# *Machine Learning - Neural Networks*

# *Handbook*

*thanks to Andrew Ng*

# *Preface*

This is a handbook, where the basic algorithms and notions around Machine Learning are analysed. I was motivated to write it after attending the online course of Machine Learning organized by Stanford University. The initial attempt was to write a brief manual, only including the equations and the algorithms needed for the most basic algorithms of Machine Learning, so that one familiar with Machine Learning concepts can easily trace back. However, in the meanwhile, I realised that in this way, the book would be somewhat useless. So, the motivation behind this manual has been slightly changed as the time went by. I ended up including some short introductory descriptions of the algorithms. However, I tried to stay as brief as possible, while in the same time format and index the algorithms and the equations, so that someone can easily locate them.

To sum up, this manual can be used by someone with no knowledge around Machine Learning (perhaps with a minimal knowledge of numerical analysis) in order to get a fast glimpse of the field of Machine Learning and see some applications. Nevertheless, it can also be used by someone having previous experience in Machine Learning, but just not remembering exactly some algorithms and willing just to refresh the formal definitions.

Raptis Dimos

23/09/2015

## *Table of Contents*

# *1. Introduction*

Machine Learning is used nowadays in many applications. Some examples of applications using Machine Learning are the spamming email filters, the search engines, the photo tagging applications etc.

Machine Learning grew out of Artificial Intelligence (AI). However, it was designed as a new capability for computers. Machine Learning has specifically high usage in applications that cannot be programmed by hand (such as helicopter autonomous driving, computer vision etc.). It has also high usage in Database Mining applications (i.e. medical records, biology).
They are also used for self-customizing programs, like Amazon recommender systems.

The most common definition of Machine Learning is the following :

***Machine learning is a type of artificial intelligence (AI) that provides computers with the ability to learn without being explicitly programmed. Machine learning focuses on the development of computer programs that can teach themselves to grow and change when exposed to new data.***

Another thing that is true about Machine Learning is that "***it's not a matter of who has the better algorithm, but who was more data***".This is due to the fact that the field of Machine Learning has significantly evolved during the last years and the most known and basic algorithms exhibit satisfactory performance and accuracy. So, the one that has the biggest quantity of data in order to train efficiently the algorithm will be the one that will have the highest accuracy from his algorithm.

## *1.1 Supervised vs Unsupervised Learning*

The 2 main categories of Machine Learning tasks are :

- Supervised Learning
- Unsupervised Learning

In ***supervised Learning***, the computer is presented with example inputs ($x_i$) and their desired outputs ($y_i$), given by a "teacher", and the goal is to learn a general rule that maps inputs to outputs.

In ***unsupervised Learning***, no labels ($y_i$) are given to the learning algorithm, leaving it on its own to find structure in its input ($x_i$). Unsupervised learning can be a goal in itself (discovering hidden patterns in data).

There is also one last category called Reinforcement Learning, that will be not extensively covered here.

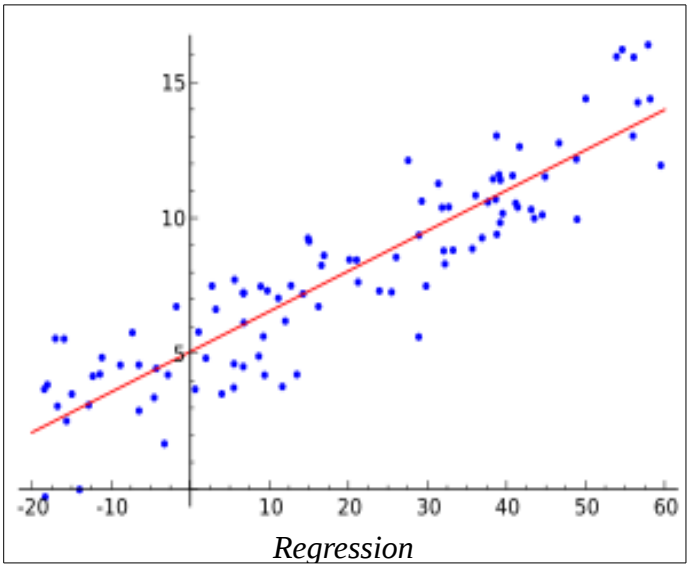
In ***Reinforcement Learning***, a computer program interacts with a dynamic environment in which it must perform a certain goal (such as driving a vehicle), without a teacher explicitly telling it whether it has come close to its goal or not. Another example is learning to play a game by playing against an opponent. Reinforcement learning differs from the supervised learning problem in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected.

## *Supervised Learning*

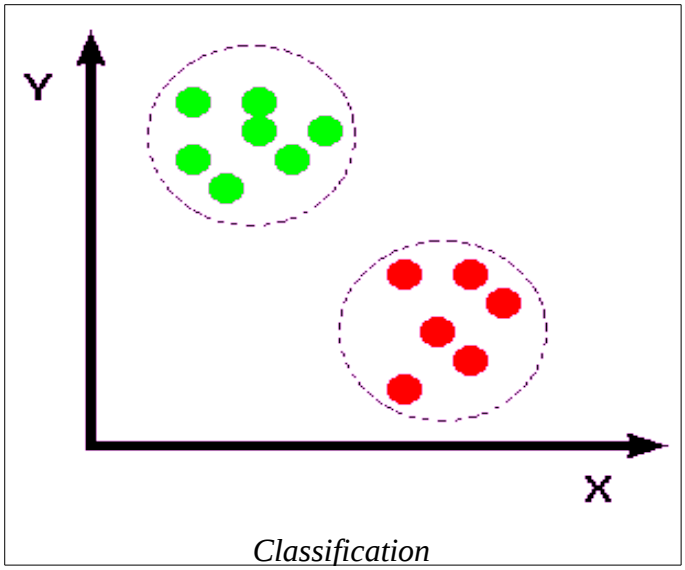
A little deeper, supervised learning is divided into 2 main categories of problems :

- Regression
- Classification

***Regression*** predicts continuous valued output (i.e. the price of a product)
***Classification*** predicts discrete valued output, aka if the sample belongs to a specific class (i.e. if cancer is benign or malignant).



*Regression*



*Classification*

Some common approaches towards supervised learning are :

- Regression Analysis (regression)
- Artificial Neural Networks (both regression and classification)
- Decision Trees (both regression and classification)
- K-nearest neighbours (both regression and classification)
- Support Vector Machines – SVMs (both regression and classification)

and many more …

## *Unsupervised Learning*

In machine learning, the problem of unsupervised learning is that of trying to find ***hidden structure in unlabeled data***. Since the examples given to the learner are unlabeled, there is no error or reward signal to evaluate a potential solution. This distinguishes unsupervised learning from supervised learning and reinforcement learning.

Unsupervised learning is closely related to the problem of density estimation in statistics. However unsupervised learning also encompasses many other techniques that seek to summarize and explain key features of the data. Many methods employed in unsupervised learning are based on data mining methods used to preprocess data.

Approaches to unsupervised Learning include :

- clustering (k-means, mixture models, hierarchical clustering)
- Approaches for learning latent variable models
  - Expectation – Maximization algorithm
  - Method of moments
  - Blind signal separation techniques
    - Principal Component Analysis (PCA)
    - Independent Component Analysis
    - Non-negative matrix factorization
    - Singular Value Decomposition (used to separate 1 signal from mixture of signals)

## *1.2 Linear Algebra Review*

Knowledge of Linear Algebra is really important for Machine Learning.

Basic knowledge around the following topics is necessary :

- Matrices and Vectors
- Addition and Scalar Multiplication
- Matrix Vector Multiplication
- Matrix Matrix Multiplication
- Matrix Multiplication properties
- Inverse and Transpose

Note that there are some cases where, in the normal equation (the one that minimizes the sum of squares of errors) the array $X^TX$ is non invertible. This is the problem of Normal Equation Non-invertibility. It is probably due to :

- redundant features (linearly dependent) → delete redundant features
- too many features (m < n) → delete some features, or regularization

However, in cases of non-invertibility you can use the pseudo-inverse matrix instead and it will calculate the right value for theta. There is a suitable function in Octave/Matlab for the calculation of pseudo-inverse matrix, called **pinv().**

## *1.3 Vectorization*

Whether you are using Octave, Matlab, Python or any other language, all these languages have built into them or have numerical linear algebra libraries. Those are usually highly optimized and specialized in numerical computing. And when you implement Machine Learning algorithms, if you want to take full advantage of these linear algebra libraries, you should call their functions instead of writing custom code to do the same things. More specifically, you should try the so-called vectorized implementations of your algorithms, so that instead of for-loops matrices are used. In this way, you can take better advantage of any parallel hardware your computer may have. And you also end up with less code this way.

An example of vectorization is the following, where θ,x are assumed to be column-vectors :

$$\boxed{h_\theta(x)=\sum_{j=0}^{n}\theta_j x_j} \quad \rightarrow \quad \boxed{h_\theta(x)=\theta^T x}$$

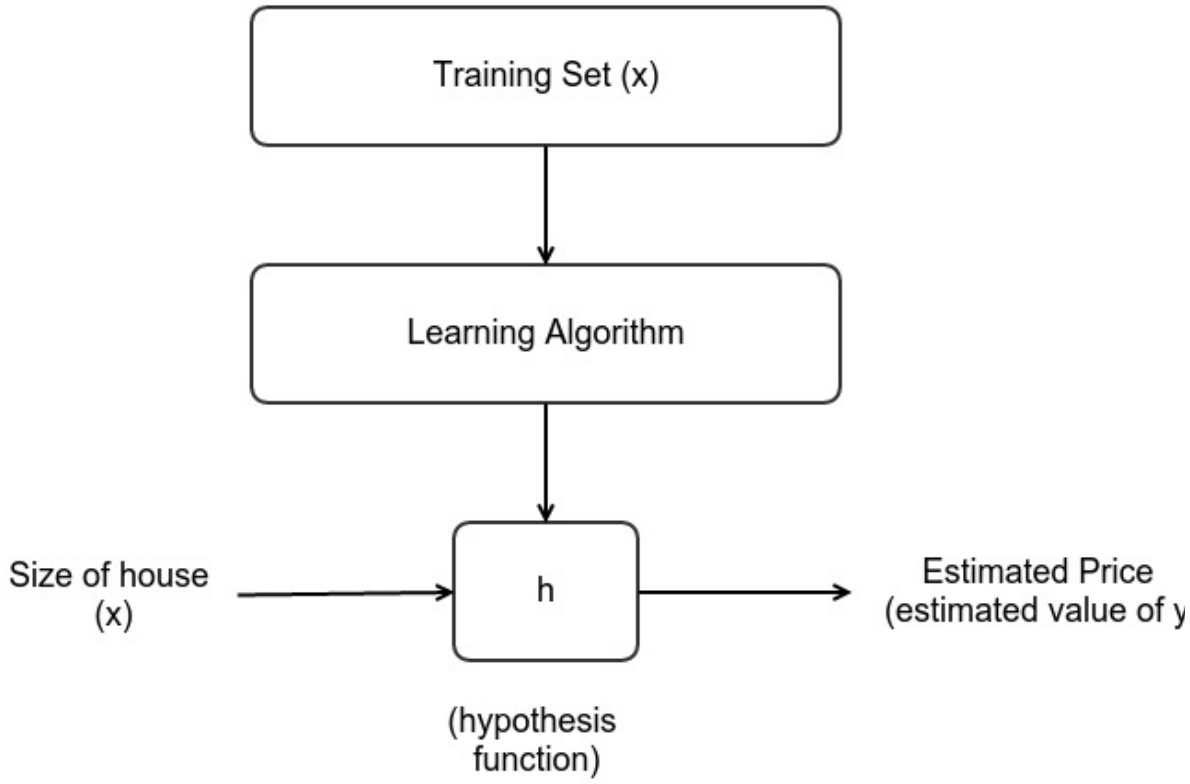
## *2. Linear Regresion with 1 variable*

Linear Regression is an approach for modeling between a scalar dependent variable ***y*** and one or more explanatory variables (or independent variables) denoted ***x***. The case of one explanatory variable is called simple linear regression (or linear regression with 1 variable – *univariate linear regression*). The result of linear regression is an equation that given x, can "predict" the values of y for this specific x.

An example of linear regression problem is the case where we are given several houses sizes along with their prices and we want to find an equation that will predict a price of a new house by its size.

## *2.1 Model and Cost Function*

For each linear regression problem we have the following notations :
x : input values
y : output values
h : hypothesis function (prediction equation)
m : number of training examples n : dimension of variables (here 1)



The hypothesis function is the following :

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

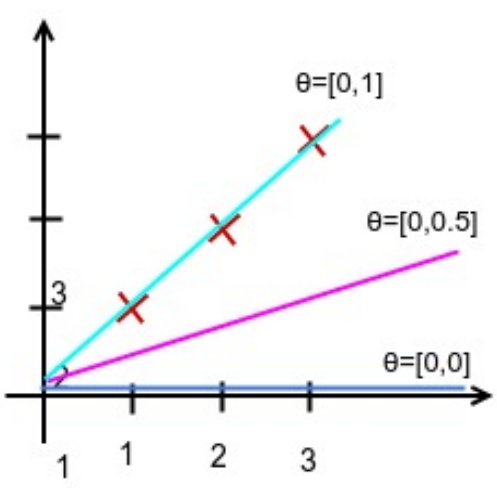
Depending on the parameters $\theta_0$ and $\theta_1$ , there are various linear equations that can be used with varying distances from the real values y. From our intuition, we understand that we want to find the equation that minimizes the "average distance" from all the given points $(x_i, y_i)$ . So, we have to define a ***cost function***, which is the following :

$$J(\theta_0,\theta 1)=\frac{1}{2m}\sum_{j=1}^{m}\left(h_\theta(x_i)-y_i\right)^2$$

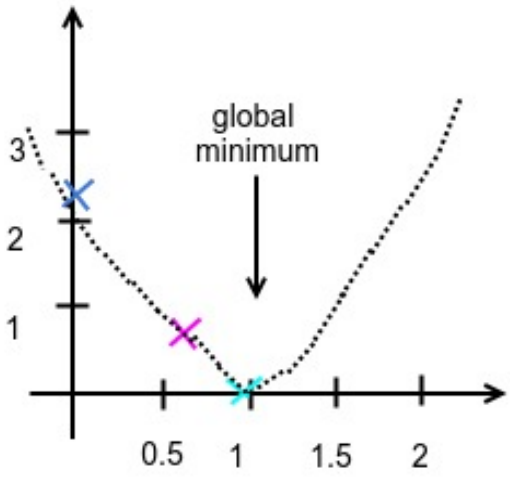
And the goal of the linear regression is the following :

$$\underset{\theta_0,\theta_1}{minimize}\, J(\theta_0,\theta_1)$$

For linear regression, the cost function will be a convex function. Thus, there will be a unique global minimum. This means that the solution that will be found will be the right one always.



various hypotheses



Cost Function

## *2.2 Parameter Learning*

So, the goal of linear regression is to minimize the cost function in order to find the optimum values for thetas (also called parameters).

This can be done using analytical processes. For example, we can use the first degree derivative of the cost function to find the place of minimization. Those processes are commonly used when we have low-order regressions (small number of independent variables). For instance, it is used for univariate linear regression, since it is quite fast. However, in Machine Learning we usually have to deal with many more variables and bigger problems. In this cases, such methods are really slow. In some cases, a lot of matrices calculations are required, raising the complexity in $O(n^3)$.

So, in Machine Learning there are other processes are used in regression, and generally in cases where we have to minimize a function. One of the most commonly used algorithms for minimizing a function is ***Gradient Descent.*** This is the algorithm that we will cover in this chapter.

The main idea behind Gradient Descent is the following :

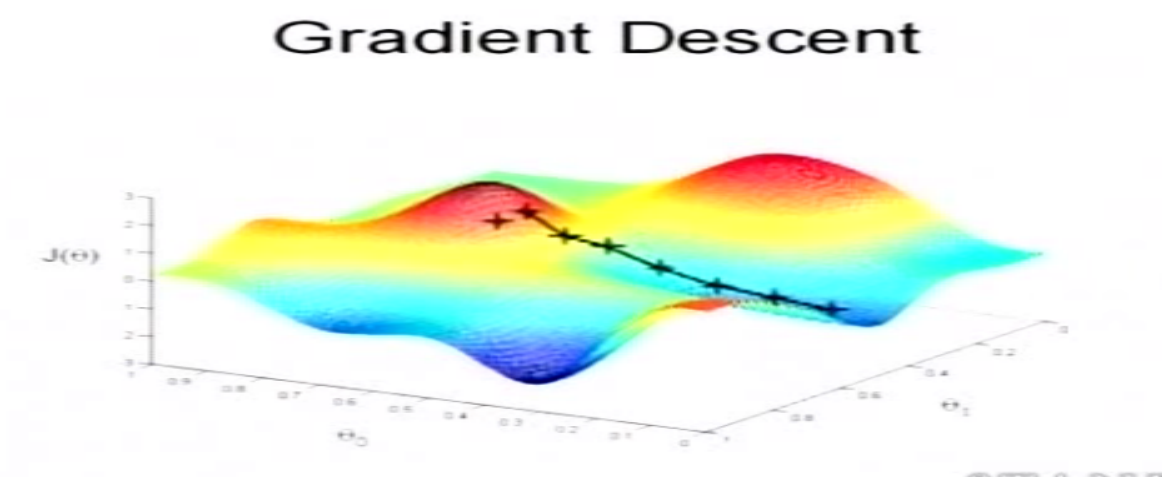
I have some function $J(\theta_0,\theta_1)$ (or generally $J(\theta_0,\theta_1,\theta_2,\dots,\theta_n)$ )

I want to $\min_{\theta_0,\theta_1} J(\theta_0,\theta_1)$ (or generally $\min_{\theta_0,\theta_1,\dots,\theta_n} J(\theta_0,\theta_1,\theta_2,\dots,\theta_n)$ )

Outline :

- Start with some random parameters $\theta_0,\theta_1$
- Keep changing the parameters to further reduce $J(\theta_0,\theta_1)$

until we hopefully end up at a minimum



The formal definition of the Gradient Descent Algorithm is the following :

*repeat until convergence*

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

*for all j*

The value ***α*** is called ***learning rate*** and corresponds to the size of the steps that the algorithm of Gradient Descent will take on each iteration. The selection of this value will have to be done after thought. The process for selecting the value of alpha rate and side effects will be discussed later.

However, in order for the algorithm to perform as expected, we have to be aware of one more small detail. The updates to thetas parameters should be done simultaneously.

So, you can see in the following equations what to avoid when implementing the algorithm :

| ***Correct Simultaneous Update*** : | ***Incorrect*** : |
|---|---|
| $temp0 = theta_0 - a \frac{\partial}{\partial theta_0} J(\theta_0, \theta_1)$ | $temp0 = theta_0 - a \frac{\partial}{\partial theta_0} J(\theta_0, \theta_1)$ |
| $temp1 = theta_1 - a \frac{\partial}{\partial theta_1} J(\theta_0, \theta_1)$ | $theta_0 = temp0$ |
| $theta_0 = temp0$ | $temp1 = theta_1 - a \frac{\partial}{\partial theta_1} J(\theta_0, \theta_1)$ |
| $theta_1 = temp1$ | $theta_1 = temp1$ |

Note : There are cases where, the function we want to minimize is not a convex function (i.e. non-linear regression). In this cases, it's not guaranteed that the algorithm will converge to a global minimum. There is also a possibility that it will converge to a local minimum.

## *Gradient Descent Intuition*

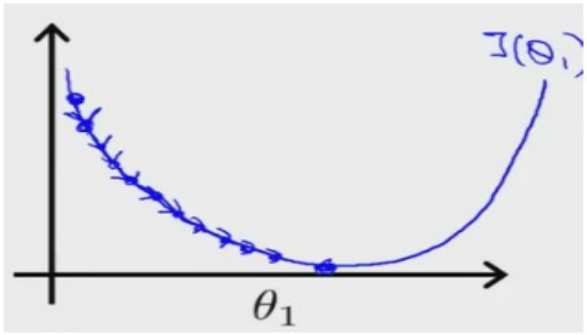
Now, let's see what is the logic behind the equation shown before that is used for Gradient Descent. Looking at the equation for each theta parameters, we conclude that :

- if partial derivative is positive, then the theta parameter will decrease (move forward)
- if partial derivative is negative, then the theta parameter will increase (move backwards)

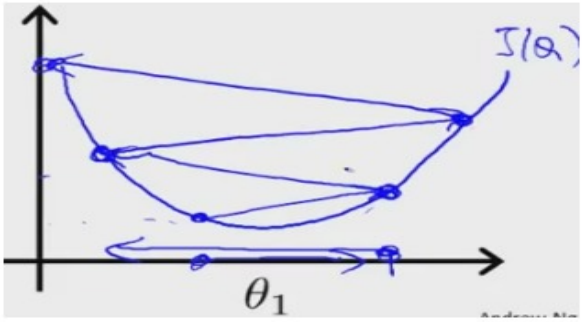
Thus, we understand that theta parameter will be "moved" towards the direction in which the partial derivative will be negative (aka the cost function will decrease).

The learning rate α determines how big steps the algorithm will take towards the minimum. So, if we select a small value for α, the algorithm will converge slowly. On the contrary, if we select a large value for α, the algorithm might converge faster, but there is also a possibility that the algorithm will never converge or it might also diverge.

If α is too small, gradient descent can be slow




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



One alternative approach would be to have a quite big learning rate initially, so that the algorithm will make some big steps in the beginning. But we have to gradually decrease the learning rate, as we approach the minimum, so as to be sure that the algorithm will finally converge.

Note: With a fixed (suitable valued) learning rate alpha, gradient descent can converge to a local minimum. This is because the derivative/slope will automatically decrease when it approach a local minimum. So, there is no need to decrease alpha over time.

## *Gradient Descent for Linear Regression*

The formal definition of Gradient Descent Algorithm for Linear Regression is the following (after having calculated the derivatives):

$$
\begin{aligned}
&repeat\ until\ convergence\{ \\
&theta_0 = theta_0 - a\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i) \\
&theta_1 = theta_1 - a\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x_i) - y_i)x_i \\
&\}
\end{aligned}
$$

where theta parameters should be updated simultaneously !

# *3. Linear Regression with multiple variables*

## *3.1 Multivariate Linear Regression*

The linear regression, when having not only one but multiple variables (called features), is called multivariate Linear Regression. The notation in this generic case is the following :

n : number of features

$x^{(i)}$ : input features of i$^{th}$ training example

$x_j^{(i)}$ : value of feature j in i$^{th}$ training example

For example, if we might have more features for the example of houses :

| Size (feet$^2$) | Number of bedrooms | Number of floors | Age of home (years) | sPrice (1000$) |
|---|---|---|---|---|
| 2104 | 5 | 1 | 45 | 460 |
| 1416 | 3 | 2 | 40 | 232 |
| 1534 | 3 | 2 | 30 | 315 |
| 852 | 2 | 1 | 36 | 178 |
| … | … | … | … | … |

$$x^{(2)} = [1416\ 3\ 2\ 40]$$

$$x_3^{(2)} = 2$$

Furthermore, the hypothesis function will have the following form :

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

For convenience of notation (and easier vectorization), we define $x_0 = 1$ ( $x_0^{(i)} = 1$ )

So, the vectorized form of multivariate linear regression is the following :

$$X = \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_n \end{bmatrix} \quad \Theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \dots \\ \theta_n \end{bmatrix}$$

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \dots + \theta_n x_n = \Theta^T X$$

The new generic cost function is the following :

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

And the new Gradiend Descent generic algorithm is the following :

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

$$\downarrow$$

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

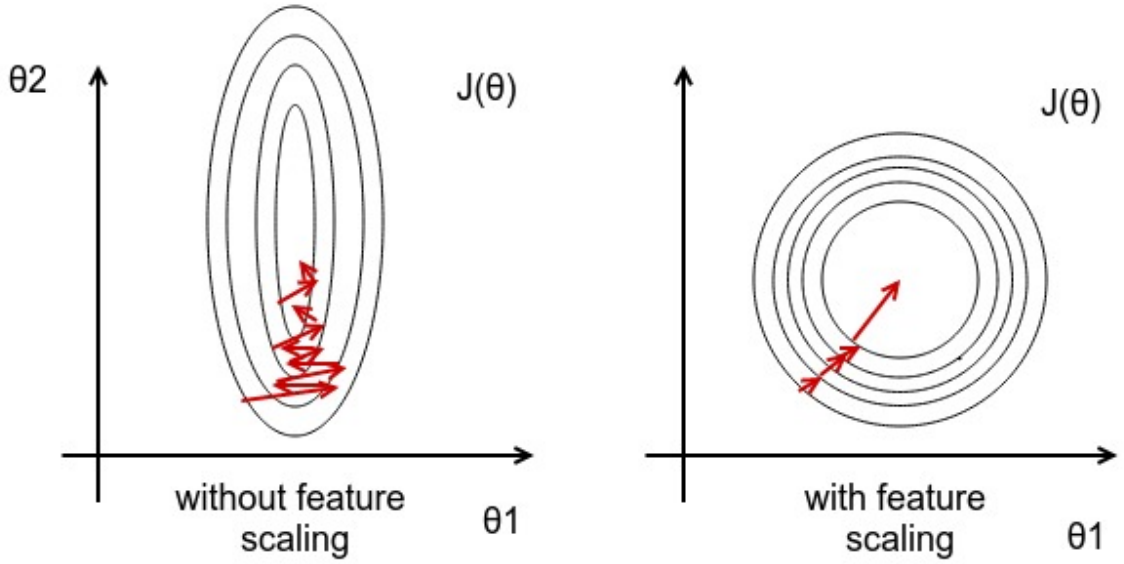
$$\theta_1 = \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} \left( h_\theta(x^{(i)}) - y^{(i)} \right) x_1^{(i)}$$

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

$$\dots$$

### *3.1.1 Feature Scaling*

Feature Scaling is a method used to make sure that features are in the same scale (range of values). In data processing, it is also known as *data normalization* and is generally performed during the data preprocessing step. Note that when features are in different scales, this might provoke a worse performance in gradient descent.




So, the target of feature scaling is to get every feature into approximately a $-1 \leqslant x_i \leqslant 1$

To accomplish it, we simply divide by the range of values of the specific feature. However, when we apply feature scaling could also benefit by making sure that features have approximately zero mean. This procedure is called ***mean normalization*** and is made by simply subtracting the mean of each feature from each input value for this feature.

As a result, there are 2 basic methods for feature scaling :

- rescaling, that makes features' scale range from -1 to 1 (and have zero mean)
- standardization, that make the values of each feature have zero mean and unit variance

The equations for both methods are the following :

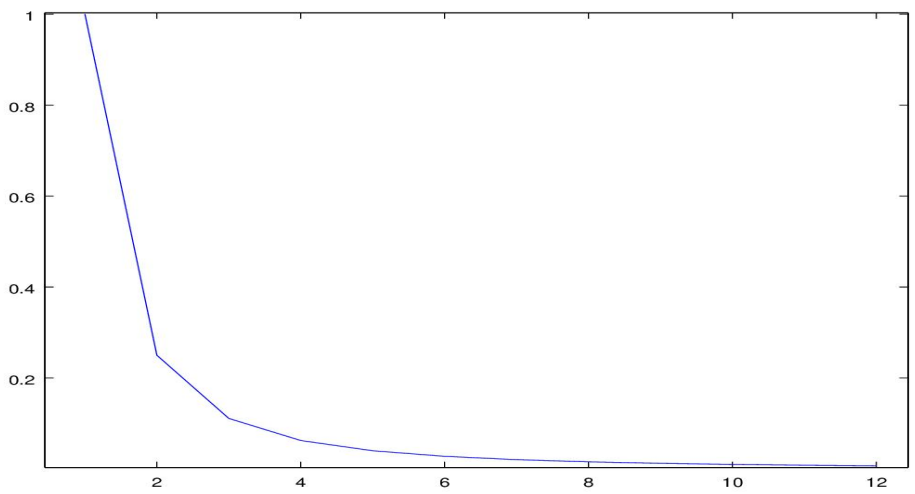
| *Rescaling* : | *Standardization* : |
|---|---|
| $x_i = \frac{(x_i - min\{x_i\})}{max\{x_i\} - min\{x_i\}}$ | $x_i = \frac{(x_i - \mu_i)}{\sigma_i}$, $\sigma_i$ : *standard deviation of* $i^{th}$ *feature* |

## *3.1.2 Learning Rate*

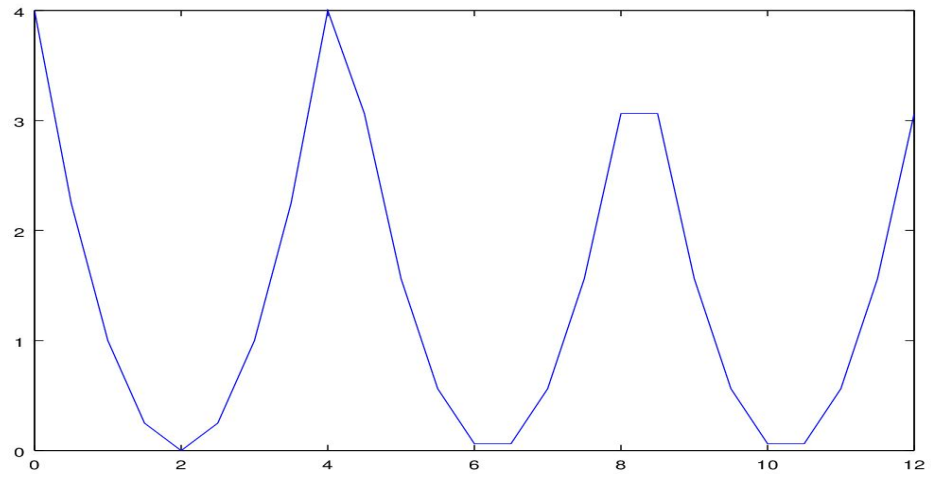
There are 2 topics regarding learning rate that should be discussed :

- Debugging, how to make sure gradient descent is working properly
- how to choose a correct learning rate

In order to debug our gradient descent algorithm, we have to make a plot of J(θ) versus the number of iterations. If the algorithm is converging, then the plot should be steadily decreasing. In other cases, the algorithm is not converging and we should select a smaller learning rate α.




*Gradient descent converging - correct α*




*Gradient Descent not converging - should select smaller α*

In case of convergence, we can see that the curve is becoming horizontal in the end, where the algorithm is near the minimum. Thus, we can check the level of convergence of the algorithm by checking how much the J(θ) is currently decreasing.
So, an example automatic convergence test could be the following :

" *Declare convergence if J(θ) decreases by less than $10^{-3}$ in one iteration* "

***Summary :***

- If α is too small, slow convergence
- If α is too large, J(θ) may not decrease on every iteration ; may not converge

To choose a, we can try various values by multiplying by 3, as below :

..., 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, ...

## *3.2 Features and Polynomial Regression*

There are cases, where the data cannot be fit with a linear regression in a satisfactory level. In this cases, polynomial regression can be used. By watching various polynomial equations, you can get the intuition that by using higher order polynomials we can theoretically fit any possible data, since extremely high order polynomials can describe extremely complicated plots.

For example, in the example with the houses, the price of the house might be better predicted by the equation

$$h_\theta(x)=\theta_0+\theta_1 x+\theta_2 x^2+\theta_3 x^3$$

rather than the linear one :

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

Note: In case of high order polynomials, the utilization of feature scaling becomes more important.

## *3.3 Normal Equation*

Given a matrix equation $Ax=b$, the normal equation is the one that minimizes the sum of the square differences between the left and the right sides :

$A^T Ax=A^T b \rightarrow x=A^T b$

It is called a normal equation because *b-Ax* is normal to the range of *A*.

In the case of regression, we have the following equations :

$$\boldsymbol{m\ examples}\left(x^{(1)},y^{(1)}\right),\left(x^{(2)},y^{(2)}\right),\ldots,\left(x^{(m)},y^{(m)}\right);\boldsymbol{n\ features}$$

$$x^{(i)}=\begin{bmatrix}x_0^{(i)}\\x_1^{(i)}\\\ldots\\x_n^{(i)}\end{bmatrix}\quad X_{(\text{design matrix})}=\begin{bmatrix}\cdots\left(x^{(0)}\right)^T\cdots\\\cdots\left(x^{(1)}\right)^T\cdots\\\ldots\\\cdots\left(x^{(m)}\right)^T\cdots\end{bmatrix},Y=\begin{bmatrix}y_0^{(i)}\\y_1^{(i)}\\\ldots\\y_n^{(i)}\end{bmatrix}$$

$$X^T X\theta=X^T Y$$

Now, assuming that the matrix $X^T X$ is invertible, we can multiply both sides by $(X^T X)^{-1}$ and get :

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

In Octave, this is computed with ***pinv(X'*X)*X'*Y***

In cases, where the normal equation is used to find the theta parameter, feature scaling does not provided any benefits in performance.

| *Gradient Descent* | *Normal Equation* |
|---|---|
| **Need to choose α** | **Don't need to choose α** |
| **Needs many iterations** | **Don't need to iterate** |
| **Works well even when n is large** | **Slow if n is very large**<br>**Needs to compute $(X^T X)^{-1}$ that has complexity O($n^3$)** |

*Comparison of methods*

## *4. Logistic Regression*

Logistic Regression is a method for classifying data into discrete outcomes. You will see that, in fact, logistic regression is a more specific version of linear regression used to classify discrete data (instead of continuous valued data).

## *4.1 Classification and Representation*

In machine learning and statistics, classification is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is know.

In the terminology of machine learning,[1] classification is considered an instance of supervised learning, i.e. learning where a training set of correctly identified observations is available. The corresponding unsupervised procedure is known as clustering, and involves grouping data into categories based on some measure of inherent similarity or distance.

Some examples of classification problems are the following :

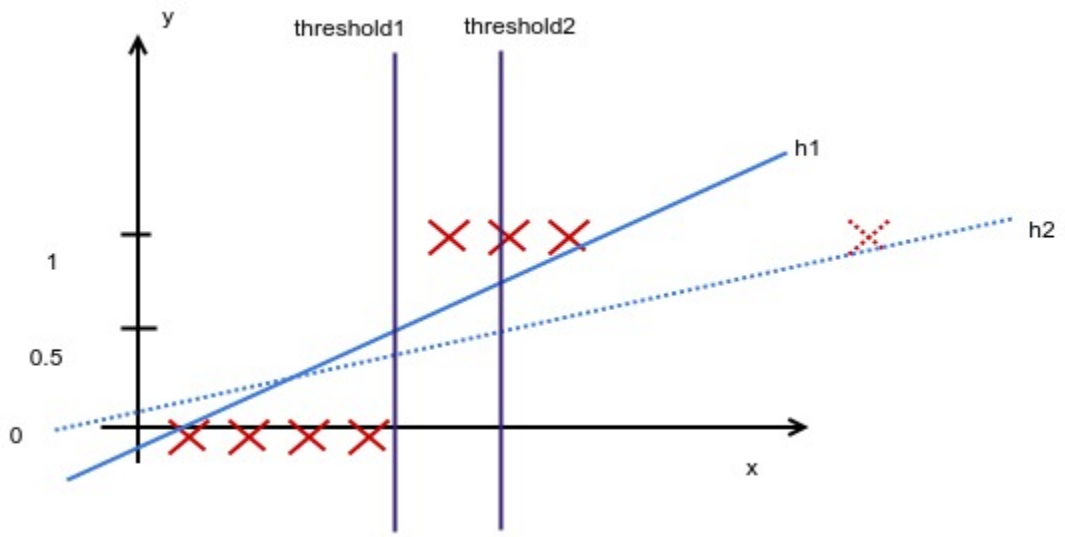
- Email : spam/not spam
- Online transactions : fraudulent (Yes/No)
- Tumor : Malignant Benign

So, in this case the output is a boolean value

$$y \in \{0,1\}, where\ 0 : negative\ class,\ 1 : positive\ class$$

However, there are cases, where there are more than 2 classes amongst which we have to distinguish a measurement. This case is called multiclass classification and we will discuss it later.

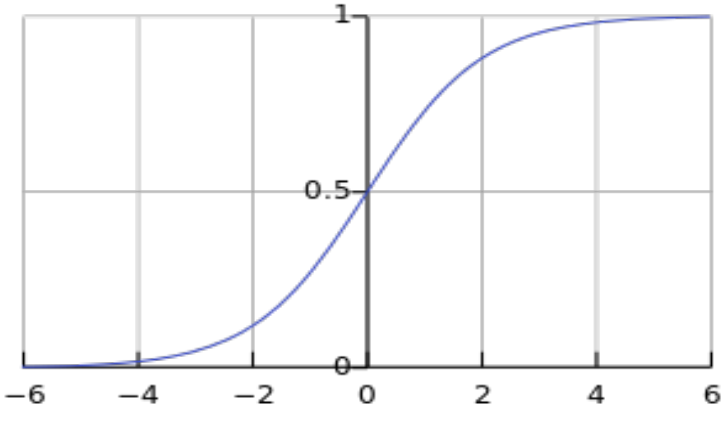
We could use linear regression to fit the data and then depending on a threshold value of the hypothesis function, predict accordingly if the value if 0 or 1. However, this process would be hurt by outlying measurements that would affect the threshold value spoiling the hypothesis function with no reason.




Another strange thing is that in the previous case, while y can take only values 0 or 1, the hypothesis function can take values bigger than 1 and less than 0. This is somewhat strange and we would like a hypothesis function that takes values between 0-1. There is an algorithm called logistic regression that is used for classification problems (ignore the "regression" since it is due to historical reasons) and uses such an hypothesis. We will analyse this algorithm now.

As discussed before, in the logistic regression model, we want $0 \leq h_\theta(x) \leq 1$
For this reason, we will use the sigmoid function :

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




So, the hypothesis function in logistic regression now becomes :

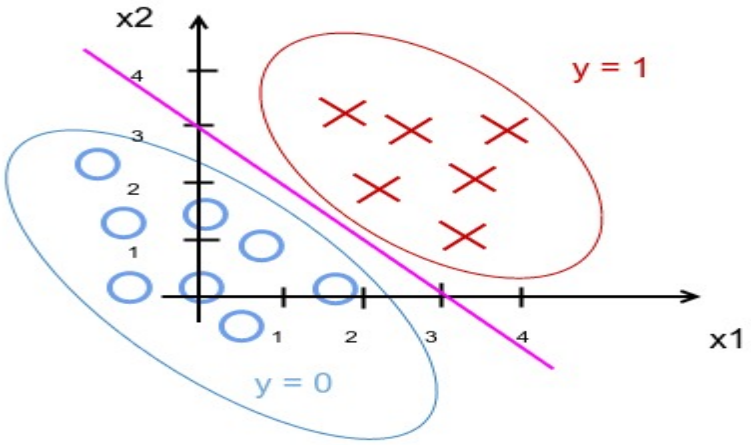
$$Linear - Polynomial\ Regression \rightarrow Logistic\ Regression$$
$$h_\theta(x) = \theta^T X \rightarrow h_\theta(x) = g(\theta^T X)$$

The interpretation of hypothesis output is that $h_\theta(x)$ is the estimated probability that y = 1 on input x. So, $h_\theta(x)=0.7$ would mean that there is a 70% chance of the tumor to be malignant. In a more formal definition :

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

## *4.2 Decision Boundary*

The Decision Boundary (or Decision Surface) is a hypersurface that partitions the underlying vector space into 2 sets, one for each class. For instance, assume we have the following data :




where the hypothesis function is :

$$h_\theta(x)=g(\theta_0+\theta_1 x_1+\theta_2 x_2)$$
$$h_\theta(x)=g(-3+1x_1+1x_2)$$

then the decision boundary will be the equation :

$$x_1+x_2=3$$

Note: there can also be non-linear decision boundaries, but they can only be represented by non-linear (higher order) polynomials. This means that the hypothesis function will contain higher order powers of features.

## *4.3 Logistic Regression Model*

If we go back to regression cost function we will see the equation :

$$J(\theta_0,\theta 1)=\frac{1}{2m}\sum_{j=1}^{m}(h_\theta(x_i)-y_i)^2$$

which can be converted into the following equation :

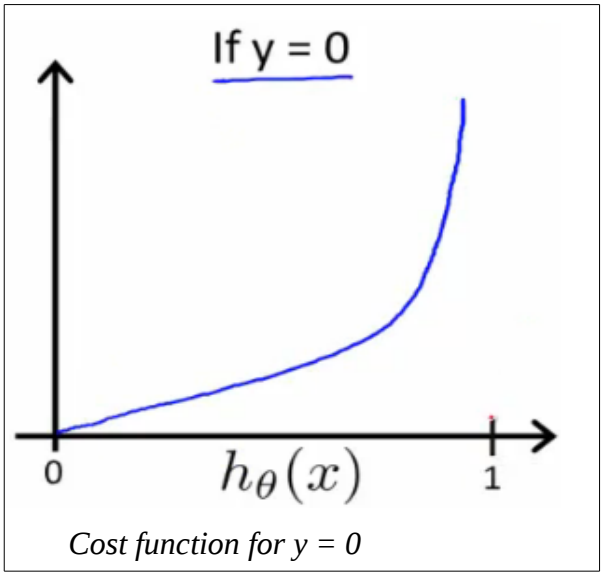
$$J(\theta_0,\theta_1)=\frac{1}{2m}\sum_{j=1}^{m} Cost\left(h_\theta(x^{(i)}),y^{(i)}\right)$$

$$where\ Cost\left(h_\theta(x^{(i)}),y^{(i)}\right)=\left(h_\theta(x^{(i)})-y^{(i)}\right)^2$$

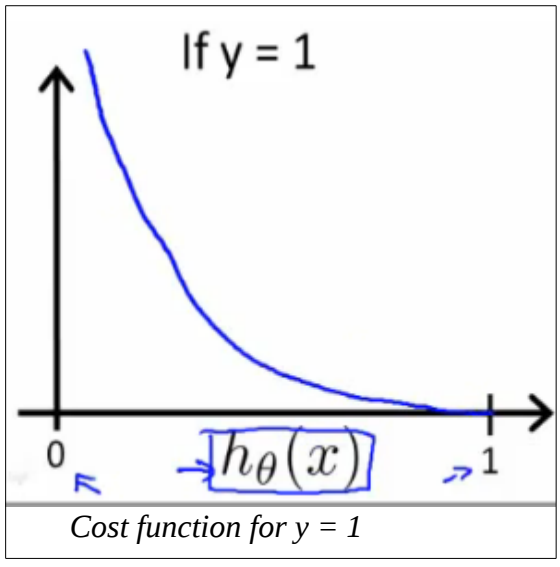
However, if we use this function for logistic regression, this is a non convex function, which means that it does not have a unique global minimum, but it has several local minimums. This is due to the non-linearity of the hypothesis function (that was caused by the sigmoid function). So, if we were to use this cost function for gradient descent, the algorithm could work, but we could not be sure that each execution of the algorithm will end up in a global minimum.

So, we would like a different convex Cost() function, that will be the following:

$$Cost\left(h_\theta(x),y\right)=\begin{cases}-\log\left(h_\theta(x)\right), \text{if } y=1\\ -\log\left(1-h_\theta(x)\right), \text{if } y=0\end{cases}$$



*Cost function for y = 0*



*Cost function for y = 1*

So, when we're right, the cost function is 0. Else, the cost function slowly increases as we become more wrong, reaching infinity if we are totally wrong (aka predict 0 with 100% possibility and it was 1 or inverse).

With this particular cost function, J(θ) is going to be convex and avoid local

minimum. The updated J(θ) function is the following :

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

And the Gradient Descent algorithm will be the following :

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

where we remind that :

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

## *4.4 Advanced Optimizations*

Besides Gradient Descent, there are also other algorithms used for minimizing a function that are built into libraries. Those algorithms are optimized and we could benefit from using them instead of Gradient Descent. One good think is that they have the same philosophy with Gradient Descent. They only need to be provided with the cost function and the derivatives of it.

Given θ, we have code that can compute
- $J(\theta)$
- $\frac{\partial}{\partial\theta_j}J(\theta)$ for (j = 0, 1, ..., n)

| Optimization Algorithms | Advantages - Disadvantages |
|---|---|
| Conjugate Gradient | No need to manually pick α |
| BFGS | Often faster than gradient descent |
| L-BFGS | More complex |

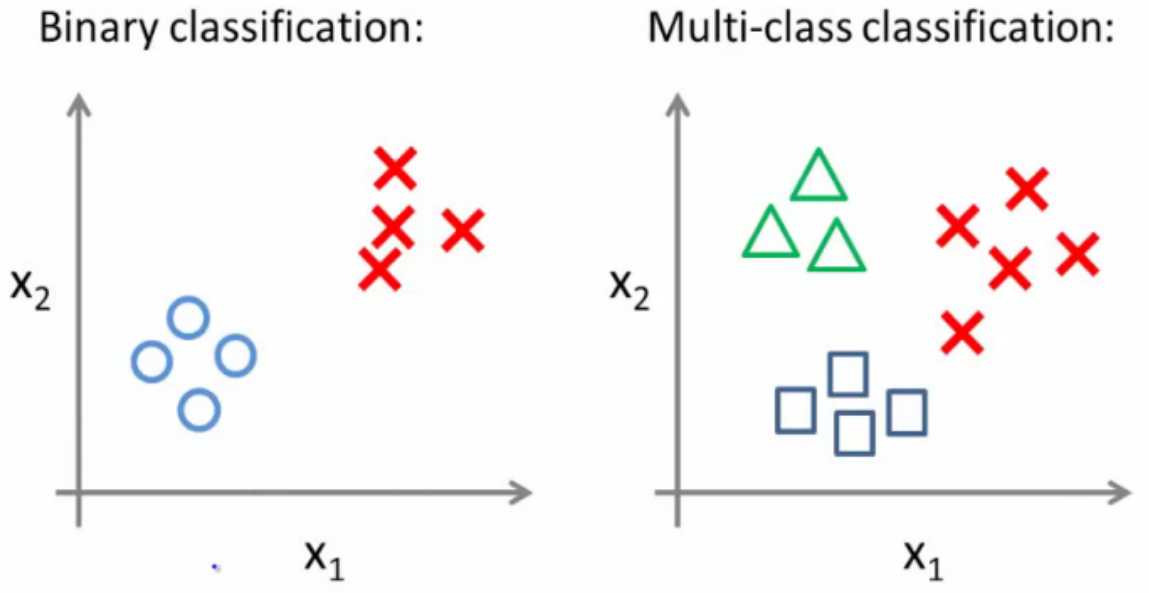
For example, in Octave you can use the fminunc function to minimize a function after having defined a function that calculates the cost function and the partial derivatives and providing an initial vector for theta parameters.

```
options = optimset('GradObj', 'on', 'MaxIter', '100');
initialTheta = zeros(2,1) ;  %assume dimension 2

function [jVal, gradient] = costFunction(theta)
...

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

## *4.5 Multiclass Classification : One-vs-All*

In machine learning, multiclass or multinomial classification is the problem of classifying instances into one of the more than two classes (classifying instances into one of the two classes is called binary classification).

Here, we will see an algorithm called One-vs-All that is used for this kind of problems. This algorithm makes use of the algorithm of logistic regression presented previously.




The idea behind this algorithm is that a multi-class classification problem with 3 classes (as the one presented above) can be solved by solving 3 separate binaries classification problems, where in each problem we apply logistic regression to classify between the one selected class and all the other together. Then, the classification of a new input is done by getting the biggest possibility amongst the different classes. The idea is the following :

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

On a new input x, to make a prediction, pick the class i that maximizes

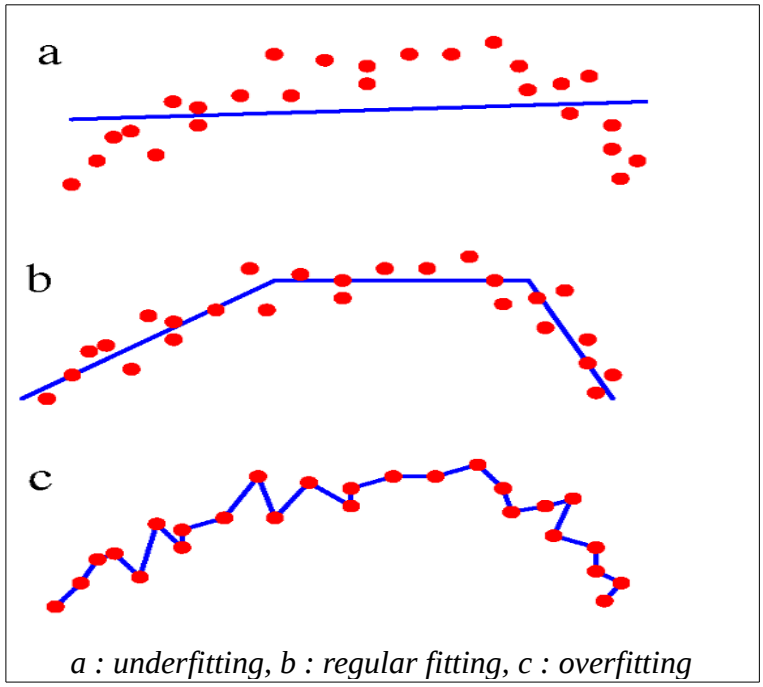
$$\max_i h_\theta^{(i)}(x)$$

## *4.6 Regularization*

***Overfitting*** : If we have too many features, the learned hypothesis may fit the training set very well (cost function ~ 0), but fail to generalize to new examples (predictions on new examples).

Addressing Overfitting :

1. Reduce number of features
    - manually select which features to keep
    - model selection algorithm (later in course)
2. Regularization
    - Keep all the features but reduce magnitude/values of parameters $\theta_j$
    - Works well when we have a lot of features, each of which contributes a bit to predicting y.




*a : underfitting, b : regular fitting, c : overfitting*

Note: There is also the phenomenon of underfitting, where we have little features and the error in the training examples is quite big.

***Regularization*** works by penalizing theta parameters of higher order features, so that they do not take big values and the hypothesis function remains simpler and less prone to overfitting.

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

Note: $\theta_0$ is excluded in regularization !

λ is called lambda parameter or regularization parameter. If it is extremely big, we are in danger of underfitting. On the other hand, if it is extremely small, we are in danger of overfitting. So, we have to select a suitable value for λ and there is a specific process for this reason

## *4.6.1 Regularized Linear Regression*

Gradient Descent :

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

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

$$\theta_j=\theta_j(1-\alpha\frac{\lambda}{m})-a\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)})-y^{(i)})x_j^{(i)},\ j=1,2,3,\dots,n(not\ j=0!)$$

Normal Equation :

$$\Theta=(X^TX+\lambda\begin{bmatrix}0&&&\\&1&&\\&&\dots&\\&&&1\end{bmatrix})^{-1}X^TY$$

### *4.6.2 Regularized Logistic Regression*

Gradient Descent :

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

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

$$\theta_j=\theta_j(1-\alpha\frac{\lambda}{m})-a\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)})-y^{(i)})x_j^{(i)},\ j=1,2,3,\ldots,n(not\ j=0!)$$

## *5. Advice for Applying Machine Learning*

## *5.1 Evaluating a Learning Algorithm*

Suppose you have implemented regularized linear regression and when you test your hypothesis on a new set of data, you find that it makes unpredictably big errors in predictions. What you should try next ?

- Get more training examples
- Try smaller sets of features
- Try getting additional features
- Try adding polynomial features $(x_1^2, x_2^2, x_1x_2 etc.)$
- Try decreasing λ
- Try increasing λ

All these options require a lot of time. You would be really happy if you had a way of knowing in advance which of those can improve your algorithm.

***Machine Learning diagnostic*** is a test that you can run to gain insight what is/isn't working with a learning algorithm, and gain guidance as to how best to improve its performance.

Diagnostics can take time to implement, but doing so can be a very good use of your time (since you are saving time from implementing the previous options without results).

### *5.1.1 Evaluating your hypothesis (Train/Validation/Test sets)*

A common mistake in the field of statistics and machine learning is that people evaluate their hypothesis using the training data, that were previously used in training the algorithm. As it is evident, it is completely expected to have small errors, since the algorithm had been trained in exactly the same data. In this way, it is impossible to detect problems like overfitting.

So, one good way of evaluating your hypothesis is to split your data into 2 sets:

- the ***training set*** (used for training the algorithm)
- the ***test set*** (used for evaluating your algorithm)

A good ratio for splitting your data is to keep 70% of the data for the training set and leave the rest 30% of the data for your test set. However, ***beware*** that you have to randomly sort the data before you split them, so that each set comprises of randomly selected data !!

So, the training/testing procedure is the following :

- Learn parameters θ from training set
- Compute test error (using test set) with the same error sum

---

Even though this is perfectly reasonable, solely for logistic regression, there is also a different error metric used, called ***0/1 misclassification error*** :

$$err\left(h_\theta(x), y\right) = \begin{cases} 1, \text{if } h_\theta(x) \geq 0.5, y = 0 \text{ or } if\ h_\theta(x) < 0.5, y = 1 \\ 0, otherwise \end{cases}$$

$$Test\ error = \frac{1}{m_{test}} \sum_{i=1}^{mtest} err\left(h_\theta(x_{test})^{(i)}, y_{test}^{(i)}\right)$$

---

However, there are also cases, where we have not fully decided about some parameters of our algorithm (such as the value of the learning parameter, or the order magnitude of the hypothesis function). From what said before, we could use the training set to train multiple algorithms with different values for the undecided parameter and then use the test set to measure the performance of each value of the parameter, so that we choose the best one.

But, in this case in which set will we evaluate our hypothesis finally ? Because, if we evaluate in the test set, we are in danger of underestimating as before the generalization error, since we selected the parameter based on the test set.

So, generally there is need for splitting the data in 3 sets :

- ***training set***
- ***cross validation set***
- ***test set***

So, let's assume that we have not selected the form of hypothesis function and we want to use cross validation set for this purpose. The process to be followed is the following :

$$\textit{Train all those different regressions with } \textbf{\textit{training set}} \begin{cases} 1.\ h_\theta(x) = \theta_0 + \theta_1 x \rightarrow \Theta^{(1)} \\ 2.\ h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x \rightarrow \Theta^{(2)} \\ 3.\ h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x + \theta_3 x \rightarrow \Theta^{(3)} \\ \dots \\ 10.\ h_\theta(x) = \theta_0 + \theta_1 x + \dots + \theta_{10} x \rightarrow \Theta^{(10)} \end{cases}$$

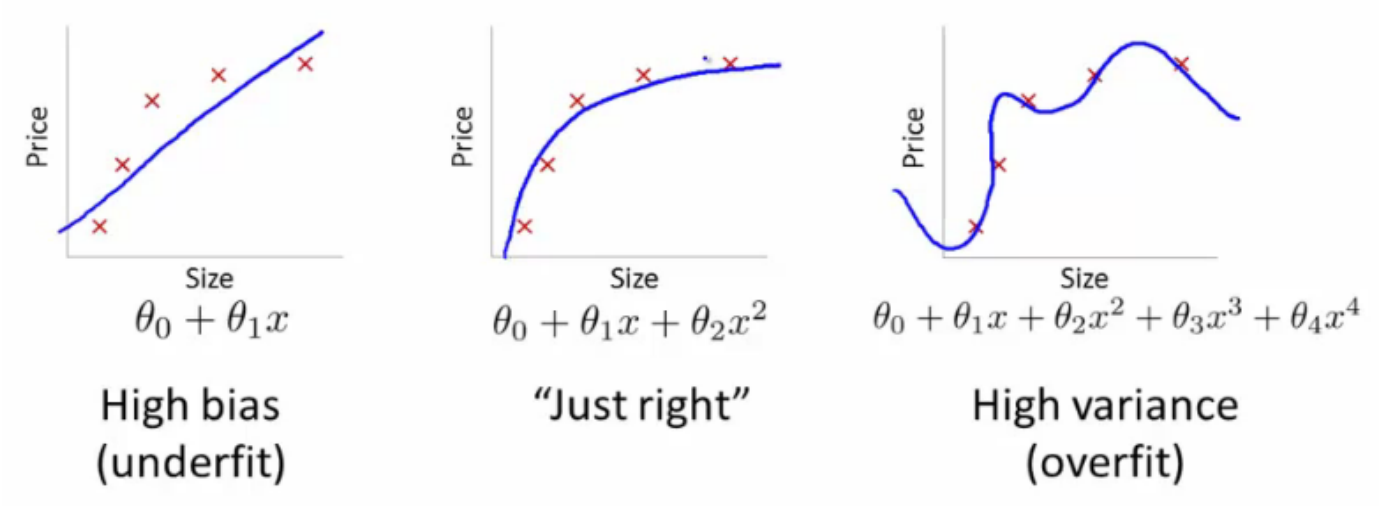
$$\textit{Use } \textbf{\textit{cross validation set}} \textit{ to compute } J_{cv}(\Theta^{(1)}), J_{cv}(\Theta^{(2)}), J_{cv}(\Theta^{(3)}), \dots, J_{cv}(\Theta^{(10)})$$

$$\textit{Pick } \Theta^{(i)} \textit{ with the smallest } J_{cv}$$

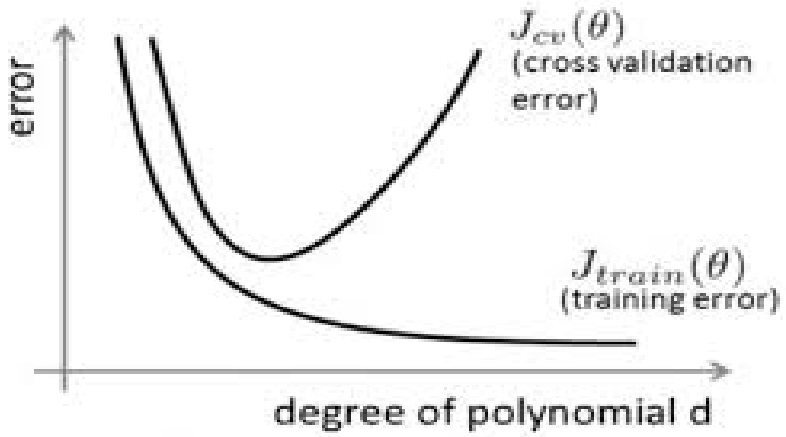
$$\textit{Estimate generilization error } J_{test}(\Theta^{(i)}) \textit{ for the selected hypothesis}$$

## *5.1.2 Bias vs Variance*

If your algorithm is not doing very well, then probably your algorithm has either high variance or high bias. In other words, it's a problem of underfitting or overfitting. So, it's useful to know whether your algorithm has high bias or high variance (or both), because so you will know which methods are capable to improve your algorithm.




If we plot the errors of training set and cross-validation test as a function of the degree of polynomial, we can see that it has the following shape. We can deduce various conclusions from this plot :




From this plot, we can draw the next conclusion :

***High Bias*** $(\textbf{\textit{underfit}})$:

$J_{cv}(\theta) \approx J_{train}(\theta)$

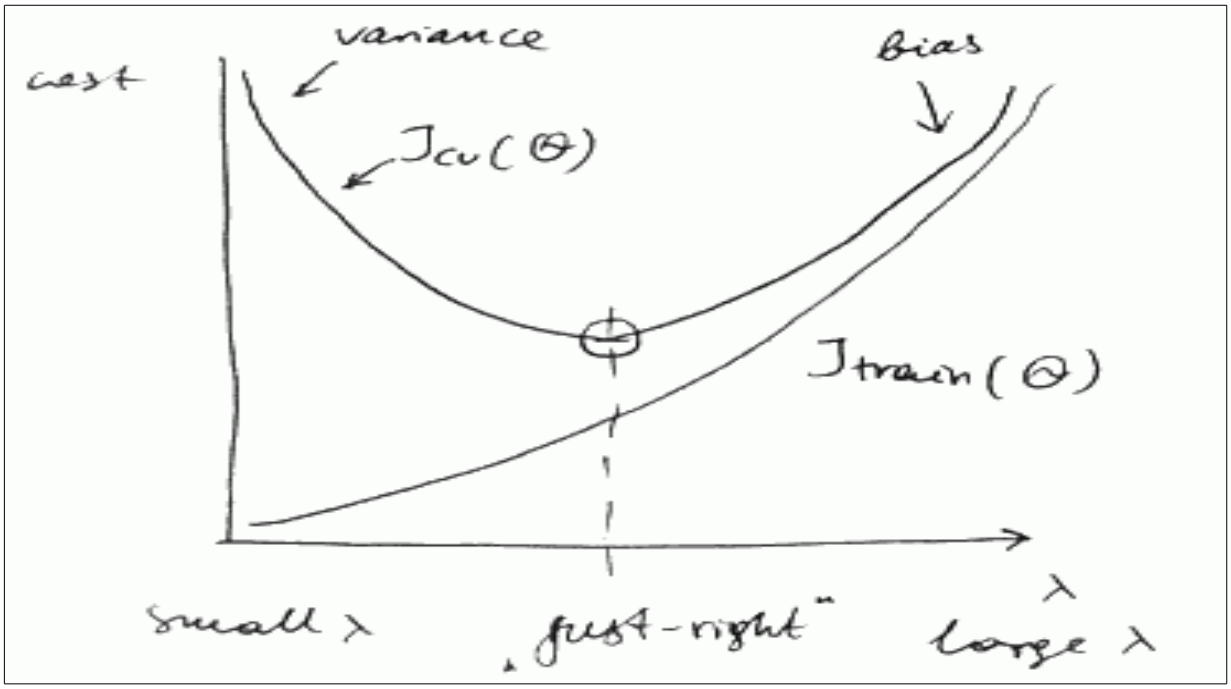
*they will both be high*

***High variance*** $(\textbf{\textit{overfit}})$:

$J_{cv}(\theta) \gg J_{train}(\theta)$

$J_{train}(\theta)$ *will be low* , $J_{cv}(\theta)$ *will be high*

Now, let's also investigate the effect of learning parameter value to the bias-variance of our learning algorithm. If we plot again the errors as a function of the values of the learning parameter λ, we come up with the following shape that is also confirmed by our intuition :



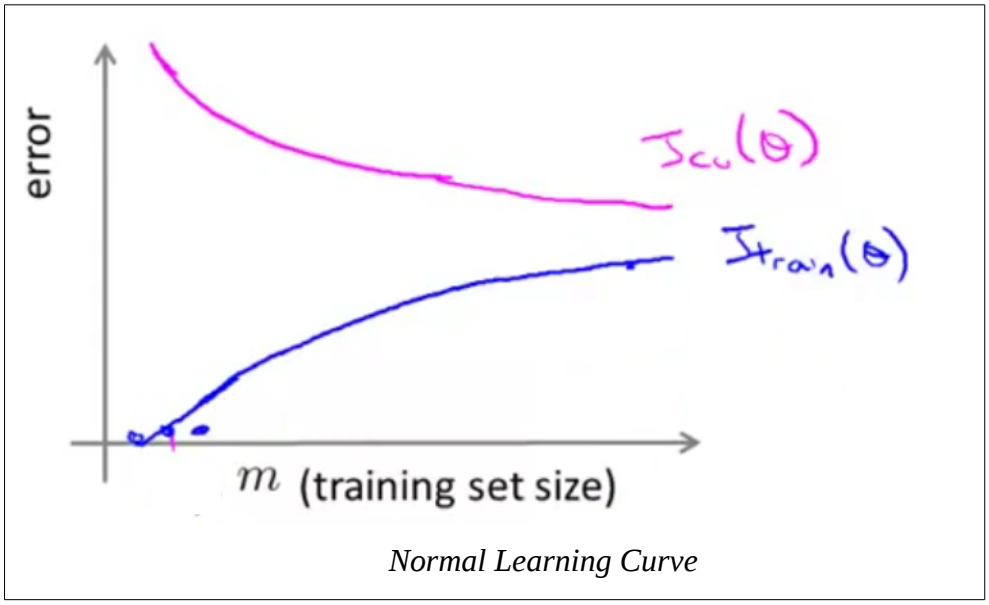

As we have seen in the initial analysis of regression, regularization was "invented" to prevent overfitting (high variance). So, it is expected that larger values of λ will correspond to high bias, since thus high variance is prevented.

So, in order to select an almost optimum value of learning parameter λ, we can use the previously described procedure trying various values of λ in cross validation data and then keep the one with the lowest errors. One good sequence of values that can be tested are the following :

λ = 0, 0.01, 0.02, 0.04, 0.08, 0.016, ..., 10.24

### *5.1.3 Learning Curves*

The learning curves are specific plots that are useful to be plotted when implementing an algorithm, as they give as insight to whether the algorithm performs well and if we have a problem of high bias or high variance. One useful learning curve is the plot of the errors as a function of the number of examples in training and cross validation sets. In order to plot this graph, we have to iteratively execute the algorithm with gradually increasing datasets. The plot will be somewhat like the following :

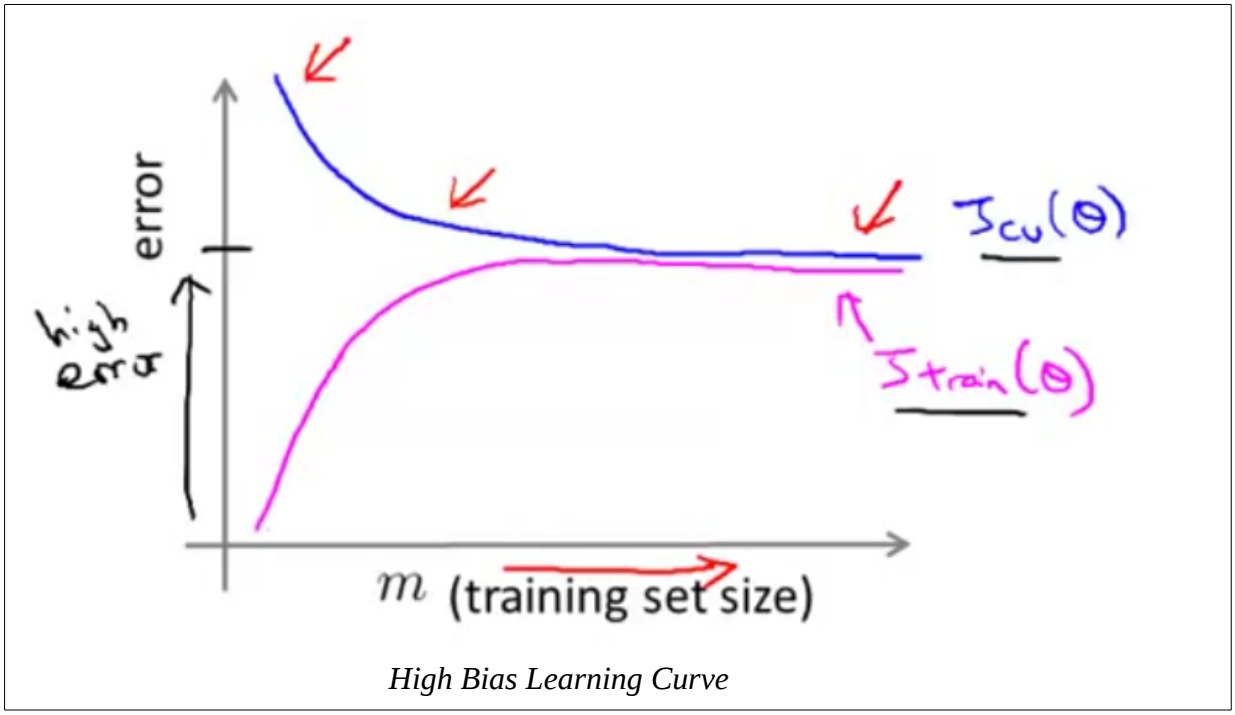



*Normal Learning Curve*

With few data, the hypothesis matches perfectly the training data, but makes large errors in the cross validation data. As the dataset grows, training error is steadily increasing, since the hypothesis cannot anymore fit all the data, but the cross validation error is dropping, since a bigger training set helps the algorithm select a more appropriate curve.

---

Now, let's investigate the case of high bias. The same graph will be like the below one :




*High Bias Learning Curve*

In case of high bias, we see that increasing the training set will increase training errors to a greater extent, since the hypothesis is not in a suitable form to fit the data at all. We also observe that cross validation errors also remain big and do not reduce significantly. Thus, we can see that **in case of high bias, getting more training data will not help (by itself) much**.

---

In case of high variance, we will have the following plot :

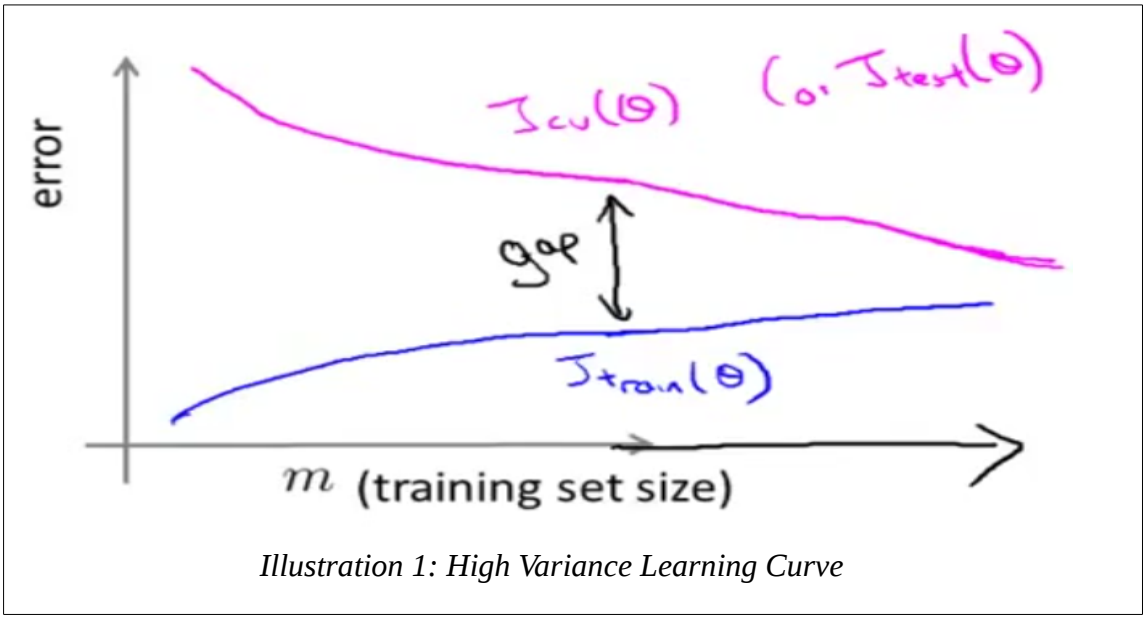
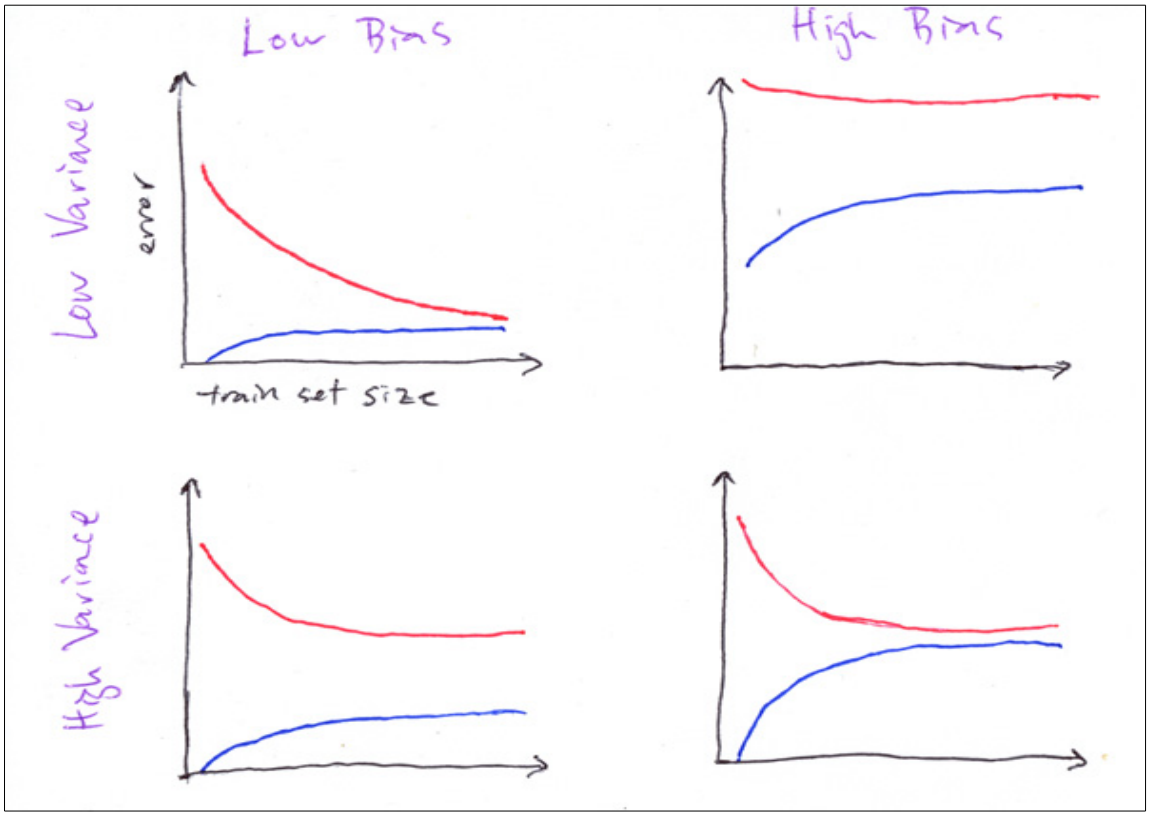



*Illustration 1: High Variance Learning Curve*

In case of high variance, we see that training errors increase with small pace, since the hypothesis has high order features and can fit the the data succesfully. Also, if we extend the plot, we see that the 2 curves of errors are tend to converge.**So, in case of high variance, getting more training data is likely to help**.

---

Now, we can also see a more general case of combinations of high variance and high bias problems.




## *5.2 Deciding what to do next*

Now, we are able to take decisions about the best options to follow to improve our algorithm depending on the problem the algorithm has :

- Get more training examples → fixes high variance
- Try smaller set of features → fixes high variance
- Try getting additional features → fixes high bias
- Try adding polynomial features → fixes high bias
- Try decreasing λ → fixes high bias
- Try increasing λ → fixes high variance

As far as neural networks are concerned, the following general notices are valid :

Small neural networks (fewer parameters or/and hidden layers):

- more prone to underfitting (add hidden layer to address)
- computationally cheaper

Large neural networks (more parameters or/and hidden layers):

- more prone to overfitting (use regularization to address)
- computationally more expensive

## *6. Machine Learning System Design*

In Machine Learning, the quantity of data is one very important factor. So, in a lot of cases, the first thing people think of when trying to decide where to devote their time is to gather much more data. However, there are a lot of thing that have to be done. So, people had better brainstorm a list of things that could improve their algorithm and then estimate how much each change would affect the results of their algorithm.

## *6.1 Error Analysis*

Error analysis is a concept that will help you take these decisions in a more systematic way.

Recommended Approach when implementing ML 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
- 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. So, this is a process, where we are trying to understand the structure of errors.

So, for example in a spam classifier, the error analysis might examine the errors (misclassified emails) and categorize them based on:
- what type of email it is
- what cues (features) you think would have helped the algorithm classify them correctly.

Using this categorization, we can make conclusions regarding the category of mails, where we should focus. More specifically, we could end up with specific features that we should add in our model.

On the other hand, there are cases when it is hard to try to understand what should be developed, by just analysing the errors. The only thing that would help is to try and see if it works. So, here comes the ***importance of numerical evaluation***. By having 1 simple evaluation metric, we can test our ideas and see if they improve our evaluation metric in order to decide whether it is worthwhile implementing them.

## *6.2 Handling Skewed Classes*

However, there is particularly a case, where it can be quite tricky to come up with an appropriate error metric. This is the case of skewed classes.

Skewed classes are classes, where the positive (or the negative) examples constitute an extremely high percentage of the examples. Take for example the cancer classification example, where the people having cancer constitute a really small percentage of the total population. In this case, using misclassification error as an error metric is not vey useful, since we can have naïve algorithm that predicts always a result of not having cancer and this algorithm would have significantly low errors.

On the other hand, a more appropriate error metric for these cases is the ***precision/recall metric***. Let's get familiar with the following notions first :

| | | **Actual Class** | |
|---|---|---|---|
| | | 1 | 0 |
| **Predicted Class** | 1 | True positive | False positive |
| | 0 | False negative | True negative |

***Precision*** (of all the patients we predicted y = 1, what fraction actually has cancer ?) :

$$\frac{True\ positives}{predicted\ positives} = \frac{True\ positives}{True\ positives + False\ positives}$$

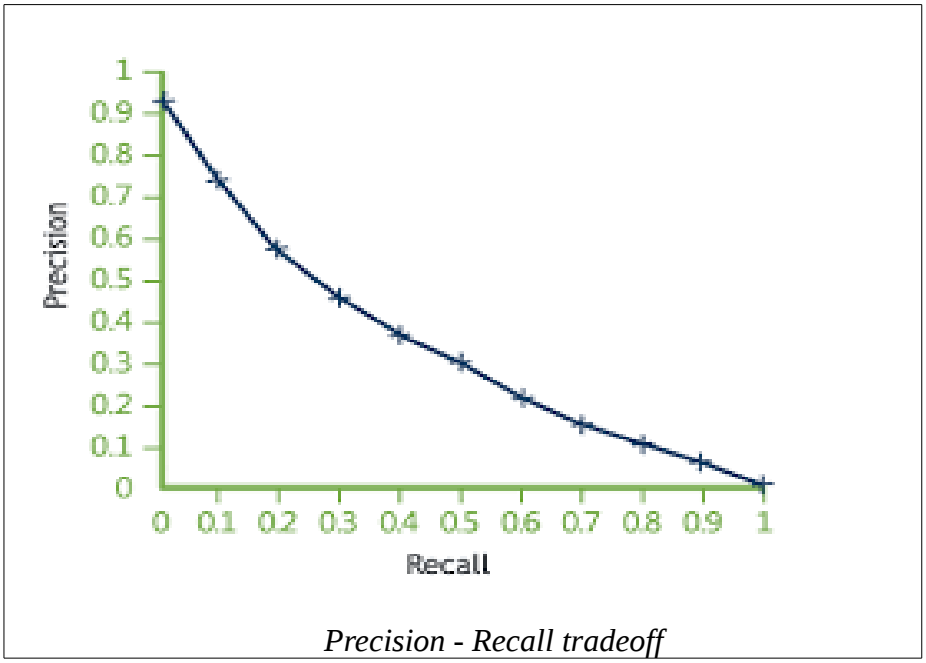
***Recall*** (of all patients that actually have cancer, what fraction did we correctly detect as having cancer ?) :

$$\frac{True\ positives}{actual\ positives} = \frac{True\ positives}{True\ positives + False\ negatives}$$

Note : In precision/recall error metric, we assume that y = 1 corresponds the the rarer class.

As you can understand, there is a constant tradeoff between precision and recall.

- If we want to predict y = 1 only if very confident, then we can raise the threshold value (i.e. to 0.7). This will give us a higher precision and lower recall.
- On the other hand, if we would like to avoid missing too many cases of cancer (avoid false negatives), then we should lower the threshold value. This would give us a higher recall and lower precision




*Precision - Recall tradeoff*

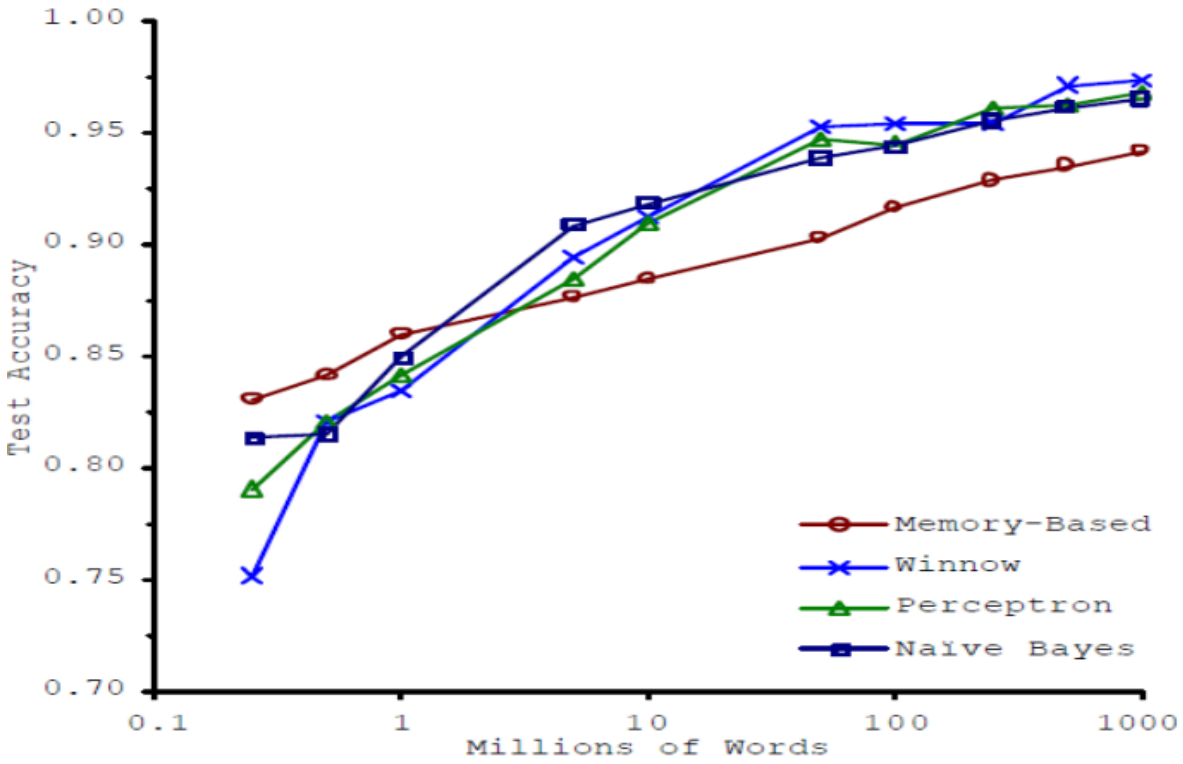
In order to take into consideration this tradeoff, we use F1 score as an evaluation metric :

$$F_1\,score = 2\frac{PR}{P+R}$$

And we want this metric to be as near to 1 as possible.

## *6.3 Data for Machine Learning*

As we have previously described, data are very important for machine learning. In a survey being conducted around 2001 by Banko and Brill, several algorithms have been compared using various sized different data sets. Their comparison can be seen in the following plot :




The conclusions drawn from this plot are the following :

- All algorithms show a monotonic increase in accuracy as we add more and more data
- This increase somewhat starts reaching 100% when we reach very big datasets
- The data seem to be more important than the algorithm. This means that if we compare 2 algorithms, a worse algorithm can exhibit better performance if more data are provided

From those conclusions, the following quote is derived :

***"It's not who was the best algorithms that wins. It's who has the most data."***

However, this is not completely true for all problems. We will describe the large data rationale to get a better insight :

- For roughly estimating where we can benefit from more data (except from bias vs variance etc), we can make the following distinction. If we give the problem to a human expert in the specific field, will he be able to find the right answer ? If yes, then it is highly possible that more data are not going to help.

For example, let's check 2 problems. In the first problem, we want to find the word that best fits a sentence. If we give this problem to an english language expert, he will solve it easily. So, more data will probably not help in this type of problem. Now, let's investigate the problem of predicting the price of a house using specific features. If we go to a real estate agent, he might not be able to predict accurately the price. So, in this problem, a bigger amount of data is going to help.

So, the strategy behind large data rationale is the following :

- Use a learning algorithm with many parameters (e.g. logistic regression/linear regression with many features; neural networks with many hidden units.
- Use a very large training set (unlikely to overfit).

# *7. Neural Networks*

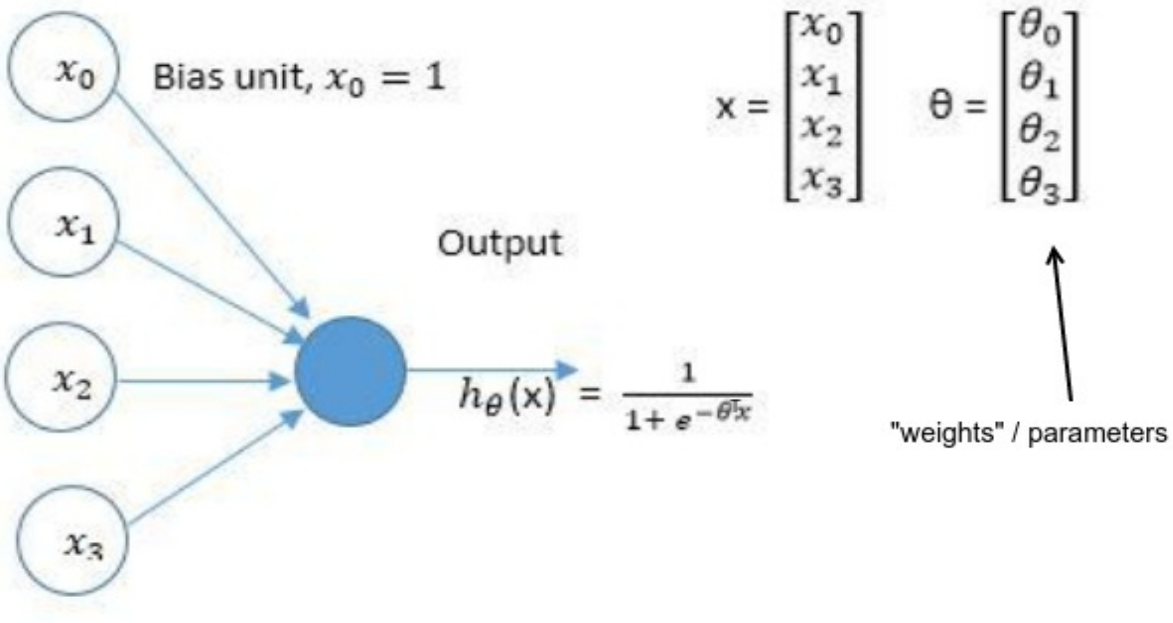
## *7.1 Motivation*

Neural Networks is a state-of-the-art technique for many machine learning problems. But, since we have linear , polynomial and logistic regression, why do we need yet another learning algorithm ?

When the number of features increase drastically, then the regression methods become slower in performance and in some cases almost infeasible. However, neural networks (that are by nature distributed) can more easily calculate more complex hypohesis functions. So, neural networks are extensively used in applications, such as computer vision, where the input is big (thousands of pixels).

## *7.2 Model Representation*

Let's start from the smallest unit of a Neural Network, the neuron.

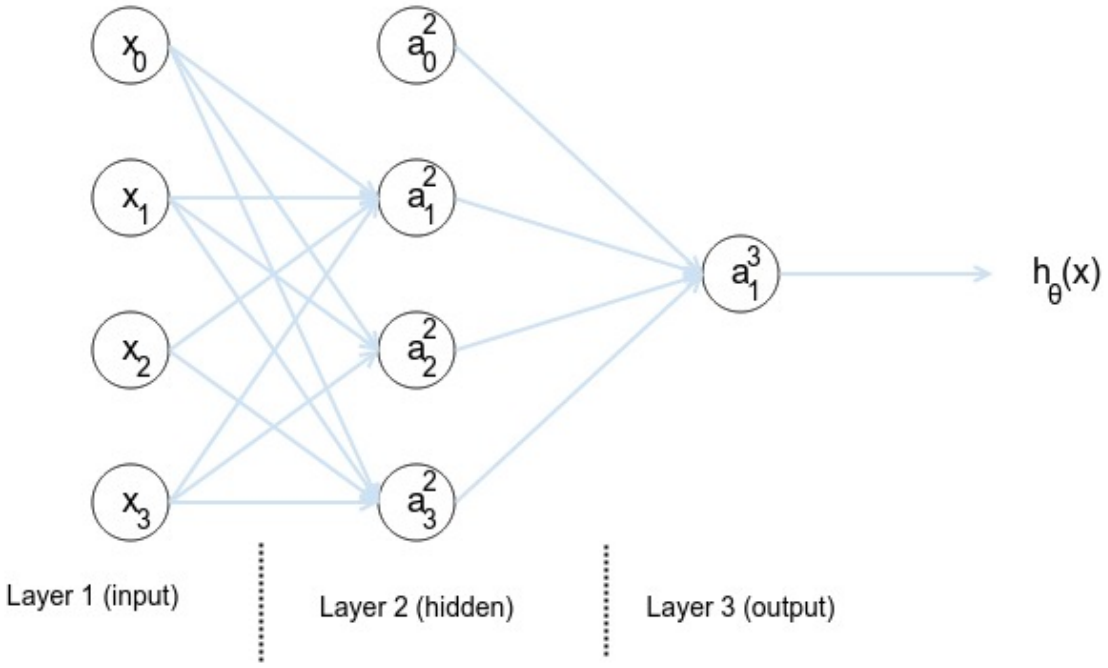



*Neuron*

Note: $\Theta^T X$ is passed through a function called ***activation function*** before being fed as output. In most cases (as above) sigmoid function is used as activation function, because its derivative can be easily calculated and because the brain neuron works with sigmoid function.

(However, some other activation functions are the step function, the linear combination, the tan-sigmoid function and the softmax function)

Let's move on now with the Neural Network representation




The algorithm of neural networks works in 2 steps :

- Forward Propagation
- Backpropagation

Forward Propagation is the step, where given the weights/parameters, the necessary calculations are made (in a forward way) in order to calculate the final output (hypothesis + cost function).

Backpropagation is the step, where the necessary calculations are made (in reverse-backwards way) in order to calculate the partial derivatives.

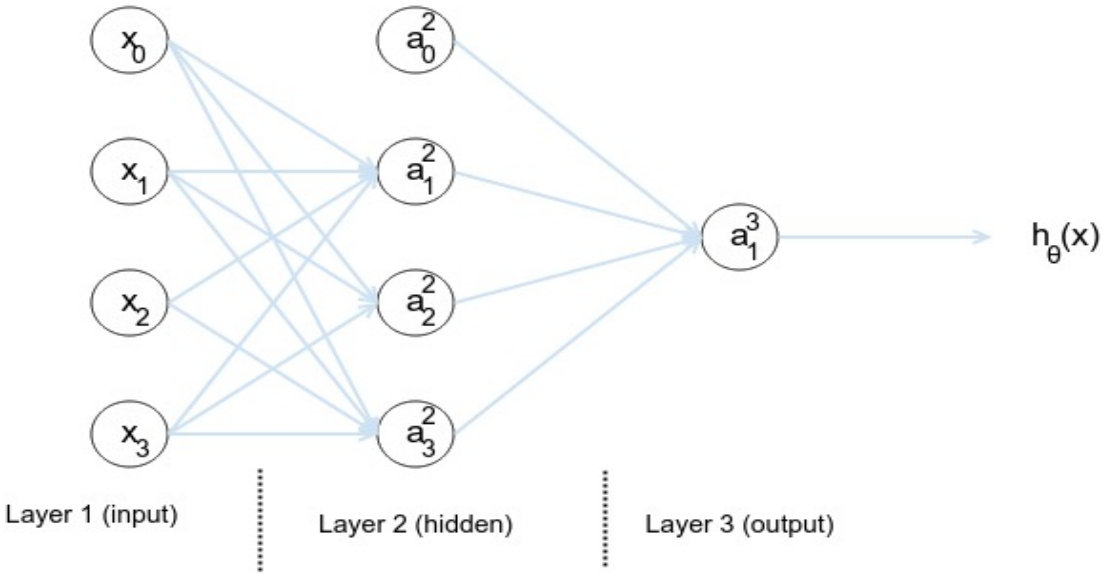
Each iteration (over 1 data example) in regression algorithms correspond to 1 execution of Forward Propagation coupled with 1 Backpropagation. After this execution, the execution (Forward Propagation and then Backpropagation) is repeated with the next data example and, after all data examples have been

processed, the cost function and derivatives are fed into gradient descent or other optimized minimization algorithms. (*for i = 1...m {Forward Propagation; Backward propagation;} parameters adaptation;*)

Note: The strength of neural networks lies in the fact that they are able to learn its own (more complex than the input) features, through their hidden layers, and so they can produce really complex non-linear hypotheses (that might be impossible in performance with other methods).

## *7.2.1 Forward Propagation*

Let's examine forward propagation with the previous example of neural network. The relationships that will be used to calculate the hypothesis function output during the forward propagation are the following :




$$\alpha_i^{(j)}: \textit{activation of unit 'i' in layer 'j'}$$

$$\Theta^{(j)}: \text{matrix } \textit{of weights controlling function mapping from layer } j \text{ to } j+1$$

$$\alpha_1^{(2)} = g\left(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3\right)$$

$$\alpha_2^{(2)} = g\left(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3\right)$$

$$\alpha_3^{(2)} = g\left(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3\right)$$

$$h_\theta(x) = a_1^{(3)} = g\left(\Theta_{10}^{(2)} \alpha_0^{(2)} + \Theta_{11}^{(2)} \alpha_1^{(2)} + \Theta_{12}^{(2)} \alpha_2^{(2)} + \Theta_{13}^{(2)} \alpha_3^{(2)}\right)$$

If network has $S_j$ units in layer j, $S_{j+1}$ units in layer j+1, then $\Theta^{(j)}$ will be of

dimension $s_{j+1}$ x $(s_j + 1)$

## *7.2.2 Vectorized Implementation*

$$a^{(1)} = x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}, \; z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix}$$

$$z^{(2)} = \Theta^{(1)} a^{(1)} = \Theta^{(1)} x$$

$$a^{(2)} = g(z^{(2)}) \; where \; add \; a_0^{(2)} = 1$$

...

$$z^{(3)} = \Theta^{(2)} a^{(2)}$$

$$h_\theta(x) = a^{(3)} = g(z^{(3)})$$

## *7.2.3 Cost Function*

The cost function for Neural Networks using sigmoid function as activation function (mainly used for classification problems) is the following :

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

where
L : total number of layers in network (including input-activation layer and output layer)
$s_l$ : no. of units (not counting bias units) in layer l
K = $s_L$ : number of output units

For neural networks with linear (“equality”) activation function, the cost function as expected is the following :

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

### *7.2.4 Backpropagation*

In backpropagation, we compute $\delta_j^{(l)}$ that by intuition corresponds to the "error" of node j in layer l. Then, we will use those errors to calculate the partial derivatives.

So, we have :

for output units : $\delta_j^{(L)} = \alpha_j^{(L)} - y_j = \left(h_\theta(x)\right)_j - y_j$

$$\delta^{(l)} = \left(\Theta^{(\lambda)}\right)^T \delta^{(l+1)} .* g'\left(z^{(l)}\right)$$

$$where\ for\ sigmoid\ activation : g'\left(z^{(l)}\right) = \alpha^{(l)} * \left(1 - \alpha^{(l)}\right)$$

for hidden layers : $\left(\text{and } for\ linear\ activation\right) = g'\left(z^{(l)}\right) = 1$

(note that .* stands for octave element-wise operation)

for input layers there are no δ values (since we have no "errors" for inputs)

Then, we use δ values to compute Δ values. After we have processed all the data examples, we compute D values that correspond to derivatives. So, the backpropagation algorithm is the following :

$Set\ \Delta_{ij}^{(l)} = 0\ (for\ all\ l, i, j)$

$for\ i = 1\ \text{to}\ m$

$Set\ \alpha^{(1)} = x^{(i)}$

$Perform\ forward\ propagation\ \text{to}\ compute\ \alpha^{(l)}\ for\ l = 2,3,\ldots,L$

$Using\ y^{(i)}\ compute\ for\ output\ units : \delta^{(L)} = \alpha^{(L)} - y^{(i)}$

$Compute\ \delta^{(L-1)}, \delta^{(L-2)}, \ldots, \delta^{(2)}\ using\ the\ previous\ equations$

$\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)} \xrightarrow{vectorized} \Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} \left(\alpha^{(l)}\right)^T$

$$D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)},\ if\ j \neq 0$$

$$D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)},\ if\ j = 0$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

## *7.3 Implementation notes*

### *7.3.1 Unrolling parameters*

When we are implementing Neural Networks, the theta parameters/weights and the partial derivatives (gradients) are matrices (and not still vectors as it was in regression). So, before feeding them into gradient descent (or any other optimized minimization algorithm we can use), we have to "unroll" them into vectors. However, we must also be able to reshape them again into matrices from the vectors format. This is done in Octave with the following commands :

$$thetaVec = [thetaVec1(:), thetaVec2(:), thetaVec3(:)]; \%in\ case\ of\ 4\ layers$$

$$Theta1 = reshape(thetaVec(1:(dim1_1 * dim1_2)), dim1_1, dim1_2);$$
$$Theta2 = reshape(thetaVec((dim1_1 * dim1_2):(dim1_1 * dim1_2 + dim2_1 * dim2_2)), dim2_1, dim2_2);$$
...

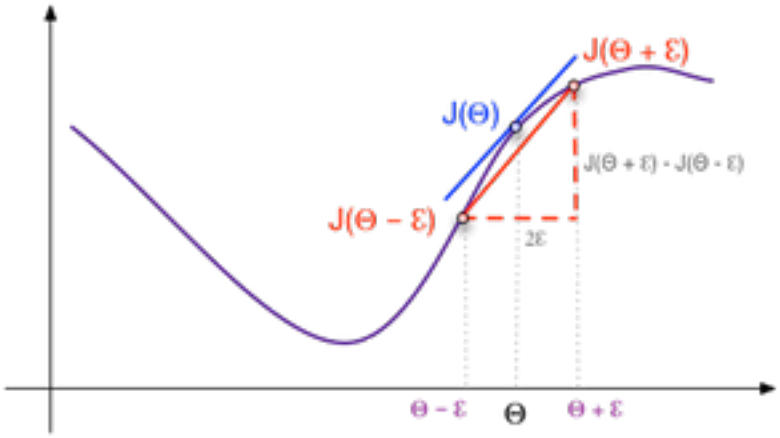
So, the learning algorithm (with unrolling parameters) works as following :

→ *Have initial parameters* $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$
*Unroll* $\Theta^{(1)}, \Theta^{(2)}, \Theta^{(3)}$ *into vector* **thetaVec**
→ *Use forward propagation/backpropagation to compute* $D^{(1)}, D^{(2)}, D^{(3)}$ and $J(\theta)$ *through*
**function**[*J grad*] = **nnCostFunction**(**thetaVec**, ...)
*This function should reshape* **thetaVec** to *execute forward propagation, backpropagation* and *then unroll the calculated* $D^{(1)}, D^{(2)}, D^{(3)}$ *into* **gradientVec** *which is returned*
→ *Call* **fminunc**(**@costFunction**, **thetaVec**, **options**)

So, we can understand that the matrix representation is more convenient for forward propagation, so that we can use the vectorized implementations and get better performance. However, the optimized minimization functions assume that we have theta parameters and gradients unrolled into a big long vector, so this is the reason for unrolling.

## *7.3.2 Gradient Checking*

As we have previously mentioned a common way to test & debug your learning algorithm is to graph a plot of J(θ) versus θ and make sure that the function is constantly decreasing as θ receives bigger values. However, with neural networks things get a little more complicated. So, this test is not adequate. Your algorithm might pass this test, but it might have deeper bugs that are harder to detect. There is another test used to further debug your learning algorithm and is called gradient checking.




The general idea behind gradient checking is to compare the calculated gradient with a numeric approximation. The formalistic definition is the following :

$$\theta=[\theta_1, \theta_2, \dots, \theta_n]$$

$$\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J(\theta_1+\varepsilon, \theta_2, \dots, \theta_n) - J(\theta_1-\varepsilon, \theta_2, \dots, \theta_n)}{2\varepsilon}$$

$$\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J(\theta_1, \theta_2+\varepsilon, \dots, \theta_n) - J(\theta_1, \theta_2-\varepsilon, \dots, \theta_n)}{2\varepsilon}$$

$$\dots$$

$$\frac{\partial}{\partial \theta_n} J(\theta) \approx \frac{J(\theta_1, \theta_2, \dots, \theta_n+\varepsilon) - J(\theta_1, \theta_2, \dots, \theta_n-\varepsilon)}{2\varepsilon}$$

$$where\ \varepsilon = 10^{-4}$$

Note that we have to set a small ε, so that the approximation better. But, if we set ε to really small values, then we are in danger of running into numerical problems.

In Octave this is implemented in the following way :

$$
\begin{aligned}
&for\ i = 1:n \\
&\quad thetaPlus = theta; \\
&\quad thetaPlus(i) = thetaPlus + epsilon; \\
&\quad thetaMinus = theta; \\
&\quad thetaMinus(i) = thetaMinux(i) - epsilon; \\
&\quad gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2 * epsilon) \\
&end \\
&\text{Check } that\ gradApprox \approx DVec
\end{aligned}
$$

**Note :**
- Implement backprop to compute Dvec (unrolled $D^{(1)}$, $D^{(2)}$, $D^{(3)}$)
- Implement numerical gradient check to compute gradApprox
- Make sure they give similar values
- Turn off gradient checking. Use backprop for learning

**Important :**
Be sure to disable gradient checking before training your classifier. If you run numerical gradient computation of every iteration of gradient descent your code will be very slow.

To implement approximation-comparison, you can use the following equation:

$$
diff = \frac{norm(numerical\ gradient - gradient)}{norm(numerical\ gradient + gradient)}
$$

$$
diff \leqslant 10^{-9}
$$

## *7.3.3 Random Initialization*

For linear regression and logistic regression, we previously saw that it is a common approach to initialise all theta parameters to zeros. There is no problem with it and the algorithm converges normally. However, the case is different with neural networks.

In neural networks, if we initialise all theta parameters to the same value

(irrespectively to zero or not), then it can be proven that there will be an undesired symmetry in the neural network, that will force all the parameters get adjusted in the same value on each iteration. In essence, this means that the neural network will have a single unique feature as input. This is not really useful as we can easily understand. For this reason, in neural networks we will always have to use ***random initialization*** for theta parameters.

## *Random Initialization: Symmetry Breaking*

$Initialize\ each\ \Theta_{ij}^{(l)}\ to\ a\ random\ value\ in\ [-\varepsilon, \varepsilon]\ (\varepsilon \approx 0.1)$

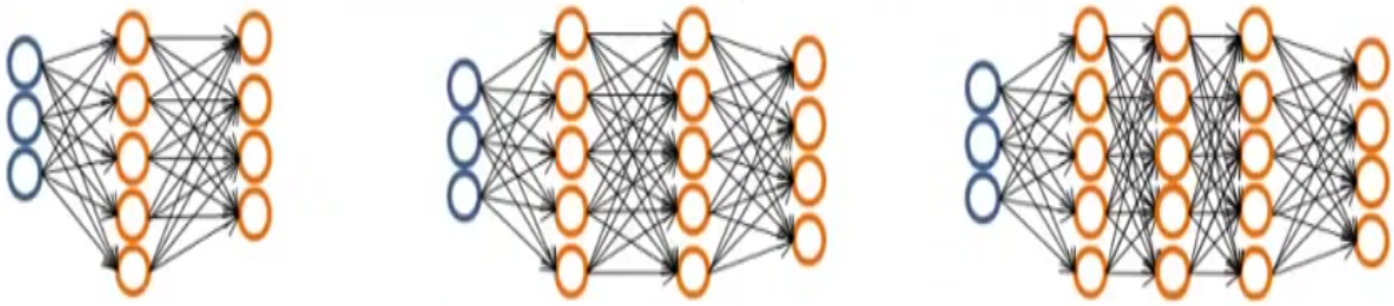
$E.g.$

$Theta1 = rand(dim1, dim2) * (2 * init_{epsilon}) - init_{epsilon}$

...

## *7.4 Architectures*

So, we have described the algorithm of neural networks. What stays to be discussed is the first thing we have to do before starting our work with a neural network, selecting the architecture. When selecting the architecture of a neural network, we have to take into account the following things:



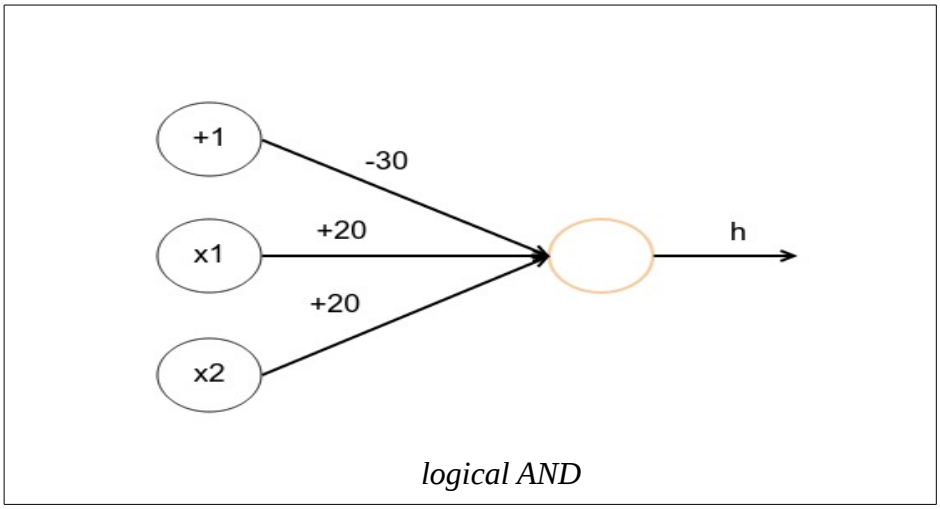
- No. of inputs: Dimension of features $x^{(i)}$
- No. output units : number of classes (in case of classification problem)
- Reasonable default : 1 hidden layer, or if >1 hidden layer, have same no. of units in every hidden layer (generally around (1-4) x input units)

The number of hidden units is basic to the performance of our neural network. The more hidden units we have the better, since the neural network learns better our data. However, we have to know that if we select too many hidden layers, then our algorithm might suffer from high variance and thus overfitting ! Furthermore, increasing the number of hidden layers, we also degrade the performance of our neural network. So, there is a tradeoff in selecting the number of hidden layers.

## *7.5 Applications*

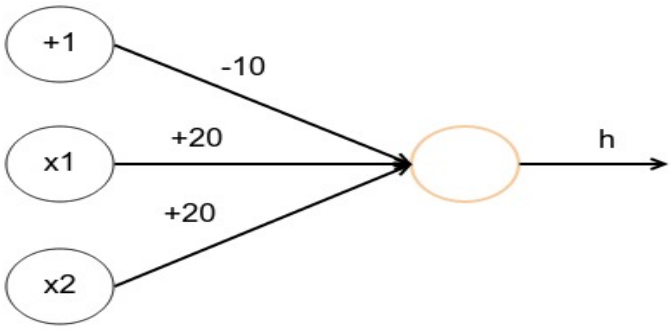
Now, we are going to see some simple examples of neural networks, to understand why they are capable of fitting complex non-linear hypotheses.

First, let's begin from some simple neural networks implementing the logical AND and logical OR :




*logical AND*

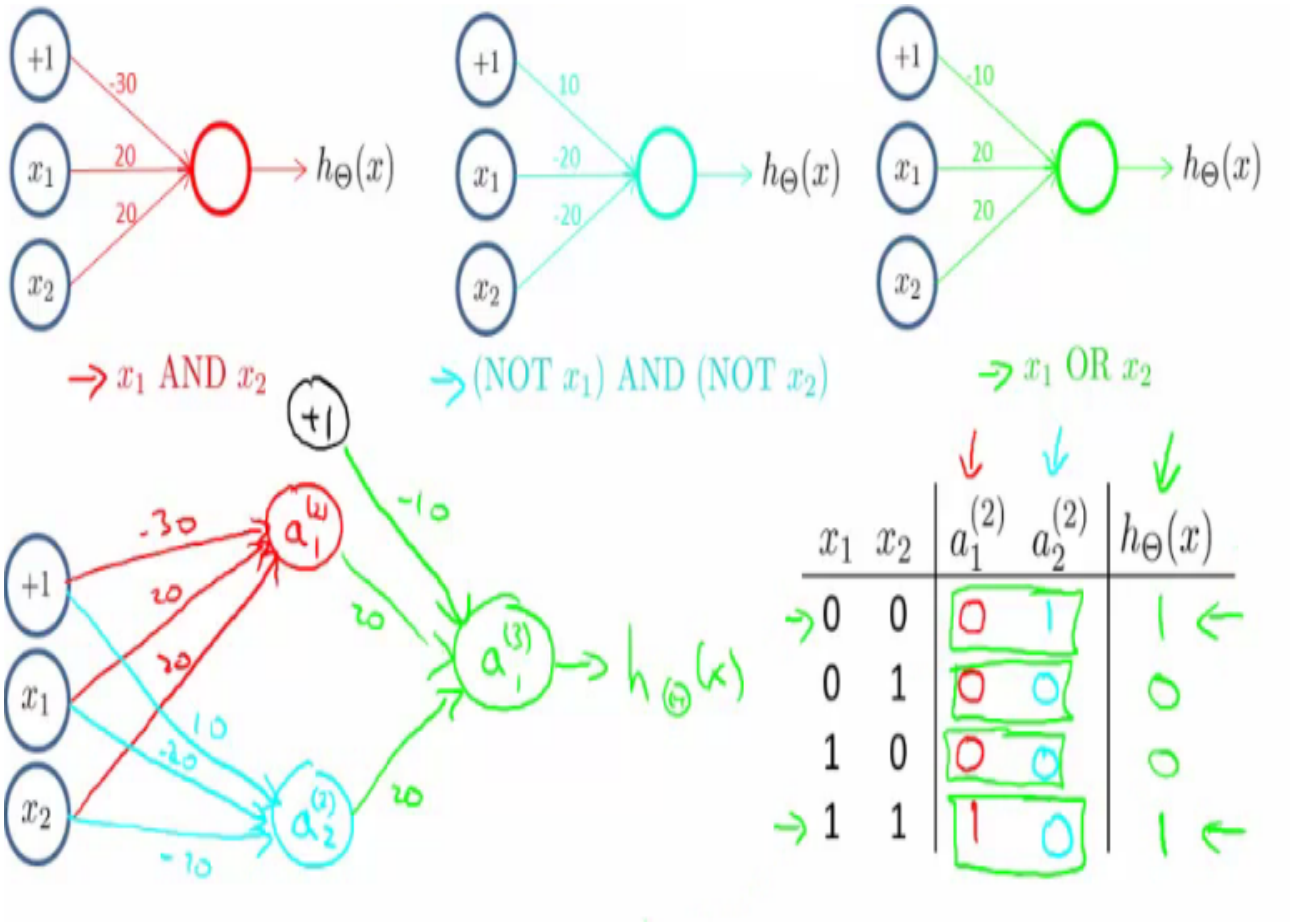
| x1 | x2 | $h_\theta(x)$ |
|---|---|---|
| 0 | 0 | g(-30) ~ 0 |
| 0 | 1 | g(-10) ~ 0 |
| 1 | 0 | g(-10) ~ 0 |
| 1 | 1 | g(10) ~ 1 |




*logical OR*

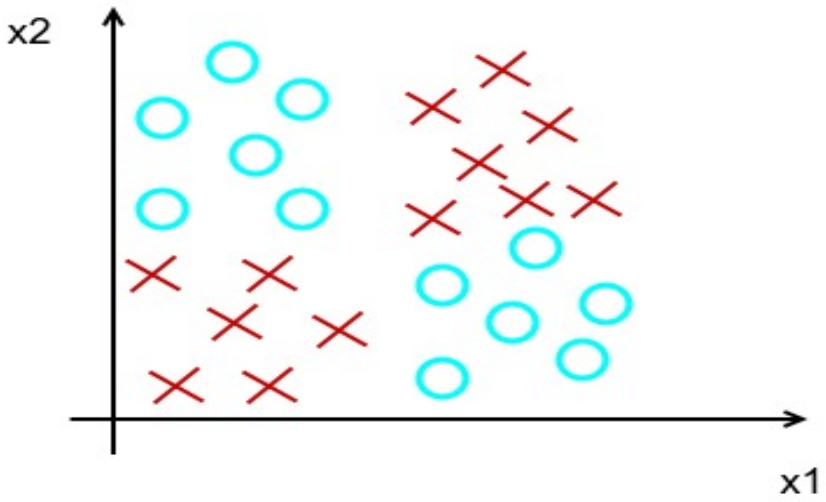
| x1 | x2 | $h_\theta(x)$ |
|---|---|---|
| 0 | 0 | g(-10) ~ 0 |
| 0 | 1 | g(10) ~ 1 |
| 1 | 0 | g(10) ~ 1 |
| 1 | 1 | g(10) ~ 1 |

Now, let's see a more complex function, the logical XNOR. To produce XNOR, we will have to use 3 basic networks and produce a network with a hidden layer as below :




*logical XNOR*

This neural network can handle complex non-linear problems like the following :



## *8. Support Vector Machines*

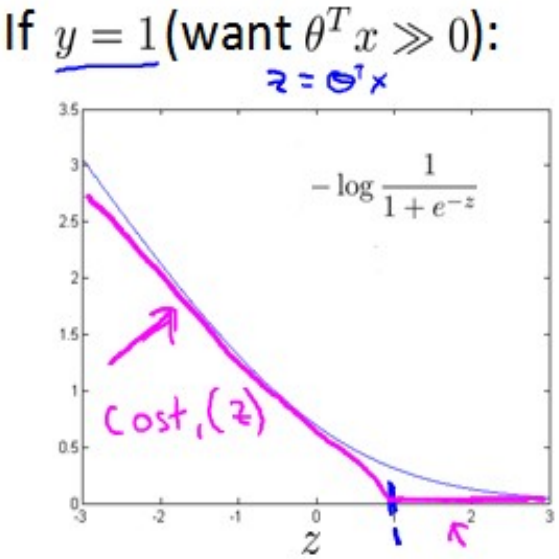
There is one more algorithm that is very powerful and widely used both in academia and industry, and that's called the support vector machines. And compared to logistic regression and neural networks, support vector machines sometimes give cleaner and more powerful way of learning complex non-linear functions. This is the last of the supervisory algorithms that will be described here.

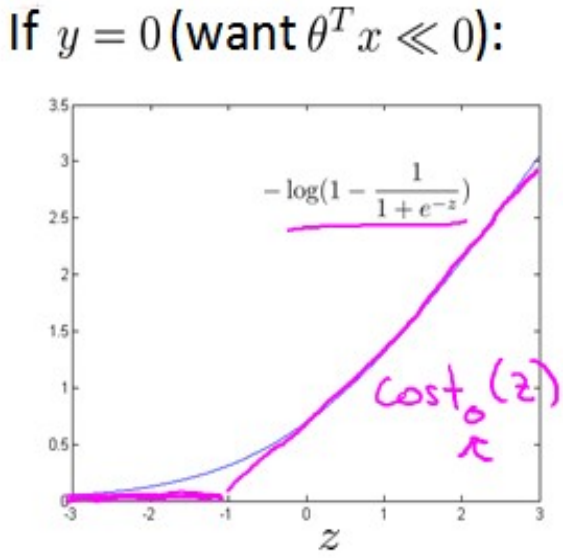
## *8.1 Large Margin Classification*

As with all algorithms, let's start with the optimization objective of the algorithm. Similar to logistic regression, in SVM case, we want to make the least errors possible. So, looking at sigmoid function shape :

- if y = 1, we want $\Theta^T x \gg 0 \rightarrow h_\theta(x) \approx 1$
- if y = 0, we want $\Theta^T x \ll 0 \rightarrow h_\theta(x) \approx 0$

So, looking at the cost function of logistic regression and dividing into 2 cases (y=0, y=1), we will approximate them with the following linear functions that will give us performance benefits :




So, the final cost function will be the following :

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

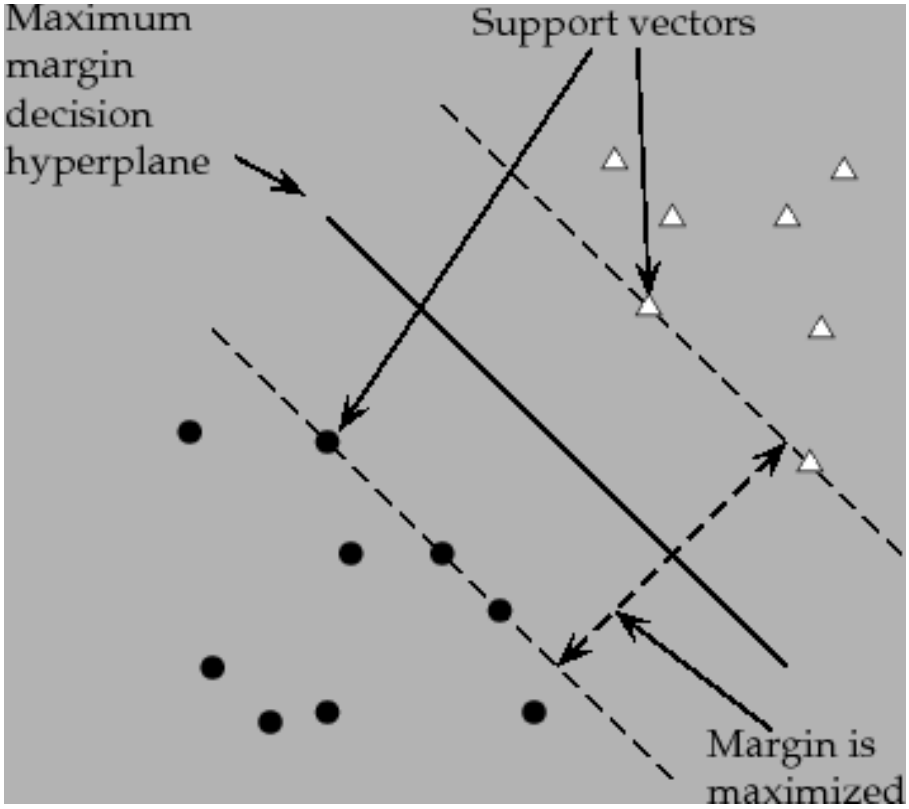
*where constant C is used instead of λ*

And the hypothesis will not output a probability (as in logistic regression), but it will instead output distinct value (0-1) :

$$h_\theta(x) = \begin{cases} 1, \text{if } \theta^T x \geq 0 \\ 0, otherwise \end{cases}$$

The algorithm of support vector machines is also called the ***large margin classifier***. This is due to the following reason :

If we observe the previous cost function plots, we will see that errors became zero not just when $\Theta^T x > 0$ (or $\Theta^T x < 0$ ), but instead only when $\Theta^T x > 1$ (or $\Theta^T x < -1$ ). This is the so-called large margin imposed by the algorithm. So, in fact, this algorithm tries to maximize the margin that exists between the classes that needs to be distinguished. And to what extent the selection of this large margin will be affected by outliers is dependent on the value of parameter C.




*Large Margin intuition*

## *8.2 Kernels*

From what we saw before, we assume that support vector machines can create decision boundaries that are straight lines. In other words, we assume that support vector machines can only be used in linear problems. However, this is not true in order to extend support vector machines so that they can also be used to develop complex non-linear classifiers, we should introduce ***kernels***.

In logistic regression, when we wanted to develop non-linear decision boundaries, we were inserting higher order features $(x_1^2, x_2^2, x_1 x_2, \ldots)$ in the hypothesis function. In kernels, we will replace the features $x_1, x_2, x_3, \cdots$ with new "complex" features $f_1, f_2, f_3, \cdots$ . However, the new features will be derived from the old ones. The conversion function will depend on the type of the kernel used. Here, we will describe the ***Gaussian kernels***. According to Gaussian kernels, we select several landmarks $l_1, l_2, l_3, \cdots$ and the features f are the similarities between those points and the input x. So :

$$f1 = similarity(x, l^{(1)}) = \exp\left(\frac{-\left\|x - l^{(1)}\right\|^2}{2\sigma^2}\right) = \exp\left(\frac{-\sum_{j=1}^{n}(x_j - l_j^{(1)})^2}{2\sigma^2}\right)$$

$$if\ x \approx l^{(1)} : f_1 \approx 1$$

$$if\ x\ far\ \text{from}\ l^{(1)} : f_1 \approx 0$$

$$\ldots$$

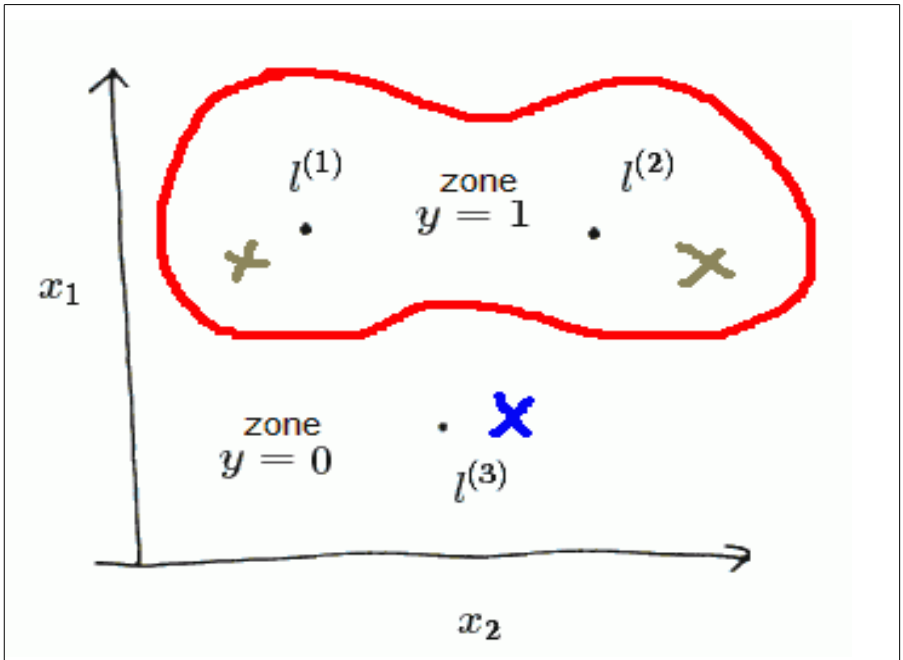
$$New\ hypothesis\ function:$$

$$h_\theta(x) = \theta_0 + \theta_1 f_1 + \theta_2 f_2 + \ldots + \theta_n f_n$$

$$predict\ y = 1, when\ h_\theta(x) > 0$$

$$predict\ y = 0, when\ h_\theta(x) < 0$$

An example of support vector machine with Gaussian kernels is the following, where 3 points were selected and they produced the following non-linear decision boundary. :




*SVM with Gassian Kernels*

Note : When having complex problems, we want a lot of landmarks. So, for each example, we place a landmark in the exact same position. So, we end up with m landmarks.

As far as SVM parameters are concerned (C, σ), the following are true :

$$C = \left(\frac{1}{\lambda}\right):$$

$$large\ C \rightarrow lower\ bias, high\ variance\ (small\ \lambda)$$
$$small\ C \rightarrow higher\ bias, lower\ variance\ (large\ \lambda)$$

$$\sigma^2: large\ \sigma^2 \rightarrow features\ f_i\ vary\ smoothly.\ higher\ bias, lower\ variance$$
$$small\ \sigma^2 \rightarrow features\ f_i\ vary\ less\ smoothly.\ lower\ vias, higher\ variance$$

## *8.3 Using an SVM*

As in other cases, there are a lot of libraries and implementations of SVMs, so there is no reason to start from scratch and try to implement an SVM again. Instead, we should use those libraries (liblinear,libsvm,..) that, most of them, are highly optimized. However, there is a need to specify :

- Choice of parameter C
- Choice of kernel (similarity function)
- possible kernel parameters (i.e. σ in case of Gaussian kernel)

Besides linear kernels (in fact SVMs without kernels), and Gaussian kernels, there are also a lot more other kernels used for specific cases, such as :

- Polynomial kernel
- String kernel
- chi-square kernel
- histogram intersection kernel
- ....

In case of multiclass classification, many SVM packages have already built-in functionality for multiclass classification. Otherwise, we use the well known one-vs-all method, by training K different SVMs (one to distinguish each class from the rest) and then we pick the class with the largest value for h.

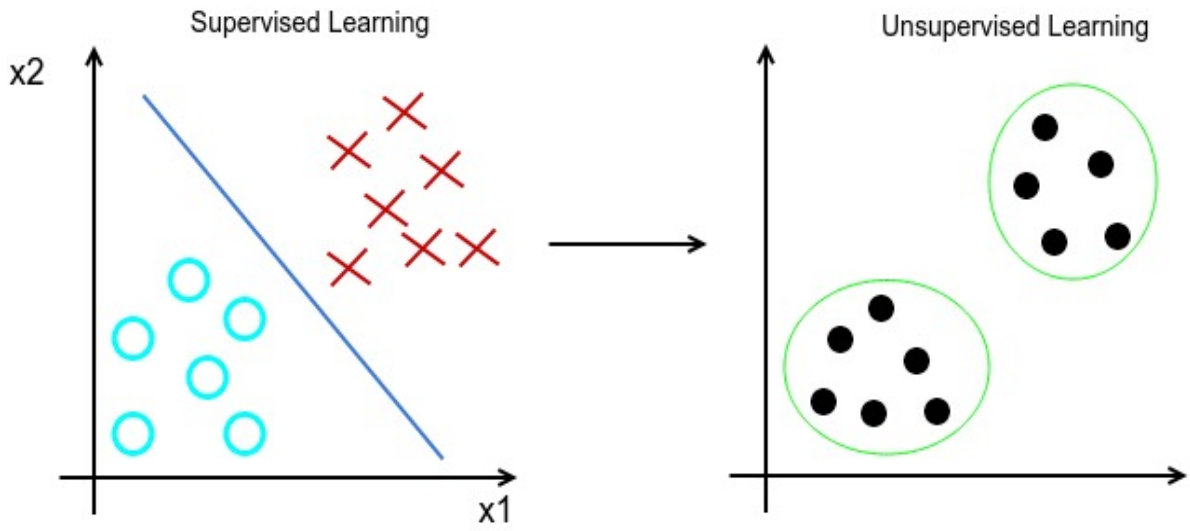
Finally, let's investigate different cases, to understand where logistic regression or SVMs are more appropriate :

- if n is large (relative to m)
  - Use logistic regression, or SVM without a kernel (“linear kernel”)
- if n is small, m is intermediate
  - Use SVM with Gaussian kernel
- If n is small, m is large
  - Create/add more features, then use logistic regression or SVM without a kernel

Note: neural networks are likely to work well for most of these settings, but are slower to train.

## *9. Unsupervised Learning*

In unsupervised learning, we learn from unlabeled data instead of labeled data (as in supervised learning).




*Unsupervised Learning - Unlabeled Data*

As we can see in the second case of unsupervised learning, our aim is to detect a structure in the unlabeled data, so that we can separate them in groups, the so-called ***clusters***. For this reason, this method is called clustering.

## *9.1 Clustering*

Clustering can be used in various applications. Some of them are the following :

- Market Segmentation
- Social Network Analysis
- Organize computing clusters
- Astronomical data analysis

One of the most famous clustering algorithms is the k-means algorithm.

## *9.2 K-means algorithm*

The basic idea of the algorithm is that we start with a specific number of points, called ***cluster centroids***, where the number is the same with the number of clusters that will be formed later. So, each cluster centroid corresponds to one cluster, and more specifically it represents its "weight center". The algorithm is an iterative algorithm and each step moves the cluster centroids in order to reduce the total distance of the points from the cluster centroids (thus reducing the overall error of the clustering). Each iteration of the algorithm contains 2 steps :

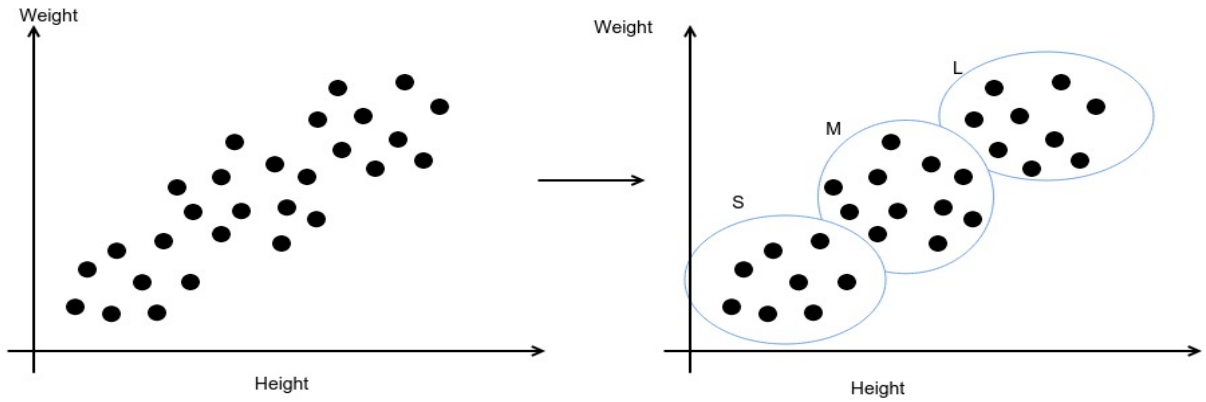
- The first step is the ***cluster assignment step***. In this step, each example is assigned to the cluster centroid that is nearest to it.
- The other step is the ***move centroid step***. In this step, for each cluster, we calculate the mean of all examples belonging to this cluster and then we move the corresponding cluster centroid to this mean.

So, the formal definition of k-means algorithm is the following :

$$
\begin{aligned}
&\textit{Randomly initialize } K \textit{ cluster centroid } \mu_1, \mu_2, \ldots, \mu_k \in \Re^n \\
&\textit{Repeat}\{ \\
&\quad \textit{for } i = 1 \text{ to } m \\
&\qquad c^{(i)} = \text{index}\,(\text{from } 1 \text{ to } K)\,\text{of } \textit{ cluster centroid closest to } x^{(i)} \\
&\quad \textit{for } k = 1 \text{ to } K \\
&\qquad \mu_k = \textit{average}\,(\textit{mean})\,\textit{of points assigned to cluster } k \\
&\}
\end{aligned}
$$

---

However, there are cases, where the data are not well separated and it initially seems that there are no clusters to be formed. We can see, for example, the following plot. However, if we execute a clustering algorithm with a suitable number of clusters, we will see that there were indeed suitable clusters that are now revealed by the algorithm. In our case, 3 clusters were selected, because the presumable company would like to have 3 t-shirts sizes (S,M,L).



Now, we are going to analyze the optimization objective of k-means algorithm. While we can just use the previous formal definition of the algorithm in order to implement k-means algorithm, the knowledge of optimization objective will help us :

- debug our algorithm and make sure it is working successfully
- handle cases, where the algorithm might get stuck into local optima

The optimization objective of k-means algorithm (as expected) is to minimize the sum of the distances of each example from its current cluster centroid (this is also called ***distortion cost***). The formal definition of the optimization objective is the following :

$$c^{(i)} = \text{index of cluster } (1, 2, \ldots, K) \text{¿ which example } x^{(i)} \text{ is currently assigned}$$

$$\mu_k = \text{cluster centroid } k \; (\mu_k \in \Re^n)$$

$$\mu_{c^{(i)}} = \text{cluster centroid of cluster to which } x^{(i)} \text{ has been assigned}$$

***Optimization objective***

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

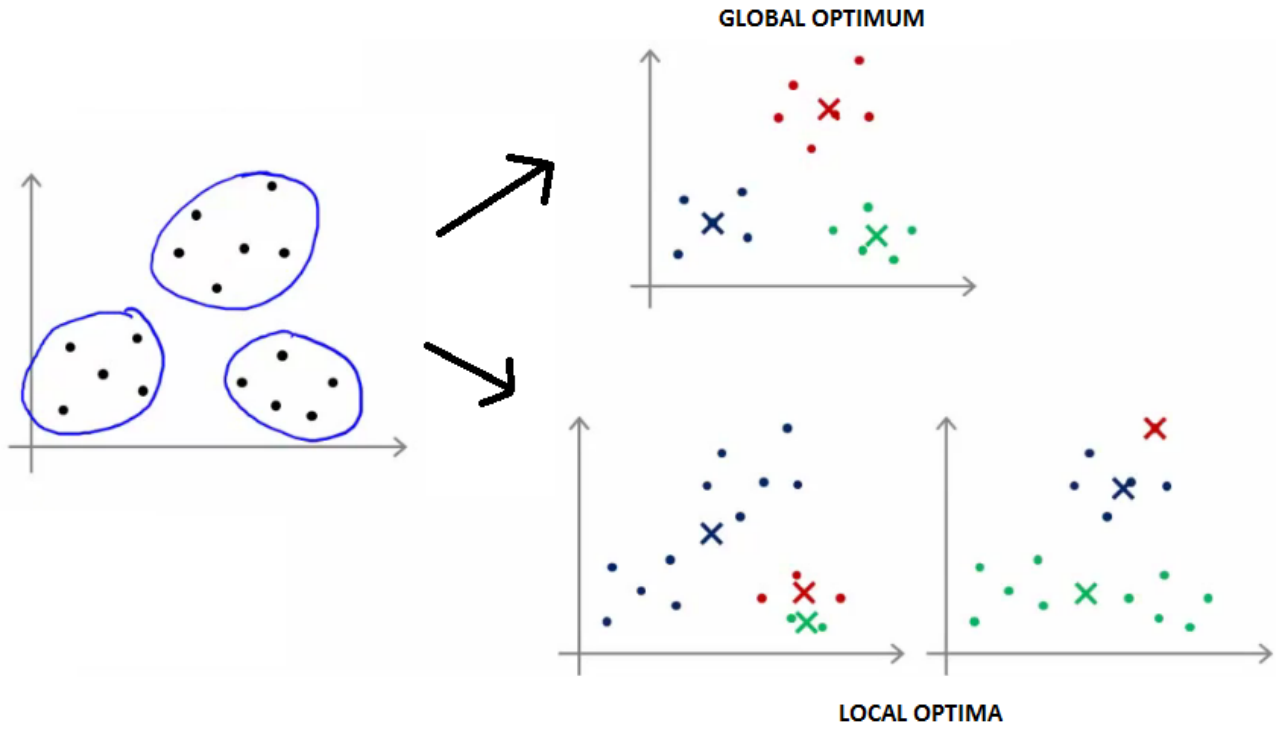
$$\min_{c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_k} J\left(c^{(1)}, \ldots, c^{(m)}, \mu_1, \ldots, \mu_k\right)$$

### *9.2.1 Random Initialization*

As we described previously, the k cluster centroids will have to be randomly initialized. But, how exactly is this done ? It is done in the most simple way :

- First of all, we should have K < m
- Randomly pick K training examples
- Set $\mu_1, \cdots, \mu_k$ equal to these K examples

However, we also previously stated that the k-means algorithm can be stuck in local optima, as it can be seen in the following case :




In order to avoid getting stuck into local optima, a common approach is the following one. We repeat the k-means algorithm multiple times using random initialization each one and we keep the solution with the minimum distortion cost. A typical range of iterations of k-means algorithm is between 50-1000.

This approach works relatively well, when we have a small number of clusters (k = 2-10). However, when dealing with more clusters, it is quite possible that all the iterations will not give us the global optimum, but merely a slightly better local optimum (than the one we got in the 1st iteration).

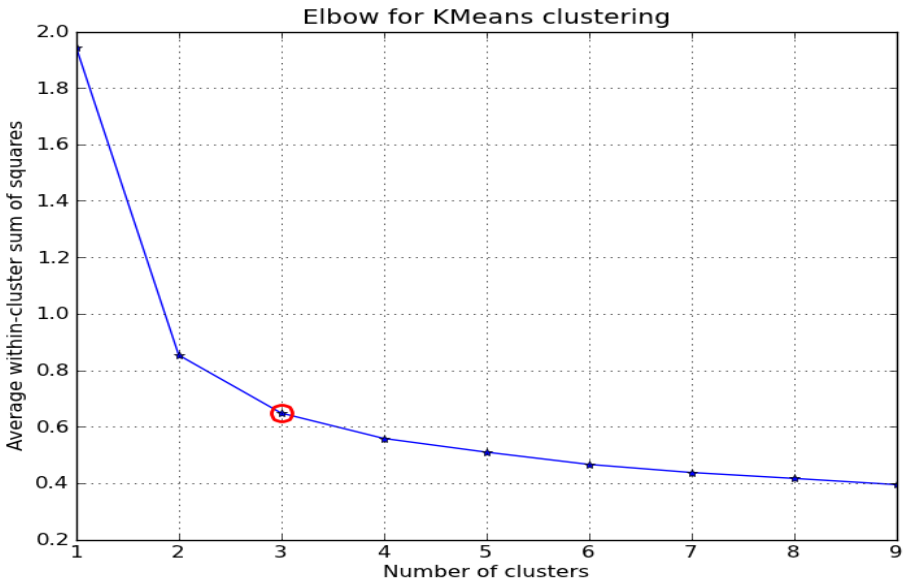
### *9.2.2 Choosing the number of Clusters*

Choosing the number of clusters can be done in many ways. The 3 most basic ways are the following :

- manual selection (perhaps using visualisation)
- elbow method
- based on a metric from later downstream purpose

In the first case of manual selection, sometimes the data can also be visualised, so that we can visually distinguish the data and manually select a number of clusters that are obvious. However, there are cases, where the visualisation is quite ambiguous.

The elbow method is a method, where we are trying to minimize the distortion function. We execute k-means algorithm with various number of clusters and we plot the distortion function. The plot usually takes the shape of a human elbow, so we select the number of clusters, where the gains from further increasing the clusters is somewhat limited.




The approach of using later downstream processes in order to define a metric is used, whenever the problems has a business or a more practical side. For instance, let's assume we have the example of the company with the t-shirts. The company hesitates between choosing among 3 (S,M,L) and 5 (XS,S,M,L,XL) sizes of t-shirts (thus clusters). Thinking from a business perspective, we can see that the company might want to reduce the number of sizes in order to reduce the costs and provide cheaper t-shirts. On the other

hand, the company might also benefit from providing more sizes (even in a higher price), since clients will be more satisfied. So, a suitable metric has to be created that takes this tradeoff into account. This metric will be later used to decide the number of sizes that will be selected.

## *10. Dimensionality Reduction*

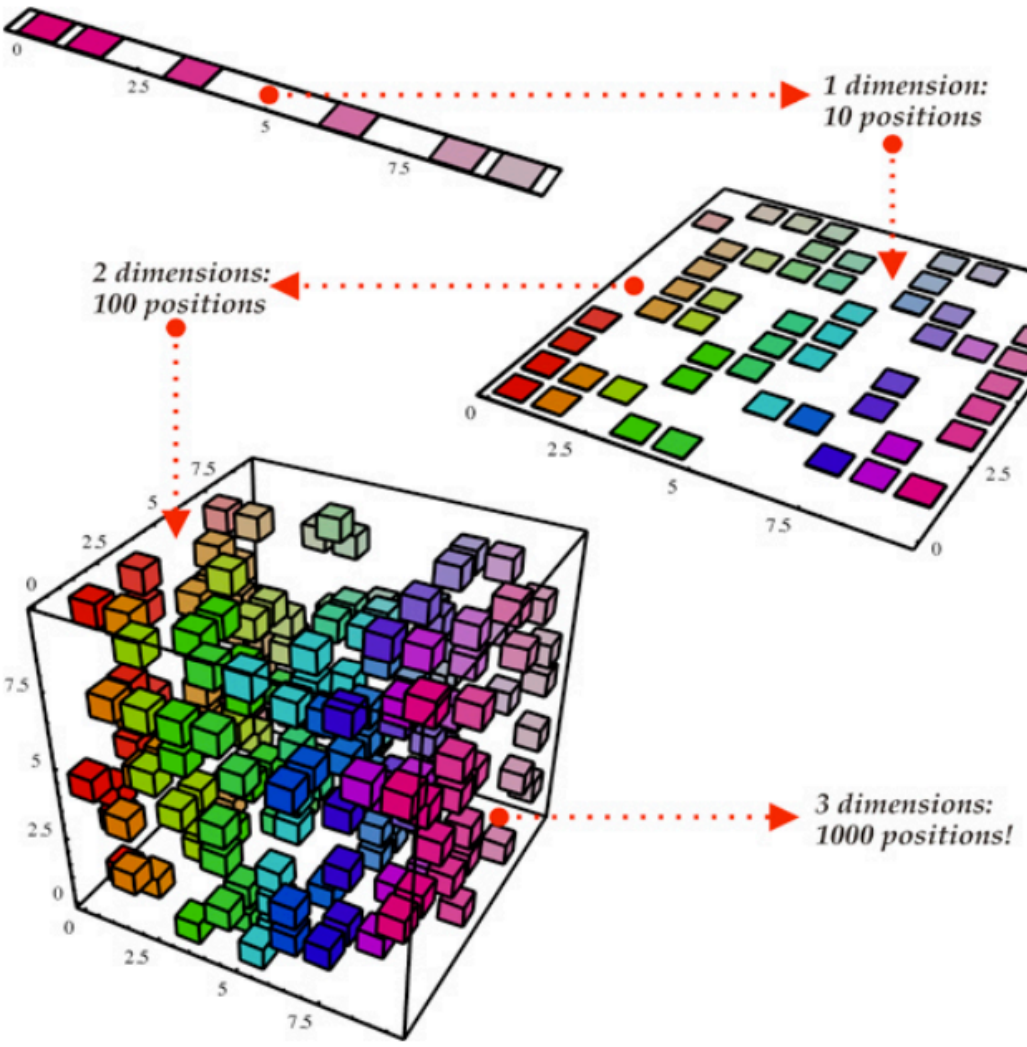
In many problems, the dimensions of the data are really large to be easily understood by humans (or be visualised). However, without data visualisations it is more difficult to gain insight about a problem. So, there is a n attempt to reduce the dimensions of a problem, so that it is more easily analysed and understood. This process is called ***dimensionality reduction***.

There are various ways, where we might have redundant data that can easily be "reduced". A simplistic case of redundant data is if we have 2 variables that correspond to the same measurement but in different units (let's say in inches and centimeters). This might quite silly, but think of big engineering teams of 20-50 people, where it is more difficult to have a tight cooperation and verify all the data. In this case, we would like to reduce the dimension of the input from 2D to 1D. However, there are also cases, where the redundance might not be so obvious. For instance, assume we measure a pilot's skills and the pilot's enjoyment as input features. Despite not being so visible to you, those 2 features are correlated and we could replace them with 1 feature (i.e. pilot's "aptitude").

So, the purposes of dimensionality reduction are :

- ***data compression*** (reduce memory-disk footprint, faster algorithms)
- ***data visualisation*** (more ease to visualise data)

In mathematical terminology, data compression is accomplished through projections. So, from 2D to 1D, we go by projecting into a straight line, while from 3D to 2D, we go by projecting into a plane.




*Data Compression*

When needed to use dimensionality reduction for data visualisation, we will probably have a lot bigger dimensional problem. For instance, we might have 100 features for each country and we want to visualise the state of each country in some way. In this case, we can "create" 2 features corresponding to the overall country size-GDP and the per person GDP. Those features will be deduced by using specific equations that incorporate all of the 100 initial features. Now, we can visualise those 2 produced features and get an insight about the countries states.

## *10.1 Principal Component Analysis*

By far the most common used algorithm for dimensionality reduction is the ***Principal Component Analysis*** algorithm. The basic idea behind this algorithm, is that it tries to find a lower dimensional surface onto which to project the data, so that the sum of squares of the distances of the points from the surface are minimized.

So, the formal definition of the algorithm is the following :

$$\textbf{Reduce from } 2-\textbf{dimension to } 1-\textbf{dimension}: \textit{Find a direction} \left(\textit{a vector } u^{(1)} \in \Re^n\right)$$
$$\textit{onto which to project the data so as ¿ minimize the projection error}$$

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

**Note:** *Even though PCA resembles linear regression, they are 2 different algorithms. In order to make this visible, check that in linear regression we are calculating the vertical distances from the line, while in PCA we are calculating the shortest orthogonal distances (which are drawn from the projection* **!!**

Now, we will see the exact algorithm used to find the vectors onto which the projection is made. The mathematical proof of the formulas of PCA are quite complicated, so they are currently omitted. The algorithm is the following :

$$\textit{After mean normalization} \left(\textit{ensure every feature has zero mean}\right) \text{ and } \textit{optionally feature scaling}$$

$$Sigma = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)}\right)\left(x^{(i)}\right)^T \underset{\textit{octave vectorized}}{\rightarrow} \textbf{sigma} = (1/m) * X' * X, \quad \textit{where } X = \begin{bmatrix} \left(x^{(1)}\right)^T \\ \ldots \\ \left(x^{(m)}\right)^T \end{bmatrix}$$

$$[U, S, V] = svd(sigma)$$

$$\textit{The first } k \textit{ columns of } U \textit{ are the needed vectors and form the matrix } U_{reduce}$$

$$Ureduce = U(1, 1:k);$$

$$z = Ureduce' * x;$$

**Note1** : Instead of octave function svd(), we can also use the function eig(). But we preferred the former, because it is more numerically stable.

**Note2** : The dimensionality reduction is made prior to adding the $x_0$ factor in the data.

## *10.2 Applying PCA*

After learning the PCA algorithm, we are now going to check some things needed when applying PCA, such as :

- Reconstruction from Compressed Representation
- Choosing the number of principles
- Advice for applying the algorithms

---

In some cases, we want to reconstruct the initial data from the compressed representation. The following equation is a way to retrieve an approximation of the initial data (aka the projections and not the initial data) :

$$x_{\text{approx}} = U_{reduce} * z$$

---

The number K of vectors that we retain for the dimensionality reduction (that correspond to the number of reduced dimensions) is also called number of ***principal components***. This number is a parameter of the algorithm. The general idea is that we want to reduce the dimensions to the lowest possible, so that the difference between the approximation and the initial data is kept small.

The algorithm for choosing the number K is the following :

$$\begin{aligned}
&\text{for}(k=1 \text{ to } m)\{ \\
&\quad \text{Compute } U_{reduce}, z^{(1)}, \ldots, z^{(m)}, x^{(1)}_{\text{approx}}, \ldots, x^{(m)}_{\text{approx}} \\
&\quad \text{Check } if \ \frac{\sum_{i=1}^{m} \left\| x^{(i)} - x^{(i)}_{\text{approx}} \right\|^2}{\sum_{i=1}^{m} \left\| x^{(i)} \right\|^2} \leqslant 0.01 \\
&\}
\end{aligned}$$

In Machine Learning terminology, this means that at least 99% of the variance is preserved. Note that at some cases, we can even accept lower percentages (some lower typical numbers are 95% and 90% in extreme cases).

However, the previous algorithm is very slow. So, we can take advantage of the svd() numerical function to calculate the same percentage. The way is the following :

$$[U, S, V] = svd(Sigma)$$
$$Pick\ smallest\ k\ value\ \text{for}\ which$$
$$\frac{\sum_{i=1}^{k} S_{ii}}{\sum_{i=1}^{m} S_{ii}} \geqslant 0.99$$
$$(99\%\ of\ variance\ retained)$$

---

Principal Component Analysis can also be used in supervised learning speedup. For instance, imagine a problem where the input dimension is 10,000 (a 100x100 pixels image). The algorithm would be extremely slow with such a big input, so we can use PCA to reduce the dimensions. However, ***note that the mapping x → z should be defined by running PCA only on the training set and this mapping can then be applied as well to the examples in the cross validation and test datasets. The same counts for feature scaling and mean normalization (range and mean are defined only by the training set).***

Summarizing the previous, we remind that PCA is used for 2 purposes, data compression and visualisation. When used for compression, the parameter k is defined by the predefined algorithm. When it is used for visualisations, in most cases, k = 2 or k = 3.

Sometimes, PCA algorithm is used to prevent overfitting, since someone might think that fewer features are less likely to overfit. This might work in some cases, but it is not a good way to address overfitting !! We should use regularization instead.

Finally, sometimes PCA is used where it should not be. So, when planning ahead for a Machine Learning project, someone might pre-plan to implement PCA as a data preprocessing step. However, if there is no problem with the data this would be futile. So, in most cases, we should implement our algorithm initially without using PCA with the original/raw data x. Only if does not do what we want, then we should implement PCA.

## *11. Anomaly Detection*

Anomaly Detection is a very common application of Machine Learning (with extensive usage in manufacturing industry). The problem of anomaly detection is the following : we have a set of unlabeled data (that are all considered normal – not anomalous). We want to produce a model p(x), that given an example, gives us the probability of a new exampe to take this value. So, if the probability of a value is lower than a (low) threshold ε, then we flag this example as anomalous. And for this modeling, the Gaussian distribution is used, since almost all "normal" phenomena-variables are ruled by this distribution.

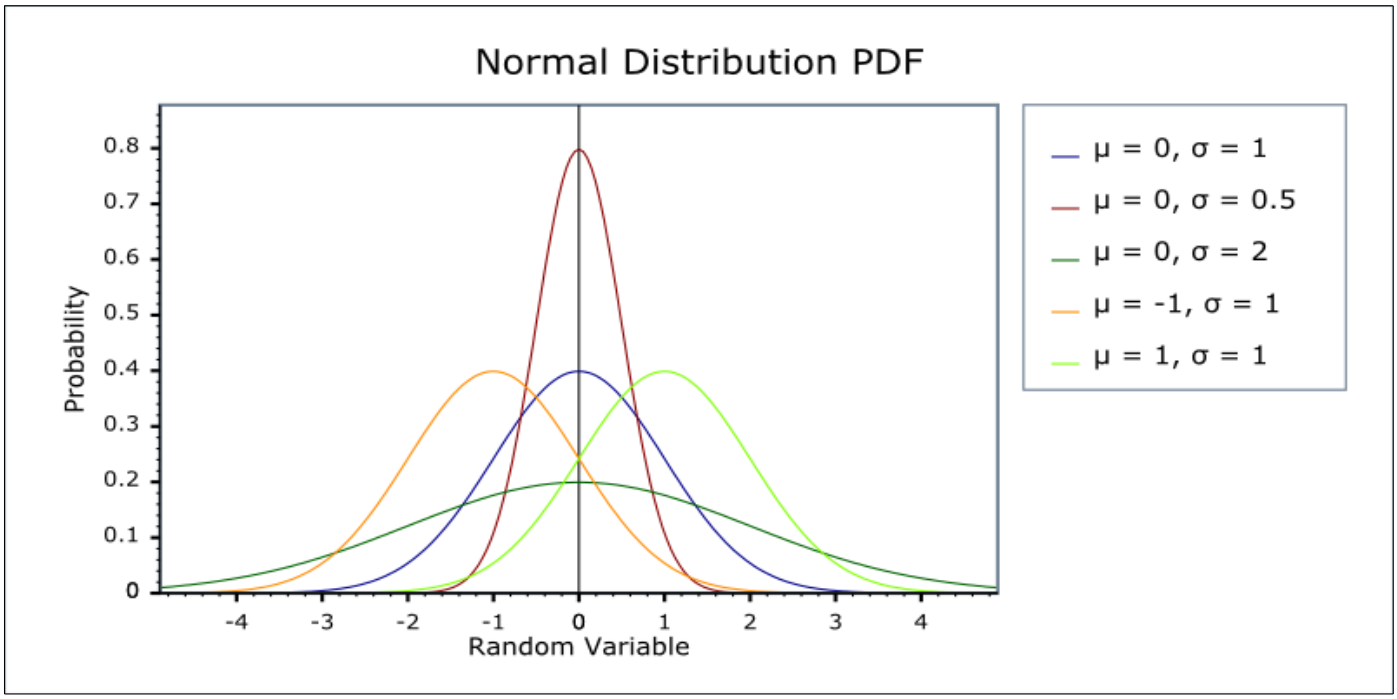
The normal distribution is parameterized by 2 values (μ, σ) that are calculated with the following formulas :

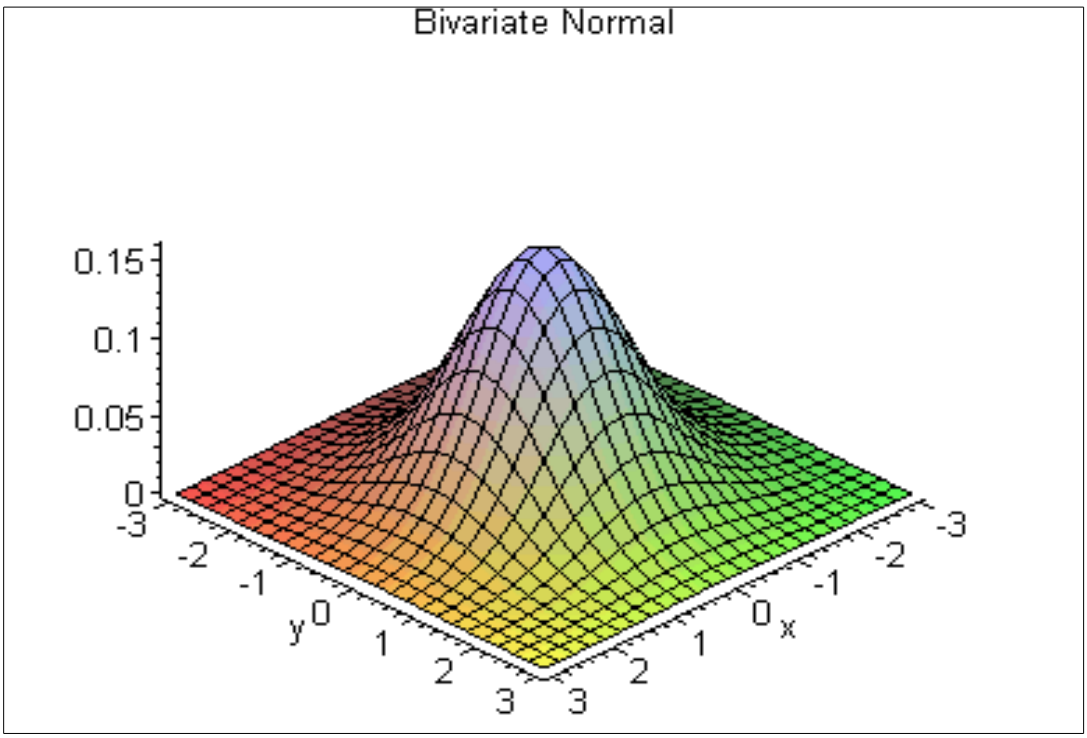
$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

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

Note: in strict Statistics, sometimes, 1/(m-1) is used instead of 1/m. But, when m (large dataset) is large, it makes almost no difference.

In the following plot, the meaning of those 2 parameters is visually described :






So, the algorithm for anomaly detection is the following :

1. Choose features $x_i$ that you think might be indicative of anomalous examples

2. Fit parameters $\mu_1, \ldots, \mu_n, \sigma_1^2, \ldots, \sigma_n^2$

3. Given new example, compute $p(x)$:

$$p(x) = p(x_1; \mu_1, \sigma_1^2) * \ldots * p(x_n; \mu_n, \sigma_n^2) = \prod_{j=1}^{n} p(x_j; \mu_j, \sigma_j^2)$$

$$\text{Anomaly if } p(x) < \varepsilon$$

Note: In order to produce this algorithm the assumption of variable independence was made. However, the algorithm will work even if the variables are not completely independent, so we should not worry about that.

## *11.1 Building an Anomaly Detection System*

### *The importance of real-number evaluation*

When developing a learning algorithm, making decision is much easier if we have a way of evaluating our learning algorithm.

So, let's investigate how the evaluation of an anomaly detection algorithm can be made. First of all, we have to see how the datasets (training-cross validation-test) are selected. For instance, let's take the next example :

## *Aircraft engines example*

$10{,}000\ good\ (normal)\ engines$

$20\ flawed\ (anomalous)\ engines$

***Selection*** **:**

$Training\ Set: 6{,}000\ good\ engines$

$CV: 2{,}000\ good\ engines, 10\ anomalous\ engines$

$Test: 2{,}000\ good\ engines, 10\ anomalous\ engines$

Note: the training set must not contain anomalous samples !!

So, the evaluation algorithm will be the following :

## *Algorithm Evaluation*

$Fit\ model\ p(x)\ on\ training\ set\ \{x^{(1)}, \ldots, x^{(m)}\}$

$On\ a\ cross\ validation/test\ example, predict$

$$y = \begin{cases} 1, \text{if } p(x) < \varepsilon\ (anomaly) \\ 0, \text{if } p(x) \geq \varepsilon\ (normal) \end{cases}$$

$Possible\ evaluation\ metrics:$

$-\ True\ positive, false\ positive, true\ negative, false\ negative$

$-\ Precision/Recall$

$-\ F_1\ score$

$Can\ also\ use\ cross\ validation\ set\ to\ choose\ parameter\ \varepsilon$

Note: The case of anomaly detection is another case of skewed data. So, we should not use accuracy as an evaluation metric, as explained in previous chapter.

---

In some cases, there is a misconception between anomaly detection and supervised learning. These algorithms can also be "exchanged", but we should carefully select which of them to implement, because otherwise the results will not be satisfactory.

In the following table, we can see some basic differences between the 2 algorithms, when we should apply each one and some examples :

| Anomaly Detection | Supervised Learning |
|---|---|
| Very small number of positive examples (y=1). (0-20 is common) | Large number of positive and negative examples |
| Large number of negative examples | |
| | |
| Many different "types" of anomalies. Hard for any algorithm to learn from positive examples what the anomalies look like; future anomalies may look nothing like the anomalous examples we have seen so far. | Enough positive examples for the algorithm to get a sense of what positive examples are like, future positive examples likely to be similar to ones in training set. |
| ***Examples*** | |
| Fraud detection | Email spam classification |
| Manufacturing | Weather prediction |
| Monitoring machines in data center | Cancer classification |

---

A last important part of anomaly detection algorithm is the selection of features to use. In order to check if a feature can be used, we can plot a histogram of the variable and check if it is normally distributed. If it is not, then we can apply a transformation to the variable (raise to power, square root or log) so that the distribution becomes normal.

However, there are also cases, where we have selected the features and our problems is that the possibility for anomalous examples is not a lot smaller than the normal examples. In this case, we probably have to think of new features to insert in our model, or in some cases, we have to think of a new combination of the current features. For instance, when monitoring datacenters, the features CPU load, Network traffic alone could not give us very good results. But if we create a new feature from the ratio of those 2 (CPU load / Network traffic), this might be a better option.

## *11.2 Multivariate Gaussian Distribution*

There are some cases, where the original anomaly detection algorithm we presented previously is not enough to cover more complex correlations between the features we have selected. Take for example, a case of a data center, where features x1 and x2 correspond to memory usage and CPU usage. With the original model, a combination of low memory usage with high memory CPU would be equally anomalous-normal to a combination of low memory usage with low CPU usage. This seems somewhat irrational, but what we could do is create a new feature x3 = x1/x2, that would cover this correlation. However, there is another generic algorithm that takes all those correlation between the features into account. This algorithm is called ***Multivariate Gaussian Distribution*** and has the following formal definition :

***Anomaly Detection using Multivariate Gaussian Distribution***

1. *Fit model* $p(x)$ *by setting*

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m} (x^{(i)} - \mu)(x^{(i)} - \mu)^T$$

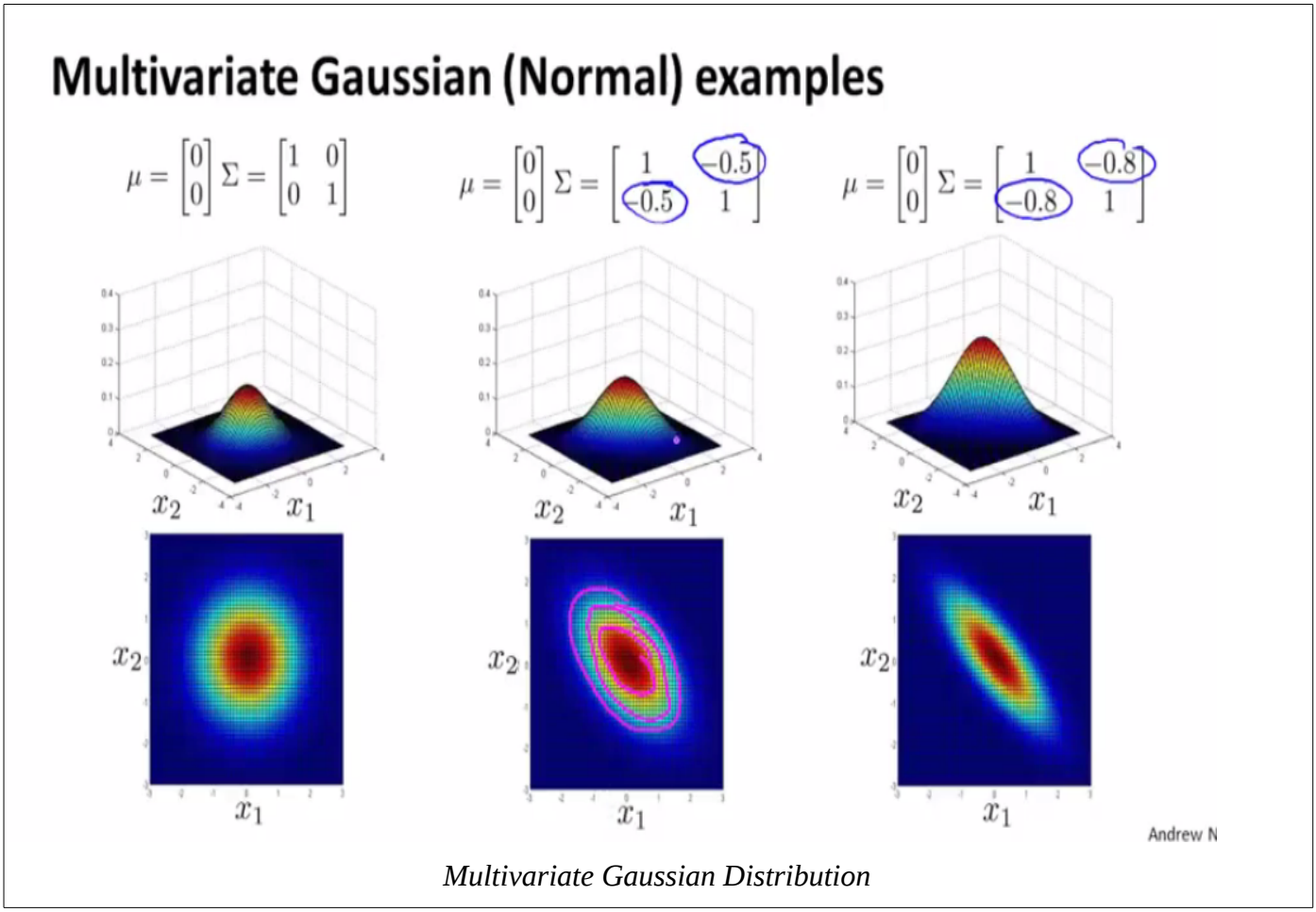
2. *Given a new example* $x$ *, compute*

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

*Flag an anomaly if* $p(x) < \varepsilon$

Note: The original model presented previously, in fact, is a special case of this more generic form, where the Σ has a specific form that makes the distribution plot be axis-aligned (while this one can also be diagonal).

In the following plots, we can see forms that can be represented by the multivariate gaussian distribution (but would be impossible with the original algorithm).




*Multivariate Gaussian Distribution*

Finally, let's summarize how we should select between the original algorithm and the Multivariate Gaussian Distribution :

| Original model | Multivariate Gaussian |
|---|---|
| $p(x)=p(x_1;\mu_1,\sigma_1^2)x\ldots xp(x_n;\mu_n,\sigma_n^2)$ | $p(x)=\frac{1}{(2\pi)^{(n/2)}\lvert\Sigma\rvert^{1/2}}\exp(\frac{-1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu))$ |
| We have to manually create features to capture anomalies where x1,x2 take unusual combinations of values (i.e. x3= x1/x2) | Automatically captures correlations between features |
| Computationally cheaper (alternatively, scales better to large n) | Computationally more expensive |
| OK, even if m (training set size) is small | Must have m > n or else Σ is non invertible (typical case : m >= 10n) |

# *12. Recommender Systems*

Recommender Systems is an extensive field of Machine Learning with many applications in industry (you can check the recommender systems of Netflix, Amazon etc.). In Machine Learning, sometimes we have a hard time trying to find the correct features that should be used. However, there are some ML algorithms that can detect the features to be used on their own, and Recommender Systems is one of those algorithms.

The formulation of the problem of Recommender Systems is the following : We have a set of movies (or songs, books, ...) and a set of users. We also have the ratings the users have given to some movies. The purpose is to create a learning algorithm that can "guess" the ratings of the users for the movies that have not been rated yet, so that we can recommend to the users some of the movies they have not yet rated and the "predicted" rating is high.

In the following sections, we will see 2 algorithms of different types :

- content based, where we have features of each movie
- collaborative filtering, where the algorithm learns the features on its own

## *12.1 Content Based Recommendations*

One type of recommenders systems use the content based approach. This means that we assume we have some features for each movie that are characterizing it, like "action", "romance", etc. An example of content based approach can be seen in the following table :

| Movie | Alice(1) | Bob(2) | Carol(3) | Dave(4) | x1(romance) | x2(action) |
|---|---|---|---|---|---|---|
| ***Love at last*** | 5 | 5 | 0 | 0 | 0.9 | 0 |
| ***Romance for ever*** | 5 | ? | ? | 0 | 1.0 | 0.01 |
| ***Cute puppies of love*** | ? | 4 | 0 | ? | 0.99 | 0 |
| ***Car chases*** | 0 | 0 | 5 | 4 | 0.1 | 1.0 |
| ***Swords vs Karate*** | 0 | 0 | 5 | ? | 0 | 0.9 |

The formal problem definition and the content based algorithm are the following :

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

$$GradientDescentUpdate:$$

$$\theta_\kappa^{(j)}=\theta_\kappa^{(j)}-\alpha\sum_{i:r(i,j)=1}\left(\left(\theta^{(j)}\right)^T x^{(i)}-y^{(i,j)}\right)x_k^{(i)}\left(for\ k=0\right)$$

$$\theta_\kappa^{(j)}=\theta_\kappa^{(j)}-\alpha\left(\sum_{i:r(i,j)=1}\left(\left(\theta^{(j)}\right)^T x^{(i)}-y^{(i,j)}\right)x_k^{(i)}+\lambda\theta_\kappa^{(j)}\right)\left(for\ k\neq 0\right)$$

## *12.2 Collaborative Filtering*

This method assumes that we do not know the features of the movies in advance, but the algorithm learns them progressively.

The basic idea is the following. If we did not know the features of each movie, but we knew the θ parameters for the ratings of the users, then we could analogously predict the features. Thus, the knowledge of θ helps us predict the features x, and reversely. The equations that are used for each case are the following :

$$\textbf{Given } \theta^{(1)},\dots,\theta^{(n_u)}\textbf{, to learn } x^{(1)},\dots,x^{(n_m)}\textbf{:}$$

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

$$\textbf{Given } x^{(1)},\dots,x^{(n_m)}\textbf{, to learn } \theta^{(1)},\dots,\theta^{(n_u)}$$

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

So, we can see that this problem is like the chicken-egg problem. Who comes first ? The algorithm of collaborative filtering is an iterative algorithm, that starts by a set of guessed θ parameters and then goes on with consecutive calculations, as below :

$$Guess\ \theta\rightarrow calculate\ x\rightarrow calculate\ \theta\rightarrow calculate\ x\rightarrow\dots$$

However, it can be mathematically proven that there is no need for this iteration and we can come up with a single cost function J, that needs to be minimized simultaneously for parameters θ and features x. This is the algorithm called ***collaborative filtering***. The corresponding equations are the following :

***Collaborative filtering algorithm :***

1. *Initialize* $x^{(1)},\dots,x^{(n_m)},\theta^{(1)},\dots,\theta^{(n_u)}$ *¿ small random values*
2. *Minimize* $J\left(x^{(1)},\dots,x^{(n_m)},\theta^{(1)},\dots,\theta^{(n_u)}\right)$ *using gradient descent* (or *other advanced optimization algorithm*) *using the following equations* :

$$x_k^{(i)} = x_k^{(i)} - \alpha\left(\sum_{j:r(i,j)=1}\left(\left(\theta^{(j)}\right)^T x^{(i)} - y^{(i,j)}\right)\theta_k^{(j)} + \lambda\, x_k^{(i)}\right)$$

$$\theta_k^{(j)} = \theta_k^{(j)} - \alpha\left(\sum_{j:r(i,j)=1}\left(\left(\theta^{(j)}\right)^T x^{(i)} - y^{(i,j)}\right)x_k^{(i)} + \lambda\,\theta_k^{(j)}\right)$$

3. *For a user with parameters* $\theta$ and *a movie with* (*learned*) *features* $x$, *predict a star rating of* $\theta^T x$.

Note: In this algorithm, there is no reason to keep the interceptor terms $x_0 = 1, \theta_0$ .

## *12.3 Low rank matrix factorization*

In this section, we are going to see the vectorized implementation of collaborative filtering, that is called low rank matrix factorization. Also, we are going to see other things we can do with this algorithm, like discover related products.

This algorithm gives us a vectorized way to calculate the predictions of ratings

$$ratings = \begin{bmatrix} \left(\theta^{(1)}\right)^T\left(x^{(1)}\right) & \left(\theta^{(2)}\right)^T\left(x^{(1)}\right) & \dots & \left(\theta^{(n_u)}\right)^T\left(x^{(1)}\right) \\ \left(\theta^{(1)}\right)^T\left(x^{(2)}\right) & \left(\theta^{(2)}\right)^T\left(x^{(2)}\right) & \dots & \left(\theta^{(n_u)}\right)^T\left(x^{(2)}\right) \\ \dots & \dots & \dots & \dots \\ \left(\theta^{(1)}\right)^T\left(x^{(n_m)}\right) & \left(\theta^{(2)}\right)^T\left(x^{(n_m)}\right) & \dots & \left(\theta^{(n_u)}\right)^T\left(x^{(n_m)}\right) \end{bmatrix}$$

***vectorized :*** $ratings = X\Theta^T$, *where*

$$X = \begin{bmatrix} \left(x^{(1)}\right)^T \\ \left(x^{(2)}\right)^T \\ \dots \\ \left(x^{(n_m)}\right)^T \end{bmatrix}, \Theta = \begin{bmatrix} \left(\theta^{(1)}\right)^T \\ \left(\theta^{(2)}\right)^T \\ \dots \\ \left(\theta^{(n_u)}\right)^T \end{bmatrix}$$

In order to find a movie j related to movie i, we have to find the movie j with the lowest distance $\left\| x^{(i)} - x^{(j)} \right\|$ .

In recommender systems, a common problem is that there might be users that have not yet rated any movie. Thus, when executing the algorithm the regularization term will force all the θ parameters (and thus ratings) of this user to become zeros, one thing that is not really logical. A way to fix this problem is to execute mean normalization in the ratings of each movie. This is implemented in the following way :

***Mean normalization***

1. *For each movie i , predict average of ratings* (from *all users*) $\mu_i$
2. From *each rating , substract the mean* and *produce a new input Y ,*
   *where the collaborative filtering is executed*
3. *For user j on movie i , predict :* $(\theta^{(j)})^T (x^{(i)}) + \mu_i$

Note: In this way, for users that have no ratings at all, the algorithm will predict the average rating for this movie, which is a lot more reasonable.

## *13. Large scale Machine Learning*

One of the reasons Machine Learning has been evolved so much lately is the fact that we have increasing amounts of data. And as we have previously explained, the more data we have the better results the learning algorithm will produce. However, Machine Learning algorithms have to be implemented in such a way, that they are capable to scale and handle big data.

Suppose, for example, that we have a training set of size m = 1,000,000. This is a huge training set, so we had better come up with a way to implement our algorithm in order to improve its performance. However, first of all, we can make a sanity check, by executing the algorithm with a smaller subset (i.e. 1,000 examples) and plot the learning curves, in order check if the algorithm has satisfactory results even when trained with a smaller dataset. However, if the sanity check shows that adding more data will significantly improve our results, then we will have to take the whole dataset and implement the algorithm in way to be able to handle this big dataset. Here, we will meet 2 alternative algorithms of gradient descent that can handle big data :

- Stochastic Gradient Descent
- Mini-batch Gradient Descent

## *13.1 Stochastic Gradient Descent*

Stochastic Gradient Descent is an algorithm that can be used in all algorithms that contain an iterative gradient descent update step (linear regression, logistic regression, neural networks). The idea is that in original gradient descent, when dataset size m is very big, each step iterates over the whole dataset and this can become very slow. So, in Stochastic Gradient Descent, instead of iterating the whole dataset, each step uses only 1 example in order to calculate the next step. The algorithm is the following :

***Stochastic Gradient Descent***

1. *Randomly shuffle (reorder) training examples*
2. *Repeat* {
   for $i = 1, \ldots, m$ {
   $\theta_j = \theta_j - \alpha \left( h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}$
   }
}

Note: When we have an extremely big dataset (i.e. m = 300,000,000), then a single iteration over the dataset might be enough for the algorithm to converge. However, in typical cases, we may need 1-10 iterations.

We now have to think of an equally efficient way to be able to debug our algorithm and ensure that it is converging. Similar to original gradient descent, we will plot the cost function as a function of iterations completed. However, instead of calculating the whole cost, we will calculate the cost for a simple example at each step and we will plot the average over a bigger sample of examples (i.e. mean of 100 examples).

Note: The algorithm of Stochastic Gradient Descent will not completely converge, but it will oscillate near the global minimum. So, in order to fix this flaw, we can adapt-decrease the learning rate α as we get nearer to the global minimum. For example, we can calculate the learning rate α from an equation like ***α = constant1/(constant2 + iterations_no)***.

## *13.2 Mini-batch Gradient Descent*

The mini-batch Gradient Descent algorithm extends a little more the idea, on which Stochastic Gradient Descent was built. The mini-batch Gradient Descent does not use a single example at each iteration, but instead uses a small set of examples (called mini-batch). The profit from this change is that now we can vectorize-parallelize the algorithm and make it a little faster. The

formal definition of mini-batch Gradient Descent is the following :

$$
\begin{aligned}
&\textbf{\textit{Mini}} - \textbf{\textit{batch Gradient Descent}} \\
&Assume\ b = 10, m = 1{,}000 \\
&Repeat\ \{ \\
&\quad for\ i = 1, 11, 21, \ldots, 991\ \{ \\
&\qquad \theta_j = \theta_j - \alpha \frac{1}{10} \sum_{k=1}^{i+9} \left( h_\theta(x^{(k)}) - y^{(k)} \right) x_j^{(k)} \qquad (for\ every\ j = 0, \ldots, n) \\
&\quad \} \\
&\}
\end{aligned}
$$

Note: If b is selected carefully, mini-batch Gradient Descent can be even faster than Stochastic Gradient Descent.

## *3.3 Map Reduce*

When we have big data as input, and we execute our algorithms in a single computer, we can be sure that no matter how many optimizations are made in the algorithm, the algorithm will still be slow. In these cases, we have to parallelize the computations, so that we reduce the computing time. When having distributed systems (many computers) in our possesion, there is a framework created for this job, called Map Reduce. So, we can leverage this framework in order to divide the work in multiple computers.

In Map Reduce framework, we have a master computer and several slaves computers. The master computer divides the work in multiple parts and "maps" (allocates) them to the slaves. When all slaves are done, all the parts are "reduced" (collected) in the master computer and a final processing is done to merge them. In case of (single) batch Gradient Descent, for example, the sum over all the examples of the dataset can be divided into sub-datasets and each slave can compute a part of this sum. Then, during the reduction the master computer will gather the results and add them to calculate the total sum.

The only bottleneck in case of usage of Map Reduce in distributed systems is the network bandwith. However, if we have multiple cores in our system, we can execute Map Reduce over our cores, so that the network bandwidth limitation is diminished. This is also automatically done, when we use optimized numerical libraries of Octave (or Matlab etc.) and create vectorized implementations of our algorithms.

## *13.4 Online Learning*

There are some cases, where we have a continuous stream of data, where we want to learn interactively and improve the decisions we take. This is a specific field of Machine Learning, called ***Online Learning***.

During Online Learning, unlike other algorithms, we do not keep track of all the previous examples. Instead, we use the example we have each time, adapt and then "throw" it. So, the algorithm is the following :

***Online Learning***

$$\text{Repeat forever}\{$$
$$\quad \text{Get}(x,y)$$
$$\quad \text{Update } \theta \text{ using } (x,y)$$
$$\quad\quad \theta_j = \theta_j - \alpha(h_\theta(x) - y)\,x_j \qquad (j=0,..,n)$$
$$\}$$

In an alternate version, we can also store the previous examples and execute the algorithms over a set of previous examples. However, the performance will become worse. Anyway, the online learning algorithm has very good results even over a single example. And the impressive when executing only with the current example is that the algorithm adapts to changes in "preferences" (behaviour-θ).

Some application examples of online learning are the following :

A shipping service website, where user comes, specifies origin and destination, you offer to ship their package for some asking price and users sometimes choose to use your shipping service (y=1) or not to use it (y=0). So, features x capture properties of user, of origin/destination and asking price. We want to learn p(y=1; x,θ) to optimize price.

## *13.5 Ceiling Analysis*

In most cases, Machine Learning systems are quite complex and constitute of multiple components. They are made following an architecture, called *pipeline architecture*. So, when having those systems with various systems, we usually evaluate the whole system. If we are not satisfied with the performance of the system, we have to find which component of the system can receive the biggest percentage of improvement. There is a procedure for this identification, called ***ceiling analysis***.

First of all, ceiling analysis is independent of metrics. So, one can use whatever attribute as the metric to evaluate the system. The procedure is the following :

- We calculate the accuracy of the whole system
- Then, we replace the 1st component by a "perfect" component (implemented manually) and we calculate the accuracy of the whole system again
- We also replace the 2nd component with a "perfect" one and calculate the new accuracy of the whole system again
- We repeat for each component of the system
- On each iteration, the observed difference in the overall accuracy shows the potential impact of an improvement in the replaced component to the whole system

So, the components that present the biggest differences are those, where we should focus on.