**Machine Learning Algorithms**

* Supervised learning
* 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.

**Regression** problem, which means to predict continuous, valued output (such as price in housing problem)

**Classification** problem is that which predicts discrete valued output ( such as breast cancer problem, can have more than two outputs als well)

Note :- as discussed in the lecture, mosty we used two features / attributes to predict our outcome, by features we means two sets of data which were plotted on the 2-d graph , but in reality one can has infinte number of data sets and storing and using these data sets is very imp for correct prediction.

## **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.

Unsupervised learning has data which doesn’t have any labels or data which has same labels. So ,we have a data set and we are not told what to do with data set or what this data set point to. It’s something like this that here is the data set, so can you find some structure in the data set.

**Clustering** algorithm – one type of unsupervised learning which means it decides and divides data sets into two different clusters, one such example is google news which clusters large number of news articles into one based on similar news. Some of the other use cases of clustering algorithm are:

* Oragnise large computer clusters – predicting which grp of computers will work together more efficiently
* Social network analysis – which are your cohesive grp of friends
* Market segmentation – companies can predict what product will be suitable for a particular group of customers
* Astronomical data analysis – analysing groups of galaxies

**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).

## **Linear Regression in one variable**

Also known as univariate linaer regression.

So we take example of predicting a cost of house provided we have the size and cost data sets with us.

Lets take a look at some notations which will be used throughout the course ( taking example of housing prediction eample)

m --- number of training examples

x’s --- input variables/ features (e.g. size in feet)

y’s --- output variables/ target variables (e.g. cost in dollars)

(x,y) --- single training example

(Xi, yi) --- ith training example

In actual, supervised learning works in below way:

**Training Set**

Fed to

**Learning Algorithm**

**h**

Size of house Estimated price

(x) hypothesis (estimated value of y)

So basically, h is a function called hypothesis which maps x’s to y’s. now we will represent h ( for sake of simplicity, we will be using single line function)

hᶿ  = ᶿ0 + ᶿ1x

short hand, we will also be using h(x)

## **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.

hᶿ  = ᶿ0 + ᶿ1x

ᶿi’s are called parameters and we will now see how to choose ᶿi’s

So the idea is to choose theta 0 and theta 1 so that hᶿ (x) is close to y for our training example (x,y)

So we will minimize ᶿ0 and ᶿ1 such that ( h(x) – y) is minimized because y is the actual value of output variable and h(x) is the predicted value of output variable with ᶿ0 and ᶿ1. So the equation which will be formed is

m

1/2m ∑ ( hᶿ(x (i) ) – y(i))2 which we have to minimize over ᶿ0 and ᶿ1

i=1

this function is known as Cost Function and is denoted by J(ᶿ0, ᶿ1) . Also known as the **squared error function or Mean squared error.** So

m

J(ᶿ0, ᶿ1) = 1/2m ∑ ( hᶿ(x (i) ) – y(i))2

i = 1

The reason that we have picked squared error cost functions over other cost functions is that this cost function works pretty well for linear regression problems.

The mean is halved ( ½ )as a convenience for the computation of the gradient descent, as the derivative term of the square function will cancel out the ½ term.

### **Cost Function – Intuition 1**

For simplification, we will be assuming that ᶿo = 0 , so our hypothesis function will be

hᶿ(x) = ᶿ1 . x

our cost function becomes and we will be minimizing it to get value of ᶿ1

m

J(ᶿ1) = 1/2m ∑ ( hᶿ(x (i) ) – y(i))2

i = 1

our hypothesis function is a function of x whereas our cost function is a function of parameter ᶿ1

### **Cost Function – Intuition 2**

Here we manually plotted contour plots of ᶿ0 and ᶿ1 and price versus size graph and we saw that how different values of ᶿ0 and ᶿ1 results in different plots for price v/s size. Since we required the minimum value of cost function, so we go on assuming values of ᶿ0 and ᶿ1 till we got the best possible graph for our hypothesis and also that graph will correspond to minimum value of cost function.

# **Gradient Descent Algorithm**

It’s an algorithm for minimizing the cost function J. It turns out gradient descent is a more general algorithm, and is used not only in linear regression. It's actually used all over the place in machine learning.

In general, gradient descent can minimize any function, J (ᶿ0,ᶿ1,ᶿ2,ᶿ3………ᶿn) over ᶿ0,ᶿ1,ᶿ2….ᶿn

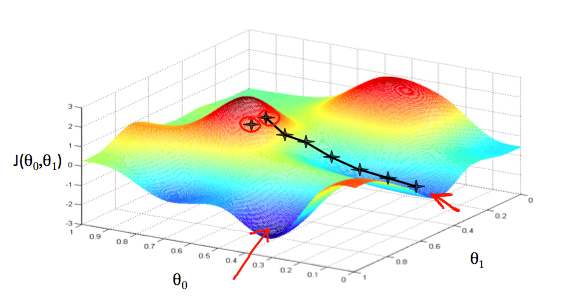
But here for simplicity, we will take only two values ᶿ0 and ᶿ1.

Outline:

* We will start with some value of ᶿ0 and ᶿ1, say 0,0
* And we will keep changing ᶿ0 and ᶿ1 to reduce j(ᶿ0, ᶿ1) until we hopefully end up at a minimum or may be at local minimum.

Theoretically in gradient descent, we start at some point and then we see in the nearby values of that point, which value should I take in order to minimize the function , J . If you don’t see such values of ᶿ parameters, then it means you are already at some minimum value. Inversely if you see such value and continue the above process until you reach the point where you are at the local minima.

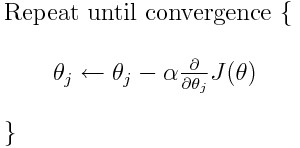
Graphically, We put ᶿ0 on the x axis and ᶿ1 on the y axis, with the cost function on the vertical z axis. The points on our graph will be the result of the cost function using our hypothesis with those specific theta parameters. The graph below depicts such a setup.



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.

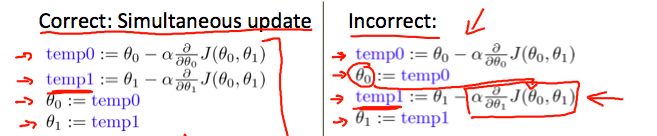
In mathematical terms, it is



j=0,1 represents the feature index number

α ----- is called learning rate and what it does is, it controls how big step we take while picking up the new values of ᶿ

At each iteration j, one should simultaneously update the parameters ᶿ1,ᶿ2,…,ᶿn. Updating a specific parameter prior to calculating another one on the jth iteration would yield to a wrong implementation.



## **Gradient Descent – Intuition 2**

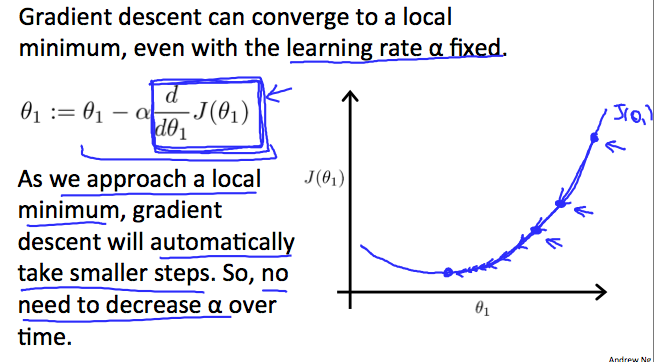
Significance of learning rate:

* If α is too small, gradient descent can be very slow.
* If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge.

Note :- 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 partial differential of J wrt ᶿapproaches 0 as we approach the bottom of our convex function. At the minimum, the derivative will always be 0 and thus we get:



## **Gradient Descent for Linear Regression**

Now we will apply gradient descent algorithm to minimize our squared error cost function.

hᶿ  = ᶿ0 + ᶿ1x

m

J(ᶿ0, ᶿ1) = 1/2m ∑ ( hᶿ(x (i) ) – y(i))2

i = 1

ᶿj := ᶿj – α(∂/∂ᶿj ( J(ᶿ0, ᶿ1 ))

Note:- Cost function is a convex function, so it will not have any local optimum. Instead, it will only have a global minimum.

Gradient descent is also known as Batch gradient descent, where batch means that, at every step of gradient descent we are using all the training examples. ( because we are using summation from i to m)

# **Matrices – Inverse and Transpose**

If A is a m\*m matrix (also known as square matix), and if it has an inverse, then

A \* A -1 = A -1 \* A = I

Let A be a m\*n matrix and let B = AT , then B is a n\*m matrix and

Bij = Aji

Note: - Matrices that don't have an inverse are singular or degenerate.

# **Multivariate Linear Regression**

## **Multiple Features**

Now we will have more features instead of just one such as number of bedrooms, number of floors, age of homes, etc. to predict price of a house. We will denote each of these features with x1, x2, x3,.. and so on for the features and y for the prediction task here the price for example.

And n = the number of features

So our hypothesis will become:

hᶿ = ᶿ0 + ᶿ1x1 + ᶿ2 x2 + ᶿ3x3 + ᶿ4x4 (since now we have four features)

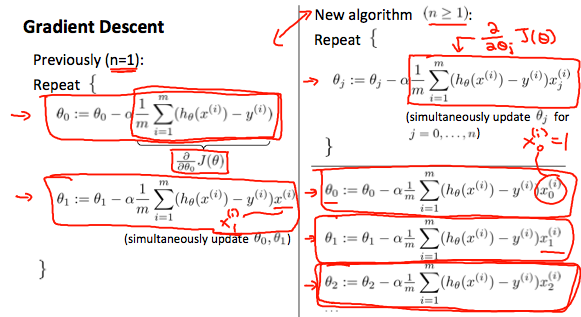
for sake of simplicity, let’s assume x0 = 1, so our hypothesis becomes

hᶿ = ᶿ0x0 + ᶿ1x1 + ᶿ2 x2 + ᶿ3x3 + ᶿ4x4

now if we define a feature vector and parameter vector as

X = vector of x0, x1, x2… xn (has n=1 elements)

ᶿ = vector of ᶿ0, ᶿ1,ᶿ2….ᶿn (has n=1 elements)



## **Gradient Descent in Practice I - Feature Scaling**

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

Make sure that the features are on same scale. We try to bring every feature approximately in the range of

-1 <= xi <= 1 or

−0.5 ≤ xi <=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**: 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:** 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.

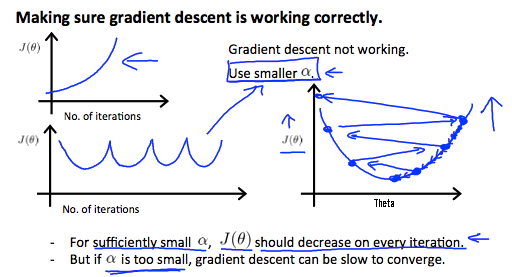
Xi = ( xi – ui) / si

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

## **Gradient Descent in Practice I – 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. It has been proven that if learning rate α is sufficiently small, then J(θ) will decrease on every iteration.



To summarize:

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

## **Features and Polynomial Regression**

We can improve our features and the form of our hypothesis function in a couple different ways. We can combine multiple features into one. For example, we can combine x1 and x2 into a new feature x3. For e.g x3 = x1\*x2 ( like if we are provided length and breadth of house as features, then we can combine them both to get a new feature area).

**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 hᶿ(x) = ᶿ0 + ᶿ1x1, then we can create additional features based on x1, to get quadratic function hᶿ(x) = ᶿ0+ᶿ1x1+ᶿ2x22.

​

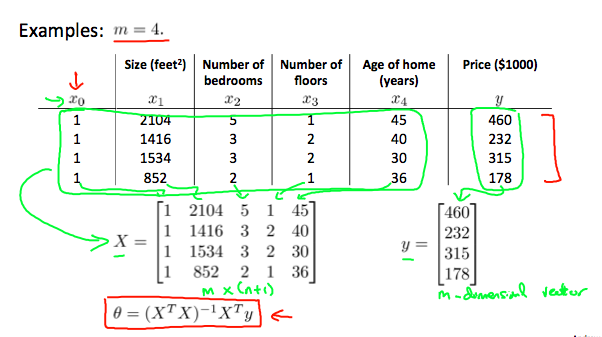
One important thing to keep in mind is, if you choose your features this way then feature scaling becomes very important.

# Computing Parameters Analytically

## Normal Equation

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

ᶿ = (XT X)-1 XT y



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

|  |  |
| --- | --- |
| Gradient Descent | Normal Equation |
| Need to choose alpha | No need to choose alpha |
| Needs many iterations | No need to iterate |
| O (kn2) | O (n^3), need to calculate inverse of XTX |
| Works well when n is large | Slow if n is very large |

With the normal equation, computing the inversion has complexity O(n^3). So if we have a very large number of features, the normal equation will be slow. In practice, when n exceeds 10,000 it might be a good time to go from a normal solution to an iterative process.

## Normal Equation Noninvertibility

If XTX 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.

# Classification

Classification problems takes discrete set of values. That is the output y, which we will be predicting will have values

y € { 0,1 }, known as binary class classification problems

where 0 : represents “Negative Class” (-) and

1: represents “Positve Class” (+)

Afterwards we will also see examples where we will predict values where y can take more than two values such as

y € {0,1 ,2,3} also known as multi-class classification problems

Linear Regression hypothesis h(x) = (theta)transpose \* x will not work for classification problems because it can give up strange results, so we will be understanding a new algorithm known as logistic regression to tackle classification problems which will give us discrete set of outputs.

## Hypothesis Representation of Classification problem

Logistic Regression Model:

0 <= h(x) <= 1

h(x) = g( (theta)transpose \* x ) = 1 / ( 1 + e- ((theta)transpose \* x) )

where g(z) = 1 / (1 + e -z)

Interpretation :

h(x) => estimated probability that y = 1 on input x

which means that for a tumor, if our hypothesis has predicted a value of 0.7 for a particular training set, then it means that tell the patient that he has 70% chance of tumor being malignant.

h(x) = P(y = 1| x,theta) ==🡺 probability that y = 1 given x and parameterized by theta.

And

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(X) >= 0.5 -----> y =1

h(x) < 0.5 ------> y = 0

Looking at the graph of sigmoid function, we can see that for all values of z >= 0 , g(z) >= 0.5 and for all values of z < 0, g(z) < 0.5. Now implementing this in our hypothesis, we get

h(x) = g( (theta)transpose \* x )

so if (theta)transpose \* x >= 0, then h(x) >= 0.5 which means y = 1

and (theta)transpose \* x < 0, then h(x) < 0.5 which means y = 0

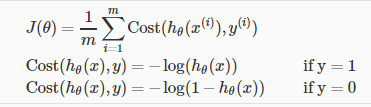
The decision boundary is the line that separates the area where y = 0 and y =1. It is created by our hypothesis function.

# Logistic Regression Model

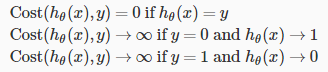
## Cost Function

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, we will define our new cost function for logistic regression, which is



This cost function gives us below points to discuss,



If our correct answer 'y' is 0, then the cost function will be 0 if our hypothesis function also outputs 0. If our hypothesis approaches 1, then the cost function will approach infinity.

If our correct answer 'y' is 1, then the cost function will be 0 if our hypothesis function outputs 1. If our hypothesis approaches 0, then the cost function will approach infinity.

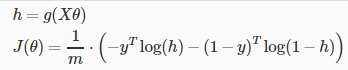
Note : that writing the cost function in this way guarantees that J(θ) is convex for logistic regression.

## Simplified Cost Function and Gradient Descent

We can fully write our cost function by joining the two terms as:



And a vectorised implementation is :



Also, vatorised form of gradeint descent is:



## Advanced Optimisation

"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:



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()"

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

initialTheta = zeros(2,1);

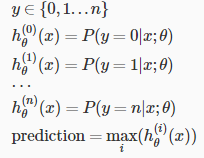
[optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialTheta, options);

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.



# Solving the problem of Overfitting

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

## Cost Function

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

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



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

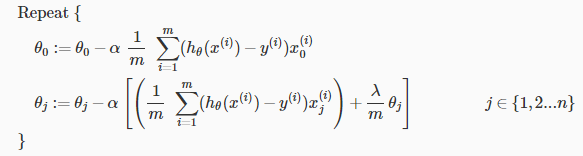
Using the above cost function with the extra summation, we can smooth the output of our hypothesis function to reduce overfitting. If lambda is chosen to be too large, it may smooth out the function too much and cause underfitting. Hence, what would happen if \lambda = 0λ=0 or is too small ?

However, we don’t penalize ᶿ0.

## Regularised Linear Regression

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.



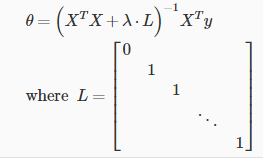
With some manipulation our update rule can also be represented as:



The first term in the above equation, 1−*α(λ/m)*​ will always be less than 1. Intuitively you can see it as reducing the value of *θj*​ by some amount on every update. Notice that the second term is now exactly the same as it was before.

Normal Equation

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

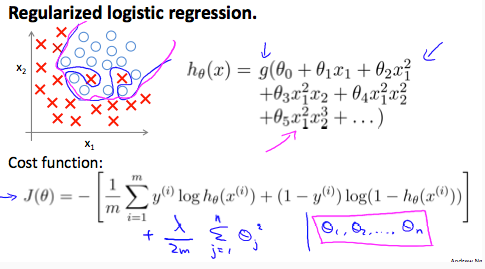


L is a matrix with 0 at the top left and 1's down the diagonal, with 0's everywhere else. It should have dimension (n+1)×(n+1). Intuitively, this is the identity matrix (though we are not including x0​), multiplied with a single real number λ.

Recall that if m < n, then X^TX is non-invertible. However, when we add the term λ⋅L, then X^TX + λ⋅L becomes invertible.

## Regularised Logistic Regression

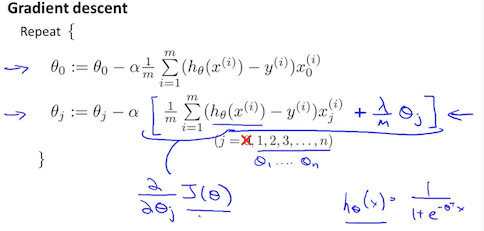
We can regularize logistic regression in the same way as linear regression and avoid overfitting.



Cost function will take the form as below:



And gradient descent as below:



# Neural Networks

## Model Representation – 1

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 x1​⋯xn​, and the output is the result of our hypothesis function. In this model our x0​ input node is sometimes called the "bias unit." It is always equal to 1. In neural networks, we use the same logistic function as in classification, 1/ 1+e^− thetaT 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:



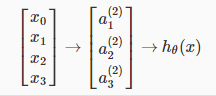
Our input nodes (layer 1), also known as the "input layer", go into another node (layer 2), which finally outputs the hypothesis function, known as the "output layer".

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

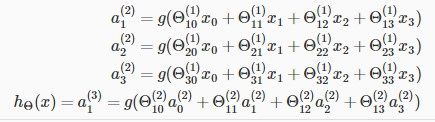
In this example, we label these intermediate or "hidden" layer nodes a20 ​⋯ a2n and call them "activation units."



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



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

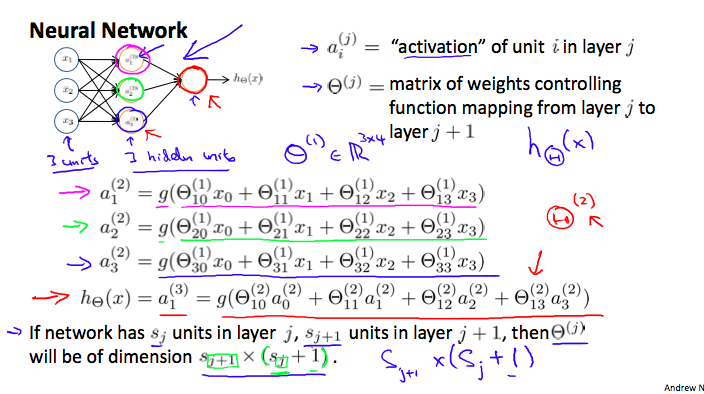


This is saying that we compute our activation nodes by using a 3×4 matrix of parameters. We apply each row of the parameters to our inputs to obtain the value for one activation node. Our hypothesis output is the logistic function applied to the sum of the values of our activation nodes, which have been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

Each layer gets its own matrix of weights, Θ(j). The dimensions of these matrices of weights is determined as follows:



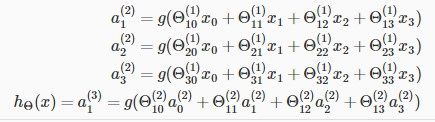
The +1 comes from the addition in Θ(j) of the "bias nodes," x0 and Θ0(j)​. In other words the output nodes will not include the bias nodes while the inputs will. The following image summarizes our model representation:



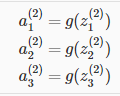


## Model Representation – 2

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



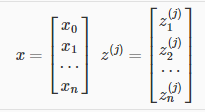
In this section we'll do a vectorized implementation of the above functions. We're going to define a new variable zk(j)​ that encompasses the parameters inside our g function. In our previous example if we replaced by the variable z for all the parameters we would get:



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

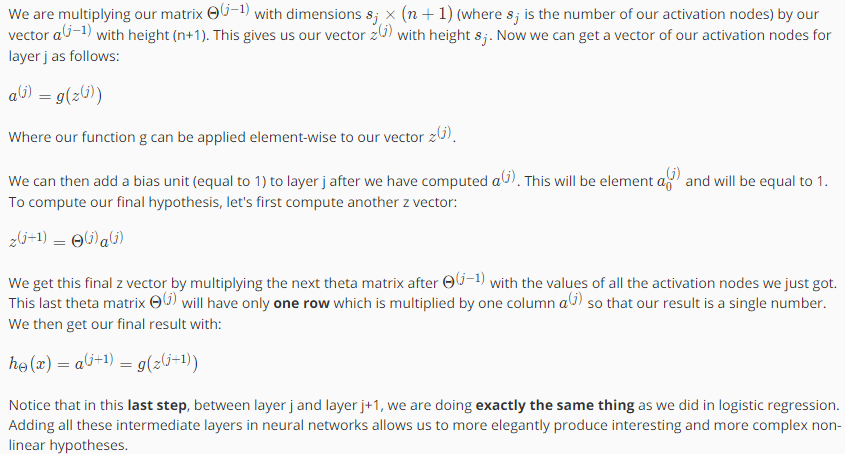


The vectorised representation of x and zj is :



Setting x = a1, we can re-write the equation as:



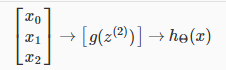


# Neural Networks Representation

## Examples and Intuitions – 1

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 function will look like this:

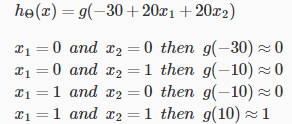


Remember that x0​ is our bias variable and is always 1.

Let's set our first theta matrix as:

Θ(1)=[−30​ 20 ​20​]

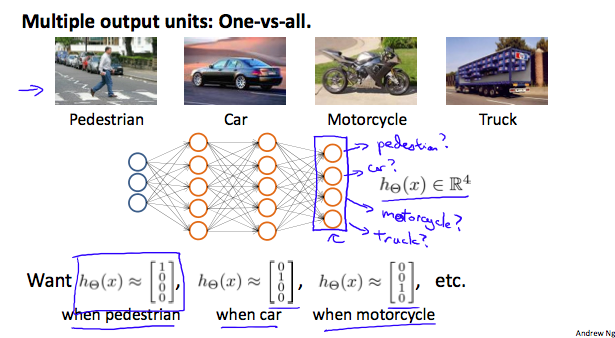
This will cause the output of our hypothesis to only be positive if both x1​ and x2​ are 1. In other words:

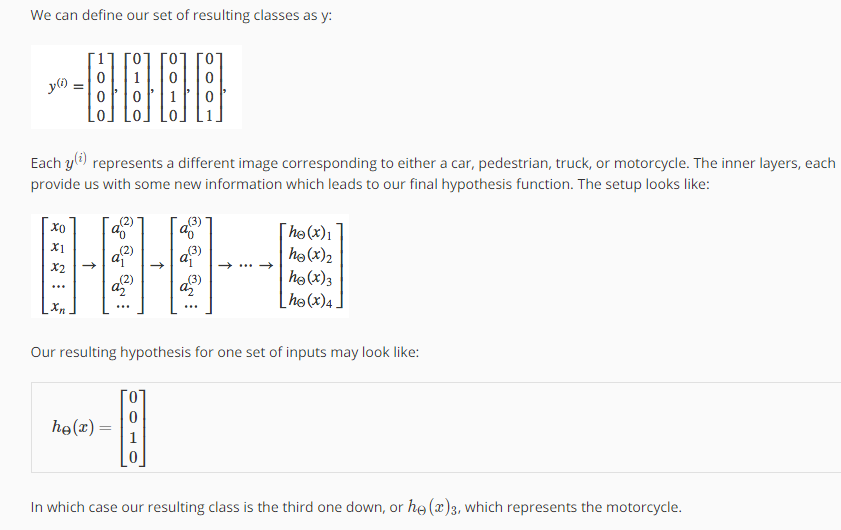


So we have constructed one of the fundamental operations in computers by using a small neural network rather than using an actual AND gate. Neural networks can also be used to simulate all the other logical gates.

## 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:





# Neural Networks Learning

## Cost Function

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

* L = total number of layers in the network
* *Sl*​ = 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*Θ​(*x*)*k*​ as being a hypothesis that results in the k^{th}*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 would be little bit more complicated:



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

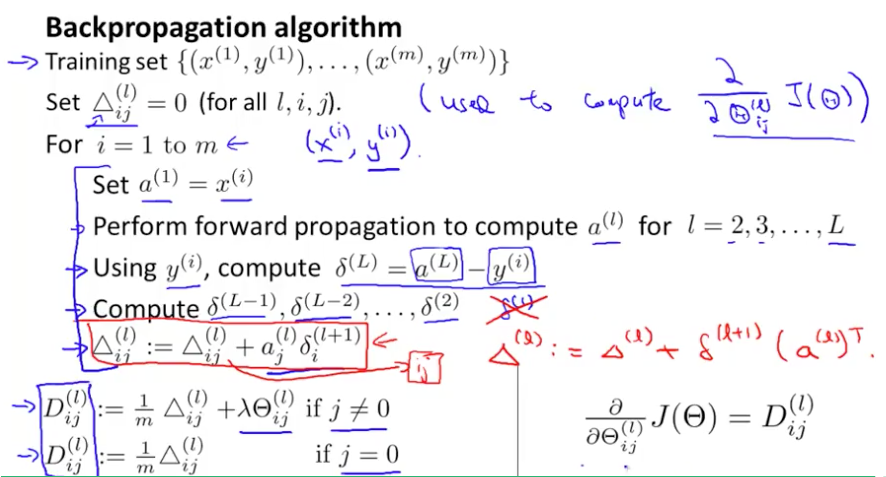
## Backpropagation Algorithm

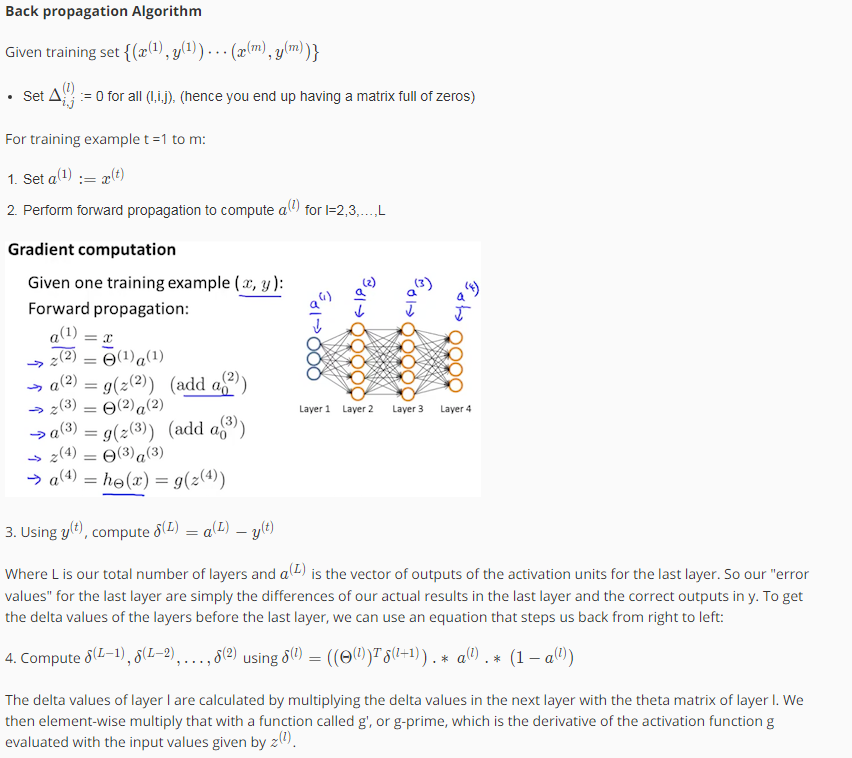
"Backpropagation" is neural-network terminology for minimizing our cost function, just like what we were doing with gradient descent in logistic and linear regression. Our goal is to compute:

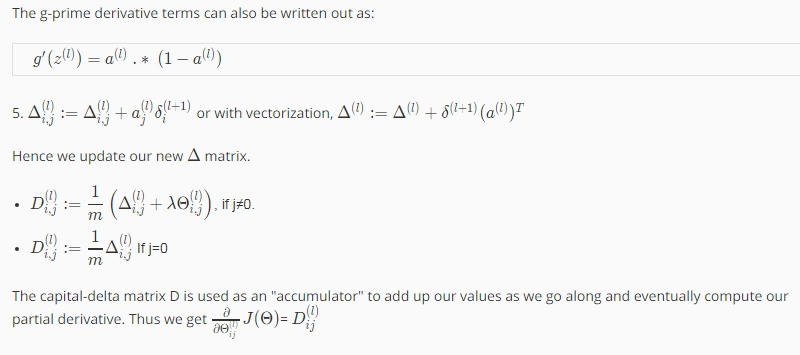
Minmise J(Θ)

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).

To do so, we use the following algorithm:







## Backpropagation Intuition

