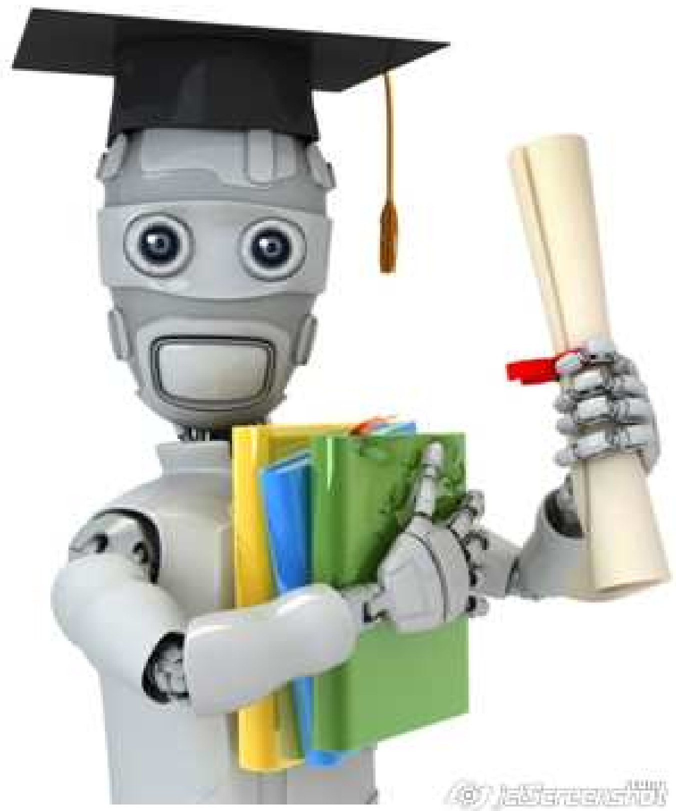
MACHINE LEARNING

04/22/2013-07/01/2013

BY PROF. ANDREW NG



Notes by: Yuval W

YWSecond@gmail.com

Contents

Introduction (Week 1)

What is Machine Learning?

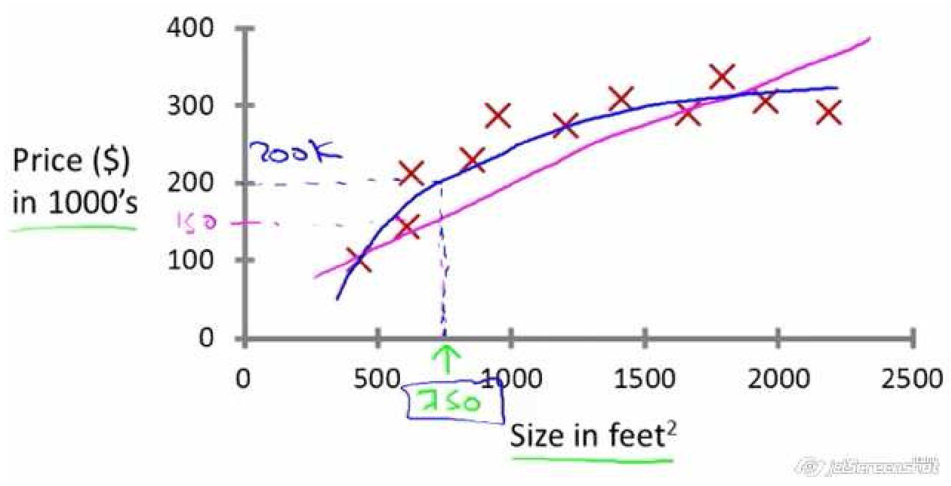
There isn’t one accepted definition of Machine Learning, one and ancient definition is by Arthur Samuel (1959): “Field of study that fives computers the ability to learn without being explicitly programmed”. Samuel wrote a checkers playing program that played 10s of thousands games against itself and recorded what board positions lead to what result, over time the computer learned what board position were leading to success and was able to beat Samuel himself.

A rather more modern and formal definition is by Tom Mitchell (1998): “A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E”.

So in the checkers game example, the task is making a play, P is the win/loss result and E is the number of games.

Supervised Learning

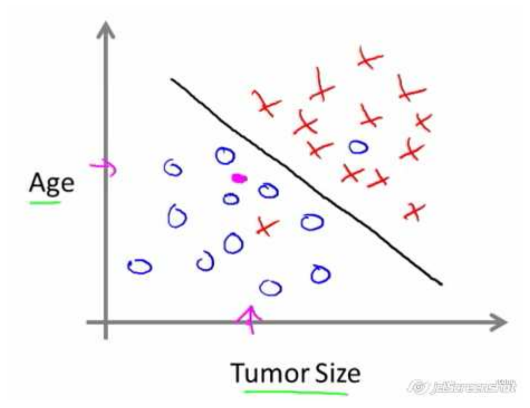
We will define Supervised Learning more formally later in the course, but for now let’s start with an example.
Assume you have the following data on home prices, each instance on the chart is a “house sale transaction” where you have the sale price and the house size. Now assume you want to predict what would be the price of a 750 square feet house. One way is fitting a straight line thru the data and see what it would predict for 750. Instead of a linear straight line we might decide that it is better to fit a second-order polynomial function which might fit the data “better” and see what it predicts for a 750 sq ft house. We will learn later how to choose what type of function (linear or polynomial) to apply to the data.



This type of learning is Supervised Learning, it is supervised because we gave the learning algorithm the “right answer” in advance, all our observations had both the dependent and independent variables.

This sort of problem also named a “**Regression**” problem; it is regression because we try to predict a continuous value output.

Let's look at another supervised learning problem.
Assume you want to predict if a breast Tumor is dangerous or not, you have data about different combinations of patient ages and their tumor size, as well as whether it was dangerous or not (supervised learning, we have the "right" answer). So we plot this information on a 2D chart where "O" denotes a non-dangerous tumor and "X" denotes a dangerous tumor, now the learning algorithm problem is to draw a line that separates between the dangerous and non-dangerous groups.

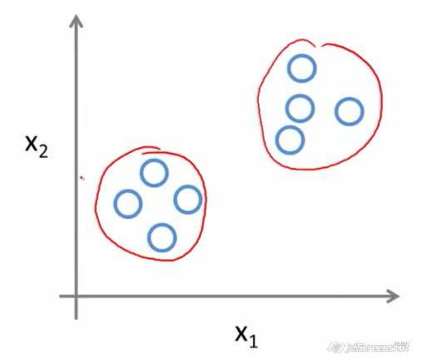


This sorts of problems are "**Classification**" problems, we want an algorithm that learns how to classify the data into different groups (not necessarily a True/False answer, it can be also a nominal group).

Here we used only two "features" or "hints" in order to make the classification (Age and Tumor size), in reality we would have much more features, and we want an algorithm that can deal with an infinite number of features, how do you do that without running out of memory, there is such algorithm that uses a neat mathematical treat to achieve this (Support Vector Machine).

Unsupervised Learning

In supervised learning our data included the "right" answer, meaning the data included information about the Tumor (whether it is dangerous or not); In Unsupervised learning we don't have this information in the data, we have only the "features" (age and size) but not the "end result" of the tumor, and it is the algorithm problem to understand that there are (if there are) two separates groups.

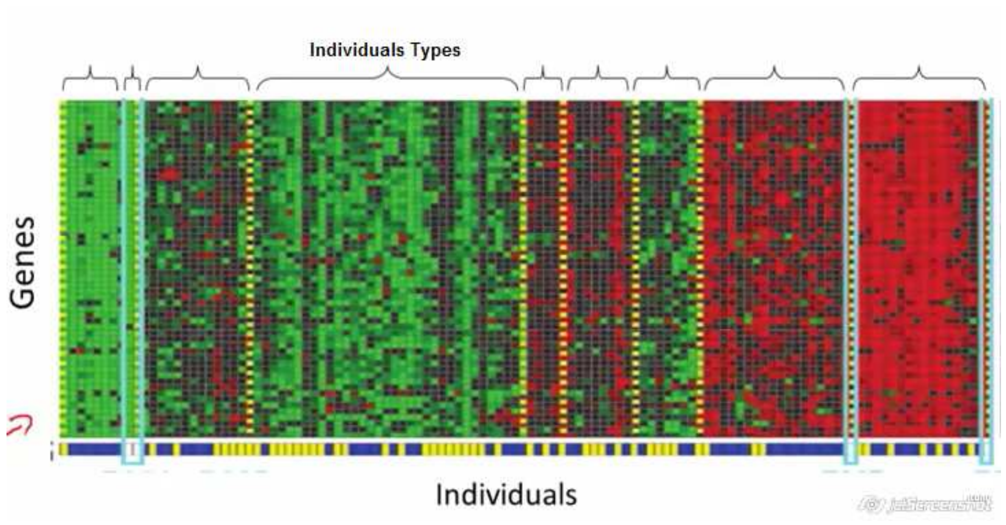


Note in the chart on the left there are no "X"s, only "O"s, we don't know (if and) what is the type of each group (and if there are any groups).
This is a "**Clustering**" problem.

An example of clustering problem is: Analyzing a big data center network transportation and find groups of computers the tends to "work together", if you can identify that you can reorganize the data center in a more efficient way.

Another clustering problem is: given a bunch of individuals and what genes do they have (not all humans have the same genes), group those individuals together. This is clustering problem because we are not giving the algorithm in advance the group type each individual belongs to (if there are groups).

Here is a result of such a classification (each row represents an individual, and each dot is the gene "amount" in the individual).



Another example of unsupervised learning problem is the "**Cocktail Party Problem**", you are in a room full of people and they all talk to each other, using several microphones recordings group the voices (meaning isolate each person voice from the recording).
A simplified version of the problem is with 2 people in a room, talking at the same time, and 2 microphones, an unsupervised learning algorithm was able to beautifully isolate each person's voice (CSI style).
While it seems like a very complicated algorithm in practice the solution is only a single line of code:

```
[W,s,v] = svd((repmat(sum(x.*x,1),size(x,1),1).*x)*x');
```

This was possible by using the Octave software as the prototyping environment.

Linear Regression with One Variable

Model Representation

As we said in the introduction a linear regression is a supervised algorithm (data include the "right" answer), and by regression we mean it is a continuous value as opposed to classification problem.

So our data for a regression problem is going to look something like this (called the **Training Set**):

Size in Sq Ft (X)	Price (Y)
2104	460
1416	232
…	…

The notations we are going to use when describing a regression problem is:

m – The number of training examples (observations)
x's – "input" variable / Feature
y's – "output" variable / "target" variable
(x,y) – one training example.
(x^i, y^i) - The i'th observation.
h – The learning algorithm "solution" or function.

How do we represent h?

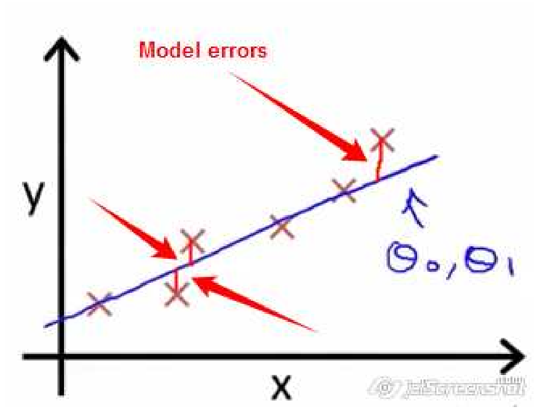
$$h_\theta(x) = \theta_0 + \theta_1 x$$

And this is a single-variable linear regression.

Cost Function

Now we are going to review how we choose $\theta_0$and θ_1 called the model parameters (Thetas). Practically the model parameters are the function slope and Y-Axis intersects.

In a linear regression we try to <u>fit</u> a straight line (h) in our training set, this won't be a perfect fit, but the fit accuracy is dependent on the model parameters (slope and intersects), here is our model line plotted against the <u>actual</u> data, the red lines (which is the difference between the model predicted value and the actual training value) is the modeling error.



Our goal is to find model parameters that will minimize the model <u>sum of squared errors</u>.

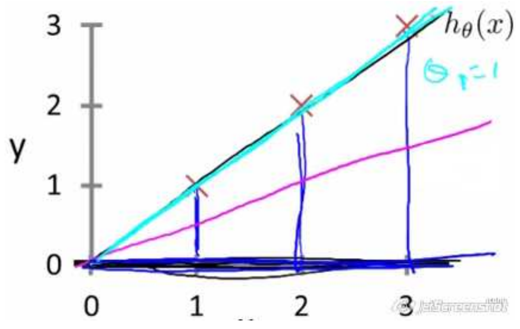
$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

J is our cost function, where $h_0\left(x^i\right)$ is the model predicted value (the model predicted y) and y^i is the training data actual value.
Our goal is to <u>minimize</u> that cost function.

How does the cost function j look like? Remember that j is a function of the model parameters (Thetas), so every time we change the model parameters we get a totally different model (different regression line), but only a different value of j. Thetas are endogenous in the cost function but exogenous in the model function h.

Lets assume our model function is $h_\theta(x) = \theta_1 x$, so theta0 equals zero, and our training set is:
(x, y) →(1, 1) (2, 2) (3, 3)

Now we plot 3 different models,
the cyan line is where theta1 = 1
the pink line is where theta1 = 0.5
the blue line is where theta1 = 0

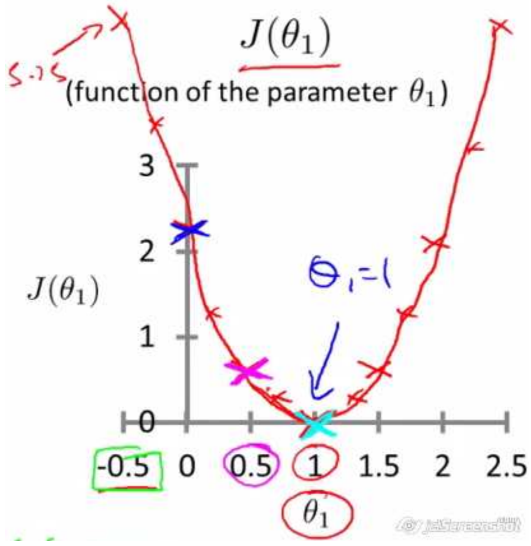


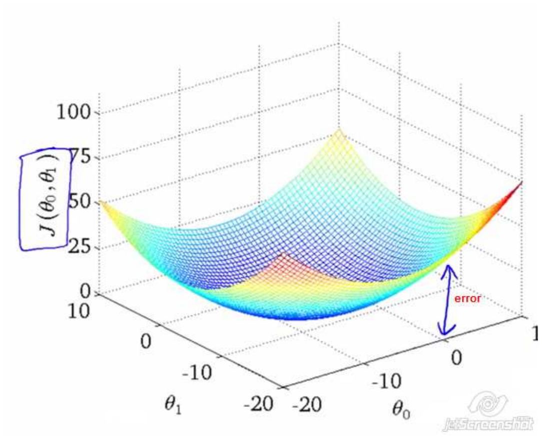
And for each of those thetas we get different value of j, where j is the sum of errors between the model value and the training value, so for example when theta1 = 0.5 (pink line):
h(x) = 0.5*x → when x=1 h=0.5, x=2 h=1, x=3 h=1.5
j(0.5) = 1/2m*[(h(1)-y(1))^2 + (h(2)-y(2))^2 + (h(3)-y(3))^2] = 1/6*[(0.5-1)^2 + (1-2)^2 + (1.5-3)^2] = 0.5833

Or if we let theta1 = 1 so our model is h(x) = 1*x →when x=1 h=1, x=2 h=2, x=3 h=3 and if we calculate the cost function we will get j(1) = 0 and this make sense as h(x)=1*x matches the training data perfectly.

Anyway we can continue and play with different values of theta1 and compute the corresponding value of the cost function, what we get is the chart on the right, note the cyan, pink and blue Xs match the cyan, pink and blue regression lines above.

And as expected we can see that the minimum of this function (the minimum of the errors) is when theta=1 and the model h(x)=1*x.



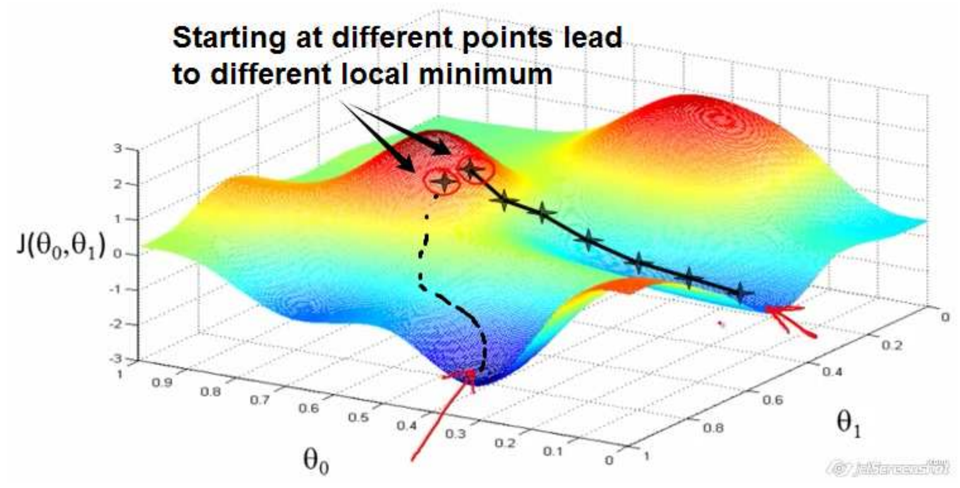


This is when we have single variable cost function as we assumed for this example that theta0 = 0, when we have both theta0 and theta1 in the model, the cost function chart is a 3D one (as we have 3 axis) and looks like the one on the left.

Gradient Descent

Gradient Descent is an algorithm used to minimize different functions; we will use this algorithm to minimize our cost function j.

The general idea behind gradient descent is we start by picking a random combination of the parameters ($\theta_1 \ldots \theta_n$) and see what the cost function value is, then we "look around" and search for the very next combination that will take us down the most, and we keep doing this until we reach a local minimum, this is a local minimum since we are not sure this is also the global minimum as we didn't pass thru all available combinations, had we chosen a different starting point we could have been ended in a different local minimum.



The gradient descent algo formula is:

Repeat until convergence {

$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ for (j = 0 and j = 1)

}

Where Alpha is the learning rate, it controls the "Steps" we are taking when we "look around" for the local minimum, alpha is then multiplied the cost function derivative.
Note that this algo is for j=0 and j=1, it means we are calculating theta0 and theta1 simultaneously, so the

way to do it is by:

$temp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$

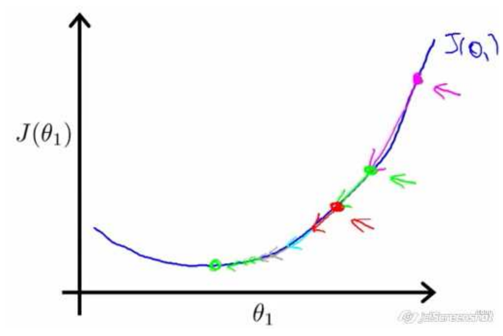
$temp1 := \ \theta_1 - \alpha \frac{\partial}{\partial 1} J(\theta_0, \theta_1)$

$\theta_0 \coloneqq temp0$

$\theta_1 \coloneqq temp1$

So we first calculate BOTH temp0 and temp1 using the current values of theta0 and theta1, and only after we assigned those values back into theta0 and theta1 and start again, this is a simultaneously calculation.

Let's look at a simple example where theta0=0, so our cost function has only 1 parameter and looks like that:



Now recall that in gradient descent we decrease theta by alpha times the cost function derivative, this derivative is equal to the slope degree of the tangent at the current point, so as we go down the slope of the tangent gets smaller and smaller hence the steps we are taking are smaller and smaller (assuming we leave alpha constant). See that the pink arrow is a bigger step than the green arrow since the slope at the pink point is larger than the slope at the green point.
This process will continue until the tangent slope would become 0 and that's our local minimum.

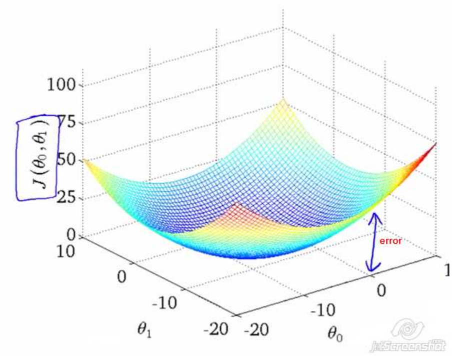
Applying the Gradient Descent to the Linear Regression

When coming to apply the gradient descent algo to minimize the cost function the "tricky" part is the derivative of the cost function.

$$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{\partial}{\partial \theta_0} \times \frac{1}{2m} \sum_{i=1}^{m} h_\theta(x^i) - y^i)^2$$

For j=0: $\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum h_\theta(x^{(i)}) - y^{(i)})$

For j=1: $\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \frac{1}{m} \sum h_\theta(x^{(i)}) - y^{(i)}) \times x^{(i)}$

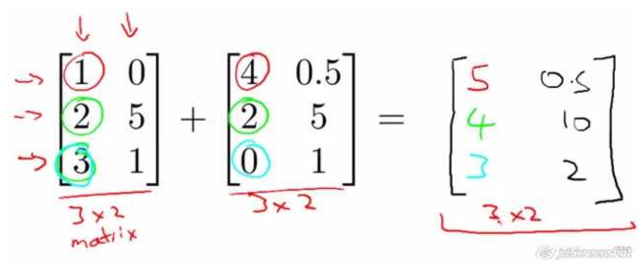


Gradient descent algo finds a local minimum, but since the cost function of a linear regression has bowl shape GD finds the global minimum.

Linear Algebra

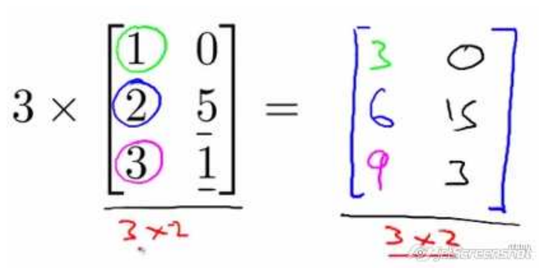
Matrix addition

You can add matrices that are only on the same degree, it is done by adding the matching items in each matrix, say we have matrix A,B so in the result matrix R we get: R(I,j) = A(I,j) + B(I,j).


$\begin{bmatrix}1&0\\2&5\\3&1\end{bmatrix}+\begin{bmatrix}4&0.5\\2&5\\0&1\end{bmatrix}=\begin{bmatrix}5&0.5\\4&10\\3&2\end{bmatrix}$
3×2 matrix
3×2
3×2

Scalar Multiplication

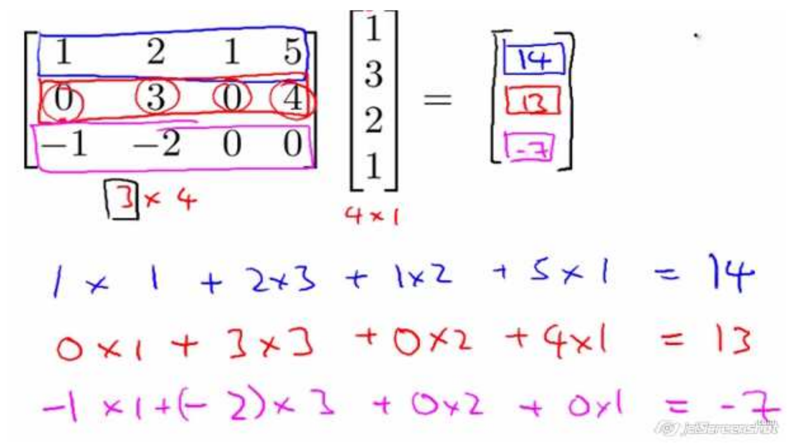
When multiplying a matrix by a scalar we just multiply each item in the scalar.
For <u>dividing</u> a matrix by a scalar we divide each item by the scalar, dividing by X is like multiplying by 1/X.


$3\times\begin{bmatrix}1&0\\2&5\\3&1\end{bmatrix}=\begin{bmatrix}3&0\\6&15\\9&3\end{bmatrix}$
3×2
3×2

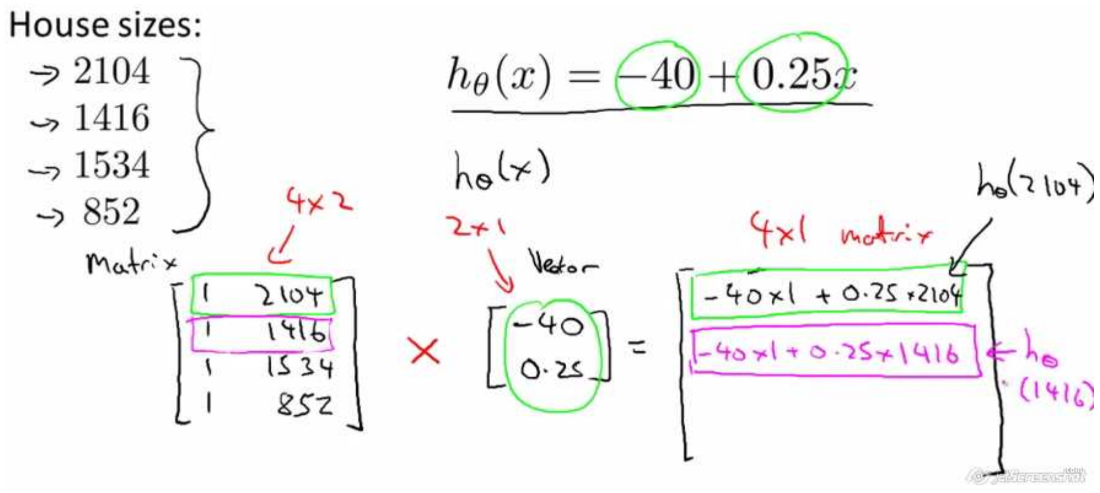
Matrix Vector Multiplication

When we multiply a Matrix A by vector v the number of <u>columns</u> in matrix A has to be equal to the number of rows in vector v, and the result is a vector with the same number of rows as in matrix A.
More formally:
For matrix A(m,n) and a vector v(n) the i'th element in the result vector r:

$$r(i) = \sum_{j=1}^{n} A(i,j) \times v(j)$$

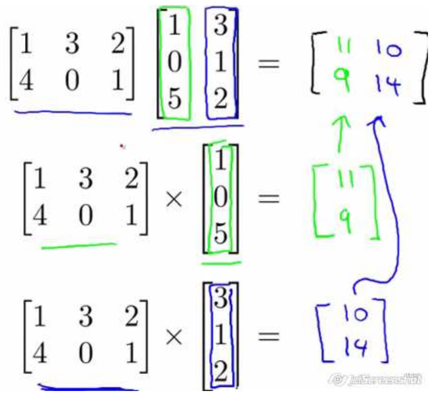

$\begin{bmatrix}1&2&1&5\\0&3&0&4\\-1&-2&0&0\end{bmatrix}\begin{bmatrix}1\\3\\2\\1\end{bmatrix}=\begin{bmatrix}14\\13\\-7\end{bmatrix}$
3×4
4×1
1×1 + 2×3 + 1×2 + 5×1 = 14
0×1 + 3×3 + 0×2 + 4×1 = 13
-1×1+(-2)×3 + 0×2 + 0×1 = -7

How can we use a Matrix x Vector multiplication? Say we have a linear regression prediction model of the regular form h(x) = theta0 + theta1*x(i), and we have a bunch of Xs we want to predict, so instead of looping thru all Xs and putting them in the model equation, what we can do is turn the Xs into a matrix and multiply that matrix by the model parameters vector, the result would be the prediction values of the model.



Matrix Matrix Multiplication

A matrix by matrix multiplication is like multiplying a matrix by several vectors and combining the results:



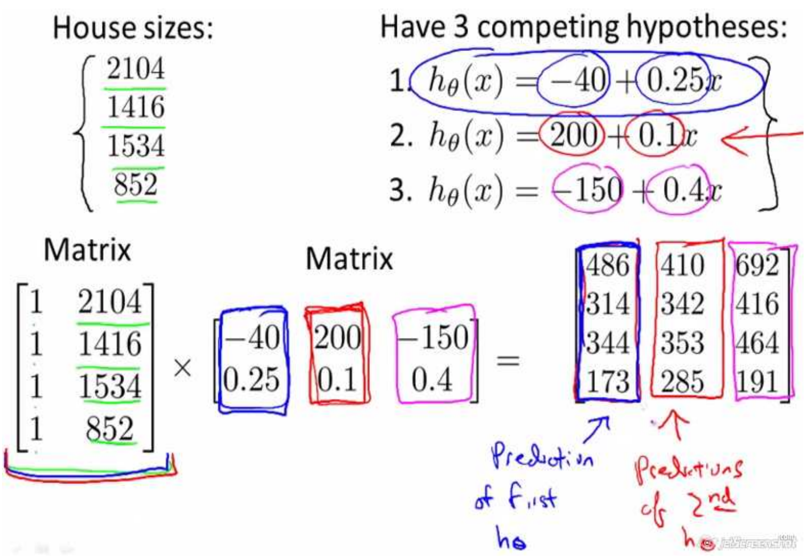
As we can see, in order to multiply two matrices together the number of <u>columns</u> in the first matrix has to be equal to the number of <u>rows</u> in the second matrix.

The result matrix has the same number of rows as in the first matrix, and the same number of columns as in the second matrix.

A(m,n) X B(n, o) = R(m, o)

and the R(i,k) element is:

$$R(i,k) = \sum_{j=1}^{n} A(i,j) \times B(j,k)$$


House sizes:
Have 3 competing hypotheses:
1. $h_\theta(x) = -40 + 0.25x$
2. $h_\theta(x) = 200 + 0.1x$
3. $h_\theta(x) = -150 + 0.4x$
Matrix
Matrix
Prediction of first h_θ
Predictions of 2nd h_θ

What can we do with Matrix by Matrix multiplications?

Say we have 3 models we want to asses, and a set of training set,by turning the training set into a matrix, and the different models parameters into a matrix we could get all the different models predictions by multiplying those.

Matrix Multiplication properties

It is **not** commutative: AxB != BxA

It is Associative: AxBxC = Ax(BxC) = (AxB)xC

The Identity matrix: For each matrix there is an “identity” matrix where multiplied by it we get the same matrix: $A_{m,n} \times I_{n,n} = I_{m,m} \times A_{m,n} = A_{m,n}$

The identity matrix is a matrix where there is “1” on the diagonal and “0” everywhere else:

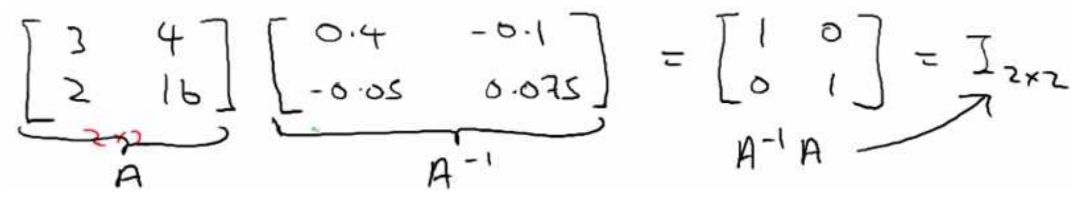
$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Matrix Inverse

When matrix A multiplied by the Inverse matrix A^{-1} the result is the identity matrix I.

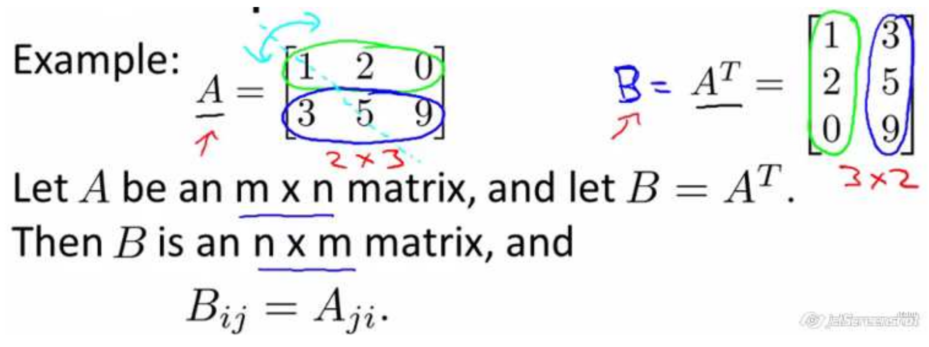
$A \times A^{-1} = A^{-1} \times A = I$

For a matrix to have an inverse matrix it has to be of a (m,m) degree, and not all matrices has an inverse matrix.


$$\underbrace{\begin{bmatrix} 3 & 4 \\ 2 & 16 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} 0.4 & -0.1 \\ -0.05 & 0.075 \end{bmatrix}}_{A^{-1}} = \underbrace{\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}}_{A^{-1}A} = I_{2\times2}$$

The way to come with the inverse matrix is by using computer

Matrix Transpose


Example: $A = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 5 & 9 \end{bmatrix}$ $B = A^T = \begin{bmatrix} 1 & 3 \\ 2 & 5 \\ 0 & 9 \end{bmatrix}$

Let A be an m x n matrix, and let $B = A^T$.

Then B is an n x m matrix, and

$B_{ij} = A_{ji}$.

Linear regression with Multiple Variables (Week 2)

Multiple Features

Up until now we had only one variable/feature in our regression model, in the house pricing model it was the size of the house. Now we want to insert more predicting features into our model, like bedrooms, floors etc, so a multiple variable model would have n features ($x_1 x_2 x_3 \dots x_n$).

New notations:
m – The number of training sets.
n – The number of features.
x^i – A vector of all features of the i'th training row.
x_j^i – The J'th feature in the i'th training row.

The new model equation to support multiple variables is:

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

Note that we have n variables, x(1)…x(n), and we have n+1 parameters $\theta_{0\dots}\theta_n$, so in order to simplify the model and make him suitable for linear algebra we add X_0=1 and what we get is:

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_n x_n$$

Important: Do not confuse, we still have only n features, but we added a "dummy" first feature and set it equal to 1.

Now we have two vectors, the features vector X and the parameters vector θ, each of these vectors is of n+1 dimension (n+1 rows and 1 column). Given a set of features how can we easily compute the predicted hθ(x) value? We transpose the parameters vector (turn it from 1 column into 1 row) and multiply it by the features vector, this gives us the predicted value.

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

Gradient descent for multiple variables

Like a one variable linear regression, also multiple variable linear regression has a cost function, this cost function measure the squared sum of all errors, where error is the difference between the model predicted value and the actual training set value.

$$J(\theta_0, \theta_1 \dots \theta_n) = \frac{1}{2m}\sum_{i=1}^{m} h_\theta\left(x^{(i)}\right) - y^{(i)})^2$$

And our goal is to minimize this cost function in regard to the parameters, so the gradient descent algo:

Repeat until convergence {

$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1 \dots \theta_n)$ Simultaneously for (j = 0 … n)

}

We can write the cost method like this (just put in the model formula instead of h):

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} \frac{1}{2m} \sum_{i=1}^{m} (\theta_0 x_0^i + \theta_1 x_1^i + \cdots + \theta_{n+1} x_{n+1}^i - y^i)^2$$

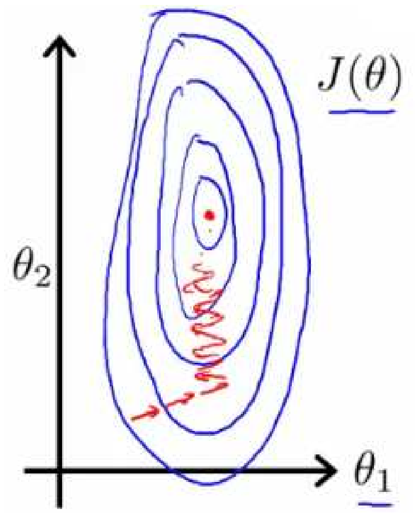
Finally we solve the derivative for j and get:

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (\theta_0 x_0^i + \theta_1 x_1^i + \cdots + \theta_{n+1} x_{n+1}^i - y^i) x_j^i := \ \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^i) - y^i) x_j^i$$

So what we do is starting with random values for our thetas, solve all n+1 equations (equation per theta) to get new values for the thetas, and solve all n+1 equations with the new thetas we got in the prev step. Note that for each equation we solve we have to loop thru all training set 1 thru m.

Gradient Descent in Practice – Feature Scaling

If you have a problem with multiple features you should make sure those features have a similar scale, it will help gradient descent to converge more quickly.



Let's assume, in our home pricing example, we have 2 features, size and bedrooms, size can take values from 0-2000 (sq ft), while bedrooms go from 1-5, when plotting the cost function using those values the contours of the shape would be very sharp (like steep slopes), this makes the gradient descent algo to have many iterations until it reaches the global minimum.

The solution for this issue is to try and scale all the features to be anywhere between -1 and 1, it shouldn't be exactly -1 and 1 "but in the area", rule of thumb is:

- -3 to +3 is generally fine - any bigger bad
- -1/3 to +1/3 is ok - any smaller bad

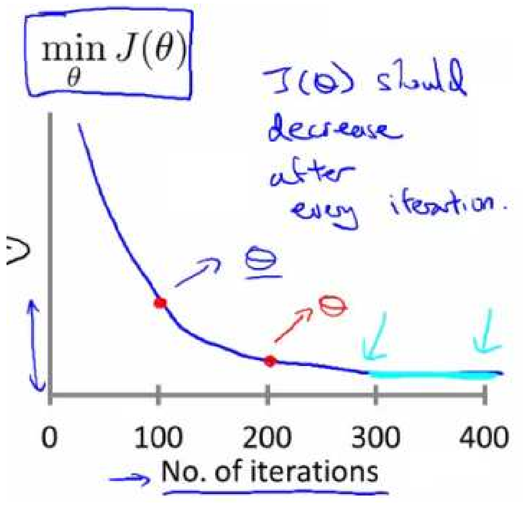
The easiest way to normalize the values is by:

$$x_n = \frac{x_n - \mu_n}{S_n}$$

where:
x(n) – is the n'th feature.
u(n) – is the n'th feature average.
S(n) – could be either the n'th feature standard deviation, or the feature range (max value-min value).
IMPORTANT: When we learn the parameters with normalization it is important to save the values used for the normalization (mean,std etc) so when it's time to make predictions we would normalize the new data using the same values we had while learning.

Gradient Descent in Practice – Learning Rate (α)

How to check if the gradient descent is working? The job of the GD is to minimize J, so we plot J as a function of the number of iterations. If GD is working properly the chart would look like this (decreasing J).

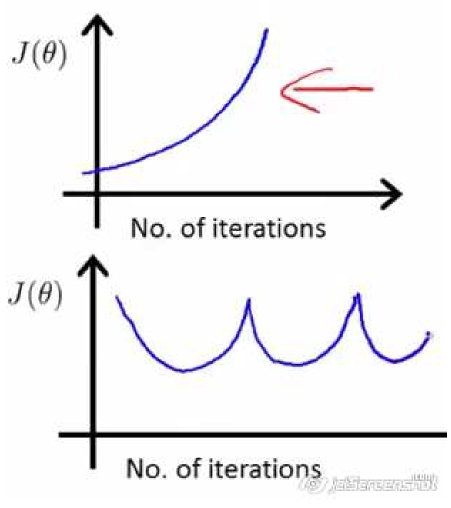


See that between 300-400 iterations J is becoming flat which means GD has converged.

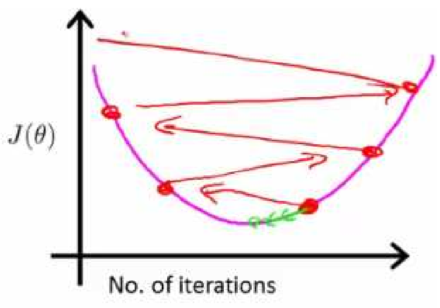
The number of iterations required in order for GD to converge may vary between models and it can't be determined in advance, that's why it's important to look at the plot and see if GD is converging.

There are some Automatic Convergence Tests the checks if GD is converging, usually they compare the change in J to some threshold (like 0.0001), but it is still preferable to look at the plot and verify that the algorithm is doing its job.

How do we know that the GD is NOT working? When we see plots like these:



Here we can see that J is not getting smaller and smaller with every iteration, usually this is due too big Alpha (assume there is no bug in the GD algo implementation). When we use Alpha that is too big the GD algo overshoots the minimum and jump to higher values of J. If for example our J(theta) function looks like that see how each iteration jumps to a higher value of J.



General rule: try alpha 0.01, 0.03, 0.1, 0.3…

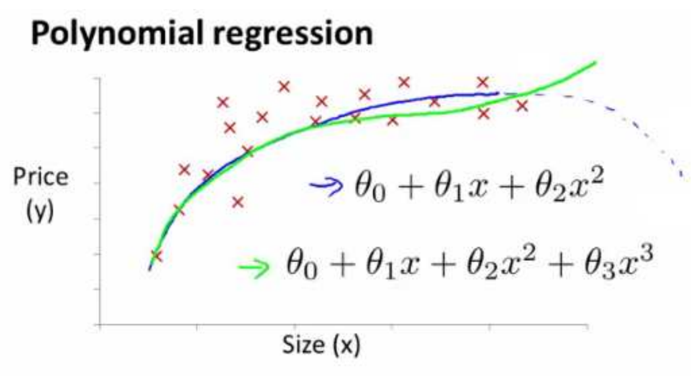
Polynomial Regression

Not always a straight line is a good for our data, sometimes we might want to use different shapes of models like quadratic model:

$$\theta_0 + \theta_1 x + \theta_2 x^2$$

Or a cubic model:

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



Note that the models above have only one feature (x), it is just we took it at squared it etc. So in our home example this could be the Size (sq ft), and the model is:

theta0 + theta1*size + theta2*size^2

How do we treat this sort of model? This is like a regular linear model, we just take the size, and size^2 and size^3 and decide they are 3 different features, so:

$x_1 = x$
$x_2 = x^2$
$x_3 = x^3$

And now we have our regular model: **theta0 + theta1*x1 + theta2*x2 + theta3*x3** and we solve it regularly.

So note there are many different shapes of models we can fit and we have to look at our data and decide, later we will learn about algos that help up decide what model describe our data best.

Important: When we start to square and quadruple our feature (like size) it is important to scale them before we run the gradient descent algo so we won't have very large numbers. We don't necessarily have to use the mean value for scaling, we can just divide by the maximum value of each feature, lets say our size feature has a maximum value of 1000, so x(1) is the size itself, so when scaling we say x(1) = size/1000. Our x(2) feature is size^2, so its max value is 1000^2, so we say x(2)=size^2/1000^2.

Normal Equation

Up until now we minimized the cost function using the Gradient Descent algorithm, for some linear regression problems the normal equation provides a better solution.

The normal equation minimizes the cost function for theta by solving the following equation:

$$\frac{\partial}{\partial \theta_j} J(\theta_j) = 0$$

When the derivative of J is equal to zero it's where the function is minimized. Now note that Theta is a vector, if for example our model is **h(theta) = theta0 +theta1*x(1) + theta2*x(2)** we have 3 thetas, so we have to derive the the J function in regard all three thetas, this will give us 3 equations with 3 variables that we have to solve. Deriving the J function is somewhat complex and we won't go into that, but we will look on how to implement the process of finding each theta.
The Normal Equation does NOT require the features to be scaled, original values are ok.

Solving the Normal Equation

Assume our training set matrix is X (including x(0)=1), and our training set target variable is vector y, then:

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

Where:
X^T – Is the transpose of X (in octave/Matlab you do it by X').
$()^{-1}$ – Is the inverse of the calculation, in octave you you do it by function **pinv()**.

Example:

X(0)	X(1)	X(2)	X(3)	X(4)	y
1	2104	5	1	45	460
1	1416	3	2	40	232
1	1534	3	2	30	315
1	852	2	1	36	178

So finally the result of the following is the vector with values of Theta:

$$\left(\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2104 & 1416 & 1534 & 852 \\ 5 & 3 & 3 & 2 \\ 1 & 2 & 2 & 1 \\ 45 & 40 & 30 & 36 \end{bmatrix} \text{X} \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix} \right)^{-1} \text{X} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2104 & 1416 & 1534 & 852 \\ 5 & 3 & 3 & 2 \\ 1 & 2 & 2 & 1 \\ 45 & 40 & 30 & 36 \end{bmatrix} \text{X} \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$$

In order to solve this use some math software like octave: pinv(X'*X)*X'*y

Normal Equation non-inevitability (optional)

There are some matrices that are not invertible hence Normal Equation can't be applied.
Why is that?
Most of the time, the reason for the invertibility is because some features are linear related, like using Size in sq ft and Size in sq meter (they are linear related because 1m=3.28 ft), solution is remove one feature.
Another reason is if you have too many features without enough training set (e.g m<=n), so in this case just delete some of the features, or use regularization (more on this later in the course).

Gradient Descent vs Normal Equation

Gradient Descent	Normal Equation
Need to chose Alpha	No need to choose Alpha
Needs many iterations	Solve in "one shot"
Works well when n is large	Slow if n is large because computing the inverse matrix has the complexity of O(n^3) and it's slow. Would say that up to n=10,000 it's OK, above – not.
Works on all sorts of models	Works only on a linear model, will not work on logistic models and others.

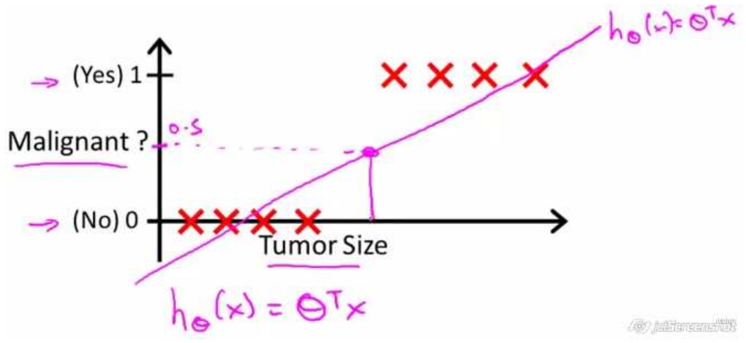
Logistic Regression (Week 3)

Classification Problems

In classification problems we try to predict a result which is a class (like true/false). Example is whether an email is spam or not, whether a financial transaction is fraud or not etc.
We start by problems whose result is binary, i.e True/False, later we would look at problems where the result class may have several classes.
So our dependent variable is:
$y \in \{0, 1\}$
where 0 is the "Negative Class" and 1 is the Positive Class.

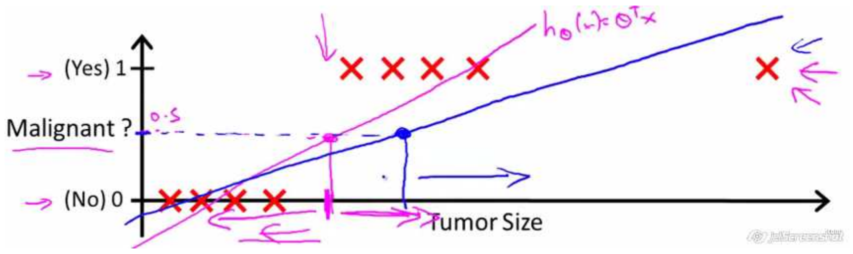
Modeling Classification Problem

Lets look at the following problem of classifying a Tumor.
One option is using the regression model h=Theta*X, the fitted line would look something like this:



We can see that the linear regression output is a continuous value while what we need is a 1/0 output, so we can decide that:
$if\ h_\theta \geq 0.5, predict\ 1$
$if\ h_\theta \leq 0.5, predict\ 0$

So now our model output is 1/0 and we can see that it is doing a good job in predicting the data. But what if our data had looked slightly different, let's say we had an extra observation of very large tumor, this extra observation would change the regression line to something like (blue line):



Now we can see that using 0.5 as the threshold for positive/negative is not good enough.

The problem is that linear regression can have any values, including values that are greater than 1 and less than 0, while our problem is defined by 0/1 values.

What should we do?
As we can't use the linear regression model there is a new model that is created specifically for classification problems, **Logistic Regression**, where its output is: $0 \leq h_\theta \leq 1$

Logistic Regression Model

Out hypothesis for the model is: $0 \leq h_\theta \leq 1$, given that our hypothesis is:

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

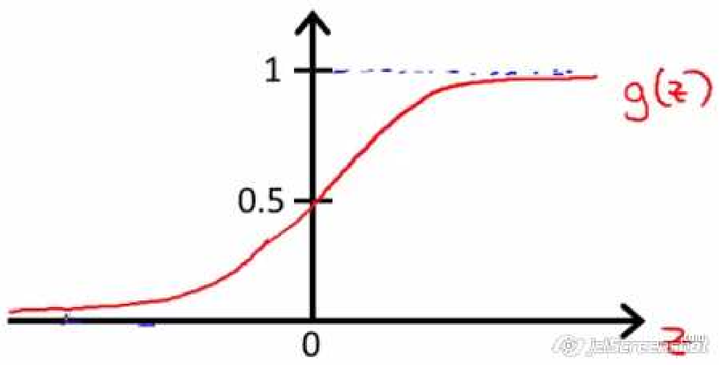
where There and X are vectors.

Where g is the Logistic function:

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

Combining the two together and we get:

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



Interpretation of Hypothesis Output

h(x) on Logistic Regression is the **estimated probability** that y=1 on input x, parameterized by Theta.

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

For example if h(x)=0.7 then there is a 70% chance that y is "true".

What is the probability that y=0 given x? Since the logistic model has a maximum value of 1, we can say that the probability of y=0 is: $1 - P(y = 1|x; \theta)$

Decision Boundary

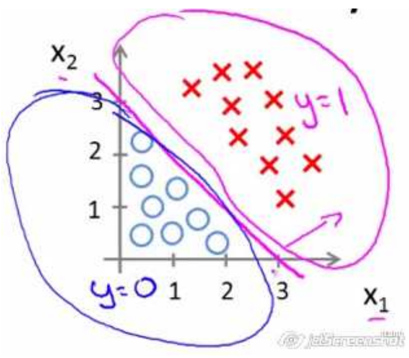
In the logistic regression we said that the prediction is the probability y=1 given x, but our "real" y data is either 1 or 0, so what we do is:

Predict y=1 if $h_\theta(x) \geq 0.5$ and Predict y=0 if $h_\theta(x) < 0.5$

Looking at the Logistic function chart above we see that g(z)=0.5 when z=0, in our logistic model **z=$\boldsymbol{\theta^T X}$** so we can say that:

Predict y=1 if $\theta^T X \geq 0$ and Predict y=0 if $\theta^T X < 0$

The Decision Boundary Line



Assume our model: $\boldsymbol{h_\theta(x) = g(\theta_0 + \theta_1 x + \theta_2 x_2)}$ = $\boldsymbol{\frac{1}{1+e^{-(\theta_0+\theta_1 x+\theta_2 x_2)}}}$

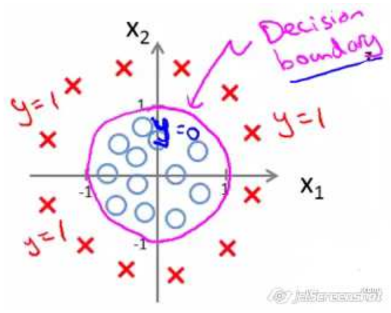
We will learn later how to find the thetas, but for now let's assume the theta vector is [-3, 1, 1], so we predict that

y=1 if $\boldsymbol{\theta^T X \geq 0}$ → y=1 if -3 + x(1) + x(2) >= 0 → y=1 if x(1)+(x2) >= 3

We can draw a line where x(1)+x(2)=3, and anything right to this line is where we predict y=1 and anything left to this line is where we predict y=0. This is our boundary line, it separates the region between where the hypothesis predicts 1 and when it predicts 0.

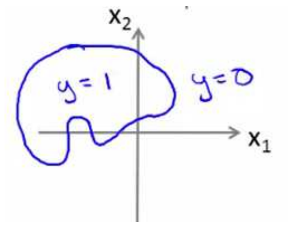
Non-Linear Decision Boundaries

What if our data look like that?



In this case we can use different hypothesis models that might describe the data better, for example something like:
$\boldsymbol{h(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2)}$
We still don't know how to estimate the thetas but assume we come up with thetas vector as [-1, 0, 0, 1, 1], so we predict y=1 if **-1+x(1)^2 + x(2)^2 >= 0 → x(1)^2 + x(2)^2 >= 1**, and if we plot that function we would see it's a circle.

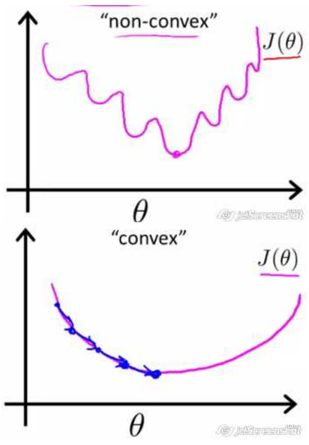
And we can use even more complicated models and get different Decision Boundaries.



$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_1^2 x_2 + \theta_5 x_1^2 x_2^2 + \theta_6 x_1^3 x_2 + \dots)$$

Cost Function

Now we are going to talk about choosing the Thetas in the Logistic model.



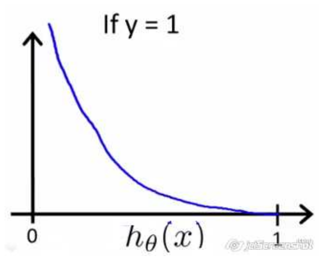
In the linear regression model we defined the cost function as the sum of the squared errors: $\boldsymbol{J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2}$

Theoretically there is no problem using it for the logistic model as well, the problem is, when we plug the logistic model $\boldsymbol{h_\theta(x) = \frac{1}{1+e^{-\theta^T X}}}$ into the cost function we get a non-convex function, meaning a function with many local minimums, which might prevent from the gradient descent algo find the global minimum.

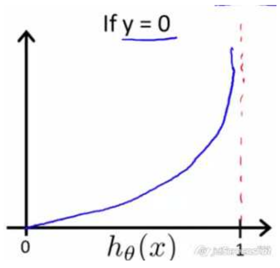
Logistic Model Cost Function

The cost function we are going to use for the logistic model is:

$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)})$ Where $Cost(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & if\ y = 1 \\ -\log(1 - h_\theta(x)) & if\ y = 0 \end{cases}$



Here is the shape of the cost function for each situation. The characteristic of this function is, when h(x) = 1 and y is actually 1 (left chart) then the cost is 0, that's ok because the model made a perfect prediction and there is a zero cost, but as the model's prediction start to go towards zero (when y is actually 1) the cost is getting bigger, till infinity, if the model had a prediction h(x)= 0.



The same intuition applies for the case when y is actually 0 (right chart).
Remember that the model's prediction is the probability of y=1 $P(y = 1|x; \theta)$

We can write the cost function in a simpler way as such:

$$Cost(h_\theta(x), y) = -y \times \log(h_\theta(x)) - (1-y) \times \log(1 - h_\theta(x))$$

As y can only have the values of 1 and 0, if we put y at the equation above we would see that we get the right cost function we defined earlier. That gives us a 1 line function we can work with and try to minimize and run gradient descent on, the complete cost function is:

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

Minimizing the Cost Function

We are going to use the regular gradient descent algorithm:

```
Repeat until convergence {
    $\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1 \ldots \theta_n)$  Simultaneously for (j = 0 … n)
} simultaneously update all $\theta_j$
```

After we make the derivative what we get is exactly like the function for the linear regression:

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

The implementation of this algo can be implemented by either "for" loops, but it is much easier to implement it using vectorization.
Also note that feature scaling is relevant when implementing the gradient descent algorithm, just like the case of linear regression.

Gradient Descent Alternative

There are several different algorithms that can be used to minimize the cost function, those algorithm are more sophisticated and their main advantages are that they don't require a learning rate Alpha, and they are often faster than gradient descent.
Such algorithms are: **Conjugate Gradient, BFGS, L-BFGS**.

The implementation of those algos is quite complex, in Octave we can use the **fminunc** function as follow:
First we define a method that returns the cost function value and the partial derivative term per theta

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

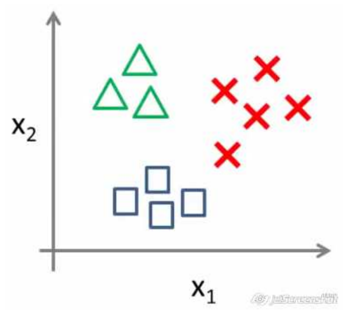
Note that jval is a scalar where gradient is a vector. Then call the fminunc function:

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

Multiclass Classification

Up until now we had only 2 classes to classify and we could used a Boolean true/false as the dependent variable (y was either 1 or 0).
A Multiclass classification problem is where we have many classes in the training set, and we can't use a Boolean as the dependent variable, for example we want to classify the weather prediction as Sunny/Cloudy/Rain/Snow etc.

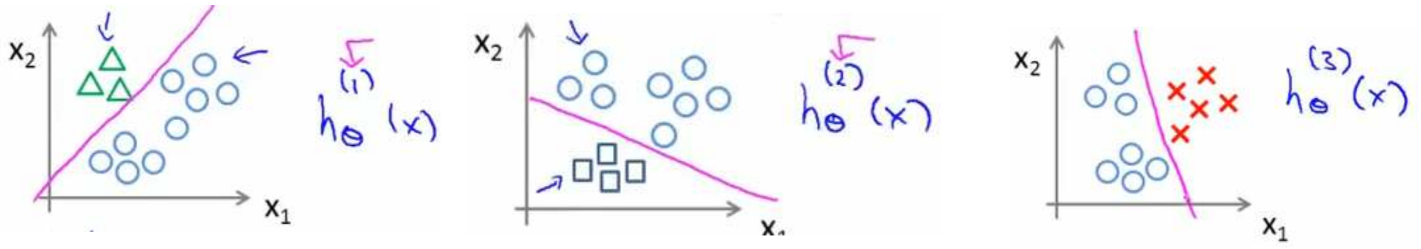
Here is how a multiclass classification problem looks like (with 2 features):



One-vs-All

In order to solve a multi class classification problem we use the one-vs-all method, in this method what we turn the problem into a binary problem, meaning with only 2 classes. In order to do that we leave one class as is, and denote it as the "positive" class (meaning y=1), and turn all the other classes into the "negative" class (meaning y=0).

This model will be denoted as: $h_\theta^{(1)}(x)$.
Subsequently, when we learn how to classify the second class, we turn the second class into the "positive" class and all other classes into the "negative" class, this model will be denoted as: $h_\theta^{(2)}(x)$, and so on.
In general, the models we have are: $h_\theta^{(i)} = P(y = i|x;\theta)\ where\ i = (1,2,3)$



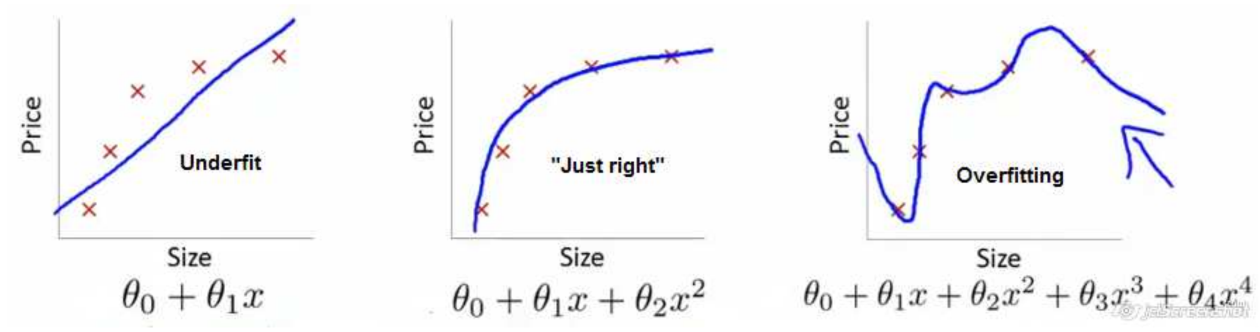
Finally, in order to make a prediction we run all classifier and we choose the class that had the highest probability $result = \max\,[h_\theta^{(i)}(x)]$

Regularization

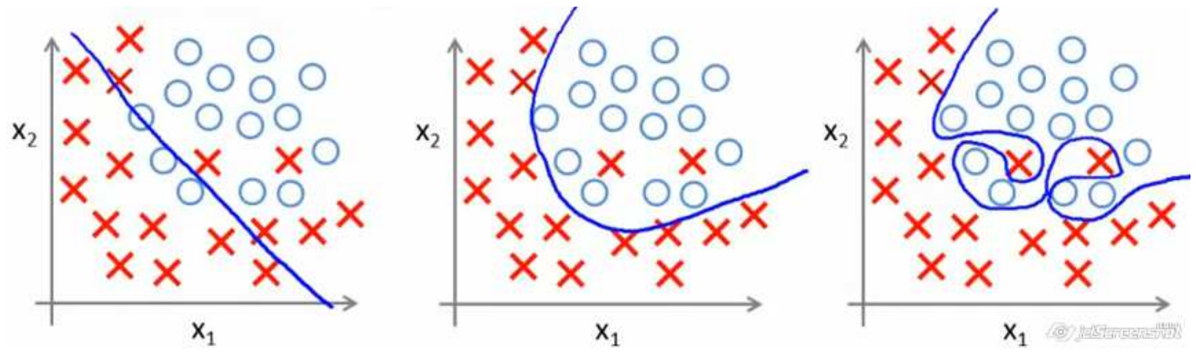
The problem of Overfitting

Overfitting: If we have too many features, the learned hypothesis may fit the training set very well (so the cost function J would be almost 0), but fail to generalize to new examples (prediction on new examples would be off).

Here we have example of 3 models, we can see that the first model "underfit", it assumes a linear relation between the Size and Price of a house which is wrong.
The second model fits the data better and assumed that as the house gets bigger in size the effect on price is reducing.
The third model has a perfect fit for the training set, but note how new data might get a very bad predictions, this is Overfitting.



Same story for the Logistic Regression, we can have a model that tries "Really hard" to fit the data and becomes overfitted, which good on the training set but on new data it will fail.



What to do if we recognize Overfitting?

1) Reduce the number of features. Either by manually selecting which features to keep, or use some model selection algorithm (later in the course).
   The problem with this method is that we throw away features that actually do help us predict y correctly.
2) Regularization – Keep all the features, but reduce the magnitude/values of parameters θ_j.

Regularization Cost Function (how it works)

In regularization we keep all the features but we reduce the magnitude of their parameters theta. What it actually mean?

Say our model is: $h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2^2 + \theta_3 x_3^3 + \theta_4 x_4^4$

And we decide we want to reduce the magnitude of Theta3 and Theta4, what we do is we change our cost function so we add a “penalty” for theta3 and theta4, as a result when we come to minimize the cost function it will take the penalty into consideration and will choose small values for theta3 and theta4.

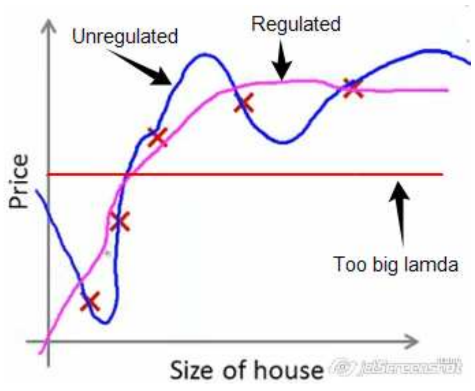
$$\min_\theta \frac{1}{2m}\underbrace{\sum_{i=1}^{m}(h_\theta(x^{(i)})-y^{(i)})^2}_{\text{The regular cost function}} \underbrace{+ 1000\,\theta_3^2 + 1000\,\theta_4^2}_{\text{Penalty for using Theta3 and Theta4}}$$

Since we have many features, and we don’t know what features we want to penalize, we will penalize all the features and will let the cost function minimization problem to select that.
This penalization results in “simpler” hypothesis which is less prone to Overfitting.
Our new cost function minimization problem is:

$$J(\theta_0,\theta_1) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda\sum_{i=1}^{n}\theta_j^2\right]$$

Where λ is called the Regularization Parameter.
Note though that we are NOT penalizing Theta0 (this is the convention).

So if our unregulated model was overfitted (blue line), the same model, when regulated can become much simpler and not overfitted.



If our Regularization Parameter (lamda) is too big we would end up minimizing all parameters, and our model will turn into a very simple model: **h(x) = Theta0** which will be an underfit (red line).

We will learn some methods for automatically choosing the right value for the regularization parameter.

Regularized Linear Regression

Our cost function for regularized models is this:

$$J(\theta_0, \theta_1) = \frac{1}{2m}\left[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{i=1}^{n} \theta_j^2\right]$$

We want to minimize it using the gradient descent algo, since we are not regularizing Theta0 we will have a 2 cases in the GD algo, one for theta0 and another for all others thetas:

Repeat until convergence {

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

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

} simultaneously update all θ_j

We can rearrange the equation above and get:

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

If we look at this term: $(\mathbf{1} - \boldsymbol{\alpha}\frac{\boldsymbol{\lambda}}{\boldsymbol{m}})$ we can see that for small Alpha this whole term is <1 and that regulates theta as $\boldsymbol{\theta_j} := \boldsymbol{\theta_j}\left(\mathbf{1} - \boldsymbol{\alpha}\frac{\boldsymbol{\lambda}}{\boldsymbol{m}}\right) - \cdots$ so every iteration we reduce Theta by multiplying it by <1 number.

The Normal Equation

We can arrive at the optimal Thetas by using the normal equation, this is a “one shot” calculation that gives as the solution $\theta = (X^T X)^{-1} X^T y$

We can apply regularization for the normal equation as well, the equation that solve the Thetas is:

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

Where the matrix in the equation is of size **(n+1, n+1)**.

One of the advantages of using the regulated normal equation is that we don’t have the non-invertibility issue if m<n.

Regularized Logistic Regression

For regularized logistic regression we take our original cost function and add the regularizing expression:

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

Now to minimize this function we are going to take the derivative and arrive at:

Repeat until convergence {

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

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

} simultaneously update all θ_j

In Octave we can use the ***fminuc*** method the minimize the cost function of the logistic regression, we do this like we did it for the linear regression (see example few pages back).
The thing to note is that the derivate term (gradient vector) we return is different for Theta0 and the rest:

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

Gradient(2…n+1) = $\left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j$

Neural Networks (Week 4)

Non-Linear Hypothesis

Why do we need neural networks? We saw, for example in classification problems, that it is a good idea to include polynomial terms of the features in the model $\rightarrow x1^2 + x1x2 + x1^2x2 + x2^2 + x2^2x1 + ...$
Using non-linear polynomial terms helps up come up with a better classifying model.

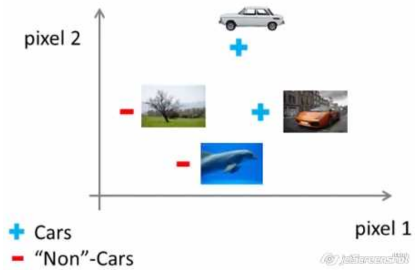
But what if we don't have only 2 features, what if we have, like we usually do, much more features, for example 100 features? If we want to build a non-linear polynomial model using all the features we would end up with lots of combinations. For example if we take just the combination of each feature with the other feature (x1x2 + x1x3 + x1x4 +…+x2x3 + x2x4 + x2x5 + … + x3x4 + x3x5 +…) we would end up with $n^2/2$ features ~ 5000 features. This is a lot of features to compute with a regular logistic regression.

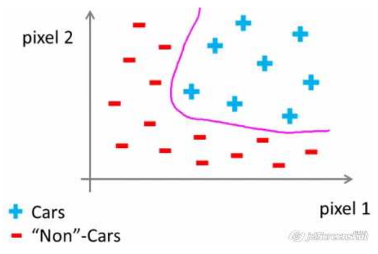
Example

Let's suppose we want to train a model to identify visual objects (like identify if a picture is a car), how can we do it? One way of doing this is taking a lot of pictures of "cars" and a lot of pictures of "not cars", and use the pictures' pixels values (intensity/brightness) as features.
For this example we assume a grayscale picture so each pixel has 1 value (and not RGB values), so if we take 2 locations on the picture (2 pixels) and plot whether it's a car or not car (+ or -), we would get something

pixel 1
pixel 2

like this:





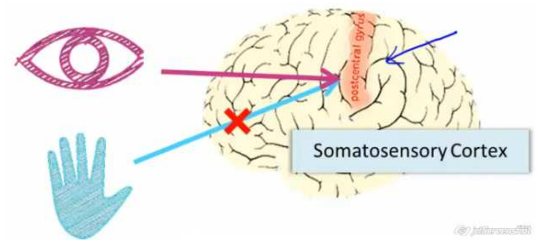
Now, what if our picture is a 50x50 pixels (small pic), we would end up with 2500 features (2500 pixels values), and if we want to have a polynomial model this is $2500^2/2$ features ~ 3 millions.

A regular logistic regression can't handle so much features effectively, this is where Neural Networks help.

Neural Networks Intro

Origin: Algorithm that try to mimic the brain. Was widely used in 80s and 90s and then diminished, but lately it started to become "popular" again, the reason is that Neural Networks are very heavy computation wise, and with the new computers capabilities it become a valid technique again.

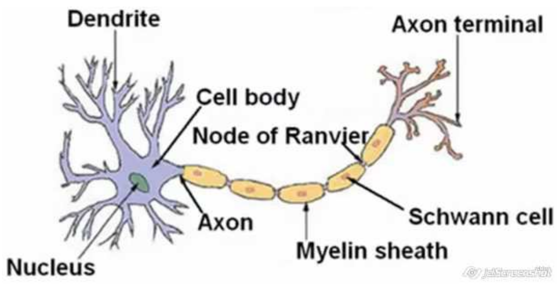
The motivation for Neural Networks was to find "one learning algorithm" that would mimic the brain learning capabilities. Few researches showed that if you re-route the wiring of the eye signals to a different part of the brain, that new part of the brain "learns" to see. Same goes for hearing and other sensors.

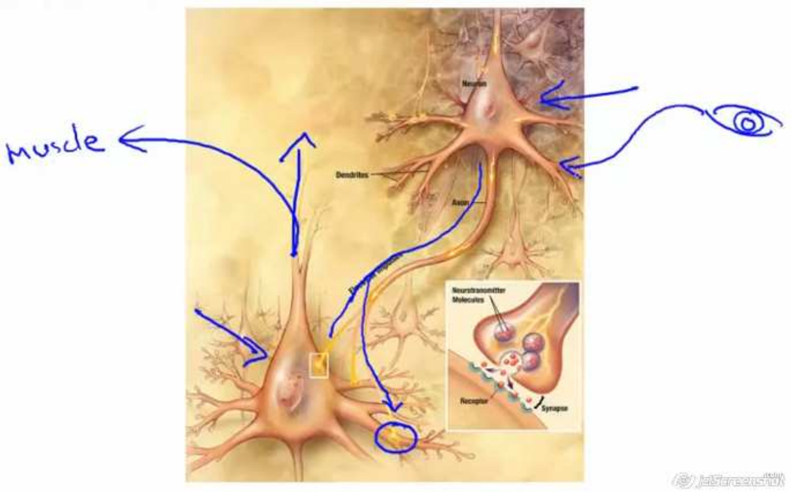


Originally the Somatosensory Cortex is responsible for sensing the "touch", but if we re-route the eye signals to the Somatosensory Cortex it start to learn to see.
And it seems we can attach any sensor to any part of the brain and that brain tissue we learn to use that new sensor.

Here is an example of learning to see with the tongue. A low-res grayscale camera is attached to the forehead of a blind person, and then each pixel is wired to a different electrode attached to the tongue, and each pixel gets a different voltage based on its brightness, it turns out that people using this technique learn to see with their tongue.

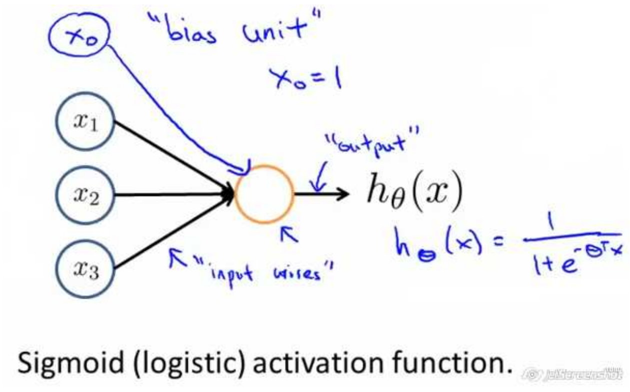
Model Representation





What is a neural network in the brain? Each neuron can be considered as a "processing unit" (Nucleus), where it has several inputs (Dendrite), and an output (Axon).
Then a neural network is a lot of neurons connected together communicating with electric pulses.

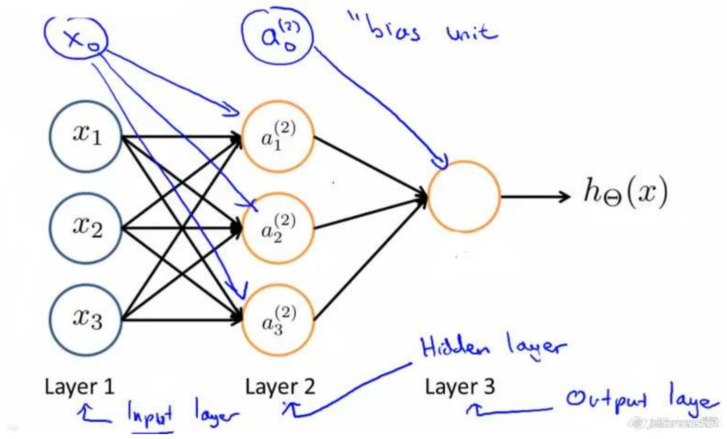
A neural network model is built from many Neurons, where each neuron is some learning model.



x_0
"bias unit"
$x_0 = 1$
x_1
x_2
x_3
"output"
$h_\theta(x)$
$h_\theta(x) = \frac{1}{1+e^{-\theta^T x}}$
"input wires"
Sigmoid (logistic) activation function.

The "neuron" (called "activation unit") has features as input, and the output is the model h(x).
The "neuron" to the left is the regular logistic regression model, with X as features and Theta as model parameters, in neural networks the Thetas can also be called as "weights" instead of "parameters".
Also x(0) is sometimes omitted from the representation but it is there (and equal to 1).

Neural Network Model



A Neural Network model is a network of many logistic units organized in layers, where each layer output is the next layer input. The very first layer input (Input Layer) is the model features x1,x2,x3, the last layer output (Output Layer) is the model h(x), and the layers in between are called Hidden Layers.

Notations:

$a_i^{(j)}$ – "Activation" on unit I in layer j.

$\Theta^{(j)}$ – Matrix of weights controlling function mapping from layer j to layer j+1 (Basically Theta(1) is the parameters of the Layer 2 etc).

So how each activation unit model looks like? (g is the Sigmoid function).

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

Note that Theta is a matrix of parameters, and since the number of features in each layer is coming from the previous layer, this Theta vector has the number of rows according to the current layer "Activation" units, and the number of columns according to the previous layer "Activation" units (plus the 0 feature x_0):

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

Note: In the discussion above, the input for the network, x(1)…x(n), is only 1 training row, when X is a matrix of many training sets you have to run the network for each training row (or use vectorization).

Vectorized Implementation (Forward Propagation)

It is easy to compute each layer's activation values using vectorization (instead of loops). Consider the example above, let's compute layer 2 values:

$$a_1^{(2)} = g\left(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3\right)$$
$$a_2^{(2)} = g\left(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3\right)$$
$$a_3^{(2)} = g\left(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3\right)$$

$$g\left(\begin{bmatrix}\Theta_{10}^{(1)} & \Theta_{11}^{(1)} & \Theta_{12}^{(1)} & \Theta_{13}^{(1)}\\ \Theta_{20}^{(1)} & \Theta_{21}^{(1)} & \Theta_{22}^{(1)} & \Theta_{23}^{(1)}\\ \Theta_{30}^{(1)} & \Theta_{31}^{(1)} & \Theta_{32}^{(1)} & \Theta_{33}^{(1)}\end{bmatrix} \times \begin{bmatrix}x_0\\x_1\\x_2\\x_3\end{bmatrix}\right) = g\left(\begin{bmatrix}\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3\\ \Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3\\ \Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3\end{bmatrix}\right) = \begin{bmatrix}a_1^{(2)}\\a_2^{(2)}\\a_3^{(2)}\end{bmatrix}$$

And then in order to compute h(x) we do the same, but now instead of X we use the second layer activation values $[a^{(2)}]$ and we need to add the intercept $a_0^{(2)} = 1$

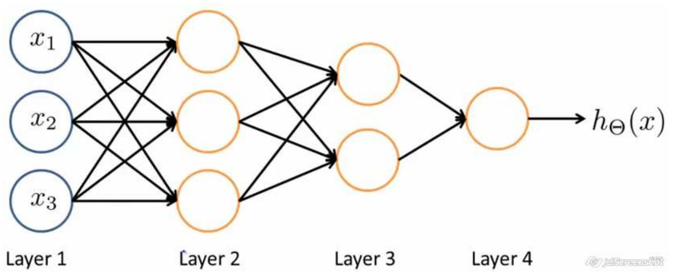
$$g\left(\begin{bmatrix}\Theta_{10}^{(2)} & \Theta_{11}^{(2)} & \Theta_{12}^{(2)} & \Theta_{13}^{(2)}\end{bmatrix} \times \begin{bmatrix}a_0^{(2)}\\a_1^{(2)}\\a_2^{(2)}\\a_3^{(2)}\end{bmatrix}\right) = g\left(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}\right) = h_\Theta(x)$$

This was the implementation for a single training row, but if X is a matrix and not a vector, the features now are in the rows and not the columns: $\begin{bmatrix}\boldsymbol{x_0^1} & \boldsymbol{x_1^1} & \boldsymbol{x_2^1}\\ x_0^2 & x_1^2 & x_2^2\\ x_0^3 & x_1^3 & x_3^3\end{bmatrix}$ so in order to multiply the parameters matrix by the features we have to transpose the input matrix (so the features will be in the columns).

$$A^{(2)} = g\left(\Theta^{(1)} \times X^T\right) = \begin{bmatrix}a_1^{(2)^1} & a_1^{(2)^2} & a_1^{(2)^3}\\ a_2^{(2)^1} & a_2^{(2)^2} & a_2^{(2)^3}\\ a_3^{(2)^1} & a_3^{(2)^2} & a_3^{(2)^3}\end{bmatrix}$$ where $a_3^{(2)^1}$ means the 2nd layer and the 1st training row.

In order to compute h(x) we have to add the "bias" unit a(0), we do this by adding it as a first row in the above matrix, then the result is: $g\left(\Theta^{(2)} \times A^{(2)}\right) = [h(x)^1 \quad h(x)^2 \quad h(x)^3]$ (one result per training row).

What Neural Network is doing?

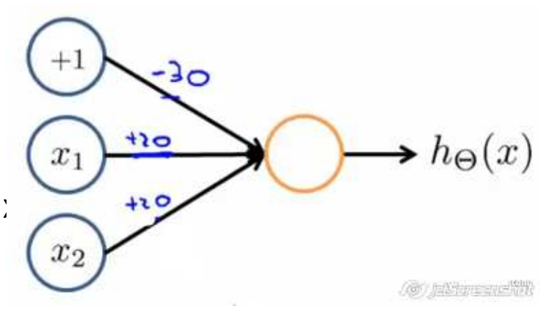


Essentially, neural network gets to learn its own features. In a regular Logistic regression we are constrained to use the raw features x(1), x(2 …x(n), it's true we can use some polynomial combinations of those, but we are still constrained to those basic X features.

In neural network (for example a 3 layers neural network), the raw features (X) is just the input for the second layer, then this 2nd layer is learning those features and finds the Theta$^{(1)}$ parameters, and finally the 3rd layer, which is the output layer, or the prediction layer, is NOT using the raw features (X) but rather it uses the learned features a(1),a(2)…a(n) and fits new parameters Theta$^{(2)}$. So basically the networked has learnt its features!

Binary Logical Operators

A single neural activation layer can be used as binary logical operators when the input features are Boolean (take either 1 or 0). This can be done by applying different weights to the features (different Thetas).



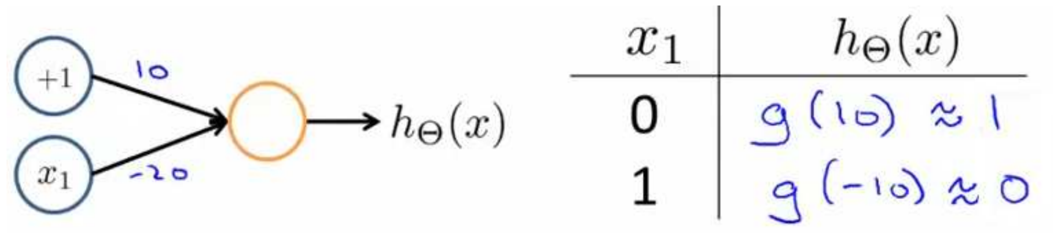
The neuron to the left function is:
h(x) = g(-30 + 20*x(1) + 20*x(2))
Lets compute h(x) values based on all combinations of x1 and x2:

We can see that we got values as if we used the **AND** operator
(x1 && x2)

0	1	g(-10) ≈ 0
1	0	g(-10) ≈ 0
1	1	g(10) ≈ 1

Similarly, by using different weights, like [-10, 20, 20] we can get the **OR** operator (x1 || x2).

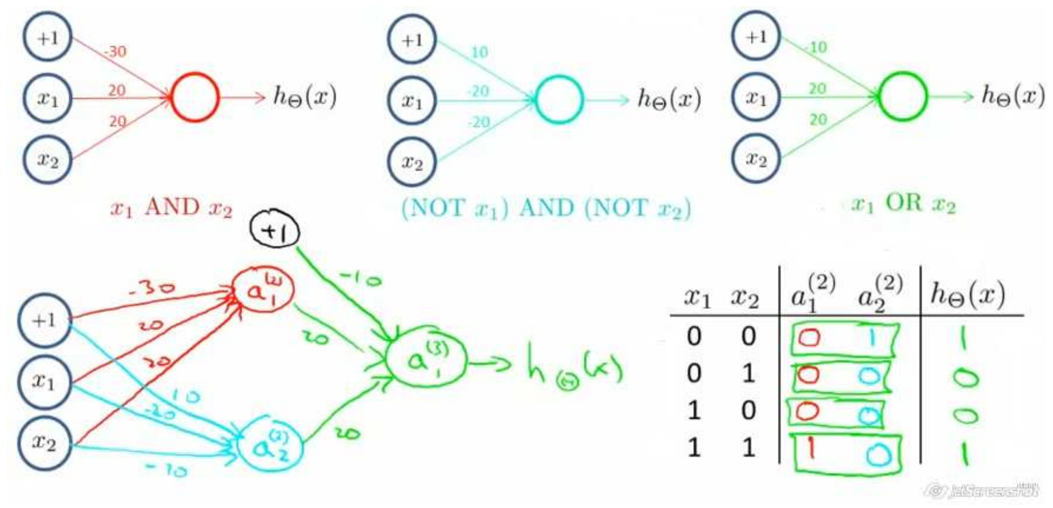
We can also implement the **NOT** operator by using weights like [10, -20]



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

When we want to negate a feature we need to give him a big negative weight.

We can compute more complex binary operators by introducing more layers, for example if we want to compute the XNOR operator (XNOR is true only when both inputs are the same, either both 1 or both 0):
XNOR = (x1 AND x2) OR [(NOT x1) AND (NOT x2)]
And this is an example of how a neural network takes the raw features and build different features for the final output layer to make a prediction.

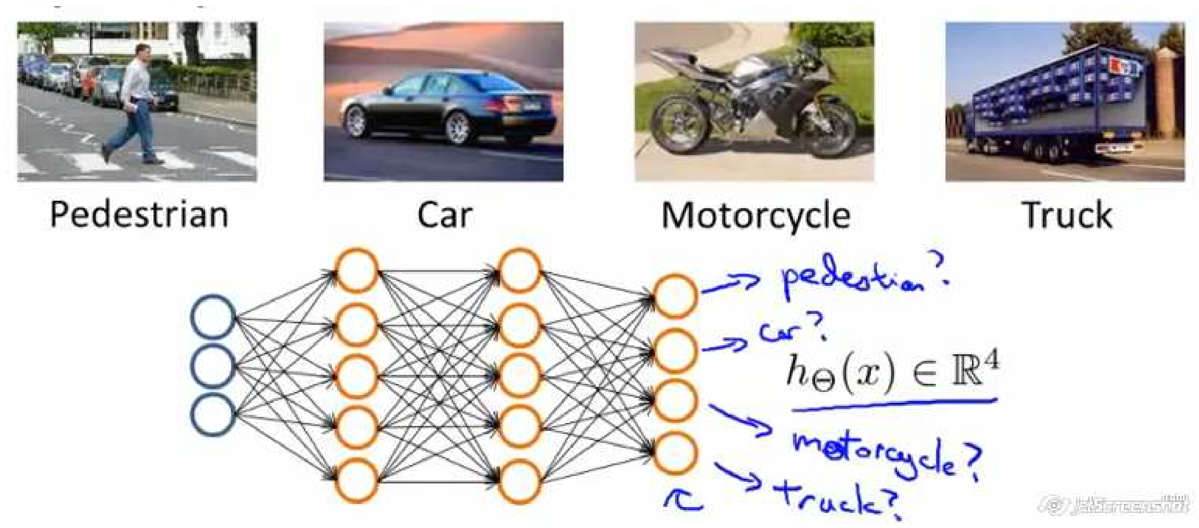


x_1	x_2	$a_1^{(2)}$	$a_2^{(2)}$	$h_\Theta(x)$
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1

Multi-Class Classification

If we have multiple categories we need to learn our network would output several values instead of a single value, where each value represents a different class.

For example, say we want to distinguish between a pedestrian, car, motorcycle and Truck.
We would build a network that would have 4 values in the output layer, where the first value gets 1 or 0 whether it’s a pedestrian or not. The second value if it’s a car or not, etc…



So the output h(x) is either: $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

And as a result, the training set $y^{(i)}$ is a vector of either: $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

Neural Network Cost Function (Week 5)

Notations

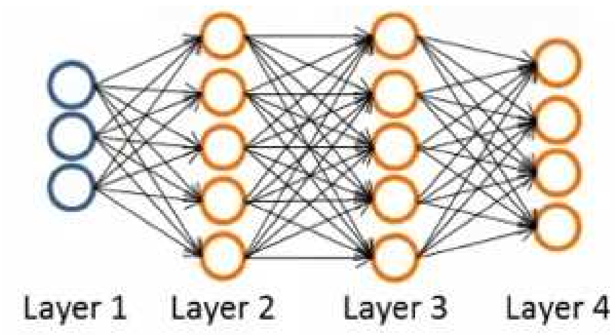
Consider the following networks:

L – Total number of layers in network

s_l - Number of units (excluding bias unit) in layer l

(S_L – is the number of units in the last layer).

K – The number of classes we want to classify (this is also the number of units of S_L.



Cost Function

For logistic regression our cost function is:

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

In logistic regression we have only 1 output variable (scalar) and one dependent variable "y", but on neural network we can have many output variables, our h(x) is a vector with dimension of K, and also our training set dependent variable is a vector, hence our cost function is a bit more complicated. Remember that <u>for each</u> training row x(i) there are K outputs.

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

<u>Explanation:</u>

$y^{(i)} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$ and h(x) tries to predict this, so $h(x^{(i)}) = \begin{bmatrix}1\\0\\0\\0\end{bmatrix}$

The cost function idea is still the same, we want to see "how far" our prediction was, but now for each row we have K predictions (as our y variable is a vector per row), so we basically loop thru all rows and then loop thru all Ks and see how our prediction h(x) was against the actual data "y".

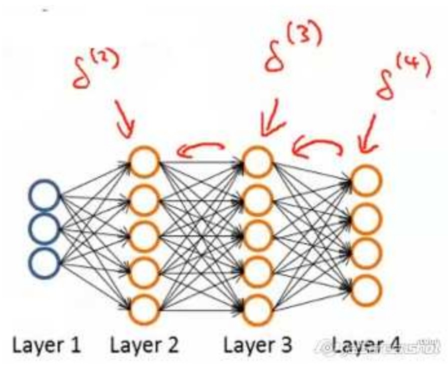
The regularization term is just a sum of the entire Θ (theta) matrix per layer, excluding Θ_0 of the bias unit (remember that for each layer theta is a matrix). The most inner loop (j) runs over the rows (the number of rows is according to the number of activation units in the <u>next</u> layer (sl+1). The second loop (i) is over the columns, the number of columns is according to the number of the current layer activation units (sl).

Backpropagation Algorithm

When we computed the prediction of the NN we used a forward-propagation method that is we started from the first layer and moved forward computing the next layer, until we computed the last layer which is the final h (x).

In order to compute the partial derivative $\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta)$ (this is the derivative of the cost function for each theta) we use the Backpropagation algo, in this method we compute the <u>last</u> layer error, and then the error of the previous layer all the way back till the second layer.

<u>Backpropagation Simple Example</u>



We start with a very simple example where we have only 1 training set ($x^{(1)}$, $y^{(1)}$) and a 4 layer network.
As noted, we start by calculating the error of the last layer, the error is defined as the difference between the activation unit prediction ($a^{(4)}_j$) and the actual value (y_j) (where j is the index of the activation unit, so in the last layer we have 4 units).

$\delta_j^{(4)} = a_j^{(4)} - y_j$

More general, we can describe lamda as a vector:

$\delta^{(4)} = a^{(4)} - y$

Then we use this value to compute the previous layer error:

$\delta^{(3)} = \left(\Theta^{(3)}\right)^T \delta^{(4)} .* \ g'(z^{(3)})$

Where g'z(3) is the derivative of the sigmoid function: $g'\left(z^{(3)}\right) = a^{(3)}.* \left(1 - a^{(3)}\right)$

When we do $\left(\theta^{(3)}\right)^T \delta^{(4)}$ what goes into each unit in $\delta^{(3)}$: $\delta_1^{(3)} = \theta_{\mathbf{11}}^{(3)}\delta_1^{(4)} + \theta_{21}^{(3)}\delta_2^{(4)} + \theta_{\mathbf{31}}^{(3)}\delta_3^{(4)} + \theta_{\mathbf{41}}^{(3)}\delta_4^{(4)}$ which is the sum of errors caused by the first parameter, then it multiplied by the derivative of the first unit value, and we get that $\delta_1^{(3)}$ contains the marginal error caused by this activation unit = its partial derivative.

$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* \ g'(z^{(2)})$

And that's it, we don't have any error for the first layer as this is the input features.

Now that we have all the errors we can compute the partial derivative of the cost function:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = a_j^{(l)} \delta_i^{l+1} \ given\ \lambda = 0$$

Remember that the matrix Theta(1) is the theta matrix used to compute the 2nd layer, so when we derive for Theta(ij) we derive for the i'th row, which is used to compute the i'th activation unit in the next layer, and column j, which is the current layer activation unit, which is also the j'th feature of the next layer, so the indexes means:
l – the current layer
j – The index of the activation unit in the current layer (which will be a feature in the next layer).
i – The index of the error unit in the next layer, this is the error unit that is affected by the i'th row in the theta matrix.

Backpropagation the General Case

In the previous example we had only 1 training row, now let's see how we do Backpropagation when we have many rows.
In the previous step we had to compute the error units for each layer in order to get to the partial derivative, in this case we also have to compute the error units for each layer, but since we have many rows we need to aggregate the error units of the layer across all training sets.
Since Theta(ij) is a matrix of parameters, we save the error "caused" by each theta, so our error term now is a matrix, we will use $\Delta_{\mathrm{ij}}^{(\mathrm{l})}$ to denote this matrix (so $\Delta_{24}^{(3)}$is the error in the 2nd unit of the 3rd layer caused by the 4th parameter. Note that the number of $\Delta_{\mathrm{ij}}^{(\mathrm{l})}$ should match the number of Thetas matrices).

<u>The algo</u>

```
For i=1 to m
{
    Set $a^{(1)} = x^{(i)}$ // Load the i'th row features into the first layer.
    Perform forward propagation to compute $a^{(l)}$ for l = 1,2,3…L // Compute all activation layers vectors.
    Using $y^{(i)}$ compute $\delta^{(L)} = a^{(L)} - y^{(i)}$ // The last layer error vector.
    Backpropagate to compute all previous layers errors: $\delta^{(L-1)}, \delta^{(L-2)}, \ldots \delta^{(2)}$
    $\Delta_{\mathrm{ij}}^{(\mathrm{l})} := \Delta_{\mathrm{ij}}^{(\mathrm{l})} + a_j^{(l)} \delta_i^{l+1}$ // This is adding the current row's errors (should be done for all j's).
                        Note that i is not the training row index but the row index in the Theta
                        matrix. This can be Vectorized for all j's:
                        $\Delta^{(\mathrm{l})} := \Delta^{(\mathrm{l})} + \delta^{(\mathrm{l}+1)}\left(\mathrm{a}^{(\mathrm{l})}\right)^{\mathrm{T}}$
}
```

Now that we have summed the errors across all training rows we can get to the partial derivative:

$$D_{ij}^{(l)} = \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = \frac{1}{m} \Delta_{\mathrm{ij}}^{(\mathrm{l})} + \lambda \Theta_{\mathrm{ij}}^{(\mathrm{l})} \quad \text{if } \mathrm{j} \neq 0$$

$$D_{ij}^{(l)} = \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = \frac{1}{m} \Delta_{\mathrm{ij}}^{(\mathrm{l})} \quad \text{if } \mathrm{j} = 0$$

So to sum up, we start from the 1st training set, doing forward propagation using x, then using y we do Backpropagation, then we add all the error units (for each theta) into the matrix that collects them.
Now do the same with the 2nd training set, and so on until the m'th training set.

<u>Implementing in Octave</u>

Our parameters we want to optimize is a matrix, and the optimizations algos (like fminunc) expects to get the thetas as vectors, so what we do is taking the matrix and turn it into a vector:
Say we have 3 matrices, Theta1 and Theta2 is 10x11, and Theta3 is 1x11
To convert all 3 into a vector: `thetaVec = [Theta1(:) ; Theta2(:) ; Theta3(:)]`

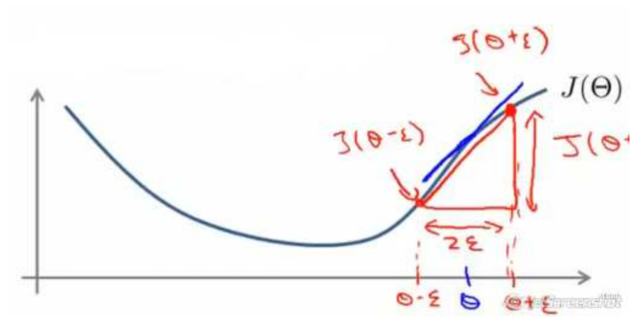
And to roll back into matrices:

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

Gradient Checking

When implementing gradient descent on a relatively complex model (like NN), there might be some subtle bugs that are not appear on the surface, meaning you'll see that the cost function is decreasing as expected, but the final result would not be the optimal because of this bug.

In order to solve this we use some a method called **Numerical Gradient Checking**.
The idea is to check that what we calculated as the derivative is really the derivative, in order to check this we need to estimate the gradient.
Estimating gradient is done by taking the slope of a line connecting 2 very close points on the cost function:



So we take some Theta, and compute the cost for Theta+Epsilon, and Theta-Epsilon, and see what the slope of this line, this is a close approximation of the derivative of the cost function at Theta.

Usually we take epsilon as 0.001.

In Octave the implementation is:

```
gradApprox = (J(theta + eps) – J(theta - eps)) / (2*eps)
```

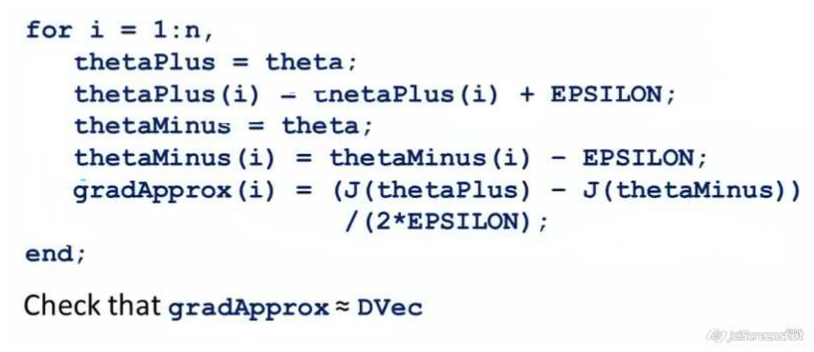
Partial Derivative Checking

What if Theta is a vector, like we really have on our cost function, and we need to check the partial derivative?
A partial derivative is just the change of the cost function when only 1 parameter is changed, so in this example below we do the partial derivative in respect the Theta1.

$$\frac{\partial}{\partial\theta_1} = \frac{J(\boldsymbol{\theta_1} + \boldsymbol{\varepsilon}, \theta_2, \theta_3 \dots \theta_n) - J(\boldsymbol{\theta_1} - \boldsymbol{\varepsilon}, \theta_2, \theta_3 \dots \theta_n)}{2\epsilon}$$

Verifying Backpropagation Algo

The final stage is to verify that our Backpropagation calculated the partial derivatives correctly.
The Backpropagation algo product is a matrix of partial derivatives: $D_{ij}^{(l)}$
We then take this matrix and unroll it into a vector, call id "dvec".
Then we take the Theta matrix and unroll it into a vector as well, we create another vector with the same size, call it "gradApprox", calculate the approximation of the partial derivative for each theta, put it in "gradApproc". Finally compare gradpApprox with dvec, they should be close in value.

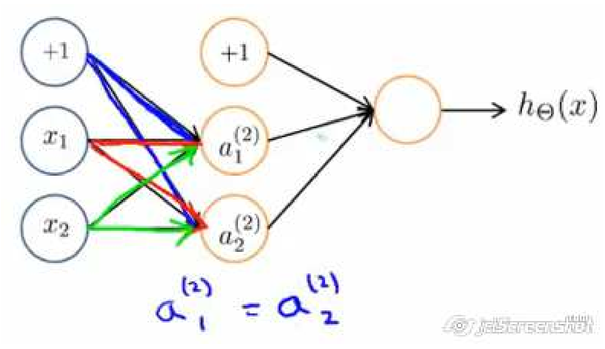


```
for i = 1:n,
    thetaPlus = theta;
    thetaPlus(i) = thetaPlus(i) + EPSILON;
    thetaMinus = theta;
    thetaMinus(i) = thetaMinus(i) - EPSILON;
    gradApprox(i) = (J(thetaPlus) - J(thetaMinus))
                    /(2*EPSILON);
end;
```

Check that `gradApprox` ≈ `DVec`

Random Initialization

Any optimization algo requires some initial thetas to start from.
Up until now we initialized all the parameters to 0, this worked ok on Logistic Regression, but for NN it is not good. When we use all zeros as the initial theta, it means that all the activation units on the second layer would have the SAME value (each unit gets the same features from the previous layer, and if the parameters are the same as well, so the sigmoid value of the units will be the same).



It is possible to prove that if you start with thetas that are all the same on the first iteration, on the following iteration the thetas would still be equal to each other (different value then the initial theta but still equal to each other), which means that again all activation units would have the same values, and when the activation units have always the same values they are redundant, so no real value in the NN.

Symmetry Breaking

In order to break that symmetry of the parameters we initialize the parameters randomly, we do this by setting the parameters to be between –epsilon and +epsilon.
For example in order to create a 10x11 Theta matrix we do:
Theta1 = rand(10, 11) * (2*eps) – eps

Putting it Together

We will go over all the steps needed to get the job done when working with NN.

Network Architecture

First thing to do is to choose network architecture, how many layers and how many units in each layer.
The number of units in the first layer is the number of feature we have (x).
The number of units in the last layer is the number of classes we have (K).
In between we need to choose the hidden layers, a good default is to start with 1 hidden layer, or if we have >1 hidden layer it is advisable to have the same number of units in each hidden layer.
Also it is advisable that the number of units in the hidden layer would be greater than the number of features.

Training a Neural Network

1. Randomly initialize thetas.
2. Implement Forward Propagation to compute h(x) for any $x^{(i)}$.
3. Implement code to compute the cost function J.
4. Implement Back Propagation to compute the partial derivatives (Algo learnt earlier).
5. Verify the partial derivatives using Numerical Gradient Checking.
6. Use gradient descent or any optimization method to minimize the cost function.
   Note that J(Θ) is non-convex, so optimization can get stuck on local minimum, but it turns out to be a non-issue most of the time.

Advice for Applying Machine Learning (Week 6)

Deciding What to do Next

Suppose we run a Linear Regression trying to predict the home prices, and when we try to apply this on un-seen data we observe that our model has big errors, what should we do next?

1) Get more training examples – Sometimes this is good idea, but not always it will improve the result of the forecasting, and there might be cheaper ways to try first.
2) Try smaller sets of features.
3) Try getting additional features.
4) Try adding polynomial features (x1^2, x2^2, x1x2, etc).
5) Try decreasing the regularization rate Lambda.
6) Try increasing the regularization rate Lambda.

In many cases people just pick randomly/intuitively items from the above list trying to increase the performance of their system, spending a lot of time just to find out at the end it was not that.

Instead one should use some Machine Learning diagnostic, which is a test that you can run to gain insight on is or isn't working with a learning algo, and gain guidance as to how best improve its performance.

Evaluating a Hypothesis

One of the problems of a learning algo when it comes to prediction is "Overfitting", when an algo is overfitted then it fails to generalize on new data.

<u>Checking for Overfitting</u>

What we can do for testing for Overfitting is split the data between training set and test set.
The training set can be taken by using 70% of the data, and testing set by the remainder 30%.
It is important that the training set and test set would both consist a mix of all data, so it is advisable to shuffle the data before (like on the OCR data we need to shuffle it so each set would contain all digits).

<u>Evaluating the Test Set</u>

After the model learned its parameter using the training set, use those parameters and run the model on the Test Set, then there are 2 ways to evaluate the errors:

1) Compute the cost function J using the <u>test set</u> data.
2) $err(h_\theta(x), y) = \begin{cases} 1 \; if \; h(x) \geq 0.5 \; and \; y = 0, or \; if \; h(x) < 0.5 \; and \; y = 1 \\ 0 \; Otherwise \end{cases}$

$$Test\ Error = \frac{1}{m_{test}} \sum_{i=1}^{m} err\left(h_\theta\left(x^i\right), y^i\right)$$

Model Selection (Validation Set)

Assume we want to select a model for our hypothesis, I.e selecting the polynomial degree of our model.
So for example we consider 10 polynomial models:
1) $h(x) = \theta_0 + \theta_1 x$
2) $h(x) = \theta_0 + \theta_1 x + \theta_2 x^2$
3) $h(x) = \theta_0 + \theta_1 x + \cdots + \theta_3 x^3$
...
We define "d" as the degree of polynomial degree.

One possibility for choosing the "best" model is:

1) Train all 10 models.
2) Compute the cost value J using the test set.
3) Choose the model with the lowest value of J (again, this is on the test set).
4) Report the generalization error (value of J of the test set) of the selected model above.

Does the reported error of the test set is a good generalization error to report? Probably **NO**, the reason is that our extra parameter "d" (degree of polynomial) was fit to the test set, hence it is an optimistic estimate to report.

Cross-Validation Set

Instead of reporting the error of the test set, what we do is split the entire data we have into **3** sets:
1) 60% of data into the training set.
2) 20% of the data into the cross-validation set.
3) 20% of data into the test set.

Choosing the mode:

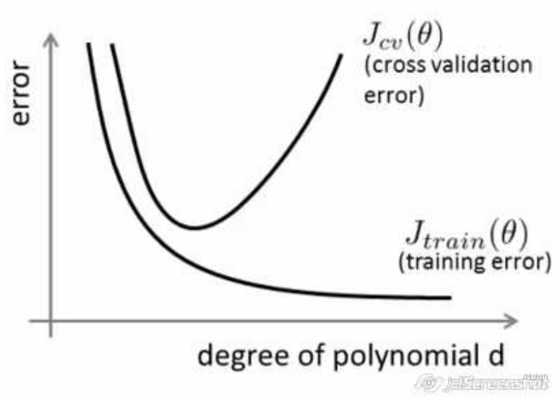
1) Train all 10 models using the training set.
2) Compute the cost value (J) for all models using the cross-validation set.
3) Choose the model with the lowest cost computed above (on the cross-validation set).
4) Report the Generalization Error as J(test), meaning compute the cost of the test set using the model selected in step 3.

The process above makes sure that the generalization error we report did not use any parameter that was fitted to test set.

Diagnosis Bias vs. Variance Problems

Those two problems of High Bias and High Variance are basically Underfit and Overfit.

What we can do to visualize this is to plot the cost function error vs the degree of polynomial, we do this for both the training set and cross-validation set.



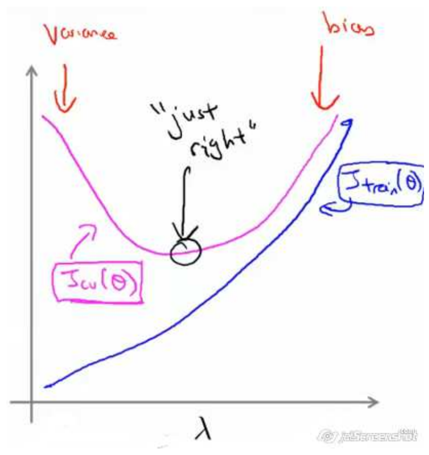
On the training set, when d is low, we have underfit hence a high error, and as we grow in d we get a lower error.

On the Cross-Validation, if d is low we are going to get a high error, then as we grow in d we get lower error that's until we arrive to values of d where we over-fit the training set and this increase the error of the cross-validation data.

If we have a high error of cross-validation, how do we know if this is high bias or high variance?
We can see from the chart is that on a lower values of d (underfit/high bias) we have higher error on BOTH the training set and cross validation set, and on higher values of d (over-fit/high variance) we have higher error only on the cross-validation data, so:
Bias (underfit): $J_{training} \approx J_{CV}$
Variance (over-fit): $J_{CV} \gg J_{training}$

Regularization Bias vs. Variance

When we train our model we use some regularization in order to prevent over-fitting. But what if we used too much regularization? or too little? How can we choose the right value of lambda (regularization term)?
Basically what we do is like we did when we wanted to choose the model polynomial degree.
We choose a range of lambda that we want to test, usually from 0-10 in multiples of 2 (0, 0.01, 0.02, 0.04, 0.08…10), this gives us 12 models to check.
We also divide our data into 3 sets: Training, Cross-Validation and Test.
Train all 12 models using the different values of lambda.
1) For each model (i.e. using the selected thetas), compute the cost (J) on the cross-validation set.
2) Choose the model which has the lowest cost on the cross-validation set.
3) Run the model using the test set data and report the cost value as the generalization error.

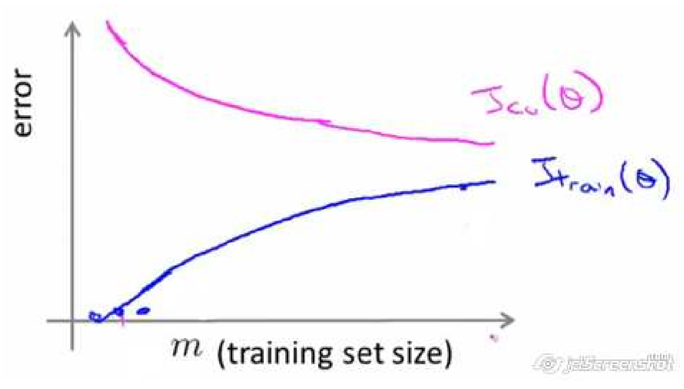


The chart to the left shows the cost-value (J) of the training set and cross-validation set over different values of lambda. Note that the cost value here is only the sum square of error divided by 2*m (i.e without counting lamda).
We can see that when lambda is small (over-fit), we get low error on the training set but high error on the cross-validation set. As we grow in lambda the error of the training set is rising and the error of the CV set is declining, until lamda become too large and we start to under-fit where the error of the CV starts to rise.

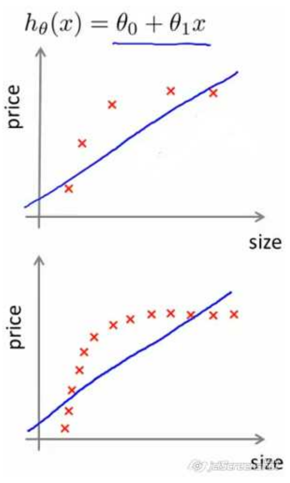
Learning Curves

Learning curve is a good sanity check for a learning algorithm.

The learning curve is a plot the training set error, and the cross validation error, as a function of "m" (the number of training examples). So if we have 100 data rows, we start of by learning on just 1 row, and then 2 rows, and then 3 rows etc. The idea is that for low number of training rows our training model will have a perfect fit (as it is easier to fit a small sample), but our CV error would be large as the trained model would fail on generalizing for other data.

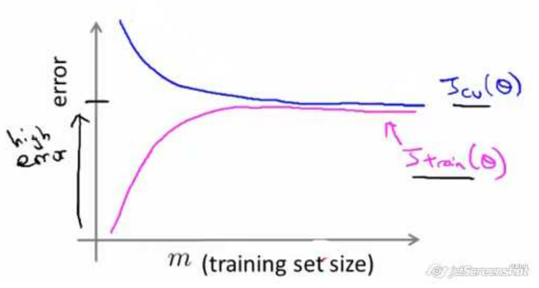


Identifying High Bias (under-fit)



For this example we try to fit a straight line to our data, so in the 2 charts we can see that it doesn't really matter how much data we use the straight line is more or less the same, and is not fitting well.

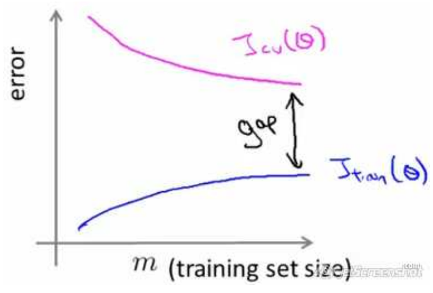
The result would be a learning curve that is leveling on a high error value (meaning that adding data doesn't improve the results of neither the training set nor CV set).



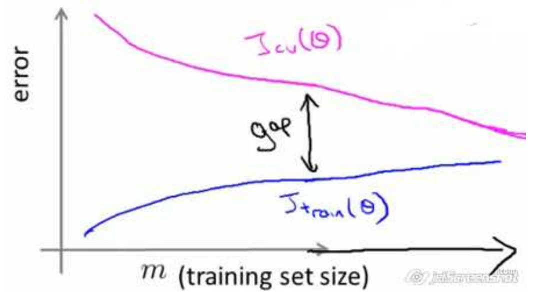
In the case of High Bias adding more training data will not (by itself) help much.
So there is no point on wasting resources of getting more data.

Identifying High Variance (Over-Fit)

Assume we are using some very high degree polynomial model with little regularization; in this case we are likely to over fit our data. So, for the training set, when "m" is low we obviously fit the data perfectly, and as we grow in m we still do a good job. As for the CV error, it starts as high error but goes down as we add more data, at the end we have a big gap between the training-set error and the CV error, but as opposed to the high bias case, here the curves do not level out, so it seems that if we add more data the error of the CV would keep going down. **In the case of High Variance, where the CV error is much larger than the training error, adding more data is likely to help.**



Adding data →

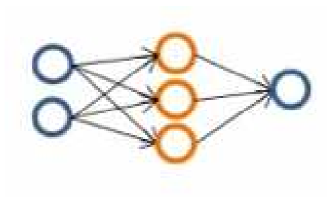


Deciding What to do Next

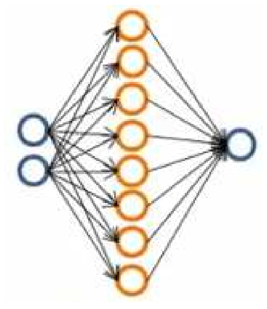
We started off this module by asking what our options are to improve our algorithm, let revisit the various options and see when we should do what.

1) Get more training examples – This could help fixing high variance.
2) Try smaller sets of features – This could fix high variance as well.
3) Try getting additional features – This could help fix high bias.
4) Try adding polynomial features (x1^2, x2^2, x1x2, etc) – This too help fix high bias.
5) Try decreasing Lambda – Too high lambda means we under-fit, so this help fix high bias.
6) Try increasing Lambda – Low lambda means we over-fit, so this helps fix high variance.

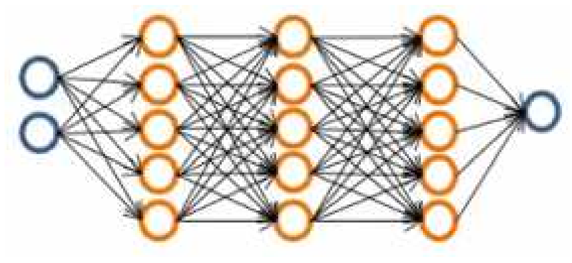
Bias and Variance on Neural Networks



Using a small NN would be prone to high bias and under-fitting as we have little parameters.
Those networks are also computationally cheaper.



Using a "large" NN with more parameters are more prone to over-fitting, in order to address this issue increase the regularization term.
Larger networks usually do better job than smaller networks, so it is better to have a large network with regularization than a small network.
Large networks are also more computationally expensive.



Another decision to make is how many hidden layers to use, starting with a single hidden layer is a good default, but in order to better choose it we should have a test. Split the data into training set, cross-validation and test set, and run different networks, for each network compute the error using the CV data, and choose the network that had the lowest CV error.

Machine Learning System Design

Prioritizing What to Work On

The example we are going to use is an email spam classifier algorithm.

The first decision is to choose how to represent the feature vector x.
We can choose a list of 100 words and define a feature vector with binary values whether a word appears or not. So our feature vector will be of size [100, 1].
In reality we wouldn't want to manually choose 100 words but rather choose the n most frequent words, this could be 10,000-50,000 words.

How should you spend your time build this classifier?
Out options are:

- Collecting more data so we would have more samples of spam vs not spam
- Develop sophisticated features based on email routing etc.
- Develop sophisticated features for message body, like should "discount" and "discounts" be treated as the same word, or "deal" and "dealer" etc.
- Develop sophisticated algo to detect misspellings like: m0rtgage, med1cine which are used by spammers to avoid recognizing them.

Among all these options it is hard to tell where you should spend your time, but it is important that your team would brain storm and think about the different options and not just go with "gut feeling".

Error Analysis

Error analysis should provide a systematic way of choosing what to work on.
The recommended approach for building a learning algorithm:

- Start with a simple algorithm that you can implement quickly. Implement it and test it on your cross-validation data.
- Plot learning curves to decide if more data, more features, etc are likely to help.
  When starting a new project you can't tell in advance what you need, so by even implementing a quick a dirty model we get some evidence on what we need next and not using gut feeling.
- Error analysis: Manually examine the examples (in cross validation set) that your algorithm made errors on. See if you spot any systematic trend in what type of examples it is making errors on.

In our spam classifier we can take all the misclassification from our cross-validation set and:

1) Group them by type, e.g is it a Pharam spam, Replica, Password stealing etc. Then we can see on what group our classifier had the most errors and concentrate on that.
2) Look at those misclassified emails and try to see what could have helped the classifier do a better job, i.e. what features are missing. Also count the number of times each feature could have helped. For example deliberate misspelling appeared X times, unusual email routing Y times and so on, then concentrate on the feature that appeared the most time and develop it.

Error analysis is the process of <u>manually</u> examining the classifier errors and trying to come up with better solution, we try to characterize the hard examples for a classifier to classify.

Numerical Evaluation

Error analysis is not always a good solution to check what path we should take. Sometimes we need to try different versions of models and compare them, that's why it is important to have a single number that will be used to compare the different models, this can be the cross-validation error.

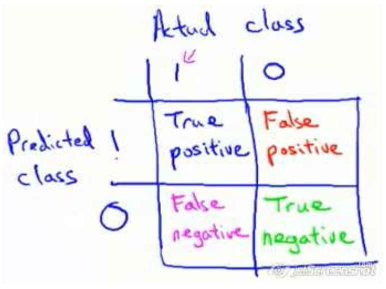
For example, Should discount/discounts/discounted/discounting be treated as the same word? We can use stemming software and we use the stem of the word instead of the word itself.
Error analysis may not be helpful for deciding if this is likely to improve performance. Only solution is to try it and see if it works. So we need a numerical evaluation (e.g., cross validation error) of algorithm's performance with and without stemming.
Also note that we are doing it against our cross-validation set and not our test set, the reason is If we develop new features by examining the test set, then we may end up choosing features that work well specifically for the test set, so J(test) is no longer a good estimate of how well we generalize to new examples.

Error Metrics for Skewed Classes

Skewed Classes is the case when we have a lot of examples from one class and almost no samples from the other class.
For example, let's say we want to predict cancer and our algorithm gets a 1% error on the test set, that's seems impressive. But now assume that our "positive" class is only 0.5% of all cases, so 99.5% of the sample is healthy people and only 0.5% has cancer.
In this case we could build a stupid non-learning "algorithm" that all it is doing is returning "false", meaning predicting y=0. This algorithm would then have a 0.5% error, better than our learning algorithm.
So it is obvious that in the case of Skewed Classes the test-set error is not a good estimate.

Precision / Recall

We divide our predictions into a True/False Positive/Negative matrix.
True Positive – We predicted "1" and the actual is "1".
True Negative – We predicted "0" and the actual is "0".
False Positive – We predicted "1" and actual is "0".
False Negative – We predicted "0" and actual is "1".



Precision – $\frac{True\ Positive}{\#\ predicted\ positive} = \frac{True\ Positive}{True\ Positive + False\ Psotive}$ - Out of all the people we thought had cancer, how many did actually had cancer. A high value of Precision is good as we didn't tell much people that we think they have cancer while they don't.

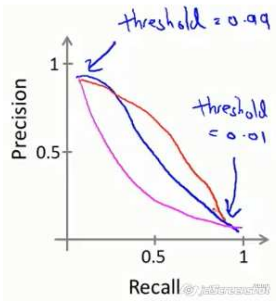
Recall - $\frac{True\ Positive}{\#\ Actual\ positive} = \frac{True\ Positive}{True\ Positive + False\ Negative}$ - Out of all people that do actually have cancer, how much we identified. A high value of recall is good as there are not much people that have cancer and we failed spotting them (or the group of people that do have cancer and we told them they don't is small).

So back to our non-learning algorithm, it predicts y=0 all the time, while it might have a low % error on the test set, it will also have a 0 recall, meaning it is not a good classifier.

Trading Off Precision and Recall

We continue with the cancer example.
Our algorithm outputs a value between 0 and 1, and we use 0.5 as threshold for true/false.
Suppose we want to predict y=1 only if we are very confident, we don't want to tell someone he might have cancer while he doesn't. In this case we want a higher precision value, so instead of using 0.5 as out threshold we can use 0.7 or 0.9, doing this will lower the number of our erroneous predictions.
The trade-off is that we would "miss" much more patients that do have cancer and we didn't spot them.

On the other hand we might decide that it is more important for us to "catch" more patients that have cancer so they would get treatment. In this case we want a higher value of recall, we can achieve this by lowering the threshold, like 0.3.



Our rate of Precision/Recall is a function of our threshold; this function can have many shapes depend on our data.

Is there a way to automatically choose this threshold?
In the beginning we had a single number that we used to compare our models, this was the %error in the test set (or cv set). Now we have 2 values that we can set algorithm on but how do we tell which is better.

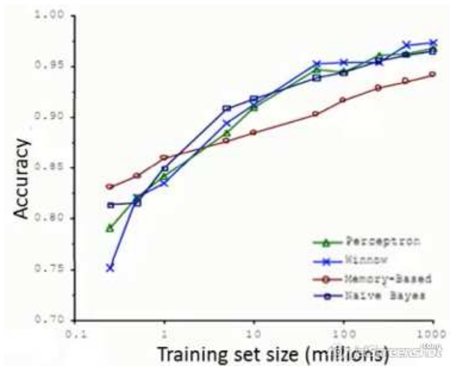
Look at the following Precision/Recall combinations:

	Precision	Recall	Average	F_1 Score
Algorithm 1	0.5	0.4	0.45	0.444
Algorithm 2	0.7	0.1	0.4	0.175
Algorithm 3	0.02	1.0	0.51	0.0392

How can we compare the different cases? One way is to use the average. This is not a good solution, look at the last algorithm, it has 1 recall and 0.02 precision, this algorithm is just a stupid algo that returns "1" for all cases, obviously it is a bad algo but it has the highest average score. More generally, any low value of Precision or Recall signals a "stupid" algorithm.

$F_1 Score = 2\frac{Precision \times Recall}{Precision + Recall}$ The F score combines both Precision and Recall but penalize for low values, in our example algorithm 1 has the highest F score.

Data for Machine Learning



We said earlier that more data is not necessarily needed in order to get a better algorithm. A study done by Bank and Brill, trying to classify confusing words (to/two/too, then/than etc) showed that it didn't really matter what type of classifier algo you used (they used 4 different), but rather the amount of data you had to train it on.
There used to be a saying:
"It's not who has the best algorithm that wins. It's who has the most data".
Next we will try to understand when it is necessary to have more data rather than trying different algorithms.

<u>Large Data Rationale</u>

Assume feature x has sufficient information to predict y accurately.

For example, when trying to predict confusing words, let's assume the feature x captures the surrounding words: "For breakfast I ate ____ eggs." In this case it is sufficient data to conclude that the word "two" is needed. That is a case when x has sufficient data to predict y accurately.

On the other hand, assume you try to predict a house price by using the size only. In this case it's not enough "features" to make a prediction. In this case x does not have sufficient information to predict y.

A useful test for the above assumption is: **Given the input x, can a human expert confidently predict y?** Answering yes to the above question is the first condition to have in order to justify a huge training set.

Suppose the above assumption occur, so now assume we use some algorithm that fits the data very well (like a neural network with many hidden units).
In this case we would have high variance and our J(train) will be small.
Now if we use a very large training set we are unlikely to over-fit and we are more likely to have J(train) and J(test) close to each other, and since J(train) is small we hope that J(test) would be small either.
Another way to look at it, we want algos that have low bias and low variance. The low bias is achieved by using many features (e.g. NN), and the low variance is achieved by using a very large training set.

To sum up: IF the feature vector x is sufficient to predict y, and you can use a low bias algorithm, then a very large training set is useful.

Support Vector Machine (Week 7)

Support Vector Machine (SVM) is a very powerful and popular algorithm.
SVM can sometimes give cleaner solution to complex non-linear problems than logistic regression and NN.

Optimization Objective

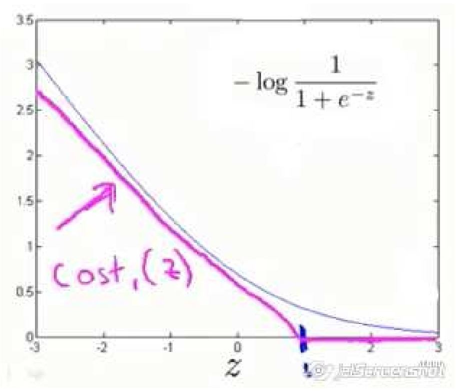
In the Logistic Regression our model was: $h_\theta(x) = \frac{1}{1+e^{-\theta^T X}} = \frac{1}{1+e^{-z}}$

Now if y=1, we want z to be as big as possible, and vice versa when y=0, so our h(x) would be close to y.

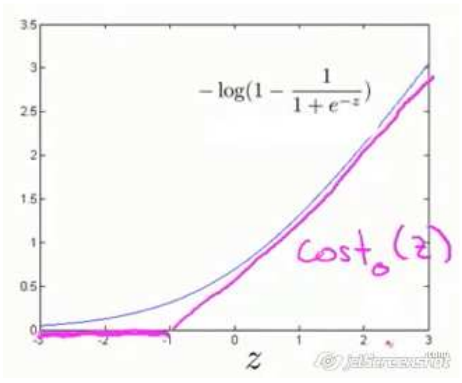
Let's look at the contribution of each training row to the cost function:

$$-y \times log\left(\frac{1}{1+e^{-\theta^T X}}\right) - (1-y) \times log\left(1 - \frac{1}{1+e^{-\theta^T X}}\right)$$

We break the cost function into 2 parts, one when y=1 (this is only the left part), and when y=0 (right part).



← Y=1 y=0 →



In this function, which is the added cost of a specific data row, we can see that as Z is larger (for the y=1 case) the cost is lower. So the logistic regression is trying to give Z a big value when it sees that y=1.
For SVM what we are going to do is consider the cost to be 0 when Z=1, so we are going to simplify the cost function and break it into 2 straight lines, this line will be called cost$_1$(z).
Similarly, when y=0 the cost is smaller as Z is smaller, and we will create a new cost function cost$_0$(z).

If we replace the new cost functions we defined in the Logistic Regression cost function we get:

$$\frac{1}{m}\left[\sum_{i=1}^{m} y^{(i)} cost_1(z) + \left(1-y^{(i)}\right) cost_0(z)\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

For SVM we tweak the cost function a little bit. First 1/m is a constant and does not change the optimization result, hence we get rid of it. Second, the regularization term lambda is controlling the portion of the regularization unit in the whole cost, as lambda is bigger the regularization unit portion in the cost is bigger and vice versa. Instead of controlling the balance between the 2 parts of the cost function with lambda we are going to do this with a constant "C" that is applied to the first term of the cost function, as if we divide the entire cost function by lambda, so C=1/lamda, our minimization problem, **which is convex**:

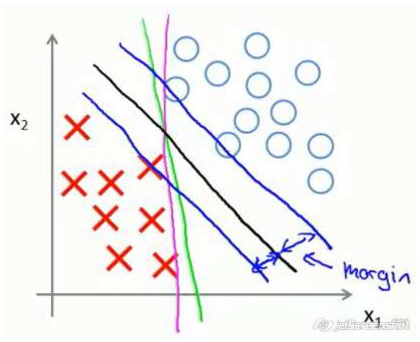
$$min\ C\sum_{i=1}^{m}\left[y^{(i)} cost_1(z) + \left(1-y^{(i)}\right) cost_0(z)\right] + \frac{1}{2}\sum_{j=1}^{n}\theta_j^2$$

Finally, the SVM hypothesis is:

$$h(x) = \begin{cases} 1\ if\ z \geq 0 \\ 0\ if\ z < 0 \end{cases}$$

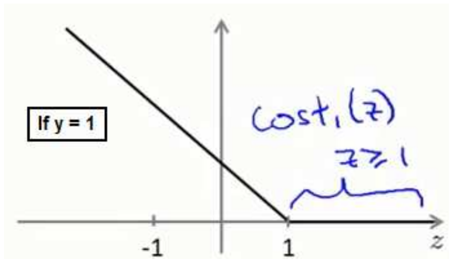
Large Margin Intuition (SVM Decision Boundary)

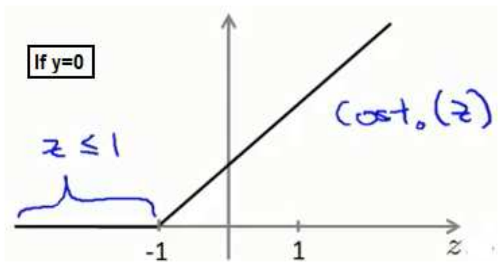
SVM is sometimes refers to as Large Margin Classifier, the reason for that is that SVM tries to find a decision boundary that has the widest distance (margin) from the dataset samples.



The green and magenta lines are decision boundaries that represents some logistic regression, and the black line is the decision boundary of an SVM, note how the black line seems more reasonable when it comes to separate the positive and the negative samples. The blue lines are the SVM decision boundary "margin", or the distance of the decision boundary from the data samples.

Why is it like so?





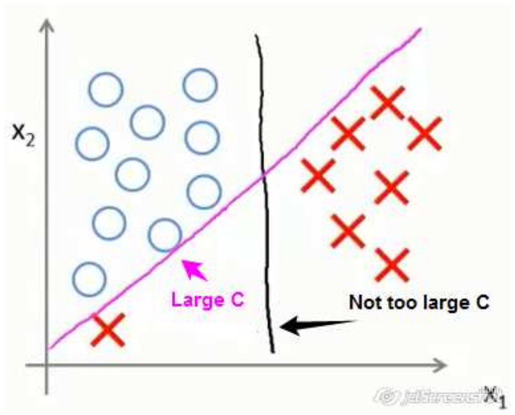
Our hypothesis is: $h(x) = \begin{cases} 1 \; if \; z \geq 0 \\ 0 \; if \; z < 0 \end{cases}$ but our cost function looked like the charts on the left. So according to our cost functions we get a cost of 0, i.e, perfect prediction, when "z" ($\theta^T X$) is either greater equal to 1 (in the case of y=1), or z lower equal to 1 (in the case of y=0).
So the SVM builds in a "safety" factor compared to the hypothesis.

Our minimization problem was:

$$min \; C \sum_{i=1}^{m} \left[y^{(i)} cost_1(z) + \left(1 - y^{(i)}\right) cost_0(z) \right] + \frac{1}{2} \sum_{j=1}^{n} \theta_j^2$$

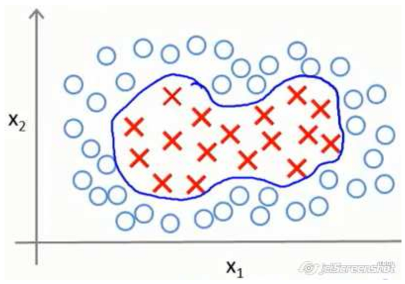
Let's choose C as a very big number, that will make the minimization process to find values for theta that eliminates the left part of the cost equation (because C is very big), and in order to zero out the left part we need to zero $cost_1(z)$ whenever y=1 and zero $cost_0(z)$ when y=0, these cost(z) will get zeroed if we find thetas that will cause to $\theta^T X >= 1$ or $\theta^T X <= -1$ (look at the cost functions charts above).
If we zeroed the entire left term on the cost equation our new minimization problem is:

$\min \frac{1}{2} \sum_{j=1}^{n} \theta_j^2 \;\; s.t \; \begin{cases} \theta^T x^{(i)} \geq 1 \; if \; y^{(i)} = 1 \\ \theta^T x^{(i)} \leq -1 \; if \; y^{(i)} = 0 \end{cases}$ When you solve this minimization problem you get the decision boundary of the SVM with its margin properties (as described in the first chart above).



SVM is more sophisticated than just choosing the large margin. If we use a dataset like the one on the left, we might choose the black line, but when C is very large the algorithm becomes very sensitive to outliers and will switch to the magenta decision boundary, and it might not be smart to switch just because of an outlier. If C is not too large then SVM becomes less sensitive to outliers.
Remember that C=1/λ, so when C is large it means lambda is small which means over-fitting, that's why we switch from black to magenta, but if we add regularization we are not over-fitting.

Kernels I

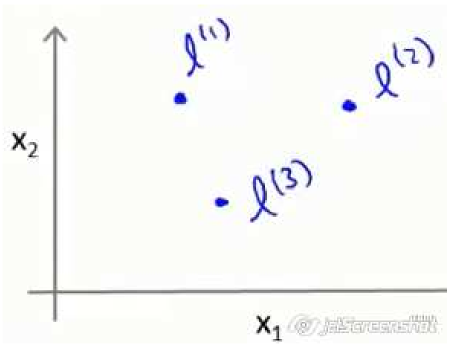


In order to get a decision boundary for the samples on the left we already saw that we need to use some high degree polynomial model, something like: $\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 + \theta_5 x_2^2 + \cdots$

We can denote each polynomial term as "f", so that:

$f_1 = x_1, f_2 = x_2, f_3 = x_1 x_2, f_4 = x_1^2, f_5 = x_2^2$

Is there a better way to choose f1, f2, f3…?



Given x, compute new features depending on the proximity to some predefined landmarks $l^{(1)}, l^{(2)}, l^{(3)}$

$$f_1 = similarity(x, l^{(1)}) = e\left(-\frac{\|x - l^{(1)}\|^2}{2\sigma^2}\right)$$

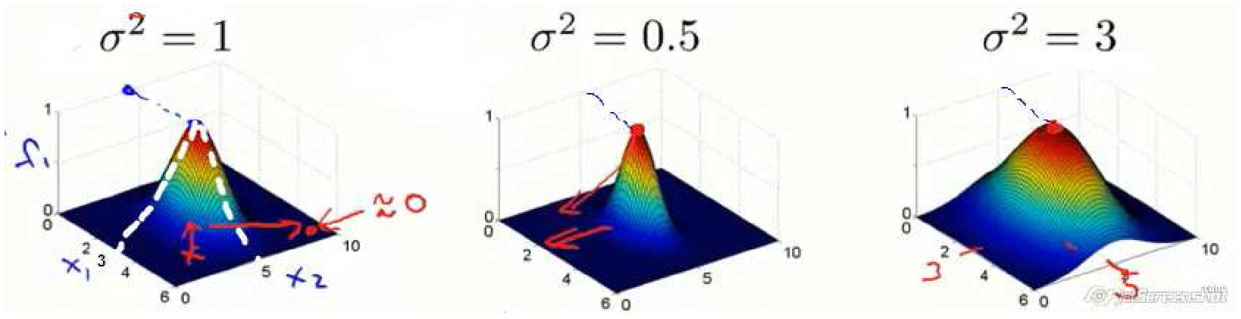
where $\|x - l^{(1)}\|^2 = \sum_{j=1}^{n}\left(x_j - l_j^{(1)}\right)^2$ which is the distance of X from l.

Instead of similarity we can say **Kernel**, and the specific kernel we choose is the Gaussian Kernel.

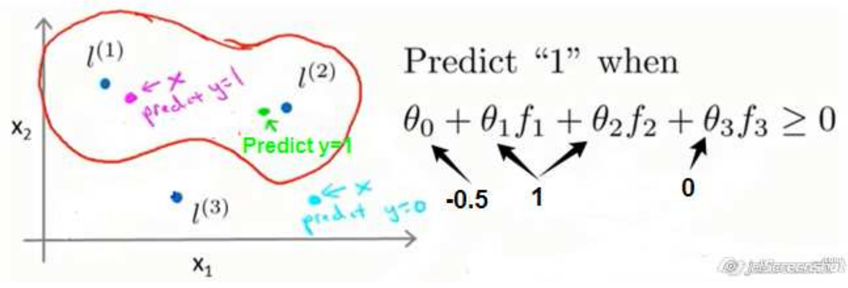
<u>What does the landmark do?</u>

If a training sample is close to L then the distance between the two would be close to 0 so f = e^0 = 1, and if X is far from L so $f = e^{-(large\ number)^2}$ = 0.

Here is a chart of "f" given the landmark **L= [3, 5]** and different values of sigma:



We can see that when X is [3, 5] it corresponds to f=1 and as we go away from 3,5 f decreases in value, the rate of which f decreases controlled by Sigma.



When X is the magenta dot, it is close to l1 so f1=1 but far from l2 and l3 so f2=0 and f=3 and we get h(x)=-0.5+1+0+0=0.5 → predict 1. Same goes for the green dot and l2: h(x)=-0.5+0+1+0=0.5 → predict 1. For the cyan dot it is far from both l1 and l2 so we predict 0.

And this is how we get our decision boundary which is built based on how far the sample dataset rows are from the landmarks <u>we chosen</u>. Given x1 and x2 we created features that are not just polynomials.

Next we would talk about how we choose the landmarks and other kernel functions.

Kernels II

How do we choose the landmarks we use?
Quite simple, we select one landmark per one training example we have, so if we are going to have "m" training examples, we are going to choose "m" landmarks, where $l^{(1)} = x^{(1)}, l^{(2)} = x^{(2)} \dots l^{(m)} = x^{(m)}$.
The benefit of this method is that now our model features measure how close the data is to our training examples. Our feature vector is of the form:

$$f^{(i)} = \begin{bmatrix} f_0^{(i)} = 1 \\ f_1^{(i)} = sim\left(x^{(i)}, l^{(1)}\right) \\ f_2^{(i)} = sim\left(x^{(i)}, l^{(2)}\right) \\ f_i^{(i)} = sim\left(x^{(i)}, l^{(i)}\right) = e^0 = 1 \\ \vdots \\ f_m^{(i)} = sim\left(x^{(i)}, l^{(m)}\right) \end{bmatrix}$$

The size of this feature vector is n = [m+1].

Hypothesis: Given x, compute features f, predict y=1 if $\theta^T f >= 0$.
Cost function:

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

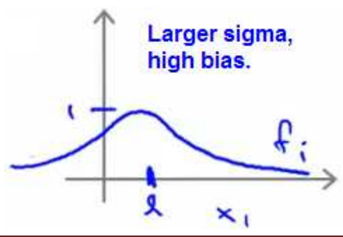
When implementing the optimization for the cost function there is a small tweak to the regularization term $\sum_{j=1}^{n=m} \theta_j^2 = \theta^T \theta\ instead\ we\ use\ \theta^T M \theta$ where M is matrix that depends on the kernel we use, so we minimize something that is a little bit different. The reason we do this is due to computational reasons, this modification let us use problems with a lot of data (like if we have 50,000 data rows, we have 50,000 features and solving it regularly is quite complicated, the adding of "M" simplifies it).
Theoretically we could have used kernels in Logistic Regression, but the simplification technique described above will not work on LR and thus solving the problem would take a lot of time.

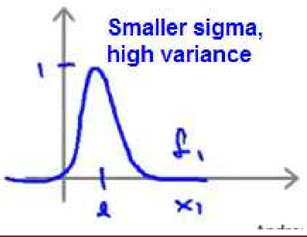
In this course we will NOT learn how to minimize the SVM cost function, we will use existing packages.

Note it is also possible to use SVM with no kernel, called "linear kernel", in this case y=1 if $\Theta^T X >= 0$.
We use this when we don't need to fit complex functions or when we don't have enough training set while a lot of feature (n is large but m is small).
If n is small and we have got a lot of data (big m) we could use the Gaussian kernel to fit the data better.

<u>SVM Parameters</u>
C - The regularization is achieved by the value of "C", which is 1/λ, so a large value of C means small value of lambda hence low bias, high variance, and vice versa for smaller values of C.

Sigma (σ) – We explained earlier that sigma controls the rate at which the feature "f" declines toward zero when the data point is getting away from the landmark. This rate corresponds to the bias/variance tradeoff, if sigma is large the rate of change is smaller and we get high bias low variance, and vice versa.





Using an SVM

When we come to learn SVM we will use some ready available libraries that knows how to minimize the cost function, some libraries, like Octave, will require us to implement the Kernel function (unless we use linear kernel == no kernel).

Function f = kernel(x, l)

$$f = e\left(-\frac{\|x-l\|^2}{2\sigma^2}\right)$$

end

Where kernel function gets “x” and “l” vectors representing a specified row and returns the feature’s value.

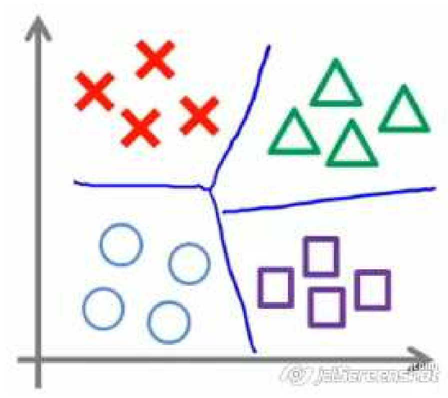
Note: It is important to perform feature scaling before using the Gaussian kernel. The reason is that features with large values (like house sq ft) would become more dominant than features with lower values (no of bedrooms) because we square the distance of the feature from the landmark. So say the landmark is a 500 sq ft house with 4 bedrooms, and the current examined training set is 80 sq ft 1 bedroom, the f corresponding to the sq ft feature would be of size $(80\text{-}500)^2$ where the other f corresponds to the bedrooms feature would be of size $(4\text{-}1)^2$, this will give extra weight to the size feature just because it has larger values, this gets eliminated when we scale the features.

Other Choices of Kernel

As we said, it is possible to use other similarity functions, but not all of similarity functions make valid kernels. Because the SVM theta solution is optimized for performance there are some assumptions that were agreed on regarding the kernel function, particularly, in order for SVM packages’ optimizations to run correctly all kernel functions need to satisfy the “Mercer’s Theorem” condition.

Many off-the-shelf kernels available:
- Polynomial Kernel: $K(x,l) = (X^T l + constant)^{power\ to\ some\ degree}$
- More esoteric: String kernel, chi-square kernel, histogram intersection kernel, etc…
Those kernels entire goal is to assess how close the training set and the landmark are similar.

Multi-Class Classification



We run a multi-class problem in the one-vs-all method.
Assume we have K classes we choose one class to be our positive class and set y=(y==k) (so y=1 if it belongs to the current class we chose as positive).
This would give us K models with $\theta^1, \theta^2, \dots, \theta^k$ where θ^1 is for the first class, θ^2 for second class etc.
Now given a new data set we run those K models and choose the model with the largest value $\theta^T X$.

Logistic Regression vs. SVM

We derived the SVM model from the logistic regression model, so when should we use what?
General rule of thumbs:

- If n is large relative to m, say we have a lot of features (like all words to represent spam email) and we have much lower email samples, in this case it is better to use Logistic Regression or SVM without a kernel. The reason is that we don't have samples to fit complex non-linear functions.
- If n is small, m is intermediate – like n is (1-1000) and m is (10 – 10,000) then better use SVM with Gaussian kernel.
- If n is small, m is large - like n is (1-1000) and m is > 50,000, in this case where we have a massive dataset then SVM will be slow to run and struggle.
  Solution is to create/add more features, then use logistic regression or SVM without a kernel.

So basically SVM with Gaussian kernel shines in situations where we have a decent amount of data, but not very big, and reasonable amount of features.

Side note: Neural network likely to work well for most of these setting (if designed well), but may be slower to train.

Clustering (Week 8)

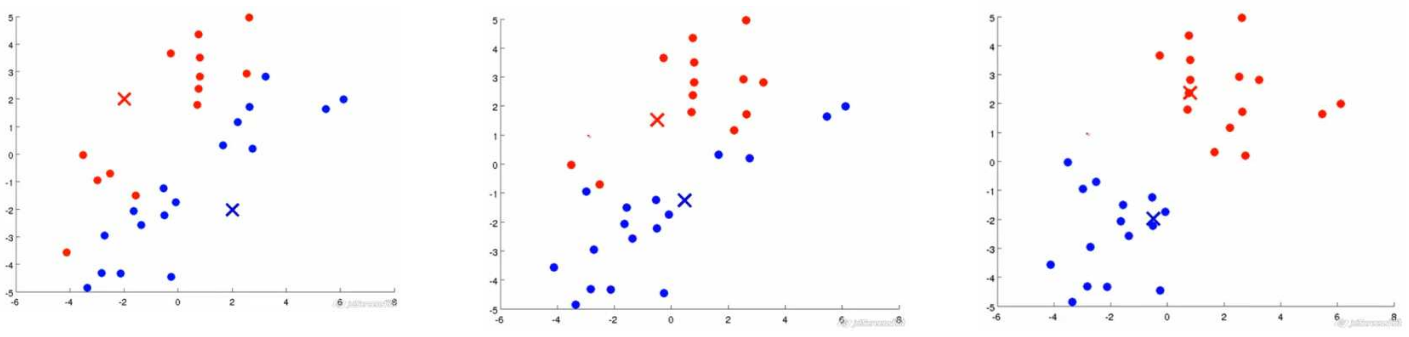
Clustering is an unsupervised learning algorithm, its goal is to find a structure in the dataset and group the data into different coherent clusters.

K-Means Algorithm

K-Means is the most popular clustering algorithm. It takes an unlabeled dataset and the data into groups.

K-Means is an iterative algorithm, assume we want to cluster the data into 2 groups:

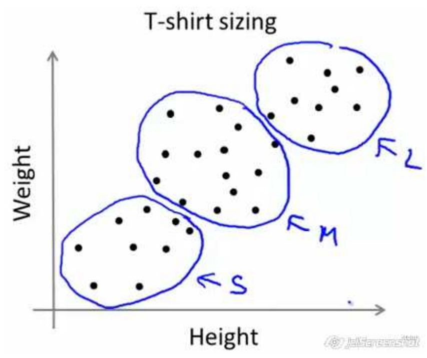
1) We start by choosing 2 random points, called the cluster centroids
2) Associate each point in the dataset to the centroid it is most close to.
3) We compute the average of each group, and move the centroids to that average.
4) Repeat steps 2-4 until there is no change between the iterations.



<u>K-Means Implementation</u>

K – The number of clusters.
Training set {x^1, x^2, x^3…} where x^i dimension is R^n (we drop x_0=1 convention).

```
Randomly initialize K cluster centroids, $\mu_1, \mu_2, \ldots \mu_k$
Repeat {
  for i=1 to m
    c(i) = The cluster index (from 1 to K) that is closest to x(i). $min_k \left\| x^{(i)} - \mu_k \right\|^2$
  end
  for k=1 to K
    u(k) = average of all points assigned to cluster k.
  end
}
```



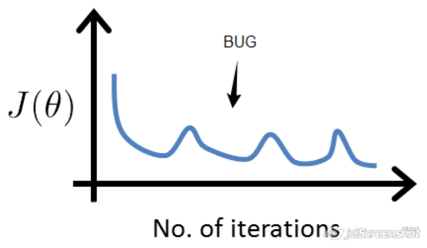
K-Mean algorithm can also become handy in separating data into groups even if there are no distinct separated clusters.
In this example we have people Weight/Height combination, the K-Mean can still separate this into groups and help us direct our product sizing (T-Shit size) better (this is Market Segmentation).

Optimization Objective

K-Mean minimization problem is to minimize the distance of each point from its centroid, this is the K-Mean cost function (or Distortion function):

$$J\left(c^{(1)}, \dots c^{(m)},, \mu_1 \dots, \mu_k\right) = \frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)} - \mu_{c^{(i)}}\right\|^2$$

Where:
c(i) – index of cluster to which example x(i) is assigned.
u(k) – Cluster centroid of group k.
u(c(i)) – Cluster centroid to which example x(i) has been assigned.



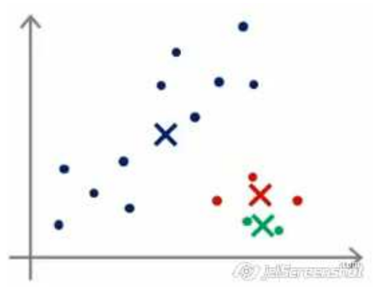
The cost function is a function if both c(i) and u(c(i)), so we need to find c and u combinations that minimizes the cost function.
If we look at the K-Mean iterative algorithm we defined earlier, we can see that the first loop is minimizing the cost with respect to c(i), while the second loop minimizes the cost with respect to u(k).
This iterative process MUST decrease the cost function in every iteration, so if we see a cost function that has rising sections we have a bug…

Random Initialization

The first step in K-Mean is randomly initializing the cluster centroid, who is it best to do this?

1) We should have K < m.
2) Randomly pick K training examples.
3) Set u to those selected training examples.

The problem is that K-Mean can get stuck at a local optima, it all depend on how it got initialized.



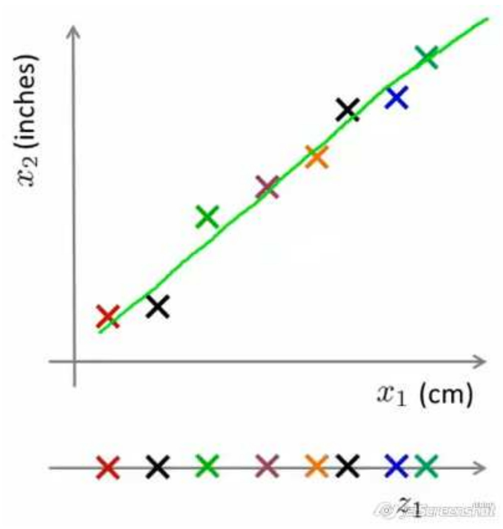
A local optima is when the K-Mean algo has converged and decreased the cost function to a minimum, but there might be a better solution, like in the example on the left.
The solution for this problem is just run the K-Mean algo several times, each time with a different random initialization, and choose the solution with the lowest cost value.
This solution might work if K is small, anywhere between 2 to 10, but if K is larger than this process might not improve the first solution that much.

Choosing the Number of Clusters

There is no “best” way of choosing the number of clusters, this is usually be done by human input depend on the problem he is trying to solve. Ask for what purpose are you running K-Mean and what is the better number of cluster that will serve this purpose.
For example if you are a T-Shirt manufacture and you want to segment your customers into sizes, you can define that 3 sizes is what you want, or maybe 5 clusters, this is a business decision. Also running the K-Mean on 3 sized and 5 sizes can answer the question “how well my T-Shirts will fit my customers”, and this can add some insight into the business decision that has to be made on how much sized to produce.

Dimensionality Reduction

Data Compression



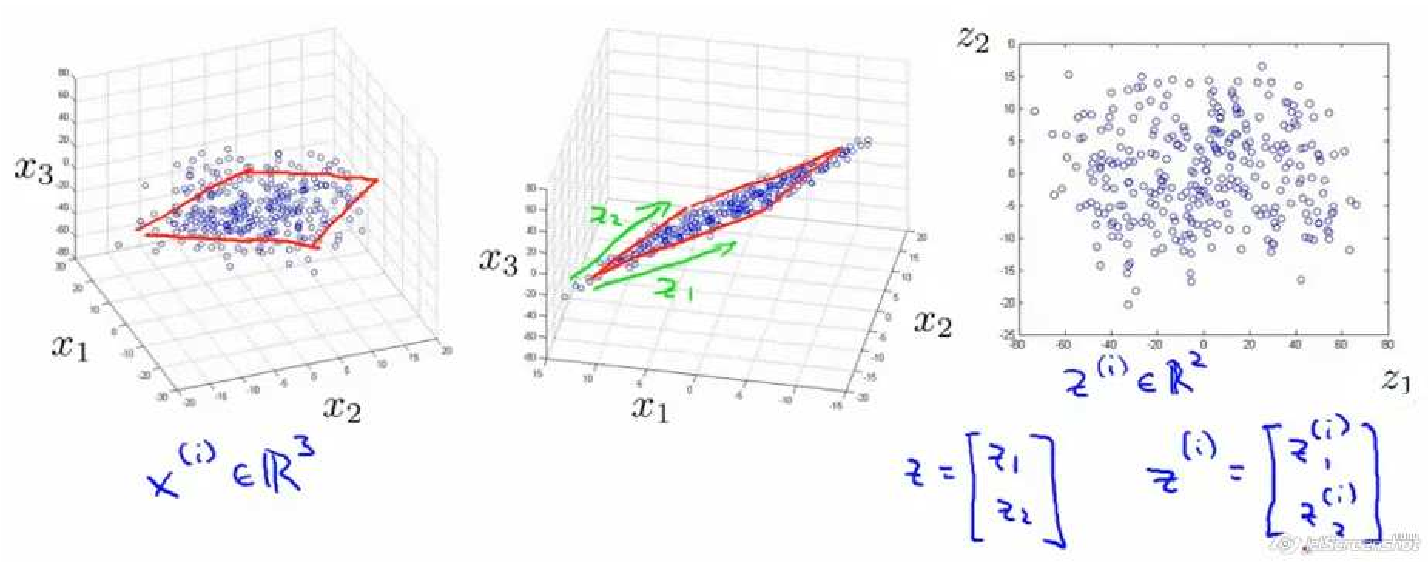
Reduce data from 2D to 1D

We start by example to explain what dimensionality reduction is. Assume we have measured something with 2 different devices, one gave us the length in inches, and the other in centimeter, and we want to use those measurements as features. Obviously those 2 features will be correlated (they are not 100% correlated because each measurement device had its own accuracy and rounding), and it is redundant to have the length in 2 different features. In that case we can reduce the dimension of the data from 2D to 1D.
The way to do this is to draw a line (green line) and project each point onto this line, denotes as "z".
So if previously $X^{(1)}$ was a feature vector with 2 elements, now it is mapped to $Z^{(1)}$ which is a (single) real number.
This new feature is an approximation of the original features, but in return it cut our memory usage and storage in half, and more importantly make our learning algorithm run faster.

Reduce data from 3D to 2D

In this example we will reduce a 3D feature vector into a 2D feature vector. The process is the same, we take the 3D vector (left image) and project it onto a 2D surface (middle image), so what we do we force all the points to be on the same plain, and once they are all on the same plain we got a 2D feature vector (right image). Of course it was possible because originally the data where more or less on the same plain.

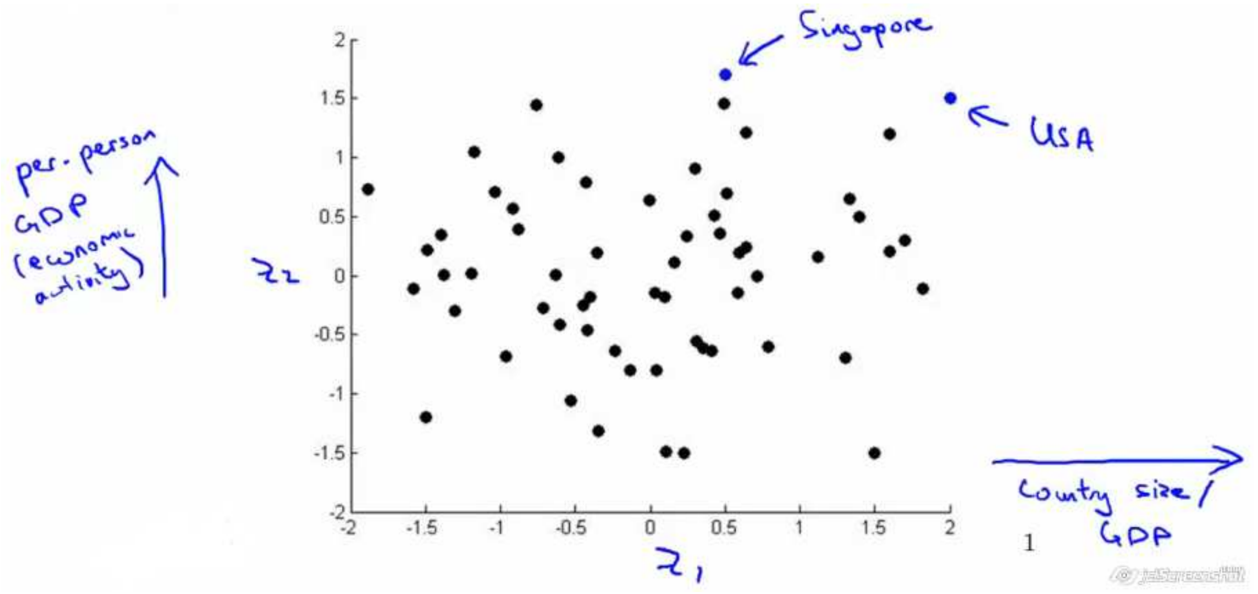
This process can be done for reducing any dimension to any other dimension, like reducing a feature vector with 1000 features into a feature vector with 100 features.



Data Visualization

In many ML problems we can find a better solution if we are able to visualize the data, dimensionally reduction can help us with that.

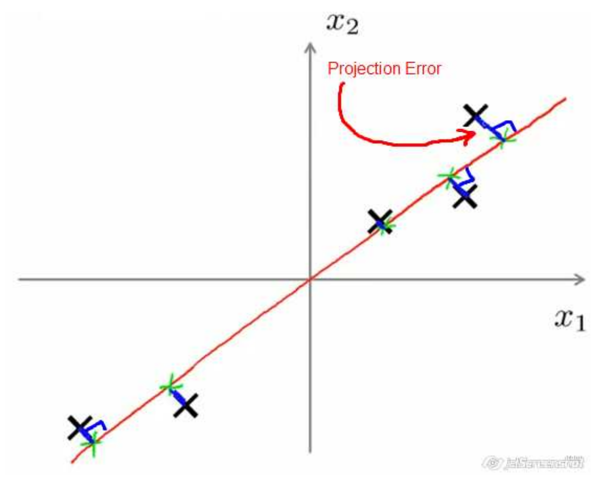
Assume we have data on many countries, and each feature vector has 50 features (like GDP, Per capita GDP, life expectancy, poverty index etc). Visualizing this 50 dimensional data is pretty difficult. Using dimensionality reduction we can reduce this 50D into a 2D data, and visualizing a 2D data is much simpler. The problem is, the dimensionality reduction algorithm doesn't give any meaning to the new features, it is up to us to figure out what it means.



So in our example this is the 2D features we get, each point represents a country, and we assume that z(1) is related to the country size/economic activity, and z(2) is related to the Per Capita GDP/economic activity. So the dimensionality reduction captures the main 2 dimensions of variations between countries.

Principal Component Analysis (PCA)

Principal Component Analysis is the most common dimensionality reduction algorithm.



In PCA what we do is finding a vector direction onto which we project our data which gives the smallest sum of squared projected errors. A vector direction is a line that goes thru the origin, and the projection error is the distance of the orthogonal (90 degrees) line from the feature to the vector.

PCA problem

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

PCA vs Linear Regression

Many confuse PCA and linear regression, but those are 2 different algorithms. PCA tries to minimize the projection error (i.e the orthogonal distance from a point to the vector) which Linear Regression tries to minimize the prediction error.
In linear regression we try to predict some value "y" while in PCA we don't try to predict anything, we treat all features equally.

Principal Component Analysis Algorithm

1) The first step in PCA is mean normalization.
   We find the mean of all the features and then set $x_j = x_j - \mu_j$.
   If the features are on different scales (x1 house size, x2 bedrooms), we also divide each feature by std dev.
2) The second step is to compute the "covariance matrix" sigma.
   $$sigma = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)})(x^{(i)})^T$$
   This gives us a [n,n] matrix.
   This calculation can be Vectorized: $sigma = \left(\frac{1}{m}\right) * X^T * X$
3) The third step is to compute the "eigenvectors" of the sigma matrix, we do this by using a "singular value decomposition", in octave:
   [U, S, V] = svd(sigma);
4) "U" is a [n, n] matrix where each column is the direction vector with smallest projection error to the data. So if we wanted to reduce our features from "n" to "k" we take from "U" only the first "k" vectors, we get a [n, k] matrix we denote as U_{reduce} and to finally get the "z" feature vector we:
   $z^{(i)} = U_{reduce}^T \times x^{(i)}$ Where x is [n,1] dimension so the result is [k, 1] dimension as needed.
5) NOTE: This does NOT performed on the bias feature $x_0 = 1$

Choosing the Number of Principal Components

How do we choose K?

PCA minimize the average squared projection error, which is: $\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)} - x_{approx}^{(i)}\right\|^2$
The total variance of the data is: $\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)}\right\|^2$

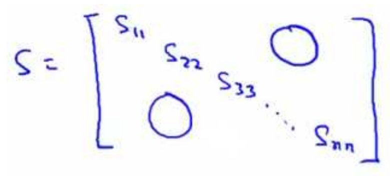
Choose K to be smallest value so that the ratio of the average squared error to variance is less than 0.01.

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)} - x_{approx}^{(i)}\right\|^2}{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}\|^2} \le 0.01\ (1\%)$$

What it means is that "99%" of the variance is retained. We can play with this and choose K so we retain "95%" of the variance etc. Sometimes by keeping 95% of the variance we can significantly reduce the number of features in the model, this is because many features are correlated in some way.

How do we calculate that? One way is set K=1 and run PCA, get U_{reduce} and Z, compute the ratio and see whether it is less than 0.01, if not do it again with k=2 and so on until you find the ratio is <= 0.01.

There is a simpler way of doing that, when calling “svd” in octave we get 3 output parameters:
[U, S, V] = svd(sigma);



What is S?
This is a [n ,n] matrix where only the diagonal has values while all other cells are zero. Using this matrix we can compute the ratio:

$$\frac{\frac{1}{m}\sum_{i=1}^{m}\left\|x^{(i)} - x_{approx}^{(i)}\right\|^2}{\frac{1}{m}\sum_{i=1}^{m}\|x^{(i)}\|^2} = 1 - \frac{\sum_{i=1}^{k} s_{ii}}{\sum_{i=1}^{n} s_{ii}}$$

If we want to check for k=3, we take the sum of the first 3 rows divided by the sum of the entire S matrix.

Alternatively we can choose K so that $\frac{\sum_{i=1}^{k} s_{ii}}{\sum_{i=1}^{n} s_{ii}} \geq 0.99$

Reconstruction from Compressed Representation

Using PCA we took a n dimension vector and turned it into a k dimensional vector, how do we go back and “decompress” the data?

What we can do is reverse the process and get the approximation of the original features:

$$x_{approx}^{(i)} = U_{reduce} z^{(i)}$$

U is [n, k] matrix, Z is [k, 1] vector, so the multiplication is [n, 1] which is like the original feature vector.

Applying PCA

Suppose we are running some computer vision ML on a 100x100 picture, this is 10,000 features.
The first step would be to run PCA and reduce the data to something like 1000 features ($Z_1 \ldots Z_{1000}$).
Next we run a learning algo on the training set: (z(1), y(1)), (z(2), y(2))…(z(m), y(m)).
When it’s time to predict, we take the new (out of sample) feature vector x, transform it into the z vector using the U_{reduce} we used for learning, and make the prediction.
Note: The mapping between X(i) → Z(i) is done using the training set only, then we apply it to the other out-of-sample data (like cross-validation and test sets).

Bad use of PCA

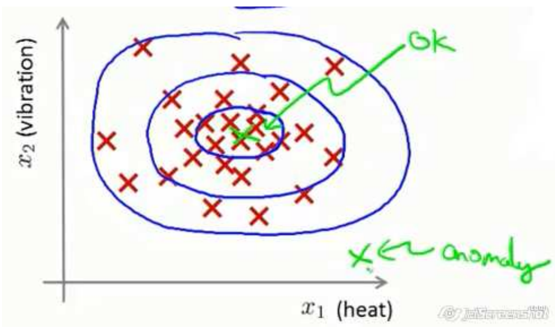
There is a common mistake for using PCA in order to reduce over-fitting, this is because PCA reduce the number of features thinking that it will reduce the over fit.
This is a bad practice and you are better off going with just regularization. The reason is that PCA “throws” features by approximation only, it does not have any information on the “y” variable, so it might throw away goo data. While regularization is achieved by minimizing a new cost function where it DOES look at the “y” variable, so if there is important data it will not remove it.

Another common mistake is automatically incorporating PCA as part of the learning process, while this might work, it is better to start with all the data, and introduce PCA only if you see it is necessary (like the executing time is too long or too much memory is used etc). There is no need to automatically include PCA in the learning process.

Anomaly Detection (Week 9)

Anomaly detection is an supervised learning algorithm that tries to detect data points that might not belong to some existing defined group.

Density Estimation



Given dataset x(1)…x(m), which we assume is normal, we want to know whether new data x(test) is anomalous, meaning what are the odds it does not belong to the group. The model we build would give us the probability p(x) that x(test) belongs to the group based on its location.
The blue circle on the chart shows different areas of probability, as x(test) is closer to the mass of other items it is more likely to be part of the group, as we go out the probability decreases.

This method is called **Density Estimation**.

$$if\ p(x)\begin{cases} \leq \varepsilon\ then\ Anomaly \\ > \varepsilon\ then\ OK \end{cases}$$

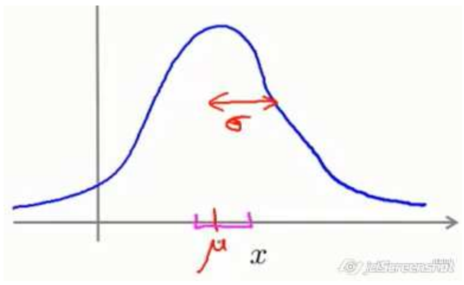
<u>Examples</u>

Anomaly detection is used to identify frauds; online website collects data about their users, so a feature vector might include features like: how often does the user log in, pages visited, number of posts on the forums, typing speed etc. Given those feature it is possible to build a model that will identify users that do not match the pattern.

Another example is monitoring of data centers. Features may be: memory use, number of disk accesses, cpu load, cpu load/network traffic etc.
Given those features if we get a computer with p(x)<epsilon it might be that this computer has a problem.

Gaussian (Normal) Distribution

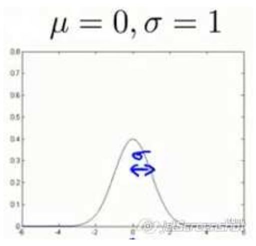
Say x is distributed Gaussian with mean u and variance sigma: $x \sim N(\mu, \sigma^2)$

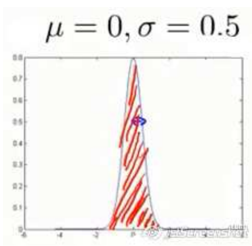


The function of the Gaussian distribution is:

$$P(x,\mu,\sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} exp - \left(\frac{(x-\mu)^2}{2\sigma^2}\right)$$

So the plot is p(x) as a function of given u and sigma.





The shape of the function is determined by the variance, if the variance is small then the function would be narrow and tall, if variance is large then the function is low and fat. <u>The area below the function is 1</u>, this is a property of a probability function.

Parameter Estimation

Say we have a dataset which we assume came from a Gaussian distribution, and we want to estimate u and sigma of the entire population, this is how we are going to do this:

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

So the estimate of u is just the average of our dataset, and the estimation of sigma is the sum square of differences of each sample from the mean, divided by "m" (In theory it should be divided by m-1, but in ML problems we use m as with a lot of data it doesn't make a difference).

Anomaly Detection Algorithm

Given data set x(1)…x(m) we compute the u and variance of each feature. Remember, each feature X_1,X_2…X_n is a vector that contains the values for this features across all rows, so each feature has its own mean and variance: $X_1 \sim N(\mu_1, \sigma_1^2), X_2 \sim N(\mu_2, \sigma_2^2), \dots, X_n \sim N(\mu_n, \sigma_n^2)$

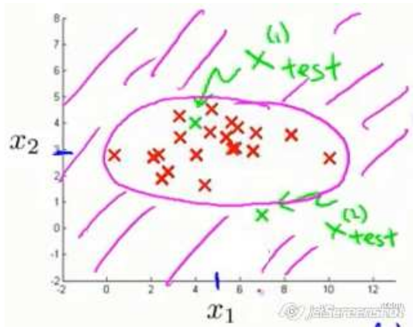
$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} x_j^{(i)}$$

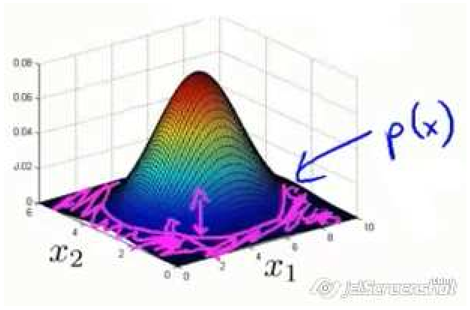
$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x_j^{(i)} - \mu_j\right)^2$$

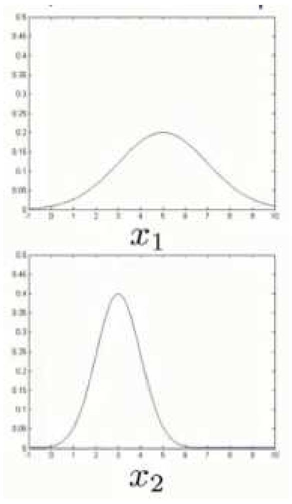
Once we have estimated the mean and variance, given new test example we compute our model p(x): (Note that p(x) is for a single example, meaning just one row).

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

Where $\prod(\dots)$is the sum of products (meaning a*b*c*…).







In the left picture we have a dataset with 2 features, the distribution of each feature is shown on the right picture. The 3D picture in the middle is the density estimation function, given the mean and variance of x(1) and x(2) it plot the value of p(x), so the z axis is the value of p(x). We can see that at the extremes the surface is low; meaning p(x) has small value representing combination of x1/x2 that are anomalies. An area where the surface is elevated means p(x) has larger value representing "normal" points. As we go "inside" towards the mean of x1 and x2 the surface (p) rises, until we get to the point where p(x)=epsilon, this defines our decision boundary, this is the magenta ellipse on the left picture, then given new data (green x) we see if it falls inside (normal) or outside (anomaly).

Evaluating an Anomaly Detection System

Anomaly detection algorithm is an unsupervised learning algorithm, meaning we don't have the "y" value that tells us if it is really anomaly or not. In this case how can we evaluate if our algorithm (meaning the features we chose) is good. We need a way to test our classification power.

When developing anomaly detection system, we start off by taking labeled data, of anomalous and non-anomalous examples (y=0 if normal, y=1 if anomaly).
From this set we make a training set that contains normal data only.
And we prepare a cross-validation set and test-set with mixed data of normal and anomalous examples.

Example

We have 10,000 good (normal) engines, and 20 flawed engines (anomalous).
We split the data as:
6000 normal engines for training set (y=0).
2000 engines for the cross validation set, out of which 10 are anomalous.
2000 engines for the test set out of which 10 are anomalous.

1) From the training set we estimate the mean and variance of the features and build p(x).
2) On CV and test set predict if x is anomaly or normal (y=1 or y=0).
3) Because the data is very skewed towards y=0 a prediction precision is not a good evaluation, so we can use methods discussed earlier in the course:
    a. Precision / Recall
    b. F_1-Score
4) We can use the CV set to try different values of epsilon so we maximize our evaluation score.

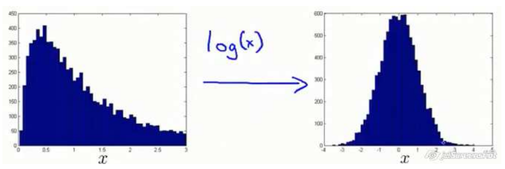
Anomaly Detection vs Supervised Learning

In the previous section we talked about evaluating anomaly detection using labeled data, if this is the case, and we have labeled data, why not use supervised learning for this task?
Here are the considerations for choosing either supervised or unsupervised learning:

Anomaly Detection	Supervised Learning
Very small number of positive (anomalies, y=1) examples (0-20 anomalies is common which we prefer to use for CV and test set)	Large number of BOTH positive AND negative examples
Large number of negative (normal data, y=0) examples (for learning)	
Many different "types" of anomalies, hard for any algo to learn from the few positive examples what anomalies look like. Future anomalies might look totally different. In this case it is better to model the negative examples (which are many).	Enough positive example to learn from and future positive examples likely to be similar to ones in training set.
Examples: Fraud detection. Manufacturing (e.g aircraft engines) Monitoring machines in data center. All of the above examples usually don't have many positives examples (i.e anomalies).	Examples: Email spam classifier Weather prediction (sunny/rain/etc) Cancer classification All of the above examples usually do have enough positives examples to learn from.

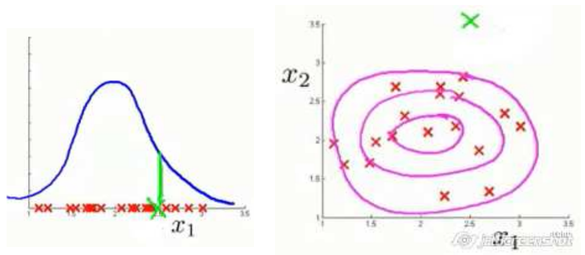
Choosing Features

In anomaly detection algorithm the features we use are crucial, so let's talk about how to choose the features.



The assumption in anomaly detection is that the features have Gaussian distribution (using histogram we want to have something like bell curve.
If the data distribution is not Gaussian, the algorithm still works ok, but it might do better if the data would be transformed into a Gaussian distribution, using the log function: x = log(x+c) where "c" is a constant >= 0. Another transformation is $x = x^C$ where c is a fraction between 0 and 1.

Error Analysis

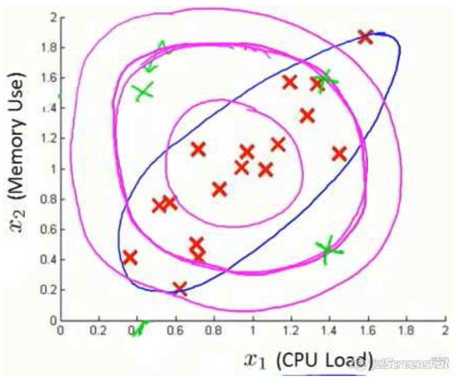


A common problem is that p(x) has a "wide base", meaning many examples are considered as normal because they get high value of p(x), like the green sample on the left most picture.

In that case error analysis can help, take the anomalies that were wrongly classified and try to learn what was the problem in that sample, for example if this is an engine try to understand what the problem was, and make a new feature out of that, maybe this new feature will add a dimension that is less "wide".

"Good" features usually take unusually large or small values in the event of an anomaly.
This can sometimes be achieved by combining two correlated features. For example in monitoring data center, we assume that if the CPU is high it means we serve a lot of users and we expect network traffic to be high as well (cpu and network goes together). Creating a new feature like "CPU / Network traffic" will take on a very large value when the CPU is high and network traffic is low, this can signal that the server is stuck in some non-ending job.

Multivariate Gaussian Distribution



Suppose we have two features that are correlated and span over a wide range, in this case the regular anomaly detection algorithm will fail to identify anomalies. The reason is that the regular algorithm tries to capture both the first feature variance and second feature variance, so this creates a large "decision boundaries". In the picture to the left we can see that the 2 features are correlated, and the magenta lines are the decision boundaries based on different values of epsilon, note how it assumes that all the green X that are on the same line have the same value of p(x), while when looking at the data it is not true. Multivariate Gaussian distribution will create a decision boundary that is more like the blue ellipse.

In the regular model we compute p(x) as a multiplication of p(x1)*p(x2)*…p(xn), so we treat each component separately. In the Multivariate model we will create a covariance matrix of the features and compute p(x) using all the features together.

Fist we calculate the mean of all the features, and the covariance matrix Sigma:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \qquad sigma = \Sigma = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \mu\right)\left(x^{(i)} - \mu\right)^T = \frac{1}{m}(X-\mu)^T(X-\mu)$$

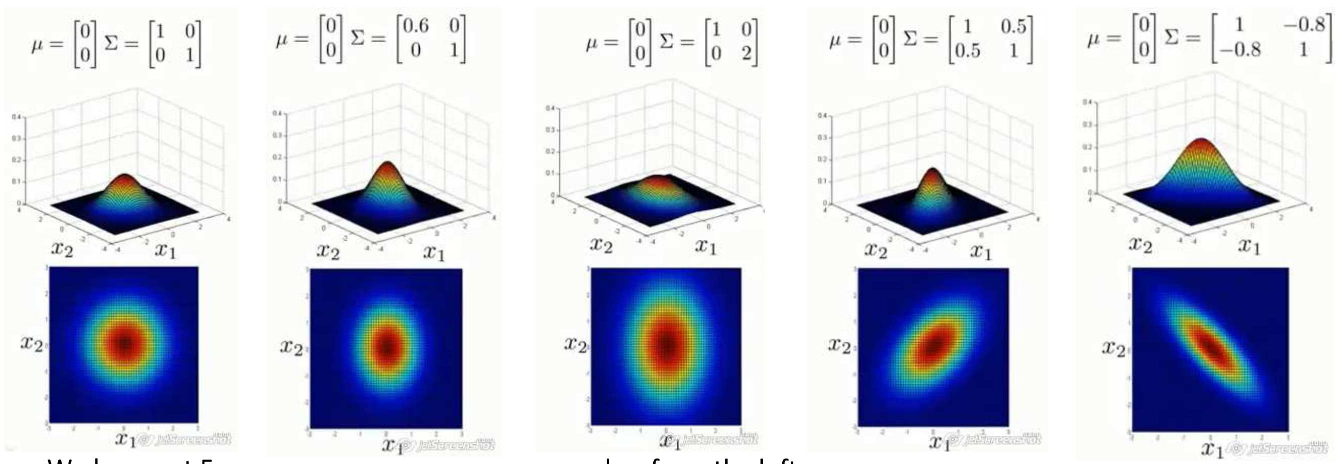
Note the u is a vector where each cell is the average across all rows of each feature, then we compute sigma where (x-u) are also vectors. The last part is the vectorization implementation of the sigma calculation, note though that X is a matrix while u is a vector, so u vector should be deducted from each column of X, this is done automatically in Octave by broadcasting (as this is not a valid matrix vector operation).

Finally we compute p(x):

$$p(x,\mu,\Sigma) = \frac{1}{(2\pi)^{\frac{n}{2}}|\Sigma|^{\frac{1}{2}}}\exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)$$

Where:
$|\Sigma|$ – Determinate of sigma, calculates in Octave using det(sigma).
Σ^{-1} – The inverse matrix of the sigma matrix.

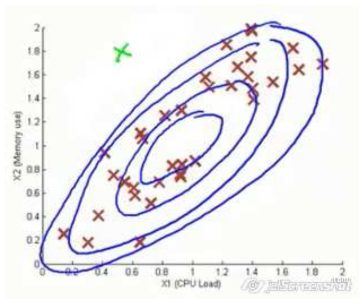
<u>How does the covariant matrix affect the model?</u>



_We have got 5 examples, from the left:

1) This is the regular model, where x(1) and x(2) use their regular variance (we set 1 in the matrix).
2) Here we set x(1) to have smaller variance while keeping x(2) variance in the original value.
3) Now we set x(2) to have a larger variance while keeping x(1) in the original value.
4) Here we left x(1) and x(2) in the original variance but we set a correlation between them, so the decision boundary rotates (this is done by setting values in the cells that are not on the diagonal of the matrix).
5) By using negative values we can rotate the boundary to the opposite direction.

In this examples we set the mean of x(1), x(2) at 0, if we change this it will just shift the center of the ellipse.

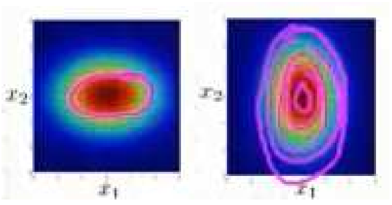


Back to our original example, using multivariate solution we would get this boundary. So we took into account the features correlation and rotated the boundary (it is done automatically once we computes the covariance matrix).

Relationship to the original model

It is possible to proof that the original anomaly detection model is a sub-case of the multivariate model. Specifically, the original model is the multivariate model where all the values on the sigma matrix are 0 except the diagonal cells that contains the variance of each feature.

$$In\ original\ model\ \Sigma = \begin{pmatrix} \boldsymbol{\sigma_1^2} & 0 & 0 & 0 \\ 0 & \boldsymbol{\sigma_2^2} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & 0 & \boldsymbol{\sigma_n^2} \end{pmatrix}$$



This constraint make the contours of the original model to be aligned to the axis, so the boundary is not “rotated”, a correlation is not taken into account.

Original model vs Multivariate model

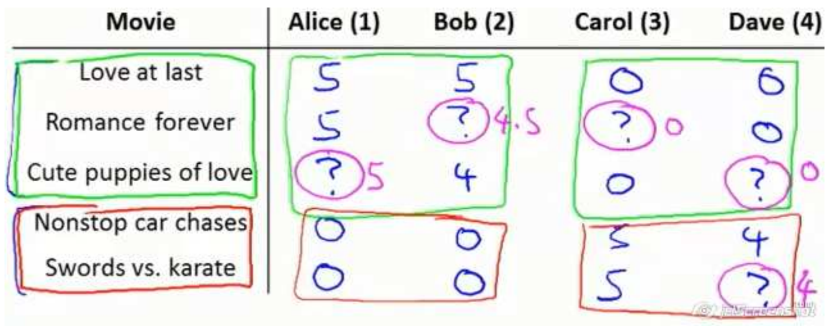
Original	Multivariate
Does not capture correlations between features, this can be solved by creating new feature x(3)=x(1)/x(2).	Automatically captures correlations between features.
Computationally cheaper, can scale better to large number of features.	Computationally expansive.
Works fine if the training set size is small.	Must have m>n or else Σ is non-invertible. In practice it is recommended that m > 10*n.
	Σ is non-invertible also if there are redundant features, like if x(1)=x(2), or x(3)=x(4)+x(5), in this case the redundant feature has to be removed.

The original model is more widely used; in case there are some features that are correlated they create another feature that captures this correlation (like dividing the correlated features in each other).
If, on the other hand, the training set is not very large, and there are not a lot of features, you can use the multivariate model which will take care of all correlations.

Recommender Systems

Problem Formulation

We start with an example in order to define the problem.



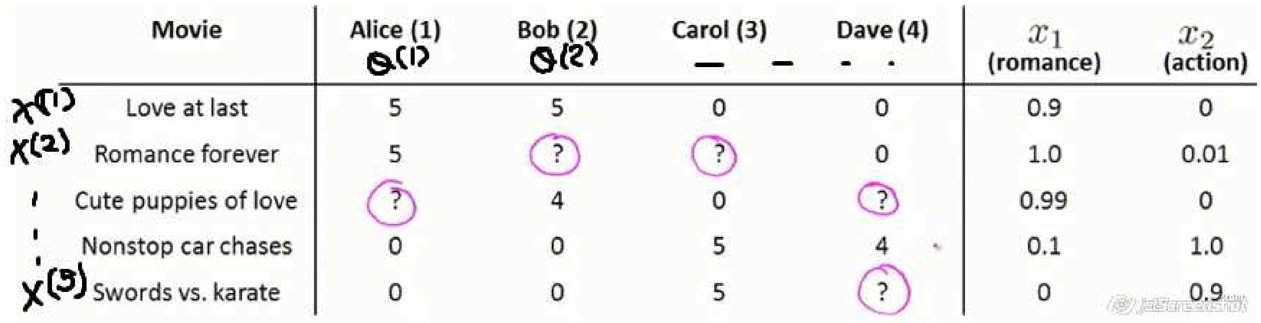
Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)
Love at last	5	5	0	0
Romance forever	5	? 4.5	? 0	0
Cute puppies of love	? 5	4	0	? 0
Nonstop car chases	0	0	5	4
Swords vs. karate	0	0	5	? 4

Assume we are a movie provider, and we have 5 movies and 4 users, we asked the users to rate the movies:
So we can see that Alice and Bob kind of liking Romance movies, and dislike Action movies, while it is the opposite with Carol and Dave. Also note that not all users watched all movies (and in real life we have much more movies), our goal is to build an algorithm that will "predict" what score would they give to each movie they did not watch, and this would be the recommendation.

Notations:
n_u – Number of users
n_m – Number of movies
$r(i, j)$ - Boolean whether user j rated movie i.
$y(i, j)$ - The rating user j gave movie I (if he rated it).
m_j - The number of movies that user j rated.

Content-based Recommendations

In a content based recommendation algorithm we assume we have some data about the items we want to recommend on (movies/products etc), this data is some features about the item.
So in our example let's say that for each movie we have 2 features, where x(1) grade how much "romantic" the movie is, and x(2) how much "action" the movie is.



	Movie	Alice (1) $\theta^{(1)}$	Bob (2) $\theta^{(2)}$	Carol (3)	Dave (4)	x_1 (romance)	x_2 (action)
$x^{(1)}$	Love at last	5	5	0	0	0.9	0
$x^{(2)}$	Romance forever	5	?	?	0	1.0	0.01
	Cute puppies of love	?	4	0	?	0.99	0
	Nonstop car chases	0	0	5	4	0.1	1.0
$x^{(5)}$	Swords vs. karate	0	0	5	?	0	0.9

So each movie has its own feature vector, where $x^{(1)}$ is the feature vector of the first movie, $x^{(2)}$ feature vector of the 2nd movie etc.
Next we are going to learn a model that predicts the rating for each user, assume we did this using linear regression and got our model parameters Theta. $\Theta^{(1)}$ is the model of the first user etc…
$\theta^{(j)}$ - Parameter vector for user j.
$x^{(i)}$ - Feature vector for movie I (including the intercept term x(0) = 1).
For user j, the predicted rating on movie $i = \left(\theta^{(j)}\right)^T \left(x^{(i)}\right)$

Cost Function

The cost function for learning the linear regression is the regular sum of squared errors:

$$min\frac{1}{2}\sum_{i:r(i,j)=1}^{m}\left(\left(\theta^{(j)}\right)^{T}\left(x^{(i)}\right)-y^{(i,j)}\right)^{2}+\frac{\lambda}{2}\sum_{k=1}^{n}\left(\theta_{k}^{(j)}\right)^{2}$$

Where i:r(i,j) means counting only the movies the user has rated.
Note: The fraction should be 1/2m for both the errors and regulations, but we can multiply the equation by (m) on both sides and it won't change the minimization problem, so we do that to get rid of "m".
Note: We don't regularize the intercept feature x(0)=1.

The above function is for learning just one user, in order to learn for all user we just add another summation over all users:

$$\min_{\theta^{(1)}\dots\theta^{(n_u)}}\frac{1}{2}\sum_{j=1}^{n_u}\sum_{i:r(i,j)=1}^{m}\left(\left(\theta^{(j)}\right)^{T}\left(x^{(i)}\right)-y^{(i,j)}\right)^{2}+\frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}\left(\theta_{k}^{(j)}\right)^{2}$$

Finally, the partial derivative of the cost function (for gradient descent) is:

$$\frac{\partial}{\partial\theta_{k}^{(j)}}J\left(\theta^{(1)}\dots\theta^{(n)}\right)=\sum_{i:r(i,j)=1}^{m}\left(\left(\theta^{(j)}\right)^{T}\left(x^{(i)}\right)-y^{(i,j)}\right)x_{k}^{(i)}\ for\ k=0$$

$$\frac{\partial}{\partial\theta_{k}^{(j)}}J\left(\theta^{(1)}\dots\theta^{(n_u)}\right)=\left(\sum_{i:r(i,j)=1}^{m}\left(\left(\theta^{(j)}\right)^{T}\left(x^{(i)}\right)-y^{(i,j)}\right)x_{k}^{(i)}+\lambda\theta_{k}^{(j)}\right)\ for\ k\neq0$$

Collaborative Filtering Algorithm

In the previous solution we assumed we have features for the movies, and using those features we learnt the parameters for the users. Alternatively, if we had the parameters for the users we could learn the movies features (by minimizing the cost function with respect to x and not theta).

But what if I don't have any of these? Collaborative Filtering Algorithm learns simultaneously BOTH the features and the parameters, resembling what a neural network is doing when learning the hidden layers. Our minimization problem is now for both x and theta:

$$\min_{x^{(1)},\dots,x^{(n_m)},\theta^{(1)}\dots\theta^{(n_u)}}=\frac{1}{2}\sum_{(i,j):r(i,j)=1}\left(\left(\theta^{(j)}\right)^{T}\left(x^{(i)}\right)-y^{(i,j)}\right)^{2}+\frac{\lambda}{2}\sum_{i=1}^{n_m}\sum_{k=1}^{n}\left(x_{k}^{(i)}\right)^{2}+\frac{\lambda}{2}\sum_{j=1}^{n_u}\sum_{k=1}^{n}\left(\theta_{k}^{(j)}\right)^{2}$$

Where the derivatives are:

$$\frac{\partial}{\partial x_{k}^{(i)}}J(\dots)=\left(\sum_{i:r(i,j)=1}^{m}\left(\left(\theta^{(j)}\right)^{T}\left(x^{(i)}\right)-y^{(i,j)}\right)\theta_{k}^{(i)}+\lambda x_{k}^{(i)}\right)$$

$$\frac{\partial}{\partial\theta_{k}^{(j)}}J(\dots)=\left(\sum_{i:r(i,j)=1}^{m}\left(\left(\theta^{(j)}\right)^{T}\left(x^{(i)}\right)-y^{(i,j)}\right)x_{k}^{(i)}+\lambda\theta_{k}^{(j)}\right)$$

Note: In this algorithm we don't add an intercept x(0), the algo can learn this feature it by itself if needed.

For user with parameters θ and movie with learned features x, predict rating $=\left(\theta^{(j)}\right)^{T}\left(x^{(i)}\right)$.

<u>Finding Related Items</u>

The feature vector that the learning process finds holds some important data about the movies; this data is not always understandable to humans as there is no explanation to the numbers. We can use this data however to recommend similar items to users.
For example, say a user is watching movie x(i), we can look for another movie, x(j) where the distance between the feature vectors is small:
$if\ \|x^{(i)} - x^{(j)}\|\ is\ small\ recommend\ movie\ j\ when\ movie\ i\ is\ selected.$

Mean Normalization

Let's look at the following user ratings:

Movie	Alice (1)	Bob (2)	Carol (3)	Dave (4)	Eve (5)
Love at last	5	5	0	0	?
Romance forever	5	?	?	0	?
Cute puppies of love	?	4	0	?	?
Nonstop car chases	0	0	5	4	?
Swords vs. karate	0	0	5	0	?

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 & ? \\ 5 & ? & ? & 0 & ? \\ ? & 4 & 0 & ? & ? \\ 0 & 0 & 5 & 4 & ? \\ 0 & 0 & 5 & 0 & ? \end{bmatrix}$$

We added a 5th user, Eve, which didn't rate any movie, when we come and learn the model for Eve, the first term of the cost function would be 0 (as she didn't rate any movie so r(i:j) is always 0), hence we end up with the regularization term. With such a cost function the learnt parameters would be all 0.
When we try to make predictions for Eve, with parameters that are all 0, we would get 0 for every movie.

In this case what we do is first normalize matrix Y by subtracting from each movie its average rating.

$$Y = \begin{bmatrix} 5 & 5 & 0 & 0 & ? \\ 5 & ? & ? & 0 & ? \\ ? & 4 & 0 & ? & ? \\ 0 & 0 & 5 & 4 & ? \\ 0 & 0 & 5 & 0 & ? \end{bmatrix} \qquad \mu = \begin{bmatrix} 2.5 \\ 2.5 \\ 2 \\ 2.25 \\ 1.25 \end{bmatrix} \rightarrow Y = \begin{bmatrix} 2.5 & 2.5 & -2.5 & -2.5 & ? \\ 2.5 & ? & ? & -2.5 & ? \\ ? & 2 & -2 & ? & ? \\ -2.25 & -2.25 & 2.75 & 1.75 & ? \\ -1.25 & -1.25 & 3.75 & -1.25 & ? \end{bmatrix}$$

Next we are going to learn the model using this new Y matrix.
When it's time to make predictions we do it the regular way but we need to add the average we subtracted from each movie: $prediction = \left(\theta^{(j)}\right)^T \left(x^{(i)}\right) + \mu_i$

So the prediction for Eve would be $0 + \mu_i = \mu_i$ which make sense…

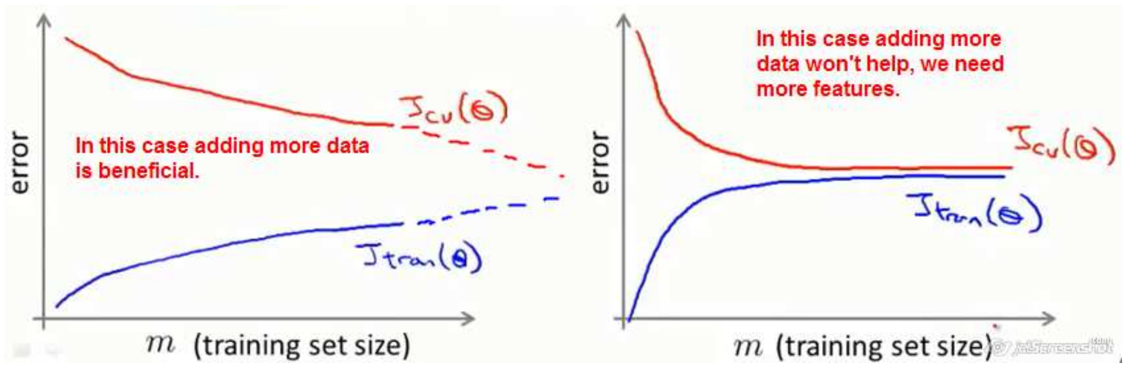
Large Scale Machine Learning (Week 10)

Learning With Large Data Set

Why would we want to use a large data set? We already saw in the course (in the "Machine Learning System Design" chapter), that if you have a low bias (=overfit) model you can get great results if you increase you training set.
So how do we cope with a training set of 100,000,000 records? This is not an unrealistic assumption; in today's world we have so much data that hundreds of millions of records is something real.

In Linear Regression for example, we have to compute the sum square of errors across all the data set, this is a loop from 1-100,000,000, and it is done for EVERY gradient descent iteration, so if we have 20 iterations in our learning, this is a very expansive computational effort.

The first thing we have to check is whether a large training set is necessary, maybe we can do ok with just a sample of 1000 training set? The way to do this is to plot the learning curve as a function of "m":

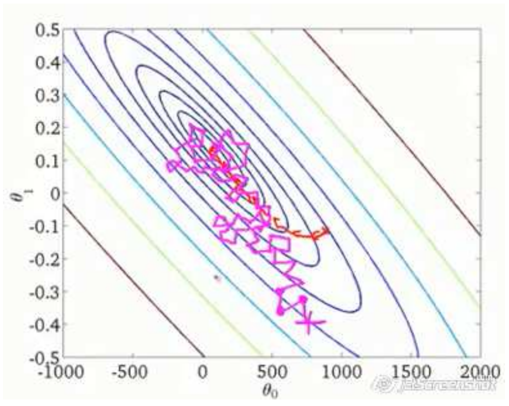


Stochastic Gradient Descent

We saw that if we have a large training set gradient descent (aka Batch GD) is very computational expansive, Stochastic Gradient Descent is an alternative that is accommodative for large training set.

In stochastic gradient descent we define our cost function as the cost of a single example:

$$cost(\theta, (x^{(i)}, y^{(i)}) = \frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$$



And the gradient descent is:
First, randomly shuffle the dataset.

```
Repeat (usually anywhere between 1-10) {
    for i=1:m {
        $\theta_j := \theta_j - \alpha(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$
        (for j=0:n)
    }
}
```

This algo updates Theta after each training set calculation, there's no need to sum over all training set first. The issue with such approach is that not every step is necessarily in the "right" direction; also it will get to the "zone" of the global minimum and not to the minimum itself, it will just continue to go around the global minimum infinitely. But while in batch GD we would take only one step after a loop from 1:m, in stochastic GD we would have taken m steps and already much closer to the global minimum.

Mini-Batch Gradient Descent

We saw that in batch GD (the original GD) we used to loop thru all examples before we update Theta.
In the stochastic GD we look only at 1 example before we update Theta.
Mini-Batch GD is somewhere in between, we loop thru “b” examples before we update Theta.

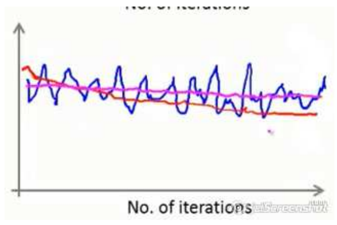
```
Repeat {
  for i=1:m {
    $\theta_j := \theta_j - \alpha \frac{1}{b} \sum_{k=i}^{i+b-1} \left(h_\theta\left(x^{(k)}\right) - y^{(k)}\right) x_j^{(k)}$
    (for j=0:n)

    i += 10;
  }
}
```

So in this algorithm we loop thru “b” examples and then update the Theta, we usually set “b” to anywhere between 2 and 100. The advantage of this approach is that you can use vectorization to do the loop thru the b samples, and if you have a good linear algebra library it can be parallelized so overall performance is not affected.

Stochastic Descent Convergence

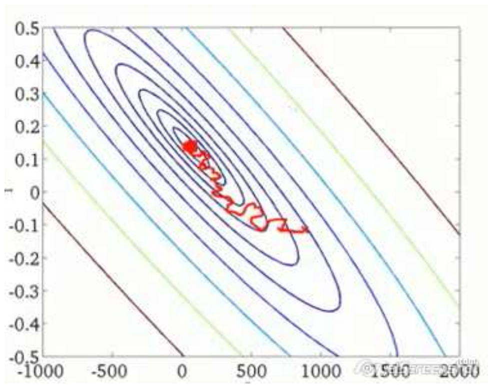
Now we are going to talk about debugging the stochastic GD algo, and choosing the learning rate alpha.

In Batch GD we plotted the cost function J as a function of the iterations, in big training sets this is not realistic as computing the cost (looping thru all data) is a heavy operation.
Alternatively, what we do, is calculating the cost on EACH iteration, just BEFORE we update theta, and then, every X iterations we plot the average cost we aggregated.



When we plot the cost function averaged over X iterations we might get a bumpy function that seems to be going sideways, what we can do is increase the average period so it would smooth the cost function, then we might see the cost decreasing (red line), or still going sideways (magenta line) which means we have a bug or poorly specified model. If the cost function is rising, we need a smaller learning rate Alpha.

Another trick that could be implemented in order to get better results, is decrease alpha as we the learning progress, so instead of keeping alpha constant set it to: $\alpha = \frac{Constant1}{iterationNumber + Constant2}$



What it does, as the learning progress, and we get closer to the global minimum, we reduce alpha so each step is smaller and smaller, forcing the algorithm to converge and not just oscillate around the minimum.
But usually the result we get without using this approach is good enough, and the overhead of choosing 2 new constant for reducing alpha does not worth the effort.

Online Learning

Online learning refers to learning from streaming data instead of an offline static dataset.
Many online websites have a constant flow of users, and with each user that come they want to be able to learn, without the need to store the user data in the database and then run the full learning process.

Assume you run a shipping company, and each time a user come to get a quote for shipment, from point A to point B, you give him a price, then the user either "take" it (y=1) or not (y=0).
You would like to build a model that will predict the chances of the user buying the service given a specific price. So price is a feature in that case, along with other features like destination and origin and user specific data, where the output is p(y=1)).
So the algo is somewhat similar to the stochastic GD where we learn from a single example and not looping thru a predefined set:

```
Repeat forever (as long as the website is running) {
  Get (x, y) corresponding to the current user
  $\theta_j := \theta_j - \alpha(h_\theta(x) - y)x_j$ // Note that we don't specify an index x(i) as there is not such index.
   (for j=0:n)
}
```

Once the learning is done for that user we can discard the data, no need to keep it anymore.
One of the benefits of this method is that we get a model that is adaptive to the change in user preferences, we continuously updating our model based on current user actions.

Note, it is not necessarily that you get only 1 data set from each interaction, for example if you display to the user 3 shipping options, and the user chooses option number 2, then you have 3 new examples to learn from, where option number 2 has y=1 and options number 1 and 3 have y=0, so you update theta based on 3 new examples.

Map Reduce and Data Parallelism

Map Reduce and Data Parallelism is a very important concept in large-scale machine learning problems. We explained earlier the problem with running a batch GD, the need to loop thru all training set and compute the cost and partial derivatives is a very computational heavy task. Instead, what if we could split our data across different machines and let each machine handle only a subset of the data, then we get the results and combine them together → That is called Map-Reduce.
More specifically, any learning algorithm that is expressed as computing sums of functions over the training set, can be split up across multiple machines (or cores) and speed up the processing time.

For example let's say we have 400 examples, we can split the <u>summation</u> part of the GD, on each iteration, into 4 machines:

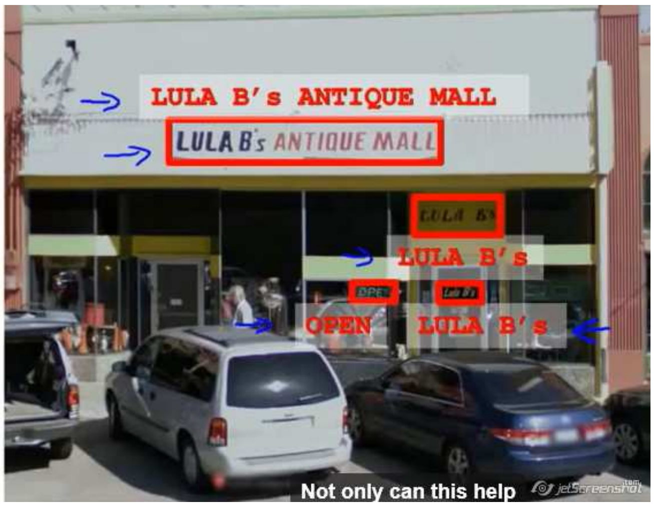
Machine no	Samples	Computation	Result
1	1-100	$temp^{(1)} = \sum_{i=1}^{100}(h_\theta(x^i) - y^i)x_j^i$	$\theta_j = \theta_j - \alpha\frac{1}{400}(temp^{(1)} + temp^{(2)} + temp^{(3)} + temp^{(4)})$
2	101-200	$temp^{(2)} = \sum_{i=101}^{200}(h_\theta(x^i) - y^i)x_j^i$	
3	201-300	$temp^{(3)} = \sum_{i=201}^{300}(h_\theta(x^i) - y^i)x_j^i$	
4	301-400	$temp^{(4)} = \sum_{i=301}^{400}(h_\theta(x^i) - y^i)x_j^i$	

This same technique can be done automatically across multiple cores by some advanced linear algebra libraries that can parallelize matrix multiplication, that's why vectorization implementation is important.

Application Example: Photo OCR

Problem Description and Pipeline

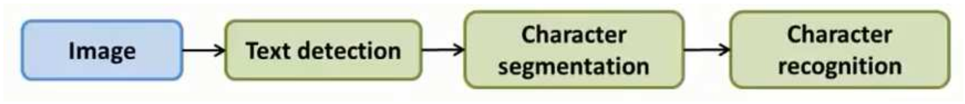
In this module we are going to talk about a Photo OCR application and describe all the staged that are required in order to make it work. A photo OCR application is an application that can read text from a given image; this is more complicated than reading text off a paper scan that is all text.



The steps that are required in order to achieve this are:

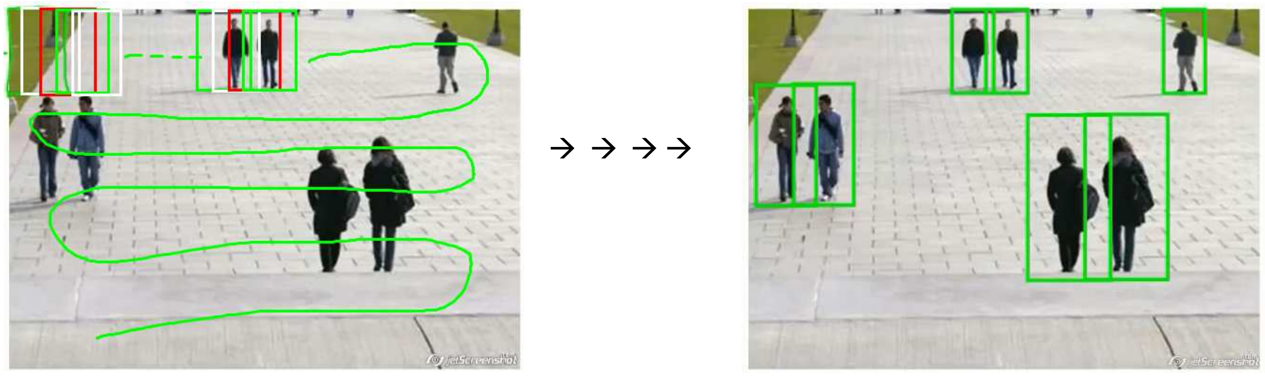
1. Text detection – separate the text from another non-text objects on the picture.
2. Character segmentation – Separate the text into single characters.
3. Character classification – Understand what each character is.

So this problem can be represented as a pipeline of tasks, each can be done by different teams or all be done by a single team, but this pipeline represents the entire machine learning problem.

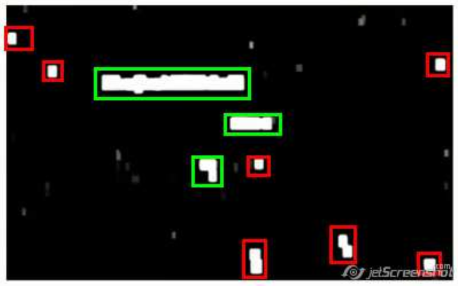


Sliding Windows

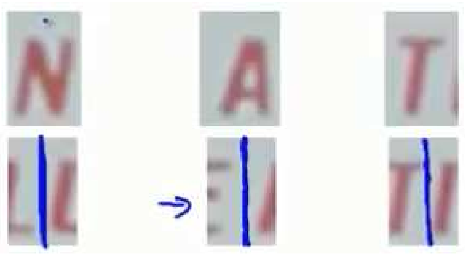
Sliding windows is a technique used to extract objects from an image. Assume we want to identify pedestrians in an image, our first step is to train a model to learn what a pedestrian is, we use some fixed ratio images (as pedestrians are taller than wider) of positive and negative examples and train the model.
Once we are done with that, we take the image we want to process, and start clipping regions with the learnt aspect ratio, and ask the model whether it is a pedestrian or not. We then slide our clipping region and clip again, repeating until we covered the entire image. Once we are done we do it again with a larger rectangle (same aspect ratio but bigger size as pedestrian appear larger when they are close to the camera and small when they are far from the camera). The sliding step is configurable.



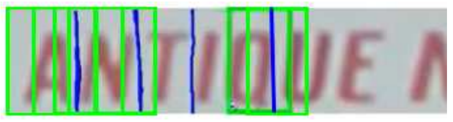
The same technique is done for OCR, we first train a model to differentiate between characters and non-characters (first 2 images).
Then we use the sliding window technique to recognize characters, once we are done with recognizing characters we expand each selection a little bit and merge overlapping selections, so now we supposedly have words. The next step is filtering all the selection whose size ratio (width/height) suggests they are not words (as words are wider than taller). This is the black and white picture, the green rectangles are considered to be words while the red rectangles are ignored.
This was the <u>text detection</u> stage.



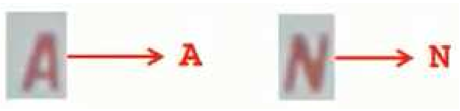
The next step is to train a model to split the words into its characters, this is done by using images of letters (negative examples), and images of the space between two letters.
Once the model is trained we use the sliding window technique again on the regions we recognized to be words, this is a 1D sliding window as the clipping area height is like the image height.



Hopefully now we have the letters spitted.
This was the <u>Character Segmentation</u> stage.



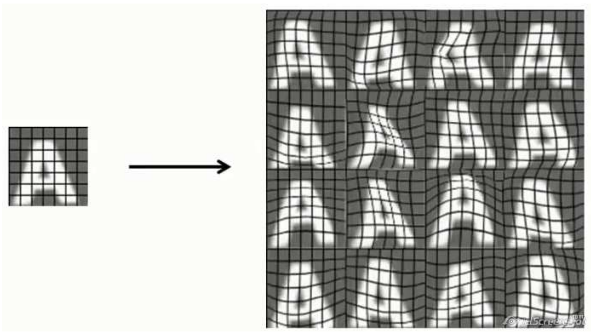
<u>The last stage is the character classification</u> which is predicting each character image what letter it is.



And we are done!!

Getting Lots of Data and Artificial Data

We said that we have a low bias algorithm (good fit) we can do better by training the model on a lot of data, where can we get the data from? Not always we have data available for training, so what we can do is artificially create new data.
For example, in the OCR application we can download different characters fonts from the internet and start make images of the different letters on different random backgrounds, this let us practically create an infinite training set. This is creating data from scratch.



Another approach is taking existing data and "amplify" it, for example we can take the current characters images we have, and apply some image processing to distort the image a little bit, rotate, blur etc; basically any processing that is needed <u>as long as we think we might encounter in such data</u>. So from 1 image we can create many images to learn from.

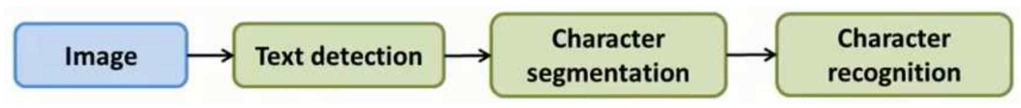
It is important to add only data that we thing is meaningful, adding just noise will not help.

<u>Some points to keep in mind</u>

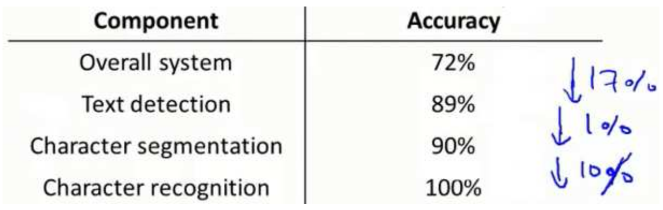
1) Before you generate more data make sure you have a low bias classifier (plot learning curve), if this is not the case add more features/hidden layer in NN. Only when you have a low bias classifier more data will help, otherwise it is time not well spent.
2) "How much work would it be to get 10x as much data as we currently have?" – Sometimes the answer for this question is "not much", and when actually thinking about how to get more data you can find that you can get 10x more data just in few days, this can boost your machine learning algorithm. How to get more data?
   a. Artificial data synthesis – this is what we talked about earlier.
   b. Collect / label data yourself – Think how much time it takes to label 1 example, and calculate how much time/effort it would be to manually label more data.
   c. "Crowd Source" (e.g. Amazon Mechanical Turk) – Crowd source is hiring cheap working hands online, this might not be the best solution, but it depends on the task, if it is a simple task hiring much more hands could be quite cheap.

Ceiling Analysis: What part of the pipeline to work next?

In machine learning applications there are usually several steps we have to go thru in order to make the final prediction, this is our pipeline. How can we know what part of the pipeline is most worth to spend time on improving it? This is done by Ceiling Analysis.



Back to our OCR application, we had our pipeline as shown above. Each part of the pipeline is the input for the next part, by ceiling analysis we take one part and make it "perfect", meaning we plug in the true results manually, so the next part would be fed with perfect data, and we then see how that improved our overall performance.



Component	Accuracy	
Overall system	72%	↓17%
Text detection	89%	↓1%
Character segmentation	90%	↓10%
Character recognition	100%	

In our example we have overall performance of 72%, then we took the "text detection" step and made it perfect, meaning we manually identified where the text is and that's what we gave as input to the "character segmentation" step. We see that the overall accuracy jumped by 17% to 89%. Meaning that we might want to put more efforts on improving the "text detection" step, it is quite far from its full potential (its "ceiling").
Then we made the "character segmentation" step perfect by manually identifying the characters and gave it to the next step. Our overall performance increased by only 1%, meaning that our current implementation of "character segmentation" is good enough, even if we improve it to be 100% correct our overall performance would increase by only 1%.
And finally we see that if we make the "character recognition" step perfect we have an overall improvement potential of 10%.
So using ceiling analysis we see that the effort should be put on Text Detection and Character Recognition.

---------- THE END ----------