

n -number of features

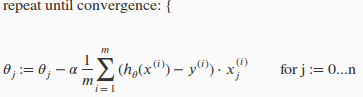
Screenshot from 2016-12-10 16-42-37.png

Hypothesis function will become-

Lets = 1;



**Gradient Descent for multivariate linear regression-**



}\

1. ***Feature Scaling.***

-will descent quickly on small ranges and slowly on large ranges, so will oscillate inefficiently down to optimum.

-We can speed up gradient descent by having our input value roughly in same range

Ideally

In featuring Scale, we generally divide input value with its range.

Mean Normalization-

Subtract average input from the value of input variable resulting in a new average value for the input variable of just zero.

To implement this-

Screenshot from 2016-12-10 17-34-16.pngScreenshot from 2016-12-10 17-34-29.png

For example, if xi represents housing prices with a range of 100 to 2000 and a mean value of 1000, then, Here 1900 is not standard Deviation.

Screenshot from 2016-12-10 17-34-56.png

1. ***Learning Rate.***

If α is too small: slow convergence.

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

**Automatic convergence test-**

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

**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 (x)=+ then we can create additional features based on x1, to get the quadratic function (x)=++ or the cubic function (x)=+++

One important thing to keep in mind is, if you choose your features this way then feature **scaling becomes very important.**

Eg. If has range 1 - 1000, then range of becomes 1 - 1000000 and that of becomes 1 - 1000000000

**Normal Equation** (another approach other than gradient descent)-

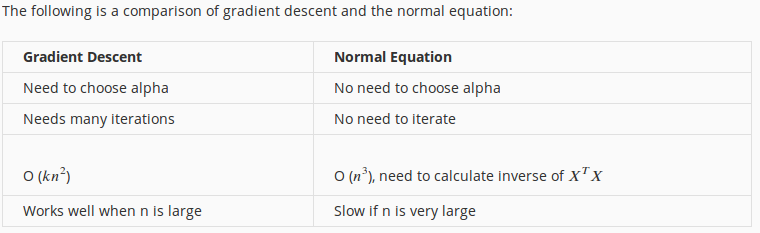
**--** This is another way of minimizing j other than gradient descend.

**--** 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:**

Screenshot from 2016-12-10 18-45-07.png



With the normal equation, **computing the inversion has complexity O().** 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.

**Non invertibility of .**

We should use pinv function in octave in place of inv. pinv will give value of even if is non invertible.

is non invertible if :-

* Redundant Features. If two features are linearly dependent.
* Too many features. E.g. m<n . In this case delete some features called regularization.

**Classification-**

Eg- spam or email, malign or benign tumor etc

We can use linear regression to do classification by mapping all predictions greater than

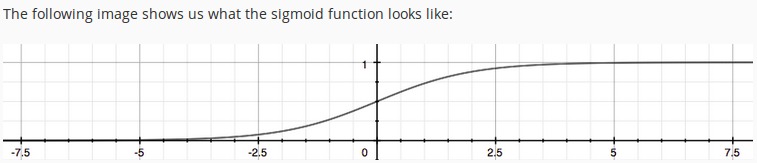
.5 to 1 and less than .5 to 0. But this does not work properly because classification is not actually a linear function. In some cases this may result into prediction >1 or <0.

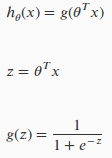
Now we will focus on **binary classification** where prediction is either yes or no / 1 or 0.

**Logistic Regression-**

It doesn’t make sense for (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 (x) to satisfy 0≤(x)≤1. This is accomplished by plugging into the Logistic Function.

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

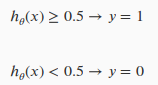


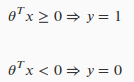


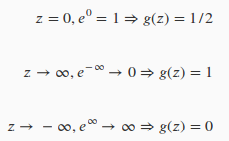
(x) will give us the **probability** that our output is 1. For example, (x)=0.7 gives us a probability of 70% that our output is 1. Our probability that our prediction is 0 is just the complement of our probability that it is 1 (e.g. if probability that it is 1 is 70%, then the probability that it is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

**Decision Boundary-**

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



g(z)>=0.5 when z>0;Remember;  **Finally we can say;**



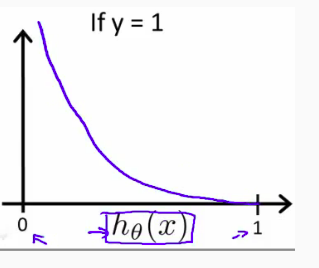
The decision boundary is the line that separates the area where y = 0 and where y = 1. It is created by our hypothesis function.

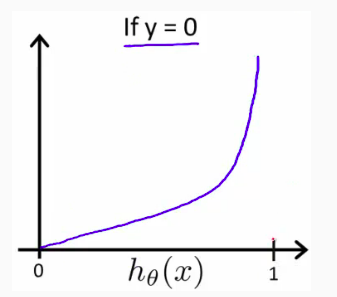
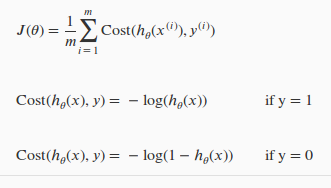
Again, the input to the sigmoid function g(z) (e.g. X) doesn't need to be linear, and could be a function that describes a circle (e.g. z=++) or any shape to fit our data.

**Logistic Regression 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.

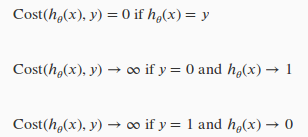
Instead, our cost function for logistic regression looks like:





**Graph of J() vs**

**Graph Intuition**



**Summary-**

If y=1->

h() -> 1, J()->0

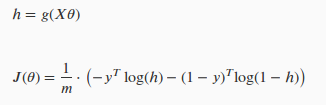
h() -> 0, J()->

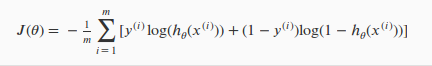
And If y=0->

h()->1, J()->

h()->0, J()->0

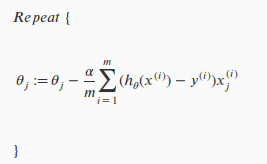
Generalised Cost-

**Vectorized Implementation**Screenshot from 2016-12-13 21-16-21.png

**Final Cost Function-**

**Gradient Descent-**

Gradient looks very similar as gradient descent for linear regression.



**A vectorized Implementation**

Screenshot from 2016-12-13 21-24-35.png

**However h()=g(X)= 1/(1+)**

**Advanced Optimization-**

3 new advanced algos-

->Conjugate gradient

->BFGS

->L-BFGS

These algos are far more sophisticated and faster way to optimize . These can be used in place of gradient descent.

We are using built-in libraries to implement them instead of writing ourselves.

We need to provide function that evaluates 2 function for a given input value

1. J() 2. d/dJ()

We can write single function to return both of these.

function [jVal, gradient] = costFunction(theta)

jVal = [...code to compute J(theta)...];

gradient = [...code to compute derivative of J(theta)...];

End

we can use octave's "fminunc()" optimization algorithm along with the "optimset()" function that creates an object containing the options we want to send to "fminunc()".

options = optimset('GradObj', 'on', 'MaxIter', 100);

initialTheta = zeros(2,1);

[optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialTheta, options);

Exitflag - tells is optimized(1) or not(0)

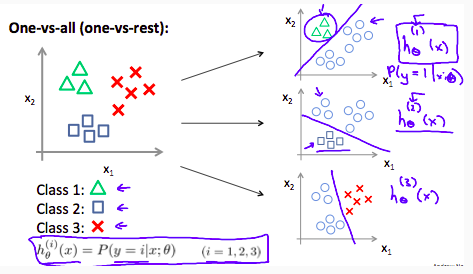
**Multiclass-Classification(One vs all method)-**

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}.

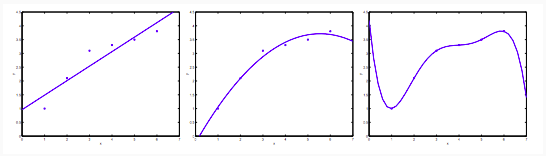
Since y = {0,1...n}, we divide our problem into n+1 (+1 because the index starts at 0) binary classification problems; in each one, we predict the probability that 'y' is a member of one of our classes.

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:*



**Problem of Overfitting-**



**Under Fitting (high Bias) Better Fitting Over Fitting (high variance)**

-> The leftmost figure below shows the result of fitting a y = +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 x2 , and fit y=θ0+θ1x+θ2x2 , then we obtain a slightly better fit to the data.

-> The rightmost figure is the result of fitting a 5th order polynomial. 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).

**Problem of overfitting is applied to both regression and classification.**

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

**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.

Lets make following function more quadratic-

Screenshot from 2016-12-14 15-10-10.png

We'll want to eliminate the influence of and . Without actually getting rid of these features or changing the form of our hypothesis, we can instead modify our cost function:Screenshot from 2016-12-14 15-13-59.png

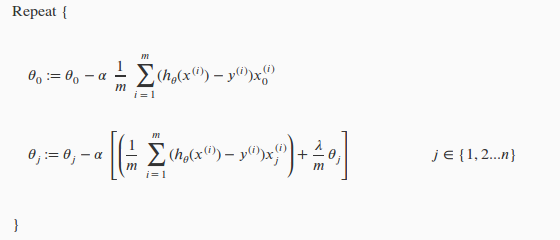
Now, in order for the cost function to get close to zero, we will have to reduce the values of θ3 and θ4 to near zero. This will in turn greatly reduce the values of θ3x3 and θ4x4 in our hypothesis function.

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

Screenshot from 2016-12-14 15-15-34.png

The λ, or lambda, is the **regularization parameter**. It determines how much the costs of our theta parameters are inflated. If is set high say 10^10, then this will cause our underfitting.

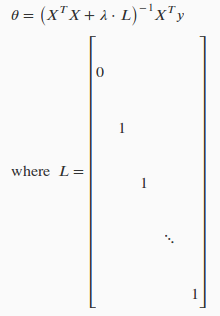
**Regularization applied to Linear Regression-**

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

With some manipulation our update rule can also be represented as:

Screenshot from 2016-12-14 16-11-12.png

( 1 − αλ/m ) will always be less than 1.



**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

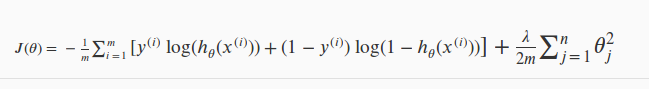
term inside the parentheses:

L should have dimension (n+1)×(n+1). Intuitively, this

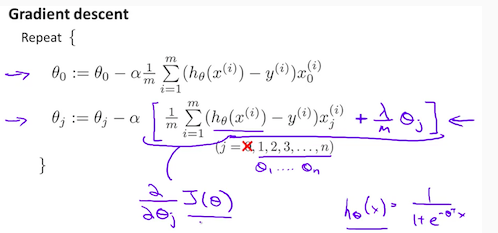
is the identity matrix (though we are not including x0), multiplied with a single real number λ.

Recall that if m ≤ n, then XTX is non-invertible. However, when we add the term λ⋅L, then XTX + λ⋅L becomes invertible.

**Regularization applied to Logistic Regression**-

We can regularize our equation by adding term to the end-

Thus, when computing the equation, we should continuously update the two following equations:



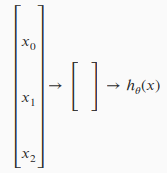
**Neural Networks (NonLinear Hypothesis)-**

If we have so many features or classes then making a non linear hypothesis will have so much complexity. Like if we have 100 features than no of variables will be 10000.

It has a complexity of O().

The solution is neural networks.

In neural networks neurons are basically computational units that take inputs (**dendrites**) as electrical inputs (called "spikes") that are channeled to outputs (**axons**).

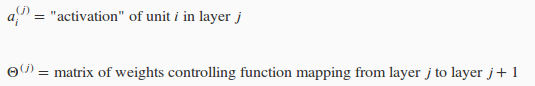
In our model, our dendrites are like the input features x1⋯xn, and the output is the result of our hypothesis function. In this model our input node is sometimes called the "bias unit." It is always equal to 1. In neural networks, we use the same logistic function as in classification, 

1/( 1+e^(−θTx) ), yet we sometimes call it a sigmoid (logistic) activation function. In this situation, our "theta" parameters are sometimes called "weights".

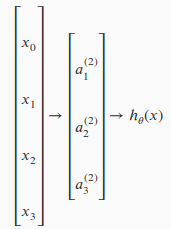
Our input nodes (layer 1), also known as the "**input layer**", go into another node (layer 2), which finally outputs the hypothesis function, known as the "**output layer**".

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

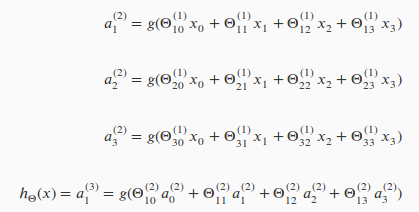
In this example, we label these intermediate or "hidden" layer nodes ⋯ and call them "activation units."

**If we have one hidden**

**layer, it looks like**



**The value of each “activation node” looks like-**

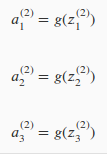


This is saying that we compute our activation nodes by using a 3×4 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been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

Each layer gets its own matrix of weights, .

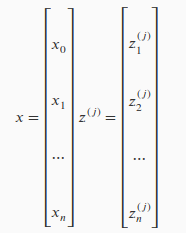
The dimensions of these matrices of weights is determined as follows:

If network has units in layer j and units in layer j+1, then will be of dimension ×(+1). +1 comes from bias node in layer .

In this section we'll do a vectorized implementation of the above functions. We're going to define a new variable 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: ->

In other words, for layer j=2 and node k, the variable z will be:Screenshot from 2016-12-15 00-57-44.png

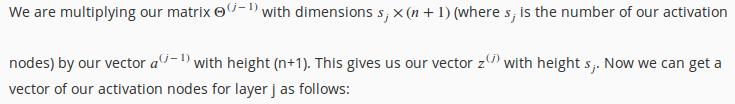
The vector representation of x and is:



Setting x as j we can obtain-

Screenshot from 2016-12-15 01-01-11.png

Screenshot from 2016-12-15 01-12-01.png



We then get our final result with:

Screenshot from 2016-12-15 01-14-20.png

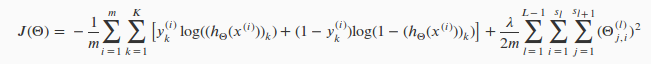
Notice that in this **last step**, between layer j and layer j+1, we are doing **exactly the same thing** as we did in logistic regression. Adding all these intermediate layers in neural networks allows us to more elegantly produce interesting and more complex non-linear hypotheses.

**Cost Function-**

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

1. L = total number of layers in the network
2. = number of units (not counting bias unit) in layer l
3. K = number of output units/classes

We denote as being a hypothesis that results in the kth output. Our cost function for neural networks is going to be a generalization of the one we used for logistic regression

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

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).

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.

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

= "error" of node j in layer l

Recall that is activation node j in layer l. For the **last layer**, we can compute the vector of delta values with:

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:

Screenshot from 2016-12-29 20-47-45.png

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

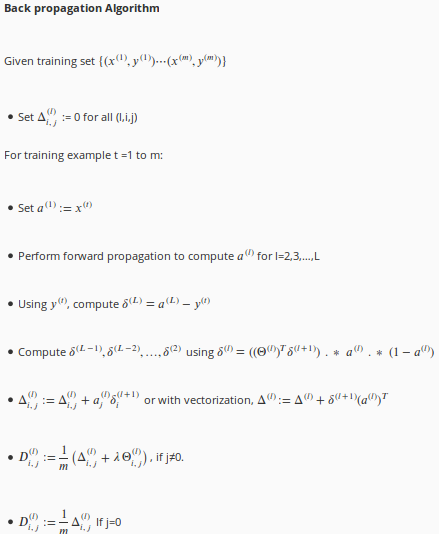
Screenshot from 2016-12-29 20-52-15.png

Screenshot from 2016-12-29 20-53-24.png

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

Screenshot from 2016-12-29 20-57-33.png

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



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

Screenshot from 2016-12-29 21-06-14.png

**Implementation-**

In order to use fminunc function to optimize our value of , we require a vector but here our is a matrix. So we must unroll(convert to long vector) to use optimizing function.

thetaVector = [ Theta1(:); Theta2(:); Theta3(:); ]

deltaVector = [ D1(:); D2(:); D3(:) ]

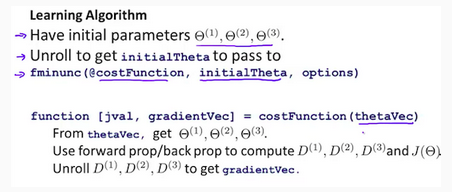
If the dimensions of Theta1 is 10x11, Theta2 is 10x11 and Theta3 is 1x11, then we can get back our original matrices from the "unrolled" versions as follows:

Theta1 = reshape(thetaVector(1:110),10,11)

Theta2 = reshape(thetaVector(111:220),10,11)

Theta3 = reshape(thetaVector(221:231),1,11)

To Summarize-



**Gradient Checking-**

It is used to check whether backpropagation algorithm is working as intended or not. We can approximate derivative of our cost function with:

Screenshot from 2017-01-08 01-06-43.png

With multiple theta matrices, we can approximate the derivative with respect to Θj as follows:Screenshot from 2017-01-08 01-08-10.png

A small value for ϵ (epsilon) such as ϵ=, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems.

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 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:

Hence, we initialize each to a random value between[−ϵ,ϵ]. 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.

Eg.

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.

**Octave Programming-**

Octave is freely available. Before implementing algorithm in higher programming language, it is firstly tested in octave. And then implemented in java, c++ etc;

% - is used for comment

; - after a statement suppress print statement

>> 5+3

ans = 8

>> 5-7

ans = -2

>> 5\*2

ans = 10

>> 5/3

ans = 1.6667

\_\_\_\_\_\_\_\_\_\_\_\_comparators

>> 5==2

ans = 0

>> 5 ~= 2 % ~= is used as a not equal operator instead !=

ans = 1

\_\_\_\_\_\_\_\_\_\_\_\_boolean operations

>> 1 && 0

ans = 0

>> 1 || 0

ans = 1

\_\_\_\_\_\_\_\_\_\_\_\_assignment operator

>> a = 5

a = 5

>> b = 'hi'; % ; suppresses print statement

>> b='hi'

b = hi

>> c = (3>=1)

c = 1

>> clear c %remove variable c

>> a =pi;

>> result= sprintf("upto 2 decimals : %.2f",a) % sprintf generate a string

result = upto 2 decimals : 3.14

\_\_\_\_\_\_\_\_\_\_\_\_\_to print

>> a

a = 3.1416

>> disp(a);

3.1416

>> disp(sprintf("upto 2 decimals : %.2f",a));

upto 2 decimals : 3.14

\_\_\_\_\_\_\_\_\_\_\_\_\_change format

>> format long

>> a

a = 3.14159265358979

>> format short

>> a

a = 3.1416

\_\_\_\_\_\_\_\_\_\_\_\_\_\_Matrices

>> A = [2 3; 4 5; 6 5]

A =

2 3

4 5

6 5

>> v = [2 3 4]

v =

2 3 4

>> v= [1;2;3]

v =

1

2

3

>> v = 1:0.2:2 % start from 1 increment by 0.2 until 2

v =

1.0000 1.2000 1.4000 1.6000 1.8000 2.0000

>> v = 1:6

v =

1 2 3 4 5 6

>> v= 2\*ones(2,3)

v =

2 2 2

2 2 2

>> v = zeros(2,3)

v =

0 0 0

0 0 0

>> v = rand(2,3)

v =

0.129371 0.841934 0.076720

0.833640 0.701954 0.927315

>> eye(4)

ans =

Diagonal Matrix

1 0 0 0

0 1 0 0

0 0 1 0

0 0 0 1

>> magic(3) %generate a magic matrix whose row and column sums to same value

ans =

8 1 6

3 5 7

4 9 2

>> c=v(1:10)

c =

4 6 7 1 8 7 2 9 7 3

>> A(3,2)

ans = 5

>> A(2,:) %fetch every element is 2nd row

ans =

4 5

>> A(:,2) %fetch every element in 2nd column

ans =

3

5

5

>> A([1,3],:) %fetch every element in 1 and 3 row

ans =

2 3

6 5

>> A(2,:) =[5 6] %as a assignment

A =

2 3

5 6

6 5

>> A = [A [6;9;12]] %append another columns

A =

2 3 6

5 6 9

6 5 12

>> A = [A ; [6 9 12]] %append another row

>> size(A) %returns a vector with 2 columns giving size

ans =

3 2

>> size(A,1) % return no of rows

ans = 3

>> size(A,2) % returns no of columns

ans = 2

>> length(v) %size of longest dimension

ans = 16

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Working with Data

>> load('featuresx.dat') % load data from featuresx.dat

>> load('pricey.dat'); % load data from pricey.dat

>> who

Variables in the current scope:

A a ans b c result v

>> whos

Variables in the current scope:

Attr Name Size Bytes Class

==== ==== ==== ===== =====

A 3x2 48 double

a 1x1 8 double

ans 1x12 12 char

b 1x2 2 char

c 1x1 1 logical

result 1x22 22 char

v 1x16 128 double

Total is 60 elements using 221 bytes

>> save v.mat c %save c in v.mat file in binary format

>> save v.txt c -ascii %save c in ascii format human readable

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Matrix Operations

. is used for element wise operators.

>> A=[2 3;5 6;7 8]

A =

2 3

5 6

7 8

>> B = [5;6]

B =

5

6

>> A\*B

ans =

28

61

83

>> A+B % not possible in this case

>>A-B %not possible in this case

>>A + 2 %add 2 to every element

>> A' % transpose

ans =

2 5 7

3 6 8

>> A .\* C % element wise multiplication

ans =

2 6

20 30

42 56

>> A.^C %element wise exponent

ans =

2 9

625 7776

117649 2097152

>> A.^2 %element wise square

ans =

4 9

25 36

49 64

>> 1 ./ A %inverse every element of A

ans =

0.50000 0.33333

0.20000 0.16667

0.14286 0.12500

>> log(A) %element wise log

ans =

0.69315 1.09861

1.60944 1.79176

1.94591 2.07944

>> exp(A) %element wise exponent

ans =

7.3891 20.0855

148.4132 403.4288

1096.6332 2980.9580

>> -A %negate all element

ans =

-2 -3

-5 -6

-7 -8

>> abs(-A) %take absolute of all -ve element

ans =

2 3

5 6

7 8

>> v

v =

4 6 7 1 8 7 2 9 7 3 8 6 4 1 9 7

>> max(v)

ans = 9

>> min(v)

ans = 1

>> mean(v)

ans = 5.5625

>> [val, index]= max(v)

val = 9

index = 8

>> [val, index]= min(v)

val = 1

index = 4

>> A<3 %element wise comparision of A

ans =

1 0

0 0

0 0

>> find(v==7) %return index where element =7

ans =

3 6 9 16

>> [row,col]=find(A>3) %return index where element>3

row = %(2,1),(3,1),(2,2),(3,2)

2

3

2

3

col =

1

1

2

2

>> sum(v) %sum all element in v

ans = 89

>> prod(v) %product of all element in

>>floor(v) %rounded down

>>ceil(v) %rounded up

>>max(A,B) %return mat of max element from 2 matrices

>> sum(A,1) %per column sum

ans =

14 17

>>sum(A,2) %per row sum

>> pinv(magic(3)) % psuedo inverse of matrix (generally we use this only for inverse)

ans =

0.147222 -0.144444 0.063889

-0.061111 0.022222 0.105556

-0.019444 0.188889 -0.102778

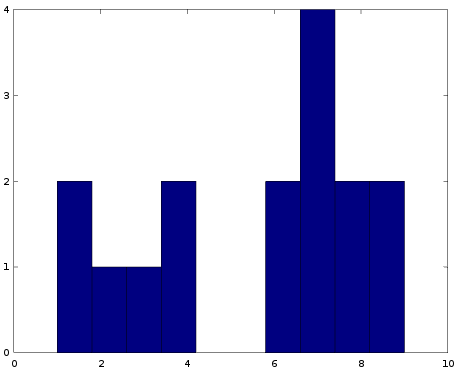
>> inv(magic(3)) % inverse of matrix

ans =

0.147222 -0.144444 0.063889

-0.061111 0.022222 0.105556

-0.019444 0.188889 -0.102778

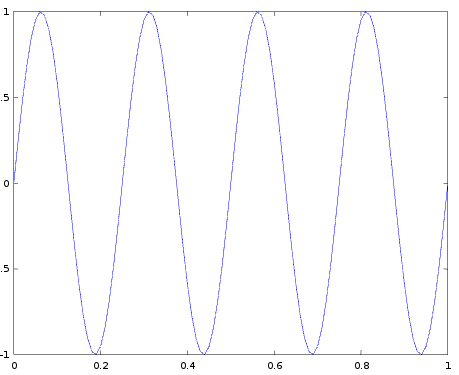
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Plotting Data\_\_\_\_\_\_\_\_

>> v = [4 6 7 1 8 7 2 9 7 3 8 6 4 1 9 7]

v =

4 6 7 1 8 7 2 9 7 3 8 6 4 1 9 7

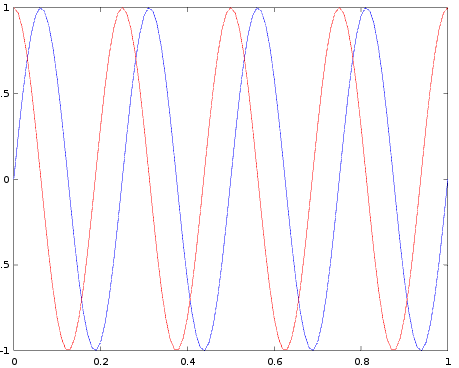
>> hist(v,10) % show histogram



>> A = [0:0.01:1];

>> B = sin(2\*pi\*4\*A);

>> plot(A,B)



>> C = cos(2\*pi\*4\*A);

>> plot(A,B)

>> hold on;

>> plot(A,C,'r'); %plot on same graph

>>xlabel(‘time’); %x axis is time

>>ylabel(‘value’); %y axis is value

>>legend(‘sin’,’cos’) %set legends

>> title('my plot') %set title

>> print -dpng 'myplot.png' %save image as png

----------For loop-----------------

>> for i=1:7,

v(i) = 2^i;

end;

>>i=1;

>> while i<7,

v(i)=100\*i;

i=i+1;

end;

>> v

v =

100

200

300

400

500

600

128

>>break;

>>continue;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_if elseif else end;\_\_\_\_\_\_\_\_\_\_\_\_\_

>> if v(1)==1,

disp("value is one");

elseif v(1)==2,

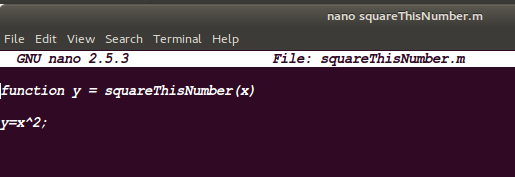
disp("value is two");

else

disp("the is unknown");

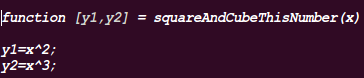
end;

the is unknown

------------------------------------------save function in files----------

>> squareThisNumber(5)

ans = 25



>> [a,b]=squareAndCubeThisNumber(2,3)

a = 4

b = 8