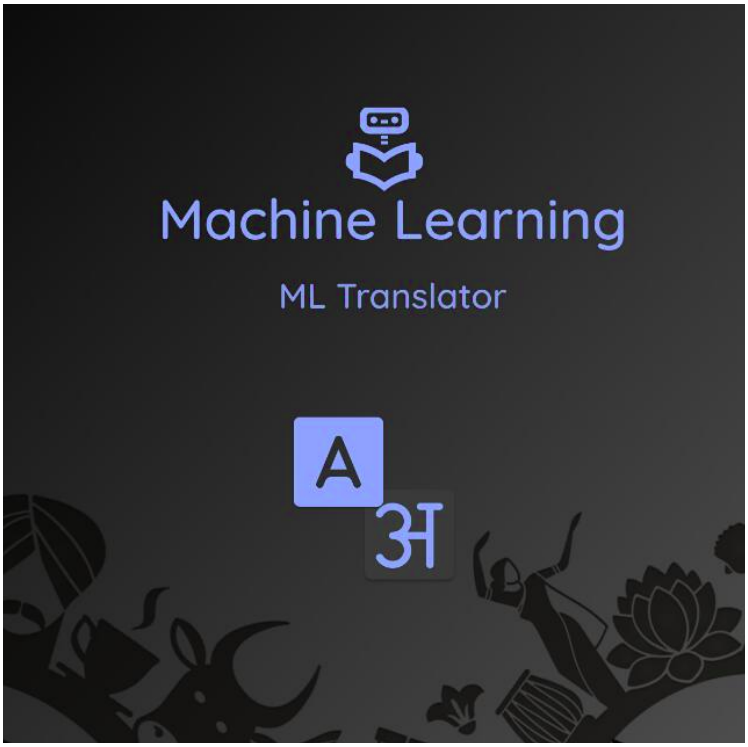
PL505 Machine Learning Lab

Visual Introduction to Machine Learning



Group: ML Translator

1901	AISHWARYA GANESH
1927	PRIYA MAJALIKAR
1930	ROCHELLE MARTINS
1943	ONKAR PARSHEKAR
1944	MINAXI PATIL

Contents

Introduction

The advancement and interest for Machine Learning, AI, Deep Learning is fast increasing. Every year about 33,000 papers are published. This book provides an introduction to various concepts with the aid of images.

What is Machine Learning?

Machine Learning is the art & science of programming computers to learn from data.

"*ML is the field of study that gives computers the ability to learn without being explicitly programmed.*" , Arthur Samuel

A more formal definition:
"*A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.*" Tom Mitchell

Why use Machine Learning?

When building non-learners, we usually follow these steps:

1. We make rules
2. We write an algorithm
3. If the algorithm performs well, we deploy. If not, we go back to step 1

However, if the problem is complex, we'll likely end up with a long list of rules that are hard to maintain and scale to other similar problems. An ML system would be much shorter, easier to maintain, and in many cases, more accurate.

We can simply train an algorithm on a large dataset, then inspect the algorithm's feature importance coefficient to gain a better understanding of the relation between the data & the problem. This is called data mining.

Supervised Machine Learning

Supervised learning is a machine learning type where machines are trained by providing it with input data as well as correct output data to the machine learning model.

The aim of a supervised learning algorithm is to **find a mapping function** to map the input variable(x) with the output variable(y).

Vector Spaces and the Candidate Elimination Algorithm

Concept Learning: Here, the machine is taught to infer the general definition of some concept, given examples labelled as members or non-members of the concept.

One approach to concept learning is the **Candidate Elimination Algorithm.**

Candidate Elimination Algorithm finds all describable hypotheses that are consistent with the training examples. The subset of all hypotheses is called the **version space**.

Given	A set of positive and negative examples
Compute	A description that is consistent with -> all the positive examples -> none of the negative examples.

Notations

?	indicates any value acceptable
value	*specifies a single value Eg. Red*
Φ	no value acceptable

Method

1. Initialize G to the set of maximally general hypotheses in H and initialize S to the set of maximally specific hypotheses in H
2. For each training example d,
    - If d is a positive example
        - Remove from G any hypothesis inconsistent with d
        - For each hypothesis s in S that is not consistent with d
            - Remove s from S
            - Add to S all minimal generalizations h of s such that h is consistent with d, and some member of G is more general than h
            - Remove from S any hypothesis that is more general than another hypothesis in S
    - If d is a negative example
        - Remove from S any hypothesis inconsistent with d
        - For each hypothesis g in G that is not consistent with d
            - Remove g from G
            - Add to G all minimal specializations h of g such that h is consistent with d, and some member of S is more specific than h
            - Remove from G any hypothesis that is less general than another hypothesis in G

Version Space method example.

Concept: Japanese Economy Car

Training Examples

Origin	Manufacturer	Color	Decade	Type	Example
Japan	Honda	Blue	1980	Economy	Positive
Japan	Toyota	Green	1970	Sports	Negative
Japan	Toyota	Blue	1990	Economy	Positive
USA	Chrysler	Red	1980	Economy	Negative
Japan	Honda	White	1980	Economy	Positive

Method:

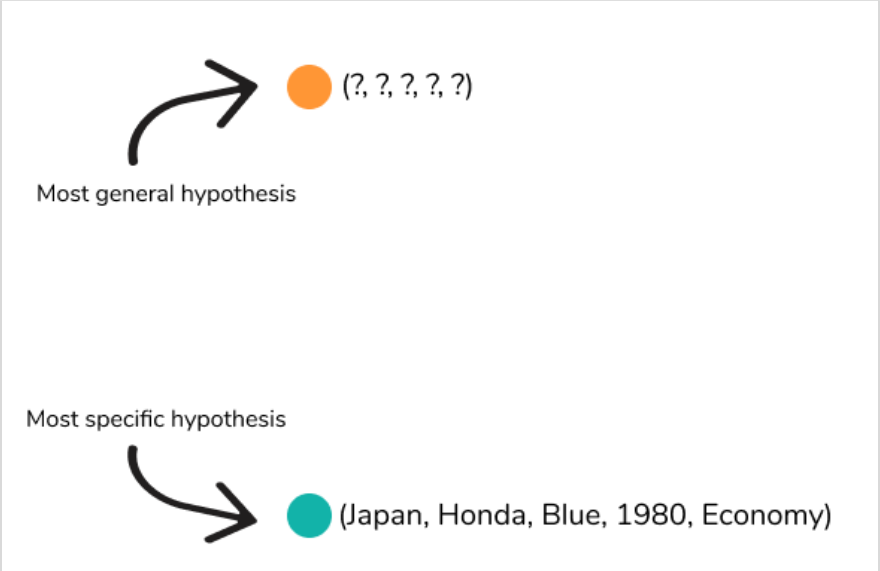
1. Start by initializing the G boundary set to contain the most general hypothesis

$$G_0 = \{(?, ?, ?, ?, ?)\}$$

Here, any values will be classified as positive.

and initializing the S to a set to include the first positive example

$$S_0 = \{(Japan, Honda, Blue, 1980, Economy)\}$$



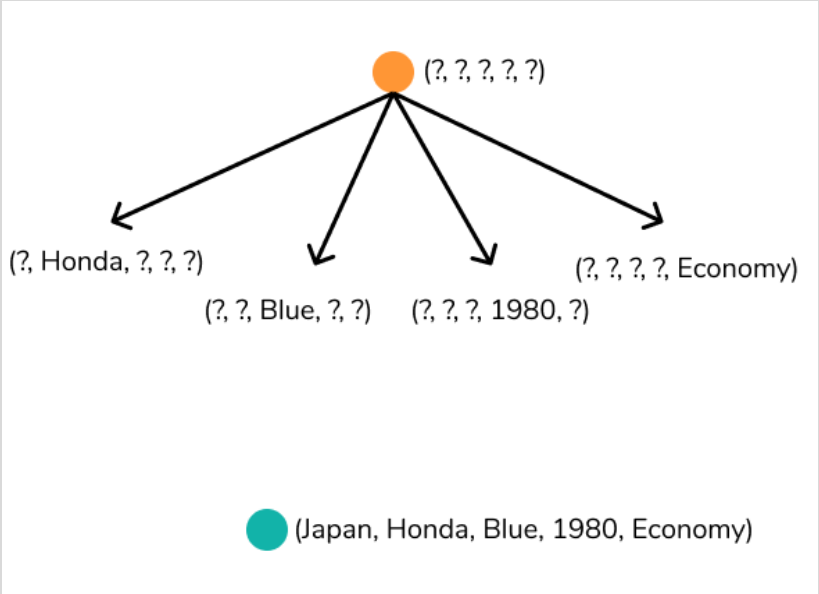
2. We now accept the second training example

Japan	Toyota	Green	1970	Sports	Negative

G, (?, ?, ?, ?, ?) classifies this training example as positive as it accepts any values for the attributes.

Hence, we specialize it so that it classifies the training example as negative.

(?, Honda, ?, ?, ?) will classify this training example as negative as Honda and Toyota don't match.
Similarly, (?, ?, Blue, ?, ?) will classify this training example as negative as Blue and Green don't match.



Likewise, for (?,?, ?, 1980, ?) and (?, ?, ?, ?, Economy)

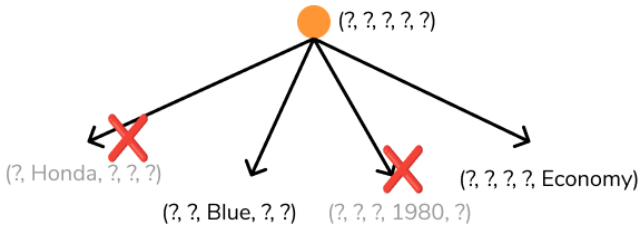
3. We now take the third training example. This is a positive example.

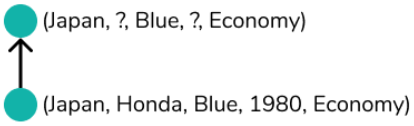
Japan	Toyota	Blue	1990	Economy	Positive

It is not consistent with the specialized set as value for **manufacturer** and **year** do not match,
S,
(Japan, Honda, Blue, 1980, Economy)

training eg,
(Japan, Toyota, Blue, 1990,Economy)

So we generalize the specific hypothesis, by replacing the unmatched values with '?', S_1 = {(Japan, ?, Blue, ?, Economy)}



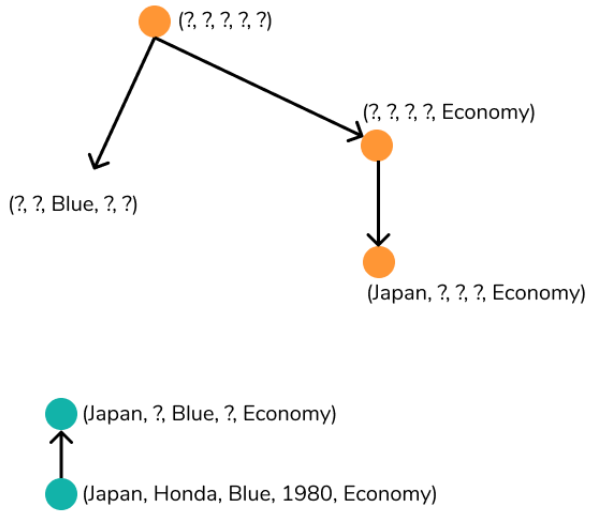


However, the generalized set is now inconsistent.
(?, Honda, ?, ?, ?) which requires the manufacturer to be Honda and the other attributes being anything, no longer holds as the model has now seen a positive example with manufacturer Toyota, and hence, is discarded.
In other words, the requirement that the manufacturer has to be Honda, and the other attribute values being anything should no longer classify an example as positive, as the model has learnt that the manufacturer can be Toyota too for a positive example.

Similarly, (?,?, ?, 1980, ?) is also discarded with the same argument as just made with the manufacturer.

4. Next, the fourth example which is classified as negative.

USA	**Chrysler**	**Red**	**1980**	**Economy**	**Negative**



We need to make sure that the general hypothesis classifies this example as negative.

(?, ?, Blue, ?, ?) from the general set classifies this example as negative as this training example as Color Red which does not match with Blue and so rightly classifies this example as negative.
However, (?, ?, ?, ?, Economy) from the general set, will classify the example as positive since this example's type is Economy, hence, we need to change this by specializing it.
If we specialize it by changing the country to be Japan, it would rightly be classified as a negative example as this example's country is the USA. So we now have (Japan,?, ?, ?, Economy)

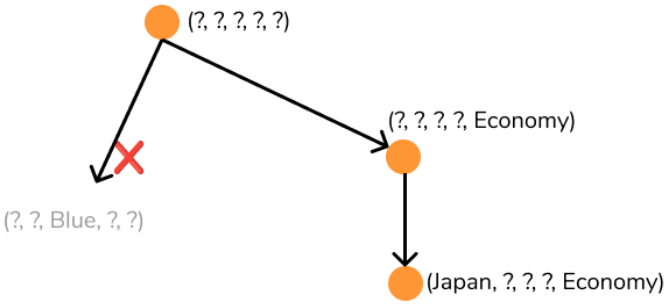
5. And finally the last example which is positive.

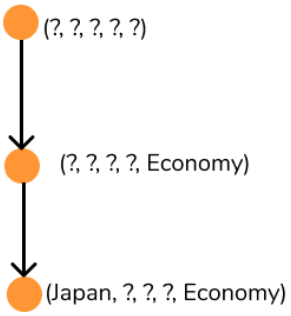
Japan	**Honda**	**White**	**1980**	**Economy**	**Positive**

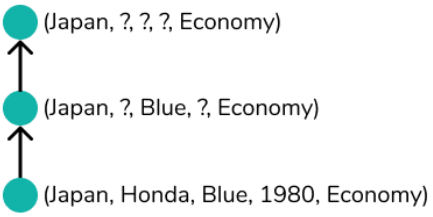
(Japan,?, Blue, ?, Economy) is what we have in the general set, and for this example we see that the color attribute is White, so we can generalize the color attribute. We get,

G = {(Japan,?, ?, ?, Economy)}

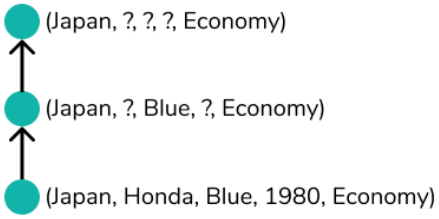
We have to also discard (?,?, Blue, ?, ?) from the generalized set as color Blue isn't a criteria to classify an example as positive anymore as our model has now seen a positive example where the color is White.







The generalized set as well as the specialized set have one element each that both match. There are no more training examples and hence, convergence is reached.



Issues

- Both positive and negative training examples should be given to train it properly, otherwise the algorithm will never converge to a single description.
- Due to breadth-first search, the need for large space requirements.
- Inconsistent data i.e. noise, like mislabeling positive examples as negative ones, will not lead to convergence.

Linear Regression

Linear Regression is a statistical model that attempts to show the relationship between two variables with the linear equation. It is used to identify the strength of the effect that the independent variables have on the dependent variable. It can also determine how much the dependent variable changes with the change in one or more independent variables.

The dependent variable for linear regression is always continuous, therefore linear regression is best used in a predictive model than a classification model.

Some of the real world examples where linear regression can be used are to forecast sales, perform risk analysis, measure profit probability etc

Univariate Linear Regression

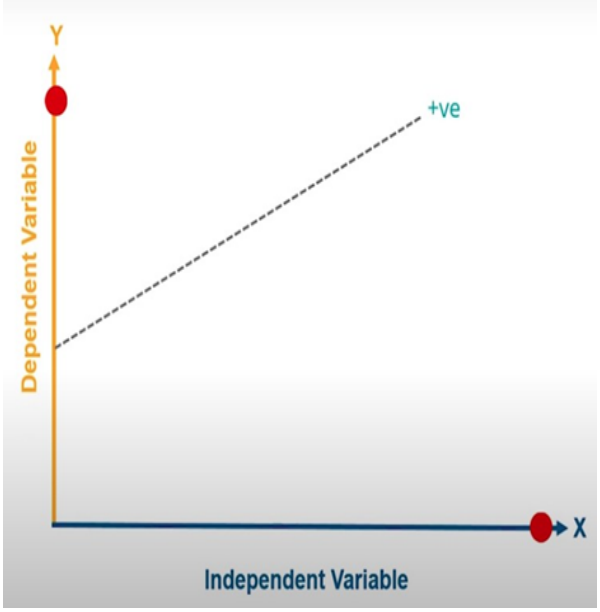
Univariate linear regression focuses on determining the relationship between one independent variable and one dependent variable.

$$Y = \alpha_0 + \alpha_1 x_1$$ equivalent to y = mx + c

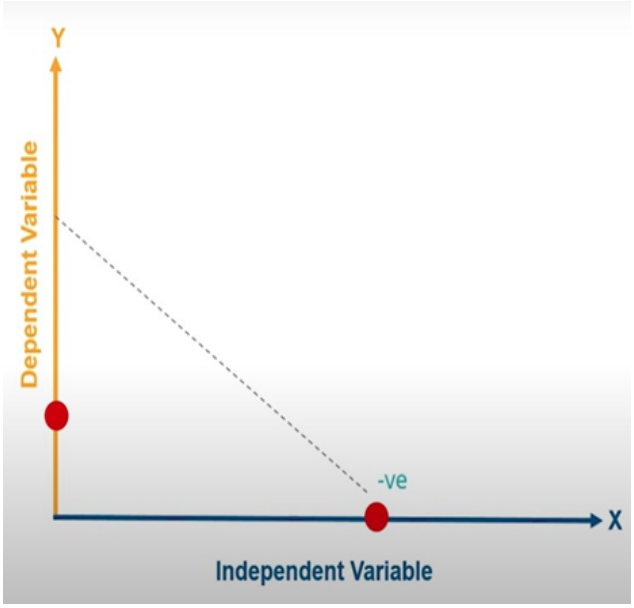
Here, x_1 = independent variable/feature

Y = dependent variable/output variable

α_0, α_1 = coefficients of regression



+ve regression, slope = +ve



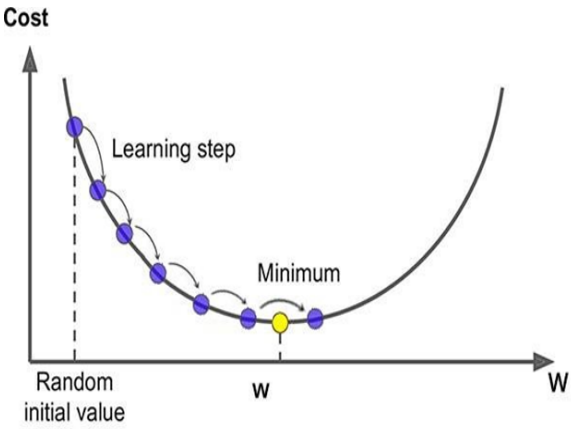
-ve regression, slope = -ve

Gradient Descent Algorithm in Linear Regression

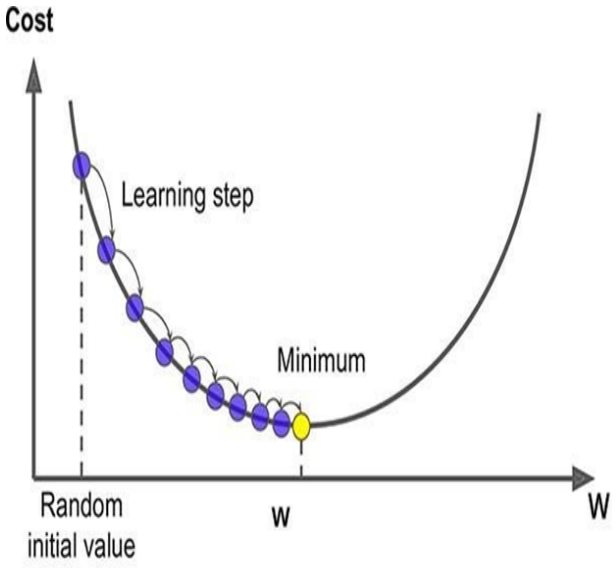
It is an algorithm used to minimize function by optimizing the parameters.

Steps:

- Start with some
- Keep changing to reduce until we hopefully end up at a minimum.



In simple words, consider this graph. Here we initially begin from a starting point at the very top, now we select a value for α (α = learning rate) and we keep descending down the graph in every iteration. If α is too large, gradient descent can overshoot the minimum. It may fail to converge, or even diverge



Whereas here in this graph, we start initially with a longer distance and as we descend, we tend to take smaller steps so that at some point in time we reach the minimum. If α is too small, gradient descent can be slow.

As previously we mentioned the hypothesis for linear regression to be:

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

Here,

$$\theta_0 := \theta_0 - \alpha \frac{d}{d\theta_0} J(\theta_0) \qquad \theta_1 := \theta_1 - \alpha \frac{d}{d\theta_1} J(\theta_1)$$

Where,

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

Multivariate Linear Regression

This is quite similar to the simple linear regression model, but with multiple independent variables contributing to the dependent variable and hence multiple coefficients to determine and complex computation due to the added variables.

$$Y_i = \alpha + \beta_1 x_i^{(1)} + \beta_2 x_i^{(2)} + \ldots + \beta_n x_i^{(n)}$$

Y_i is the estimate of i^{th} component of dependent variable y, where we have n independent variables and x_i^j denotes the i^{th} component of the j^{th} independent variable/feature. Similarly cost function is as follows,

$$E(\alpha, \beta_1, \beta_2, \ldots, \beta_n) = \frac{1}{2m} \sum_{i=1}^{m} (y_i - Y_i)$$

If we have n independent variables in our training data, our matrix X has n+1 rows, where the first row is the 0th term added to each vector of independent variables which has a value of 1 (this is the coefficient of the constant term α). So, X is as follows,

$$X = \begin{bmatrix} X_1 \\ .. \\ X_m \end{bmatrix}$$

X^i contains n entries corresponding to each feature in training data of i^{th} entry. So, matrix X has m rows and $n+1$ columns ($0^{th} column$ is all 1^s and rest for one independent variable each).

$$Y = \begin{bmatrix} Y_1 \\ Y_2 \\ .. \\ Y_m \end{bmatrix}$$

and coefficient matrix C,

$$C = \begin{bmatrix} \alpha \\ \beta_1 \\ .. \\ \beta_n \end{bmatrix}$$

and our final equation for our hypothesis is,

$$Y = XC$$

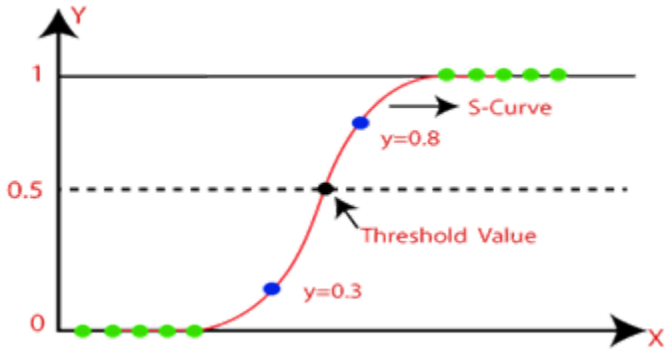
To calculate the coefficients, we need n+1 equations and we get them from the minimising condition of the error function. Equating partial derivative of E(α,β1,β2,...,βn) with each of the coefficients to 0 gives a system of n+1 equations. Solving these is a complicated step and gives the following nice result for matrix C,

$$C = (X^T X)^{-1} X^T y$$

where y is the matrix of the observed values of dependent variables.
This method seems to work well when the n value is considerably small (approximately for 3-digit values of n). As n grows big the above computation of matrix inverse and multiplication take large amount of time

Logistic Regression

Logistic regression is a basic algorithm to solve classification problems. The technique is somewhat similar to linear regression.



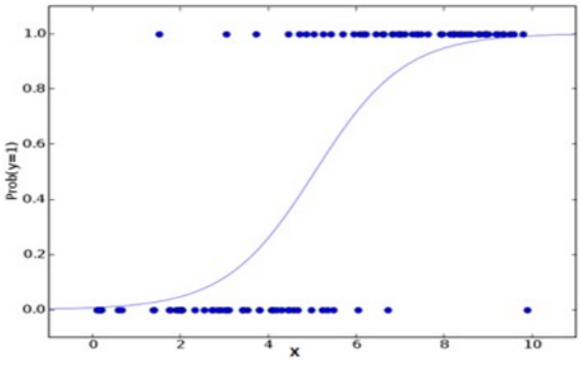
The problem is said to be a classification problem when independent variables are continuous in nature and dependent variables are in categorical form.

Some real life examples could be, to categorize whether mail is spam or not, transaction is fraudulent or not. All answers to categorical problems are yes or no.

Logistic regression uses sigmoid function to deal with outliers. It takes any real value between 0 and 1.

Defined as :

$$\sigma(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-t}}$$



Logistic regression uses a threshold value concept that defines probability of either 0 or 1. Value above threshold value tends to 1 and below threshold value tends to 0.

Logistic regression cost function:

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

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

$$\text{Note: } y = 0 \text{ or } 1 \text{ always}$$

cost=0 if y=1, $h_0(x)$=1
But as $h_0(x)$ -> 0 cost -> infinity

Gradient descent:
Repeat {

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

} simultaneously update all theta j.

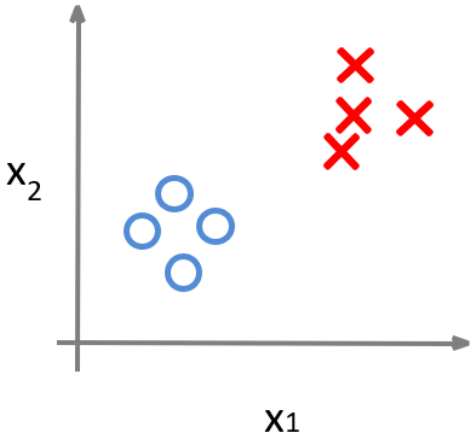
Multi-class classification: One-vs-all

Email foldering/tagging: Work, Friends, Family, Hobby

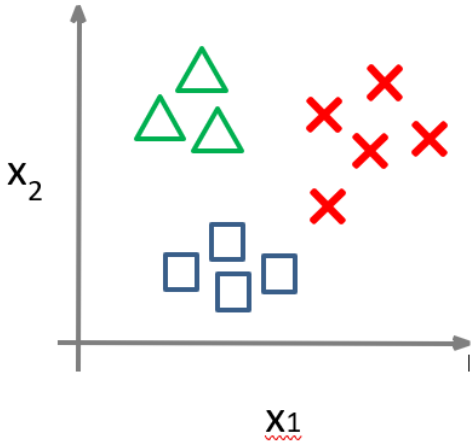
Medical diagrams: Not ill, Cold, Flu

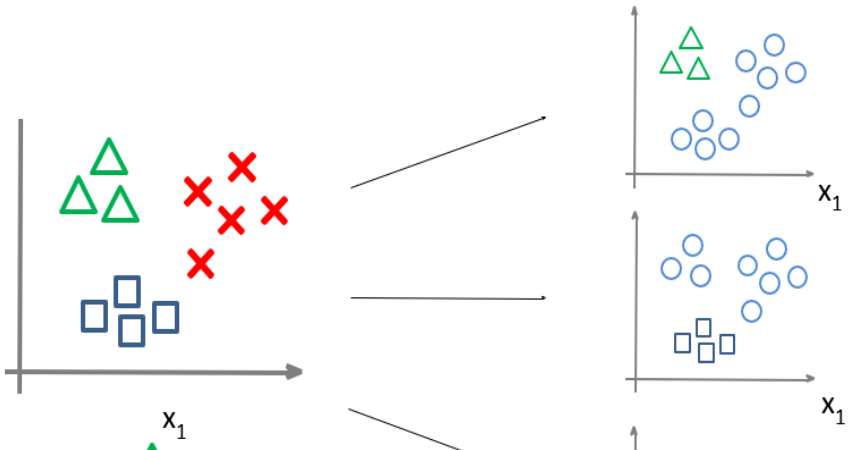
Weather: Sunny, Cloudy, Rain, Snow

Binary classification:



Multi-class classification:





Class 1: △

Class 2: □

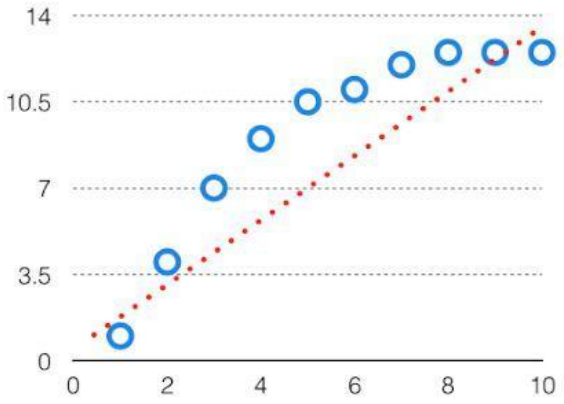
Class 3: ×

$h_\theta^{(i)}(x) = P(y = i|x; \theta)$

Bias v/s Variance

Understanding the fundamental concepts of *bias* and *variance* helps you decide the right model to apply for your data. There is a tradeoff between a model's ability to minimise bias and variance. Gaining a proper understanding of these errors would help us not only to build accurate models but also to avoid the mistake of *overfitting* and *underfitting*.

We split the data into two sets namely, the *training set* which usually constitutes around 75% of the dataset and the remaining 25% makes up the test set. Consider the example of predicting the price of the house based on its size(area). Basically, we try to find a function *f* that maps x (set of different sizes) to y (set of different prices) which is represented by the surface (line or curve) that fits in the data well. Have a look at the example below.



As can be seen in the graph above, the line does not fit all the data points properly. In other words, there is some error which can be calculated as the sum of squares of the distances between the actual data points and the predicted values, also known as least square method, which we need to minimise to get the correct results. The formula for calculating the error using least square method is as follows.

$$S = \sum_{i=1}^{n} {d_i}^2$$

$$S = \sum_{i=1}^{n} [y_i - f_{x_i}]^2$$

$$S = {d_1}^2 + {d_2}^2 + {d_3}^2 + \cdots + {d_n}^2$$

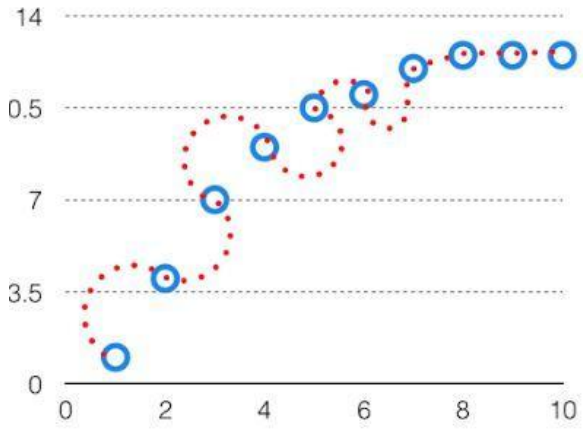
Where d_i is the distance between the i^{th} data point and the predicted value for x_i.

Bias

The inability of the model to capture the true relationship between the independent variable (x) and the dependent variable (y) is called ***bias***. It is the difference between the average prediction of our model and the correct value which we are trying to predict. Model with high bias pays very little attention to the training data and generalises the model. It always leads to high error on training data but it performs reasonably well on test data. It results in a very simplistic model that does not consider the variations very well. Since it does not learn the training data very well it is said to have **high bias** and is known as **Underfitting**. The graph above is an example of underfitting.

Variance

Model with **high variance** pays a lot of attention to training data and does not generalise on the data which it hasn't seen before. As a result, such models perform very well on training data but have high error rates on test data.



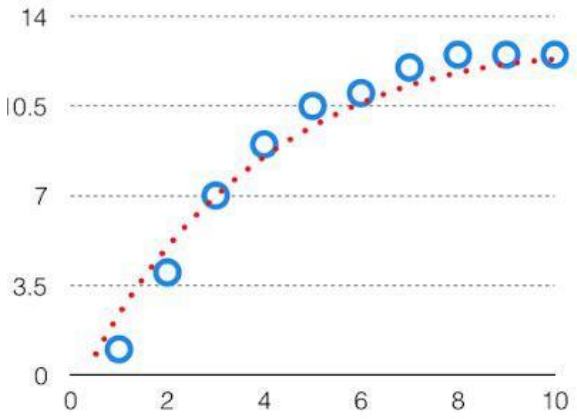
In the above graph, the model fits the data accurately. When a model has high variance it is known as **overfitting**. Overfitting is fitting the training set accurately via complex curves and high order functions but is not the solution as the error with unseen data is high. For some test data the accuracy could be very good while for some other test data the accuracy may be very poor. While training a data model, variance should be kept low as to what works accurately for training data cannot generalise for test data. Hence it leads to *very poor generalisation*.

What is the Bias Variance Tradeoff?

If our model is too simple and has very few parameters then it may have high bias and low variance. On the other hand if our model has a large number of parameters then it's going to have high variance and low bias. So we need to find the right or good balance without overfitting and underfitting the data.

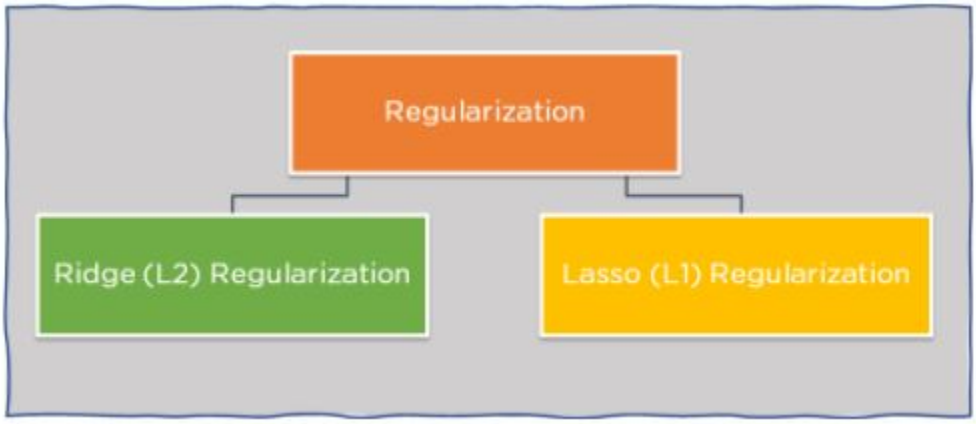
This tradeoff in complexity is why there is a tradeoff between bias and variance. An algorithm can't be more complex and less complex at the same time.

The model that perfectly fits the data would be as follows.



Regularization

There are two main types of regularization techniques: Ridge Regularization and Lasso Regularization.



Ridge Regularization

Also known as Ridge Regression, it modifies the over-fitted or under-fitted models by adding the penalty equivalent to the sum of the squares of the magnitude of coefficients.

This means that the mathematical function representing our machine learning model is minimized and coefficients are calculated. The magnitude of coefficients is squared and added. Ridge Regression performs regularization by shrinking the coefficients present. The function depicted below shows the cost function of ridge regression :

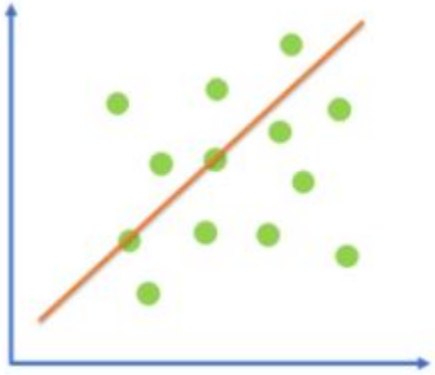
Cost function = Loss + λ x $\sum \|w\|^2$

Here,

Loss = Sum of the squared residuals

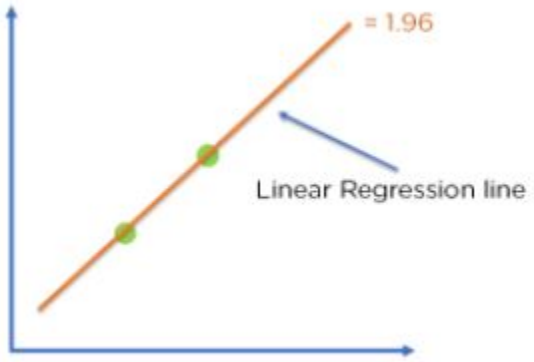
λ = Penalty for the errors

W = slope of the curve/ line



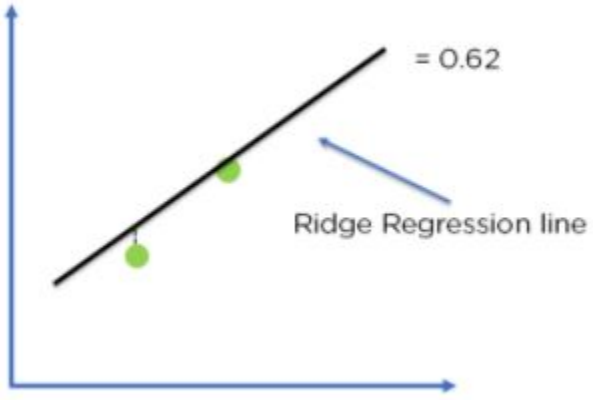
In the cost function, the penalty term is represented by Lambda λ. By changing the values of the penalty function, we are controlling the penalty term. The higher the penalty, it reduces the magnitude of coefficients. It shrinks the parameters. Therefore, it is used to prevent multicollinearity, and it reduces the model complexity by coefficient shrinkage.

Consider the graph illustrated below which represents Linear regression :

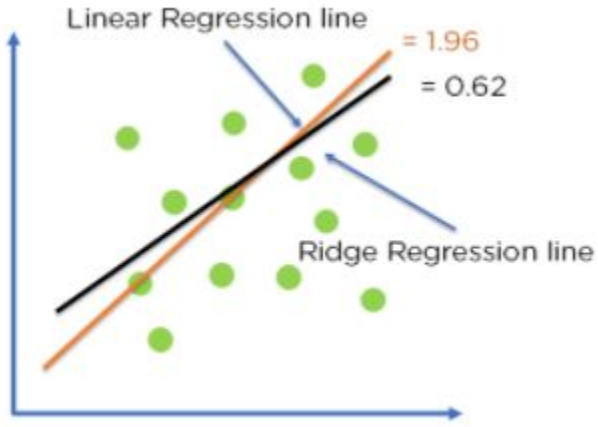


Cost function = Loss + λ xΣ‖w‖^2
For Linear Regression line, let's consider two points that are on the line,
Loss = 0 (considering the two points on the line)
λ= 1
w = 1.4
Then, Cost function = 0 + 1 x 1.42 = 1.96

For Ridge Regression, let's assume,
Loss = 0.32 + 0.22 = 0.13
λ = 1
w = 0.7
Then, Cost function = 0.13 + 1 x 0.72 = 0.62



Comparing the two models, with all data points, we can see that the Ridge regression line fits the model more accurately than the linear regression line.



Lasso Regularization

It modifies the over-fitted or under-fitted models by adding the penalty equivalent to the sum of the absolute values of coefficients.

Lasso regression also performs coefficient minimization, but instead of squaring the magnitudes of the coefficients, it takes the true values of coefficients. This means that the coefficient sum can also be 0, because of the presence of negative coefficients. Consider the cost function for Lasso regression :

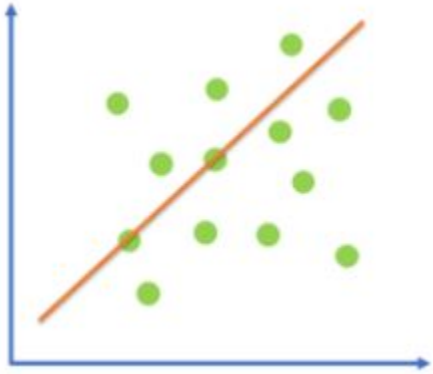
Cost function = Loss + λ x $\sum\|w\|$

Here,

Loss = Sum of the squared residuals

λ = Penalty for the errors

w = slope of the curve/ line



We can control the coefficient values by controlling the penalty terms, just like we did in Ridge Regression. Again consider a Linear Regression model :

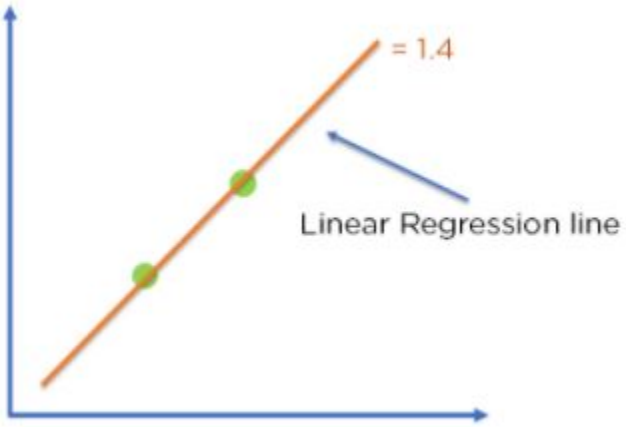


Figure 12: Linear Regression Model

Cost function = Loss + λ x Σ‖w‖

For Linear Regression line, let's assume,
Loss = 0 (considering the two points on the line)
λ = 1
w = 1.4
Then, Cost function = 0 + 1 x 1.4 = 1.4

For Ridge Regression, let's assume,
Loss = 0.32 + 0.12 = 0.1
λ = 1
w = 0.7
Then, Cost function = 0.1 + 1 x 0.7 = 0.8

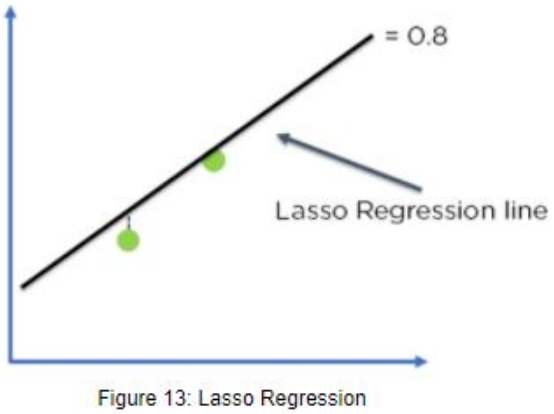


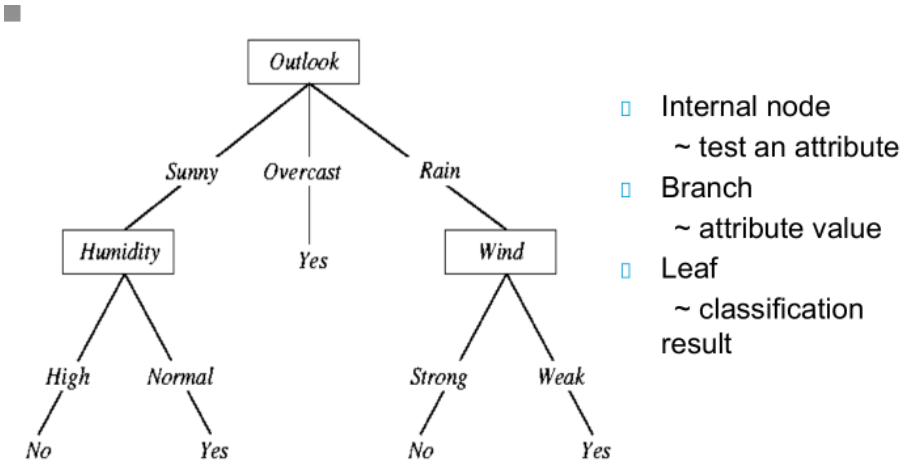
Figure 13: Lasso Regression

Comparing the two models, with all data points, we can see that the Lasso regression line fits the model more accurately than the linear regression line.

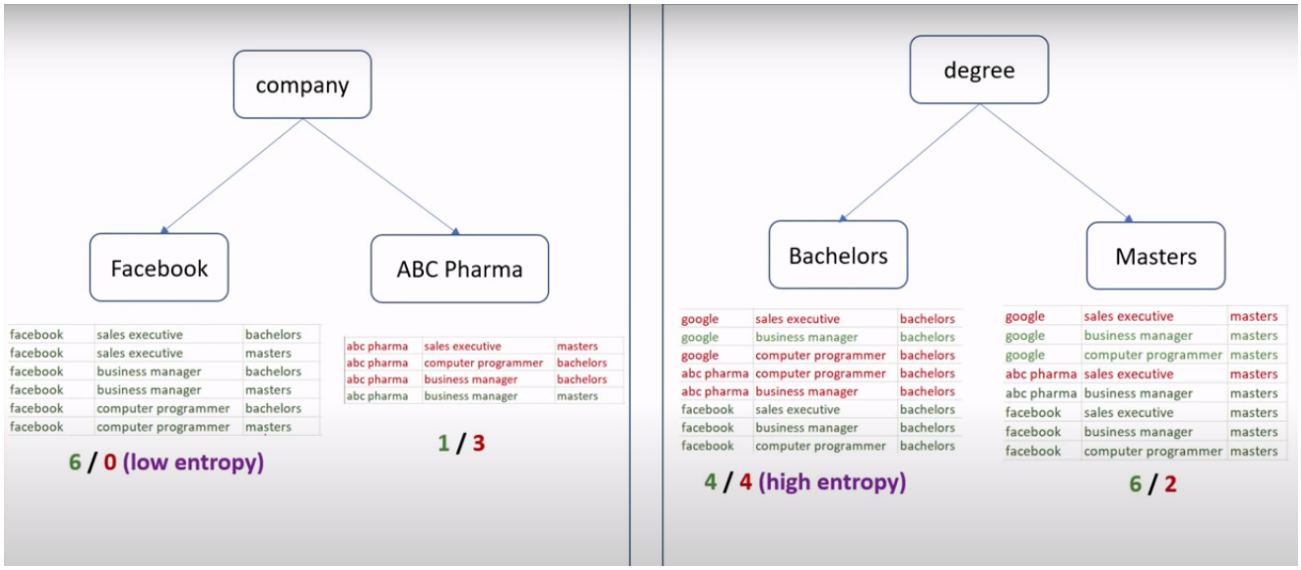
Decision Tree and Random Forest

A **decision tree** is a classification and prediction tool having a tree-like structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node (terminal node) holds a class label.

Outlook	Temperature	Humidity	Wind	Played football(yes/no)
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Overcast	Hot	High	Weak	Yes
Rain	Mild	High	Weak	Yes
Rain	Cool	Normal	Weak	Yes
Rain	Cool	Normal	Strong	No
Overcast	Cool	Normal	Strong	Yes
Sunny	Mild	High	Weak	No
Sunny	Cool	Normal	Weak	Yes
Rain	Mild	Normal	Weak	Yes
Sunny	Mild	Normal	Strong	Yes
Overcast	Mild	High	Strong	Yes
Overcast	Hot	Normal	Weak	Yes
Rain	Mild	High	Strong	No



Now, how do we decide which is the best attribute for splitting up the data? Consider the decision tree given below:



Here on the left hand side we split data using company and get a pure subset for facebook hence 6/0 (low entropy) and for ABC pharma the entropy is comparatively low ⅓. Whereas on the right hand side, we have high entropy for both Bachelors and Masters when we split the data by degree.
From this we can conclude that on the left hand side we have high information gain and on the right side we have low information gain. Hence you need to use an approach which gives you high information gain at every split.

Entropy: In machine learning, entropy is a measure of the randomness in the information being processed. The higher the entropy, the harder it is to draw any conclusions from that information. The Entropy measures the disorder of a set S containing a total of n examples of which n+ are positive and n- are negative and it is given by:

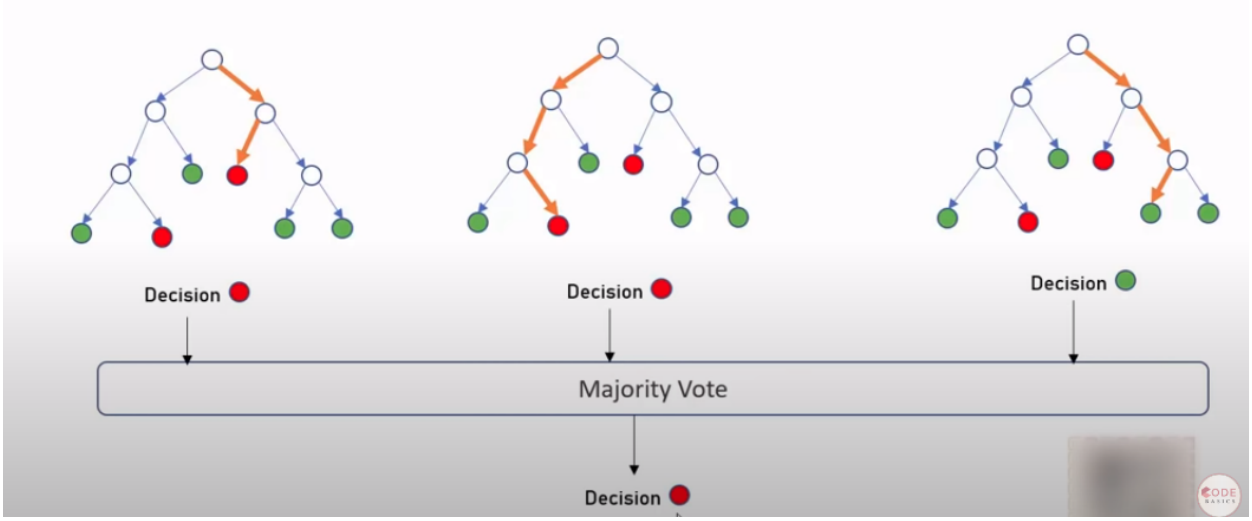
$$D(n_+, n_-) = -\frac{n_+}{n}\log_2\frac{n_+}{n} - \frac{n_-}{n}\log_2\frac{n_-}{n} = Entropy(S)$$

Information Gain: The information gain measures the expected reduction in entropy due to splitting on an attribute A.

$$Gain(S, A) = Entropy(S) - \underbrace{\sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)}$$

Random forest, like its name implies, consists of a large number of individual decision trees that operate as an **ensemble**(ensemble learning is a process of *combining multiple classifiers to solve a complex problem and to improve the performance of the model*).

Each individual tree in the random forest spits out a class prediction and the class with the most votes becomes our model's prediction.

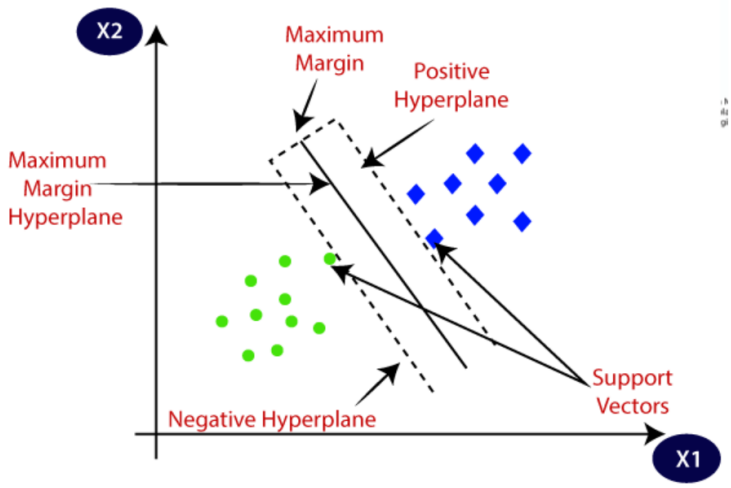


Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset. The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.

Support Vector Machine

"Support Vector Machine" (SVM) is a supervised machine learning algorithm that can be used for both classification or regression challenges.
Support vectors are the data points that lie closest to the decision surface (or hyperplane) .They are the data points most difficult to classify.

- Linear SVM: Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.



Optimal Margin classifier

$$\begin{aligned} \min_{\gamma,w,b} \quad & \frac{1}{2}||w||^2 \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \ldots, m \end{aligned}$$

We can write the constraints as

$$g_i(w) = -y^{(i)}(w^T x^{(i)} + b) + 1 \leq 0.$$

When we construct the Lagrangian for our optimization problem we have:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{m} \alpha_i \left[y^{(i)}(w^T x^{(i)} + b) - 1 \right].$$

setting the derivatives of L with respect to w and b to zero. We have:

$$\nabla_w \mathcal{L}(w, b, \alpha) = w - \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} = 0$$

This implies that

$$w = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)}.$$

As for the derivative with respect to b, we obtain

$$\frac{\partial}{\partial b} \mathcal{L}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i y^{(i)} = 0.$$

we obtain the following dual optimization problem

$$\begin{aligned} \max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle. \\ \text{s.t.} \quad & \alpha_i \geq 0, \quad i = 1, \ldots, m \\ & \sum_{i=1}^{m} \alpha_i y^{(i)} = 0, \end{aligned}$$

If αi =0 corresponding vector x is not a support vector.
If αi is very high x has greater influence on the hyperplane.

Hard Margin vs. Soft Margin

The difference between a hard margin and a soft margin in SVMs lies in the separability of the data. If our data is linearly separable, we go for a hard margin. However, if this is not the case, it won't be feasible to do that. In the presence of the data points that make it impossible to find a linear classifier, we would have to be more lenient and let some of the data points be misclassified. In this case, a soft margin SVM is appropriate.

Sometimes, the data is linearly separable, but the margin is so small that the model becomes prone to overfitting or being too sensitive to outliers. Also, in this case, we can opt for a larger margin by using soft margin SVM in order to help the model generalize better.

Non Linear SVM

- ➢ When we cannot classify data points by line we go for higher dimensions.
- ➢ Mapping function is used to map lower dimension to higher dimension.

Kernel trick:

Example:

X=(x1,x2,x3) and Y=(y1,y2,y3)

F(x)=(x1.x1,x1.x2,x1.x3, x2.x1,x2.x2,x2.x3, x3.x1,x3.x2,x3.x3)

X=(1,2,3) Y=(4,5,6)

F(x)=(1,2,3,2,4,6,3,6,9)

F(y)=(16,20,24,20,25,30,24,30,36)

(f(x),f(y))=16+40+72+40+100+180+72+180+329=1024

This is computationally expensive. To reduce the cost in finding dot products for all the points we use the kernel trick.

So above example using kernel trick can be solved as :

$K(<x,y>)=(<x,y>)^2$

$=((1).(4)+(2).(5)+(3).(6))^2$

$=(4+10+18)^2= 32^2 =1024$

Different kinds of kernel functions can be used. Few examples are :

- Polynomial kernel function:

  $=(a*b+r)^d$

  Where a and b are input/observation , r is coefficient of polynomial and d is the degree of polynomial.

- RBF (Radial basis kernel function):
  =e-r(a-b)^2

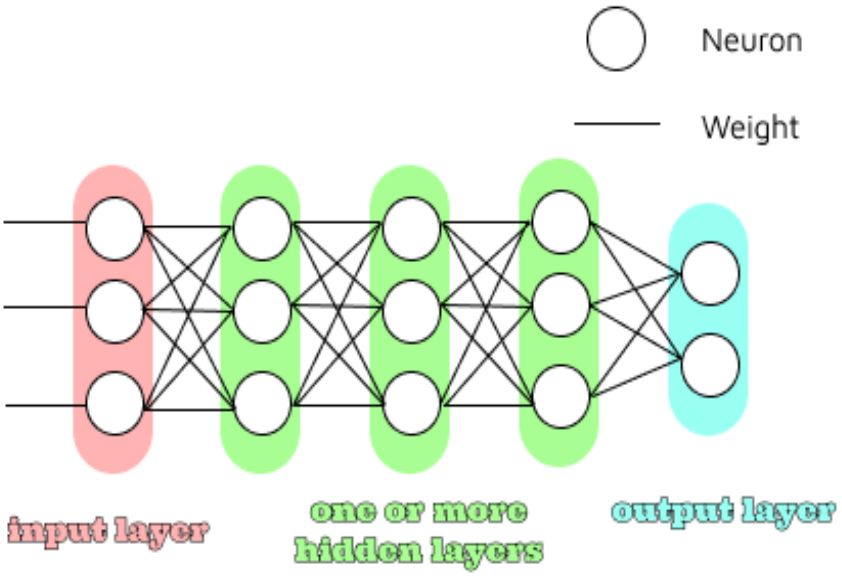
  Calculates high dimension relationships Widely used.

Neural Networks

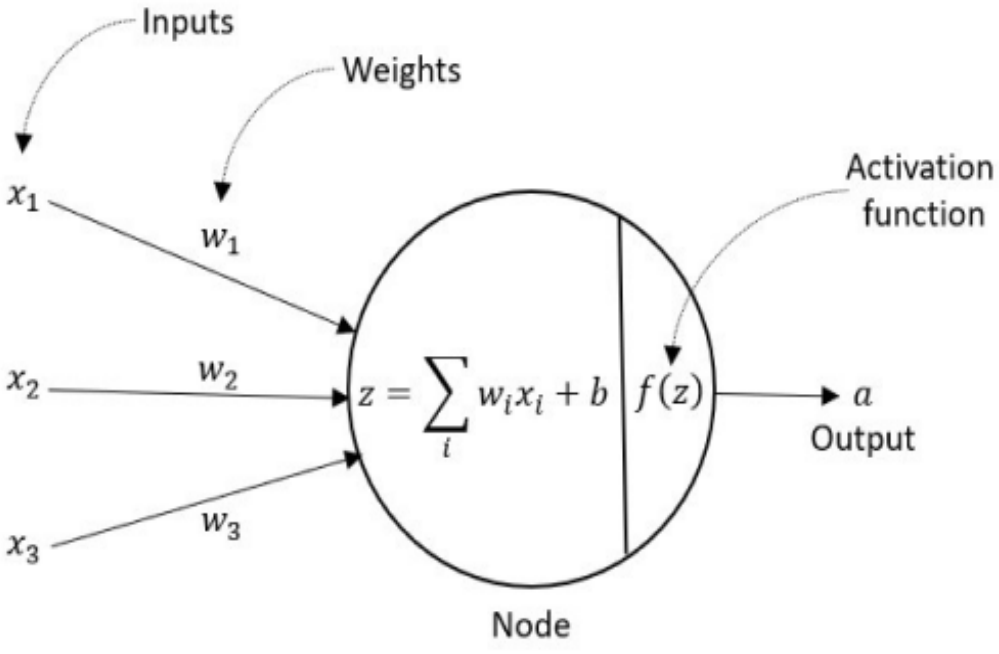
What? Model of computation inspired by the structure of neural networks in the brain

Feed Forward Neural Network

Structure

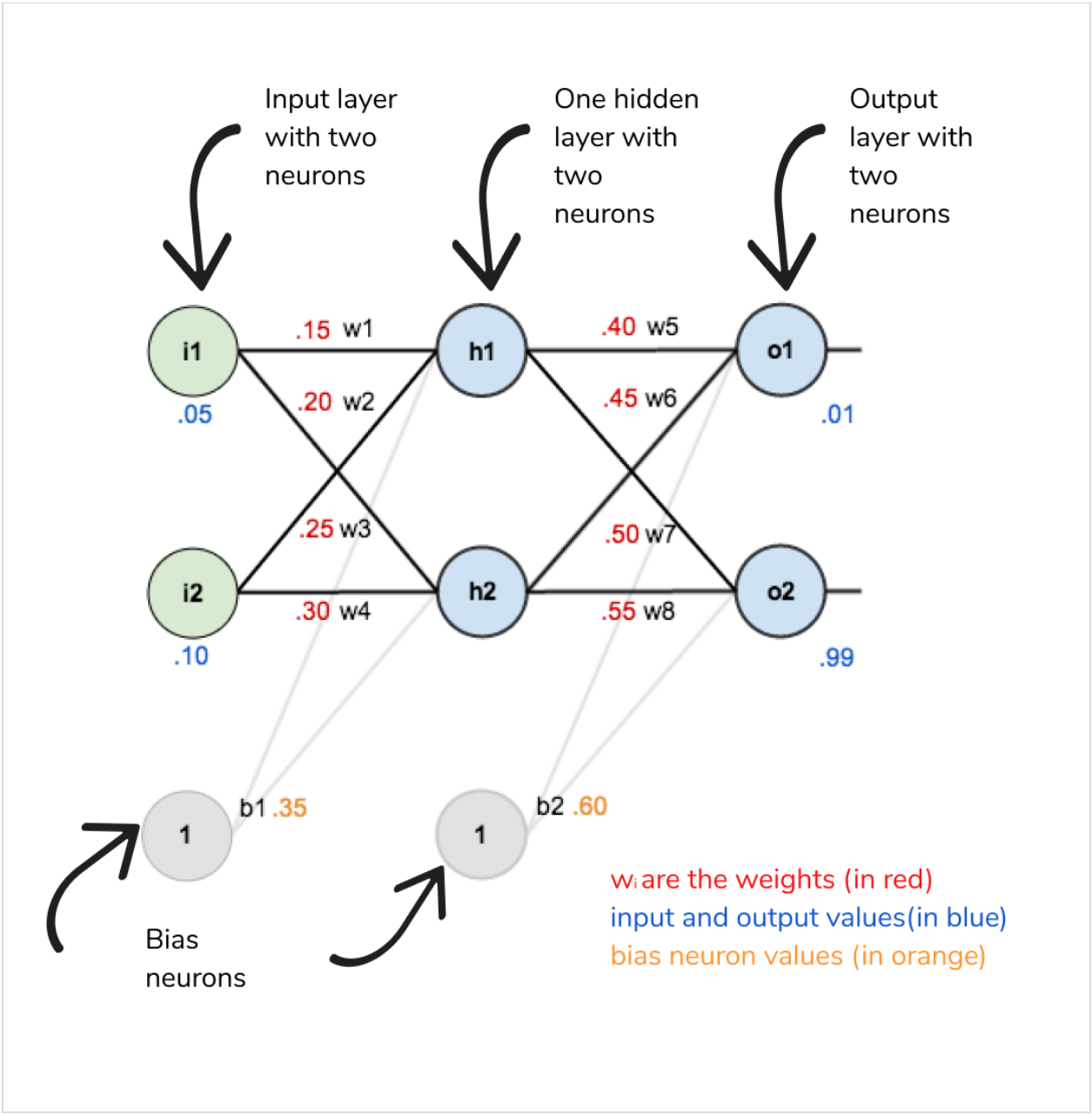


Node Computution



Training a Neural Network

Example



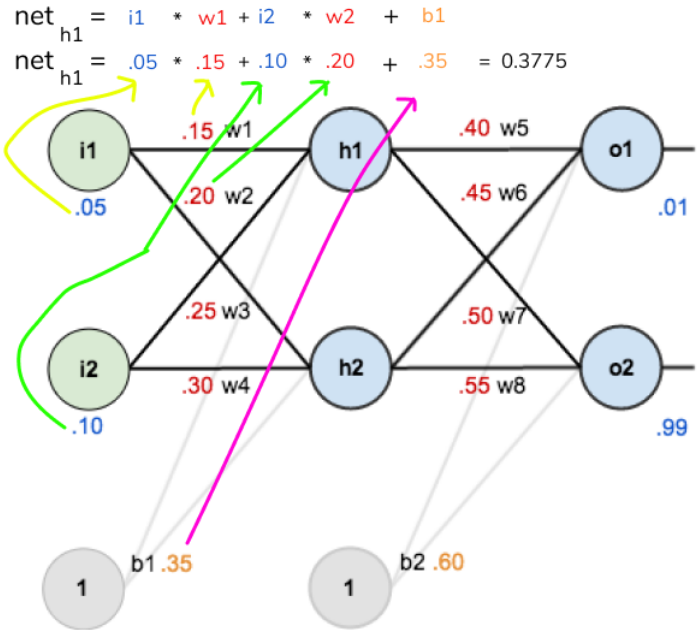
Forward Pass

We figure out the **total net input** to each hidden layer neuron, squash the total net input using an activation function (here we use the logistic function), then repeat the process with the output layer neurons.

Calculating net $_{h1}$

$$net_{h1} = (i1 * w1) + (i2 * w2) + b1$$

$$net_{h1} = (.05 * .15) + (.10 * .20) + .35 = 0.3775$$



We then squash it using the logistic function to get the output of h1, out $_{h1}$

Calculating out $_{h1}$

$$out_{h1} = f(net_{h1}) = \frac{1}{1+ e^{-net_{h1}}} = \frac{1}{1+ e^{-0.3775}} = 0.5932$$

Similarly, finding net $_{h2}$ and out $_{h2}$, we have

$$net_{h2} = (i1 * w3) + (i2 * w4) + b1$$

$$net_{h2} = (.05 * .25) + (.10 * .30) + .35 = 0.3925$$

$$out_{h2} = f(net_{h2}) = \frac{1}{1+ e^{-net_{h2}}} = \frac{1}{1+ e^{-0.3925}} = 0.5968$$

We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.

At o1,

$$net_{o1} = (out_{h1} * w5) + (out_{h2} * w6) + b2$$

$$net_{o1} = (.5932 * .40) + (.5968 * .45) + .60 = 1.1059$$

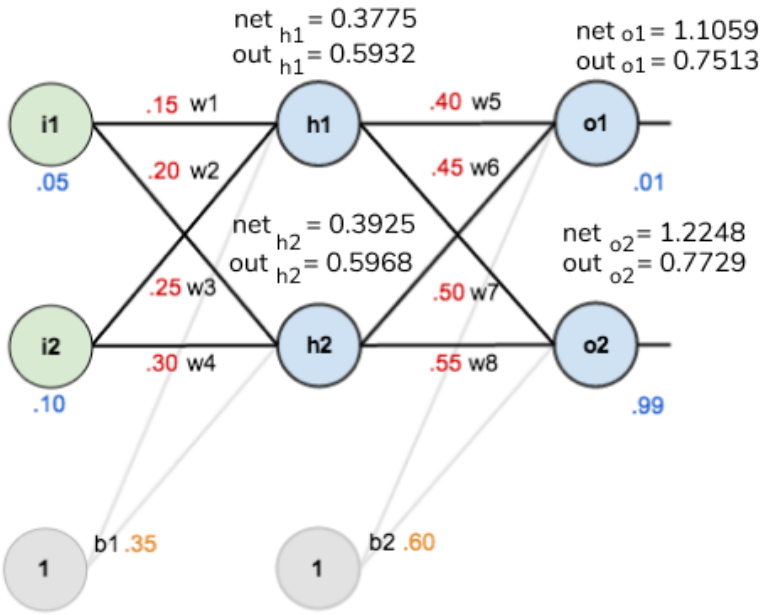
$$out_{o1} = f(net_{o1}) = \frac{1}{1+ e^{-net_{o1}}} = \frac{1}{1+ e^{-1.1059}} = 0.7513$$

At o2,

$$net_{o2} = (out_{h1} * w7) + (out_{h2} * w8) + b2$$

$$net_{o2} = (.5932 * .50) + (.5968 * .55) + .60 = 1.2248$$

$$out_{o2} = f(net_{o2}) = \frac{1}{1+ e^{-net_{o2}}} = \frac{1}{1+ e^{-1.2248}} = 0.7729$$



Calculating the total error

We now

- **calculate the error** for each output neuron using the squared error function
- sum them to get the total error

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

$$E_{total} = E_{o1} + E_{o2}$$

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$E_{total} = \frac{1}{2}(.01 - .7513)^2 + \frac{1}{2}(.99 - .7729)^2$$

$$E_{total} = 0.2748 + 0.0235 = 0.2983$$

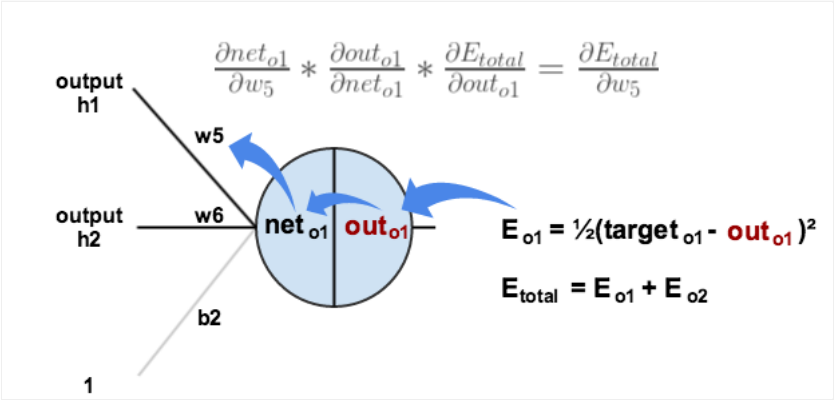
The Backward Pass

Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer to the target output, thereby minimizing the error for each output neuron and the network as a whole.

Output Layer

We update the weights that connect the hidden layer neurons to the output layer neurons.
We wish to find out how much a change in the weight affects the total error.

Consider w5, by applying chain rule,



$$\frac{\delta E_{total}}{\delta w5} = \frac{\delta E_{total}}{\delta out_{o1}} * \frac{\delta out_{o1}}{\delta net_{o1}} * \frac{\delta net_{o1}}{\delta w5}$$

Let's break down each piece in this equation,

$$\frac{\delta E_{total}}{\delta out_{o1}} = \frac{\delta[\frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2]}{\delta out_{o1}}$$

$$\frac{\delta E_{total}}{\delta out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1$$

$$\frac{\delta E_{total}}{\delta out_{o1}} = -(target_{o1} - out_{o1}) = -(.01 - .7513) = 0.7413$$

$$\frac{\delta out_{o1}}{\delta net_{o1}} = \frac{\delta(\frac{1}{1+e^{-net_{o1}}})}{\delta net_{o1}} = out_{o1}(1 - out_{o1})$$

$$\frac{\delta out_{o1}}{\delta net_{o1}} = .7513(1 - .7513) = 0.1868$$

$$\frac{\delta net_{o1}}{\delta w5} = \frac{\delta((out_{h1} * w5) + (out_{h2} * w6) + b2)}{\delta w5} = out_{h1} = 0.5932$$

$$\frac{\delta E_{total}}{\delta w5} = 0.7413 * 0.1868 * 0.5932 = 0.0821$$

We then subtract this value from the current weight, w5 (learning rate, $\eta = 0.5$)

$$w5^{+} = w5 - \eta * \frac{\delta E_{total}}{\delta w5} = .40 - .50 * .0821 = 0.3589$$

$$w6^{+} = 0.4086$$

$$w7^{+} = 0.5113$$

$$w8^{+} = 0.5613$$

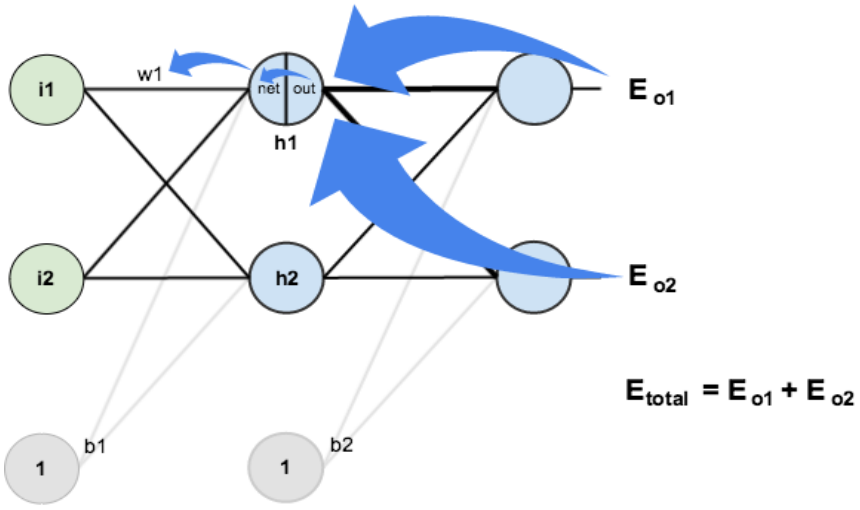
Hidden Layer

Next, we update the weights that connect the input layer neurons to the hidden layer neurons.

We're going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons.

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



Consider w1, to calculate change in total error wrt w1,

$$\frac{\delta E_{total}}{\delta w1} = \frac{\delta E_{total}}{\delta\ out_{h1}} * \frac{\delta\ out_{h1}}{\delta\ net_{h1}} * \frac{\delta\ net_{h1}}{\delta w1}$$

Let's break down each piece in this equation,

$$\frac{\delta E_{total}}{\delta\ out_{h1}} = \frac{\delta E_{o1}}{\delta\ out_{h1}} + \frac{\delta E_{o2}}{\delta\ out_{h1}}$$

Further breaking apart the first term, $\frac{\delta E_{o1}}{\delta\ out_{h1}}$

$$\frac{\delta E_{o1}}{\delta\ out_{h1}} = \frac{\delta E_{o1}}{\delta\ net_{o1}} * \frac{\delta net_{o1}}{\delta\ out_{h1}}$$

$$\frac{\delta E_{o1}}{\delta\ out_{h1}} = \left(\frac{\delta E_{o1}}{\delta\ out_{o1}} * \frac{\delta\ out_{o1}}{\delta\ net_{o1}}\right) * w5$$

$$\frac{\delta E_{o1}}{\delta\ out_{h1}} = (.7413 * .1868) * .40 = 0.0553$$

Similarly, for $\frac{\delta E_{o2}}{\delta\ out_{h1}}$, we get -0.0190

$$\frac{\delta E_{total}}{\delta\ out_{h1}} = .553 + (-.0190) = 0.0363$$

$$\frac{\delta\ out_{h1}}{\delta\ net_{h1}} = out_{h1}(1 - out_{h1}) = .5932(1 - .5932) = 0.2413$$

$$\frac{\delta\ net_{h1}}{\delta w1} = \frac{\delta (i1 * w1) + (i2 * w2) + b1}{\delta w1} = i1 = 0.05$$

Plugging in the values,

$$\frac{\delta E_{total}}{\delta w1} = .0363 * .2413 * .05 = 0.0004$$

We then subtract this value from the current weight, w1 (learning rate, $\eta = 0.5$)

$$w1^{+} = w1 - \eta * \frac{\delta E_{total}}{\delta w1} = .15 - .5 * .0004 = 0.1497$$

After computing for $w2$, $w3$ and $w4$

$$w2^{+} = 0.1995$$

$$w3^{+} = 0.2497$$

$$w4^{+} = 0.2995$$

Hyperparameter Tuning

Machine Learning is all about **curve fitting**, the curve could be **linear** or **non-linea**r depending on the relationship between the input data and the output data. So we try to approximate a function. For eg. if it is a simple line then the parameters are the slope and the intercept. Interest in ML comes from data, CPU power and algorithms.

Some hyperparameters are:

1. Number of hidden layers (More the number of hidden layers, more the feature extraction)
2. Number of neurons in the hidden layer
3. Number of epochs
4. Activation function
5. Optimizer
6. Size of the batch
7. Loss function

These hyperparameters are used for model evaluation

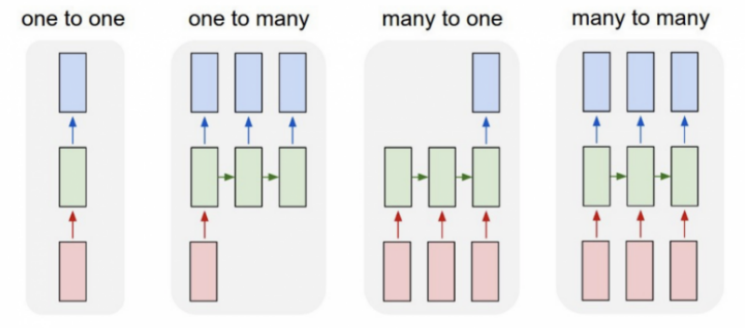
Recurrent Neural Network

Recurrent neural networks (RNN) are a class of neural networks that are helpful in modeling sequence data. Derived from feedforward networks, RNNs exhibit similar behavior to how human brains function.

While feed-forward neural networks map one input to one output, RNNs can map one to many (video captioning), many to many (translation) and many to one (classifying a voice).

Types of RNN:

- One to One
- One to Many
- Many to One
- Many to Many



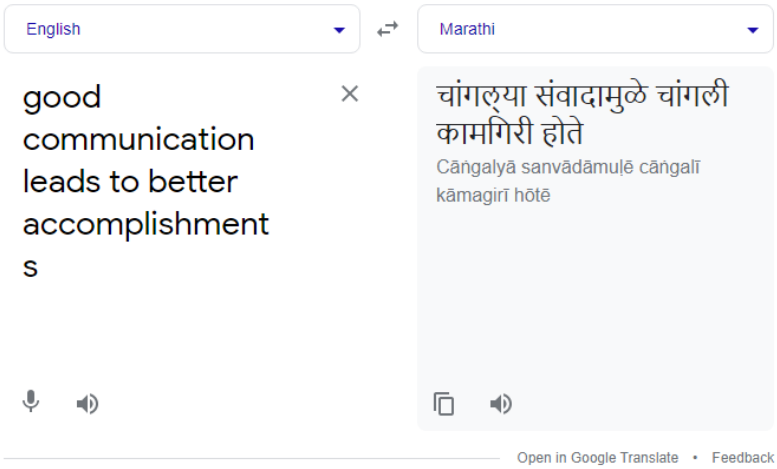
Applications of RNN

Before we begin learning how RNN works in detail, let's understand what are the different tasks that we can do using RNN. A few applications of RNN are listed below:

➔ **Autocomplete**: Used in Google Search Engine, Gmail, Google docs and so on.

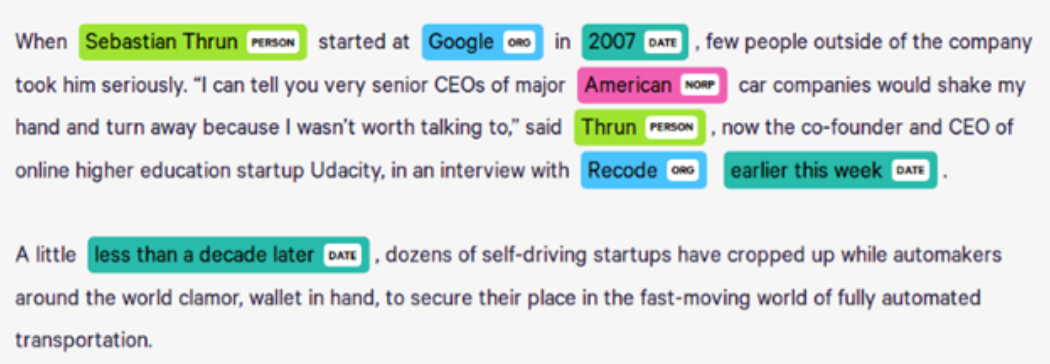
➔ **Language Translation**: A popular example of this is *Google Translate.*

- Language translation involves translation of a sentence from one language to another.

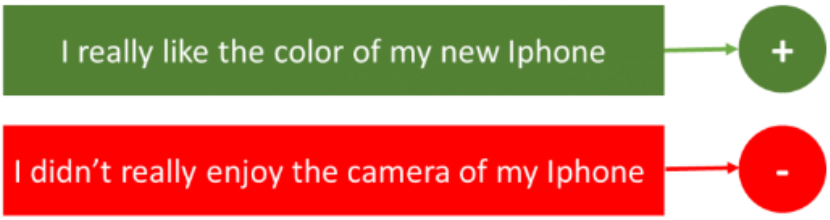


➔ **Named Entity Recognition**:

- Identifies and classifies input text into various categories such as person names, organizations, locations, medical codes, time expressions, quantities, monetary values, percentages, etc.



➔ **Sentiment Analysis**: Examples of this include classifying tweets and reviews given for a product on an ecommerce website.
- ◆ This can be a task of simply classifying a given sentence into positive and negative sentiment.

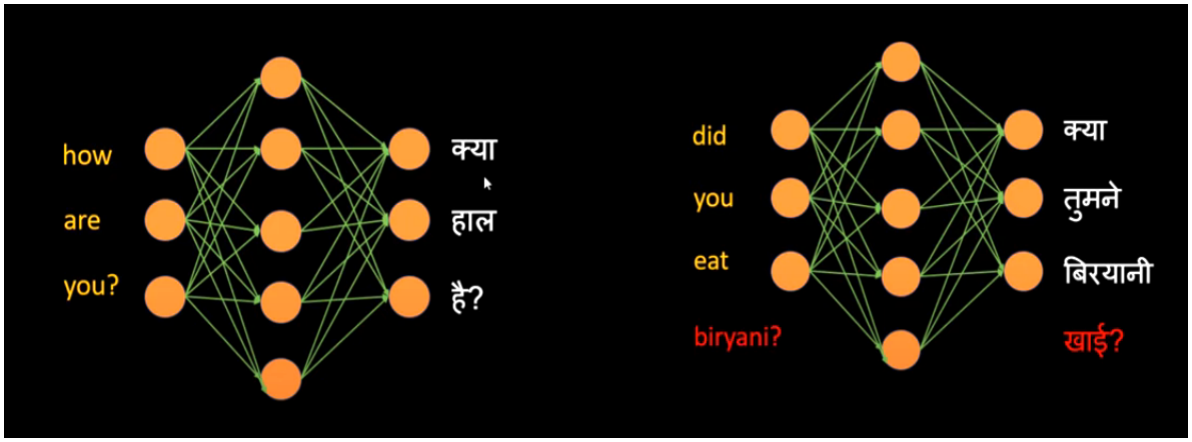


Why do we need to use RNN?

Now that we have looked at the various applications of the RNN model, let's understand some of the reasons for using the RNN model over the Multi-layer Perceptron (MLP) and other standard neural networks.

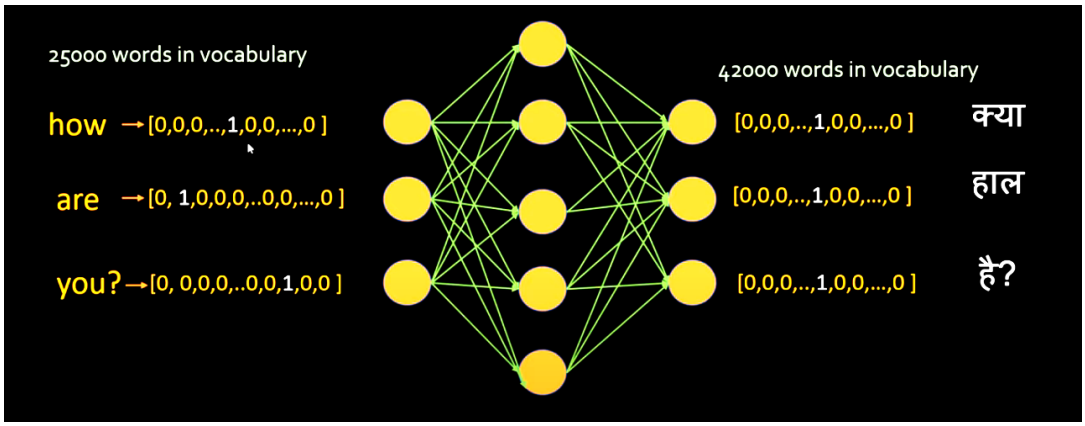
Some of the reasons with respect to *language translation* are as follows:

➔ **No fixed size of neurons in the layer:**
- ◆ The length of the input sentences may vary hence we cannot use a fixed number of neurons in the layers.



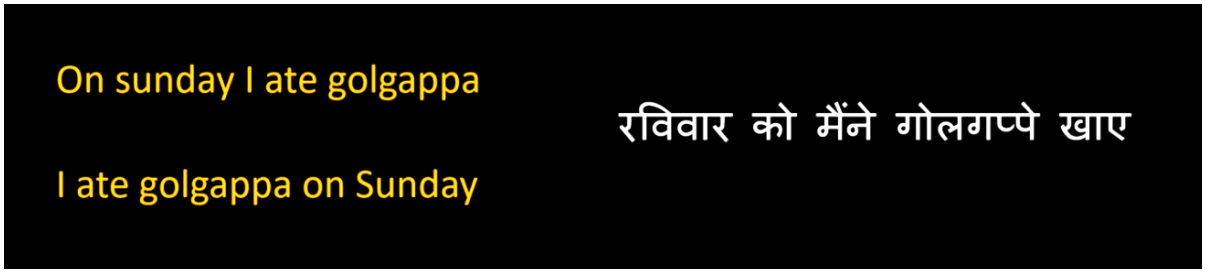
➔ **Too much computation:**

- As the length of the input sentences(vocabulary size) increases the size of the one-hot encoded vector also increases which results in complex computations.

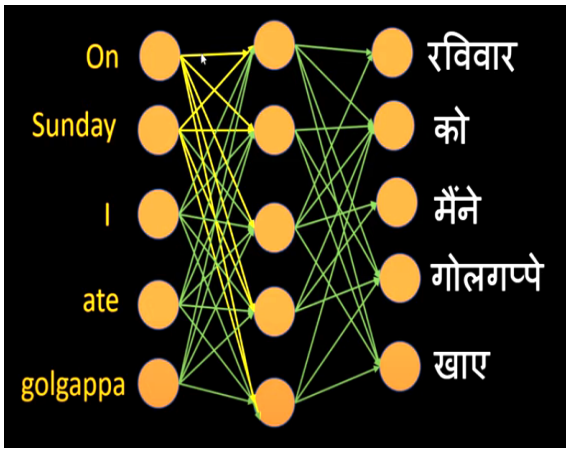


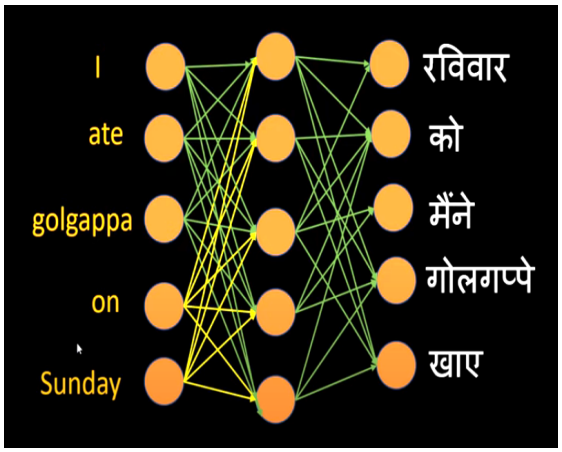
➔ **Parameters are not shared:**

- Two different sentences may have the same translation as the example given below.

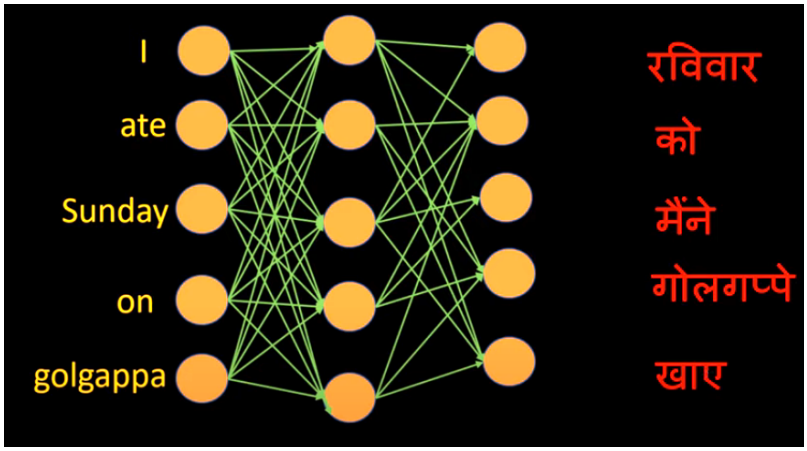


- The different sentences use different sets of edges and hence the parameters are not shared. The following diagram makes it clear to understand the problem.





➔ **Sequence of the input data is important.**



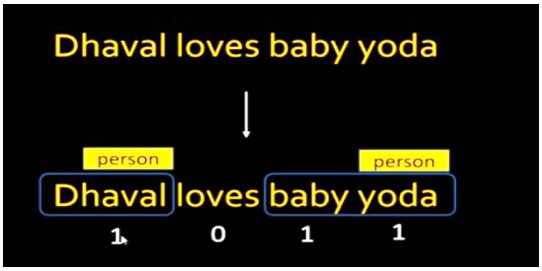
- In the above example, if we change the sequence of words of the sentence that is given as input to the model it would completely change the meaning of the sentences and sometimes the sentence could even become invalid. Hence the sequence of words in the input sentence is very important.

Hence the following are the 3 major issues of building ANN model for sequence data:

Architecture of RNN

The RNN model overcomes the above mentioned issues that exist with simple neural networks. To better understand the RNN architecture let's take an example of Named-Entity Recognition that categorizes the input sentences into certain categories.

With RNN, we represent the input sentence as a vector consisting of 1's for the words that represent person name, organisation, country etc. and 0's for the remaining words which is depicted in the image below:

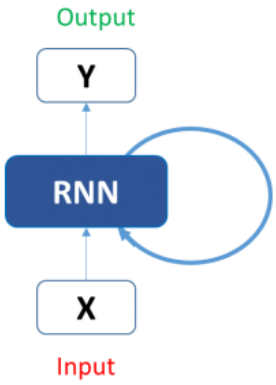


Basically the RNN functions as follows:

Each word in the sentence could be represented in the vector form using many methods such as one-hot encoding. To do one-hot encoding we create a vocabulary consisting of all the unique words, sort the words alphabetically and index all of them. To create a vector for each word, we substitute 1 in the vector representation at the index to which the word is associated with and fill all the remaining places with 0's.

The vector is given as input to the model. The model has an input layer, hidden layer and an output layer containing a number of neurons and every neuron consists of a tanh function as the activation function and softmax function (helps in normalizing the output).

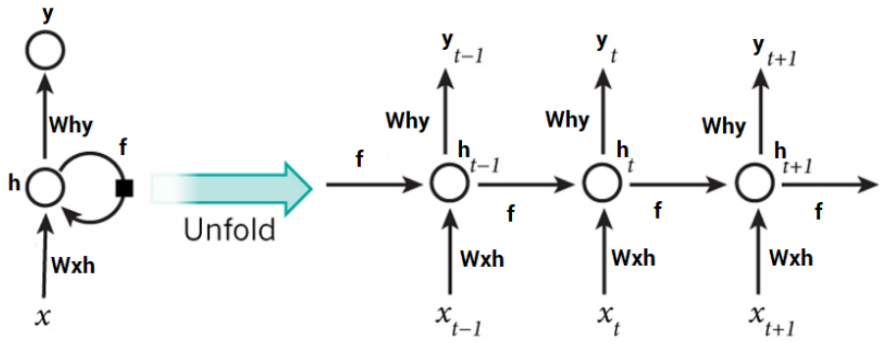
Generic Representation of RNN



We process the sentence word by word. At each step (when a word is given as input) we get corresponding output (y_t). The output of the previous step along with the next word in the sequence becomes the input of the current step. Initially for the first word we can set the output of the previous step as a vector with all zeros. This helps us retain the context (memory) of the input sentence over time and processes inputs efficiently to give better results.This process continues until all the words in the input sentence are processed. That's how the network gets the name *Recurrent Neural Network* as it recursively processes each word in the sentence.

Understanding Recurrent Neural Network in Detail

The diagram below illustrates how the RNN works. It depicts the RNN architecture unrolled over time.



At every step, the value of the activation function in the hidden layer at time t,h_{t-1} multiplied by its weight value along with input at time t, x_t multiplied by its weight value becomes the input for the hidden layer at time t.

The formula for the current state can be written as,

$$h_t = f(h_{t-1}, x_t)$$

Here, h_t is the new state at time t. The hidden layer contains an activation function (tanh) which processes the given input. If weight of the recurrent neuron is W_{hh} , weight of the input neuron is W_{xh} and b_t is the bias, then we can write the equation for the state at time t as,

$$h_t = tanh\,(W_{hh}h_{t-1} + W_{xh}x_t + b_t)$$

tanh function or *relu* (reactify linearity) whenever we need non-linearity as the activation function. With these functions the negative values are replaced with positive values. The *relu* function replaces the negative values with 0's to bring in non-linearity.

The output of the hidden layer at each step goes through a softmax function which gives the predicted output $\hat{y}$ (value could be either 0 or 1). Softmax function gives the output as probabilities. We can calculate the output state as,

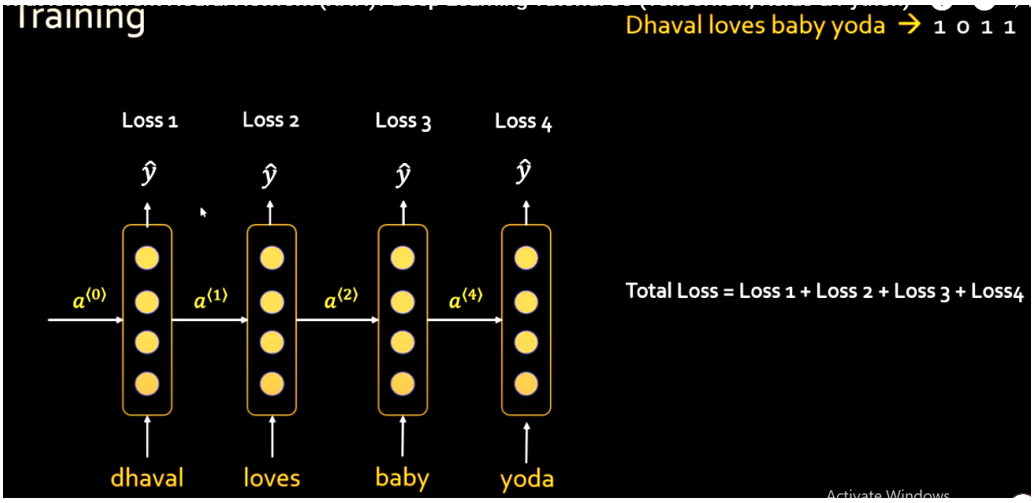
$$y_t = W_{hy} h_t + b_y$$

Training the RNN model

Having learnt about the RNN architecture now let's learn how to train the model.

X	y
Dhaval loves baby yoda	1 0 1 1
Bob told Ahmed that pizza is delivered	1 0 1 0 0 0 0
Ironman punched on hulk's face	1 0 0 1 1

The above diagram consists of the training samples where x column contains the statements and y denotes whether the given word is a name of a person or not.The training process is simplified using the following picture:



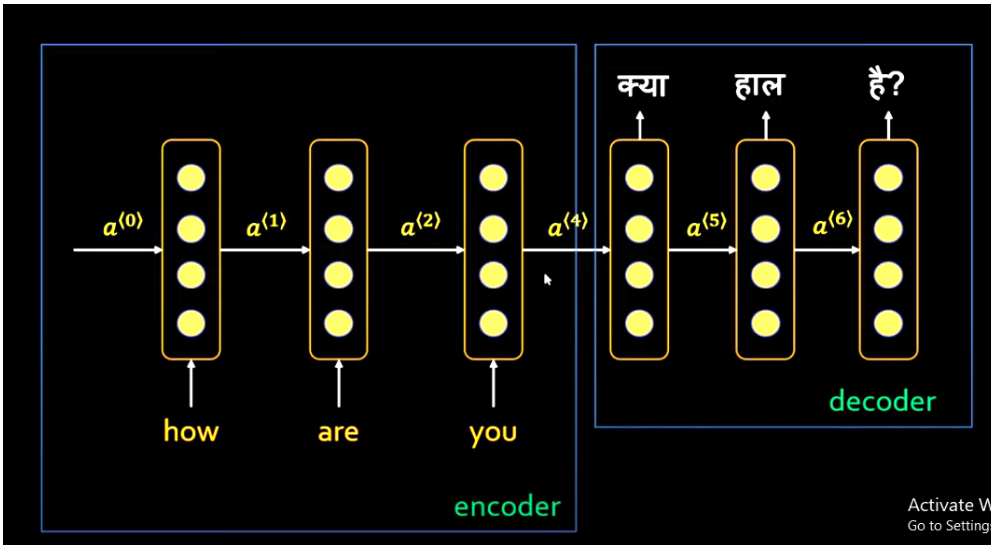
We initialize the weight values randomly in the first step. As we process words in the sentence one at a time, at each step we obtain the result for each word, known as the *predicted output* $\hat{y}$ and compare this output with the actual output that the model was supposed to give (the y column in the previous image, also shown at the top right corner of

the above image). If they do not match, we compute the loss and this is done for each step. Finally, we add up all the losses computed at each step which gives us the *total error*. Then we back propagate the model and adjust the weight values so as to reduce the total loss. The same process continues for all the training samples.

Suppose we have 1000 training samples, training the model with all the 1000 samples at once and then computing the total error and then backpropagating the model and updating weight values using gradient descent for minimizing the error is known as *1 epoch*.
After repeating the process for say 20 epochs, the total error will be minimized and that's when we can say that our model has been trained.

RNN with an example

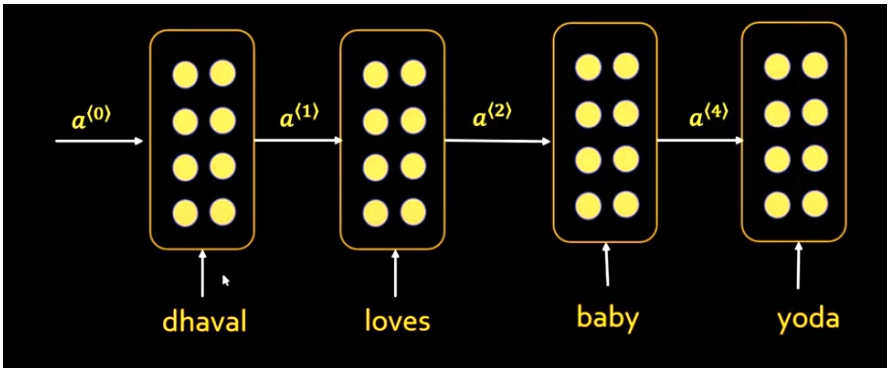
Let's understand how RNN can be used for language translation. Consider the following picture of RNN demonstrating the translation process.



The RNN consists of encoder and decoder. The encoder processes all the words in the sentence. Once all the words have been processed, the model starts translating the sentence.The decoder performs the translation of the sentence one word at a time.

Deep Neural Network

We Could also use Deep Neural Network instead for the same purpose. The difference lies in the number of hidden layers in the architecture. While RNNs consist of a single hidden layer, Deep Neural Networks could consist of multiple hidden layers. The picture below shows Deep Neural Network architecture.



Limitations of RNN

Vanishing Gradient

Due to vanishing gradient the model tends to forget what has been learnt previously. Whenever we back propagate the model to reduce the error, we find the gradient (slope) of the error either using MSE or crossentrophy. As we move backwards along the network the update in the weight value is very small or negligent such that it does not minimize the error to give the correct output.

Exploding Gradient problem

In this case, weight updating is so huge that the network cannot learn from training data hence global minima can never be reached.

Bi-directional RNN

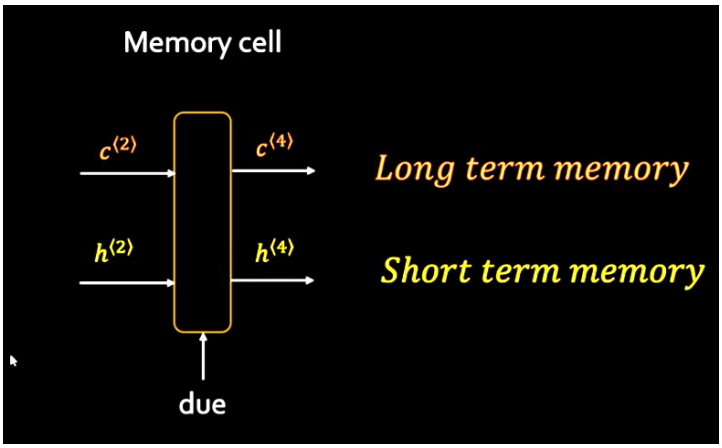
We need bidirectional RNN in some applications such as Named Entity Recognition (NER) and such. Consider the following example.

*He said **Teddy** Roosevelt served as one of the best Presidents of the US.*
*He said **Teddy** bear is her favorite.*

As the model processes given input sequentially, for example, to process the word *Teddy* in the above two sentences, it might need to look at the words that are far ahead in the sequence and then come back and process the word *Teddy* to predict the output correctly. In such cases, where we need the context of the entire sentence in order to give output we would need to use Bi-directional RNN.

LSTM

LSTM stands for Long-Short Term Memory. An LSTM has a similar control flow as a recurrent neural network. It processes data passing on information as it propagates forward. The differences are the operations within the LSTM's cells.

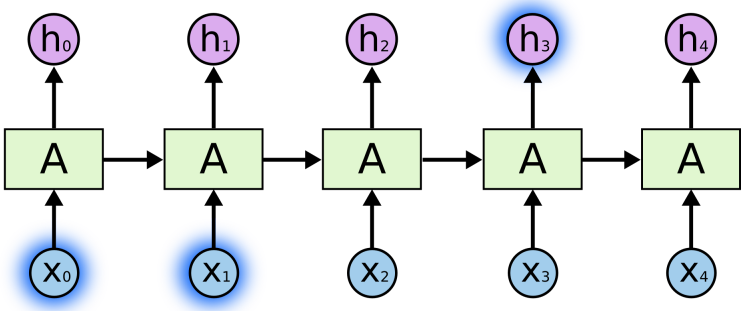


Why do we need to use LSTM?

RNNs have a short-term memory that persists data only for a short period of time. LSTMs are able to handle long-term dependencies.Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones.

If we are trying to predict the last word in "*the clouds are in the sky*," we don't need any further context because it's pretty obvious the next word is going to be *sky*.

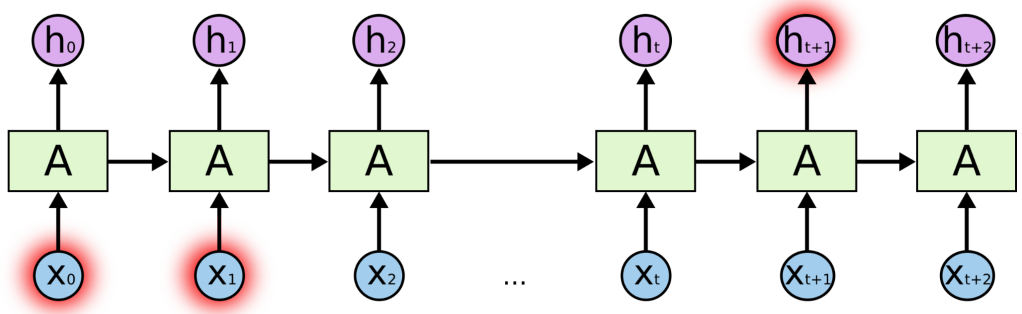
In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information as shown in the diagram below.



But there are also cases where we need more context.

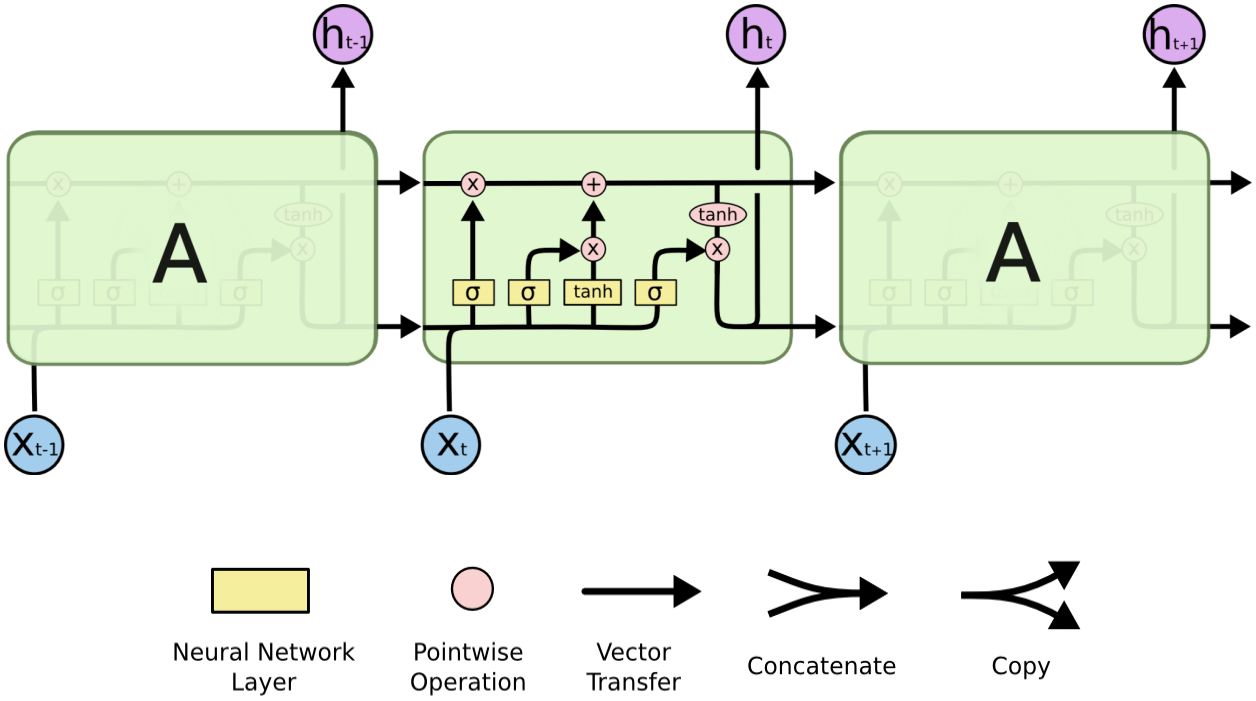
Consider trying to predict the last word in the text "*I grew up in France… I speak fluent French*."

Recent information suggests that the next word is probably the name of a language, but if we want to narrow down which language, we need the context of *France*, from further back. Unfortunately, as that gap grows, RNNs become unable to learn to connect the information.



Architecture of LSTM

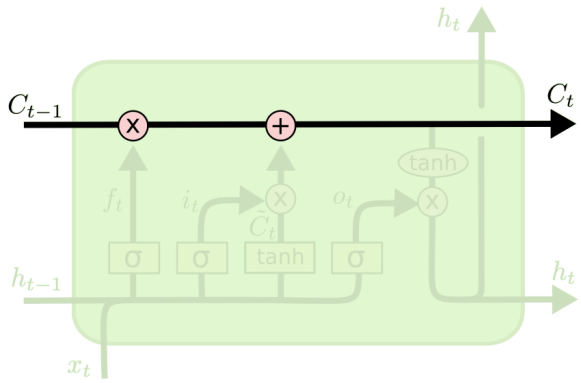
The following diagram shows the structure of the repeating modules in an LSTM.



The repeating module in an LSTM contains four interacting layers.The labels x_{t-1} to x_{t+1} are the data point at different intervals of time and h_t represents the output for corresponding cell state(Individual block for each time step).

Cell state in LSTM

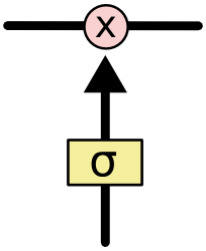
The key to LSTMs is the cell state, the horizontal line running through the top of the diagram which allows the network to remember information for a long period of time. It runs straight down the entire chain, with only some minor linear interactions.



Gates

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed out of a sigmoid neural network layer and a pointwise multiplication operation.



The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means “let nothing through,” while a value of one means “let everything through!”

A tanh layer outputs numbers between -1 and 1.

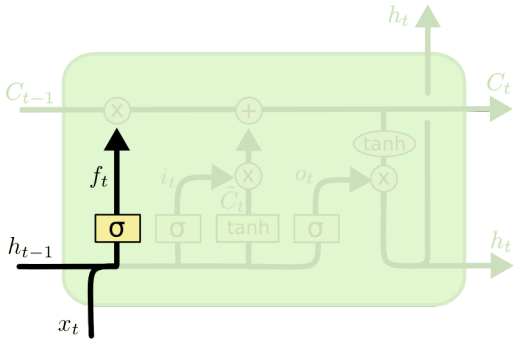
An LSTM has three of these gates, to protect and control the cell state. The whole LSTM can be simplified by using three primary gates:

Forget gate layer

The *forget gate layer* decides what information is not necessary to retain in memory and to throw it away from the cell state and allow only the necessary information to be passed onto the next state.

It looks at h_{t-1} and x_t, and outputs a number between 0 and 1 for each number in the cell state C_{t-1}. A 1 represents "*completely keep this*" while a 0 represents "*completely get rid of this.*"

Let's go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.



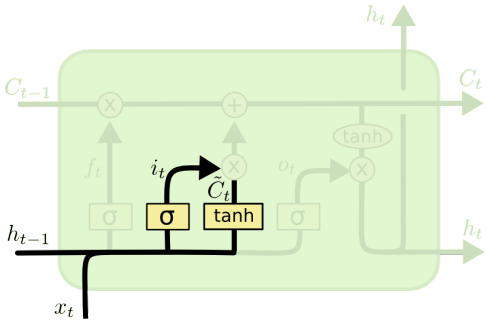
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Input Gate Layer

Input gate layer decides what new information we're going to store in the cell state. This has two parts.
First, a sigmoid layer decides which values we'll update.

Next, a tanh layer creates a vector of new candidate values, $\tilde{C_t}$, that could be added to the state. In the next step, we'll combine these two to create an update to the state.

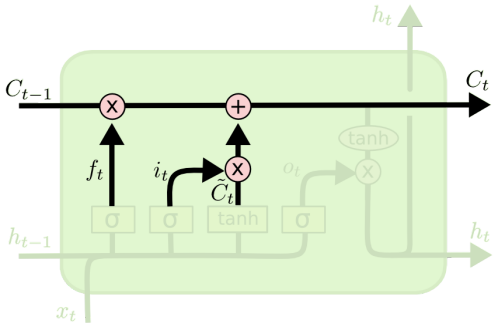
In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

We need to update the old cell state, C_{t-1}, into the new cell state C_t. We multiply the old state by f_t, forgetting the things we decided to forget earlier. Then we add $i_t * \tilde{C_t}$. This is the new candidate values, scaled by how much we decided to update each state value.

In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information.

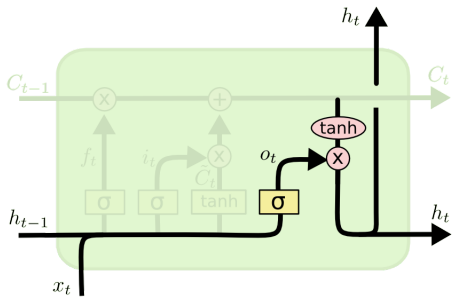


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Output gate

Last gate is the *output gate*, the output gate decides what information is relevant to the next hidden state. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between –1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.

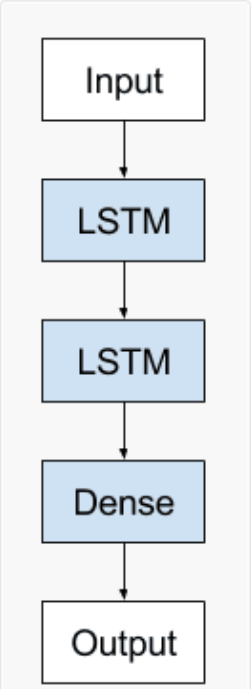


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Stacked LSTM

Stacked LSTM architecture can be defined as an LSTM model comprised of multiple LSTM layers. An LSTM layer above provides a sequence output rather than a single valued output to

the LSTM layer below. Specifically, one output per input time step, rather than one output time step for all input time steps. Stacked LSTM architecture can be understood from the following diagram.

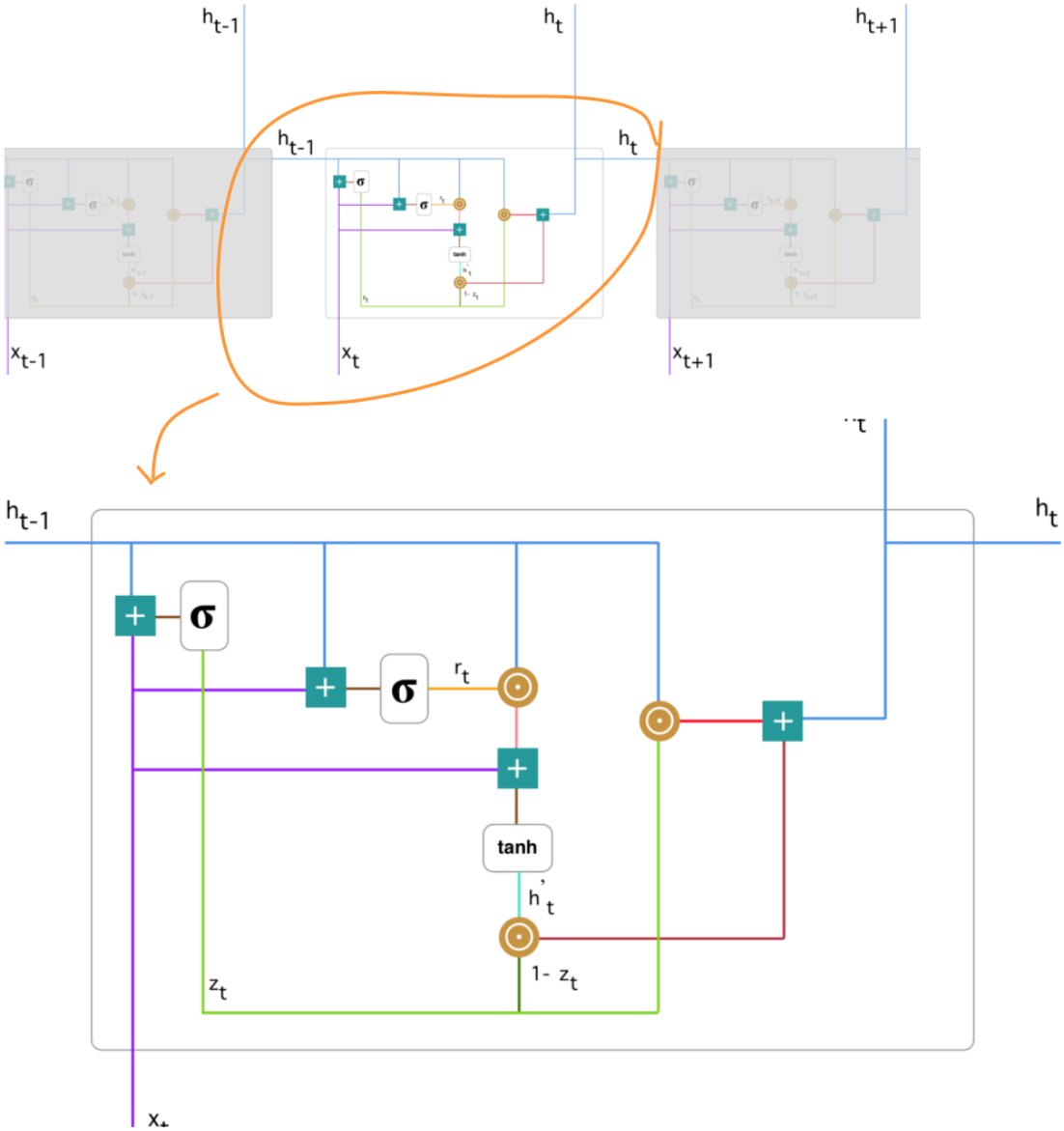


Stacked Long Short-Term Memory Archiecture

GRU

Gated Recurrent Unit (GRU) is an improved version of standard recurrent neural networks. It aims to solve the vanishing gradient problem which comes with a standard recurrent neural network.

GRU uses an **update gate** and a **reset gate.** These are two vectors which *decide what information should be passed to the output*. They can be trained to keep information from long ago, without washing it through time or removing information which is irrelevant to the prediction.



Notations

"plus" operation

"sigmoid" function

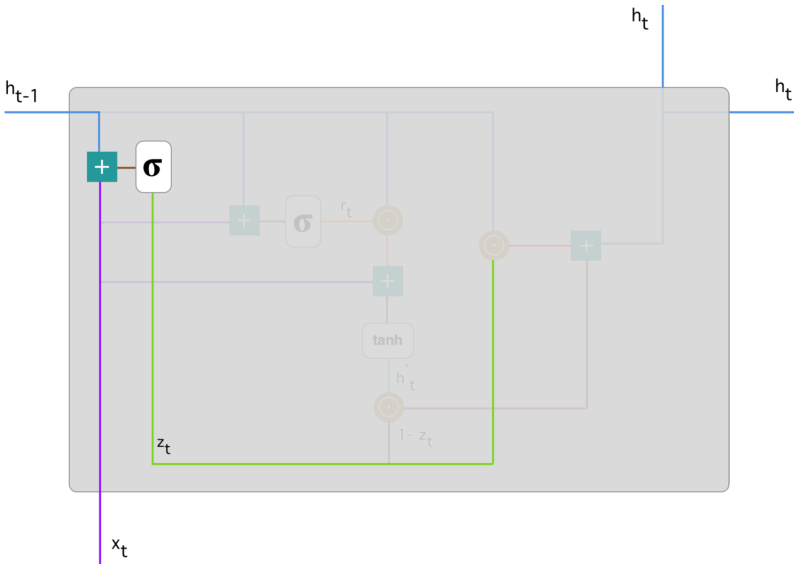
"Hadamard product" operation

"tanh" function

Update gate

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

1. x_t is multiplied by its own weight W(z).
2. h_(t-1) which holds the information for the previous t-1 units, is also multiplied by its own weight U(z).
3. Both results are added together
4. sigmoid activation function is applied so that the result is between 0 and 1.



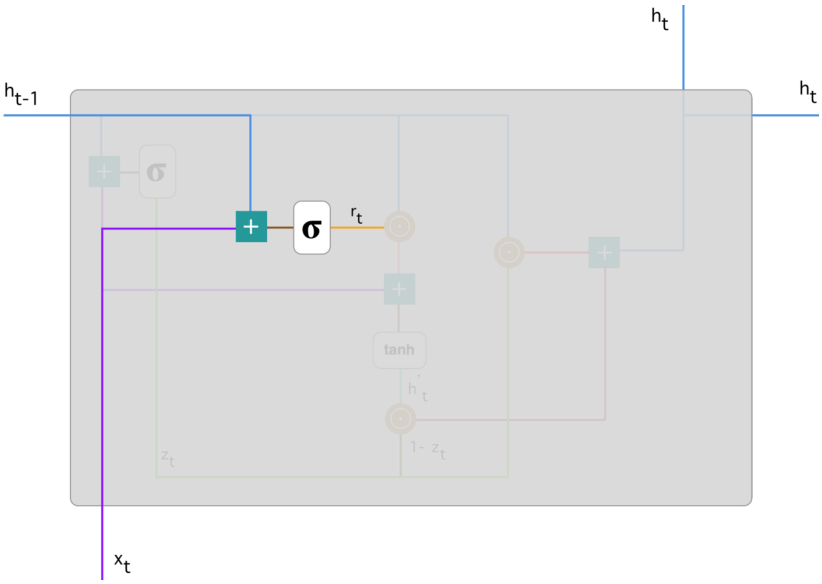
The update gate helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future.

Reset gate

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

As before, we plug in h(t-1) and x_t, multiply them with their corresponding weights, sum the results and apply the sigmoid function.

This gate is used to decide how much of the past information to forget.

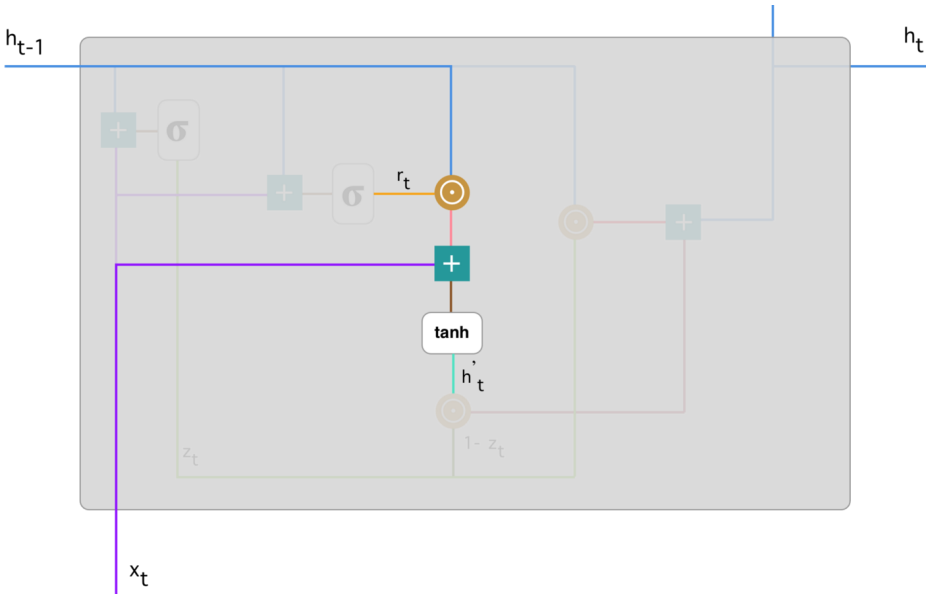


Current memory content

We introduce a new memory content which will use the reset gate to store the relevant information from the past.

$$h_t^{'} = \tanh(Wx_t + r_t \odot Uh_{t-1})$$

1. Multiply the input x_t with a weight W and h_(t-1) with a weight U.
2. Calculate the Hadamard (element-wise) product between the **reset gate r_t** and Uh_(t-1).
3. Sum up the results of step 1 and 2.
4. Apply the nonlinear activation function tanh.

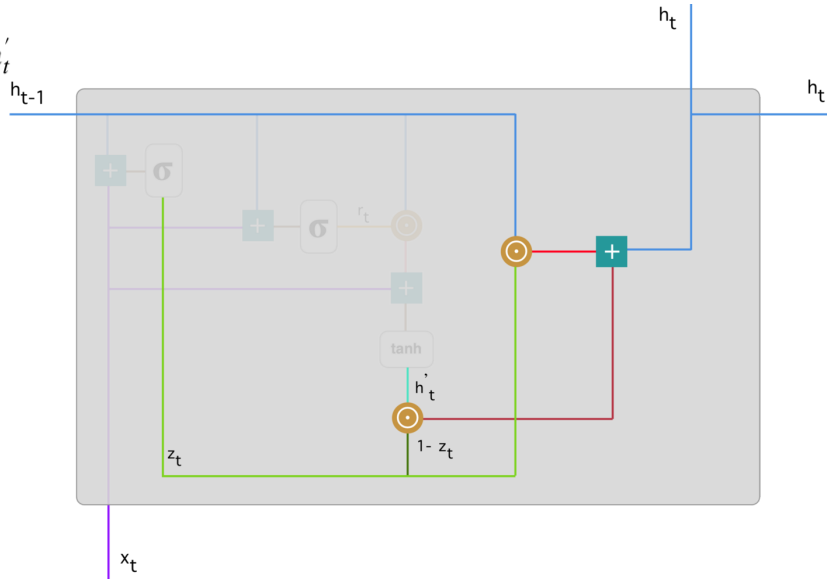


Final memory at current time step

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h_t^{'}$$

It determines what to collect from the current memory content h'_t and what from the previous steps — h_(t-1).

1. Apply element-wise multiplication to the update gate z_t and h_(t-1).
2. Apply element-wise multiplication to (1-z_t) and h'_t.
3. Sum the results from step 1 and 2.



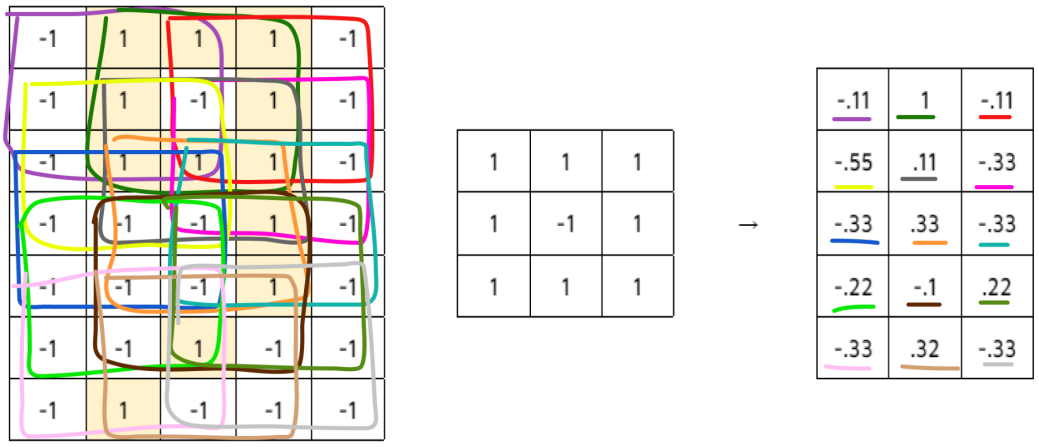
Convolutional Neural Network

A typical MLP will not handle location invariant (for e.g. classifying an animal where it's sitting differently to the same animal), hence we need CNN.

So we apply filters on an image in order to extract features.

Convolution Operation

Example of recognising an image with the digit 9. We first use a loop pattern filter.



* check for accuracy

We fit the loop filter on the image and calculate the dot product and store the result in another matrix. Follow the color encircling the 3x3 matrix in the input to see its for value for dot product with the filter matrix. We can move the filter according to any number of strides. We call the resulting matrix a feature map. A value of 1 indicates that a loop is present in this case.

We similarly apply the vertical line and the diagonal filter. In this way we get a stack of feature maps.

ReLU Operation

In order to bring non-linearity, wherever there's a negative value in the feature maps we have obtained, we replace it with 0 and keep the positive values as they are.

0	1	0
0	.11	0
0	.33	0
0	.01	0
0	.33	0

Max Pooling

We take the maximum value across our filters, and thereby reduce the dimensions and avoid overfitting.

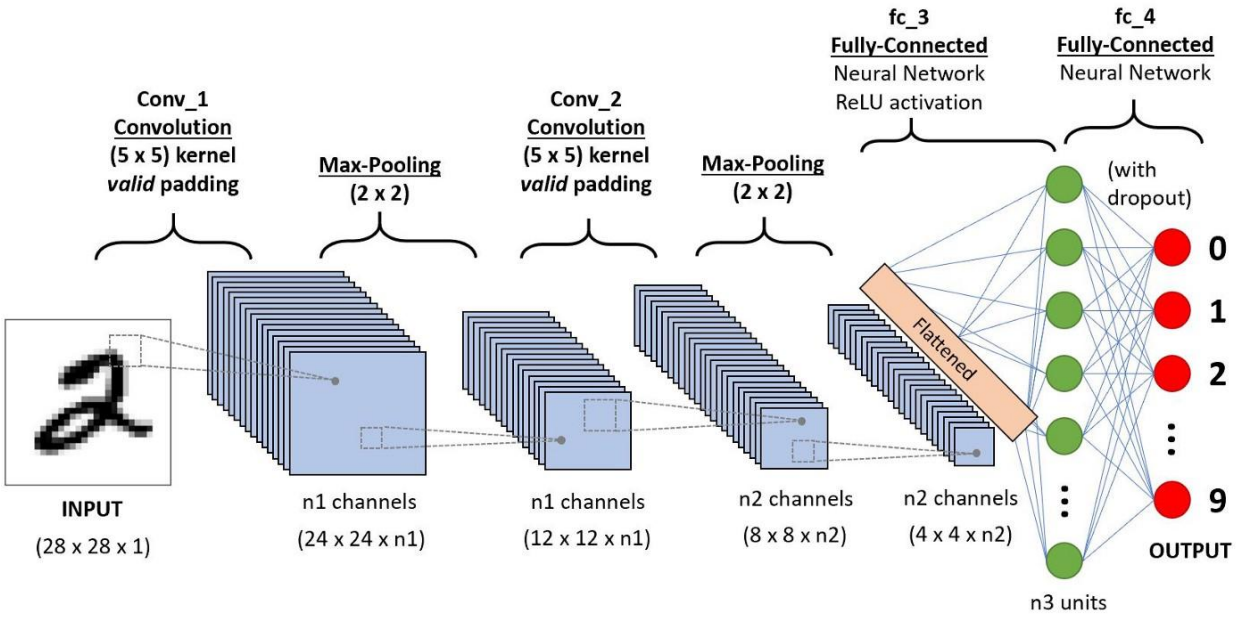
5	1	3	**9**
8	2	9	2
1	**3**	0	1
2	2	**2**	0

→

8	9
3	2

In case of average pooling, we would find the average of the matrix.

Here's an image summing up the network
(Image source:
https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53)



Vectorisation

When we have features in the form of strings, we need to convert it into a vector. This mapping of words or phrases to a vector of real numbers is called vectorisation.

One Hot Encoding

Models cannot work directly on the categorical data. For this, we require one hot encoding process. One-hot encoding deals with the data in binary format so we encode the categorical data in binary format.

One-hot means that we can only make an index of data 1 (true) if it is present in the vector or else 0 (false). So every data has its unique representation in vector format.

For example, if we have an array of data like : ["python","java","c++"] then the one hot encoding representation of this array will be :

[[1 , 0 , 0]

[0 , 1 , 0]

[0 , 0 , 1]]

There are two major issues with this approach for word embedding.

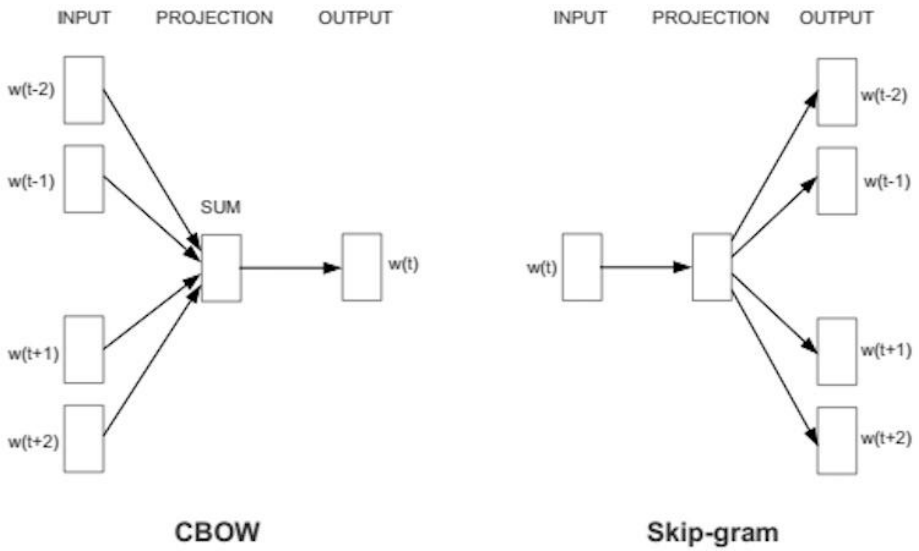
First issue is the curse of dimensionality, which refers to all sorts of problems that arise with data in high dimensions. Most of the matrix is taken up by zeros, so useful data becomes sparse. Imagine we have a vocabulary of 50,000. (There are roughly a million words in English language.) Each word is represented with 49,999 zeros and a single one, and we need 50,000 squared = 2.5 billion units of memory space. Not computationally efficient.

Second issue is that it is hard to extract meanings. Each word is embedded in isolation, and every word contains a single one and N zeros where N is the number of dimensions. The resulting set of vectors do not say much about one another. If our vocabulary had "orange," "banana," and "watermelon," we can see the similarity between those words, such as the fact that they are types of fruit, or that they usually follow some form of the verb "eat." We can easily form a mental map or cluster where these words exist close to each other. But with one-hot vectors, all words are equal distance apart.

Word2Vec

Word2vec is a two-layer neural net that processes text by **vectorizing** words. Its input is a text corpus and its output is a set of vectors: feature vectors that represent words in that corpus. The objective behind using word2vec is to have words with similar context occupy close spatial positions. Here comes the idea of generating distributed representations. Intuitively, we introduce some dependence of one word on the other words. The words in context of this word would get a greater share of this dependence.

Word2Vec is a method to construct such an embedding. It can be obtained using two methods (both involving Neural Networks): **Skip Gram** and **Common Bag Of Words**.



- **Continuous Bag-of-Words Model** which predicts the middle word based on surrounding context words. The context consists of a few words before and after the current (middle) word. This architecture is called a bag-of-words model as the order of words in the context is not important.
- **Continuous Skip-gram Model** which predicts words within a certain range before and after the current word in the same sentence.

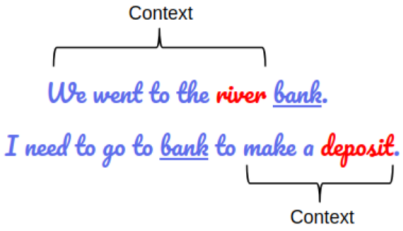
Skip Gram works well with small amounts of data and is found to represent rare words well. On the other hand, CBOW is faster and has better representations for more frequent words.

BERT

Bidirectional Encoder Representations from Transformers

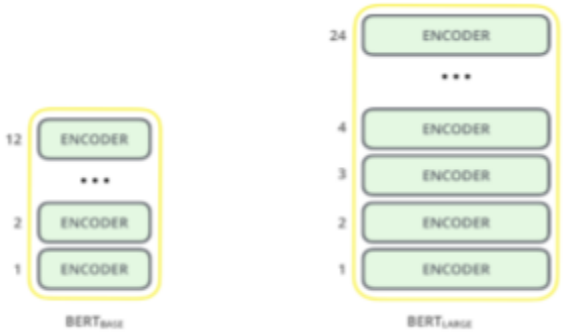
Bert model consists of a multi layer bidirectional transformer encoder. BERT is pre-trained on the entire Wikipedia(that's 2,500 million words!) and Book Corpus (800 million words). It is an open source machine learning framework for natural language processing (NLP).

language models could only read text input sequentially , either left-to-right or right-to-left but couldn't do both at the same time. BERT is different because it is designed to read in both directions at once.

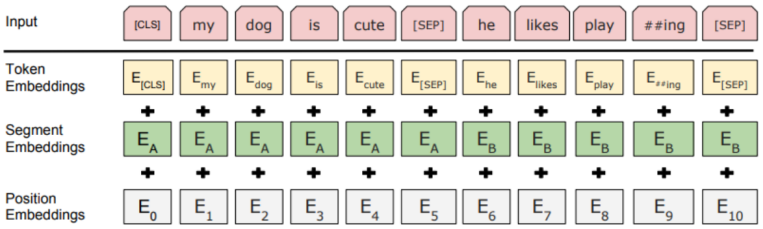


It comes in two sizes

BERT Base: 12 layers (transformer blocks), 12 attention heads, and 110 million parameters
BERT Large: 24 layers (transformer blocks), 16 attention heads and, 340 million parameters



Text PreProcessing



- Position Embeddings: BERT learns and uses positional embeddings to express the position of words in a sentence. These are added to overcome the limitation of Transformer which, unlike an RNN, is not able to capture "sequence" or "order" information

- Segment Embeddings: BERT can also take sentence pairs as inputs for tasks (Question-Answering). That's why it learns a unique embedding for the first and the second sentences to help the model distinguish between them. In the above example, all the tokens marked as EA belong to sentence A (and similarly for EB)

- Token Embeddings: These are the embeddings learned for the specific token from the WordPiece token vocabulary

BERT is pre-trained on two NLP tasks:

- Masked LM (MLM)

  - The objective of Masked Language Model (MLM) training is to hide a word in a sentence and then have the program predict what word has been hidden (masked) based on the hidden word's context.
  - Eg. Input: the man went to the [MASK1] . he bought a [MASK2] of milk. Labels: [MASK1] = store; [MASK2] = gallon

- Next Sentence Prediction (NSP)

  - The objective of Next Sentence Prediction training is to have the program predict whether two given sentences have a logical, sequential connection or whether their relationship is simply random.
  - Eg.

    Sentence A: the man went to the store .

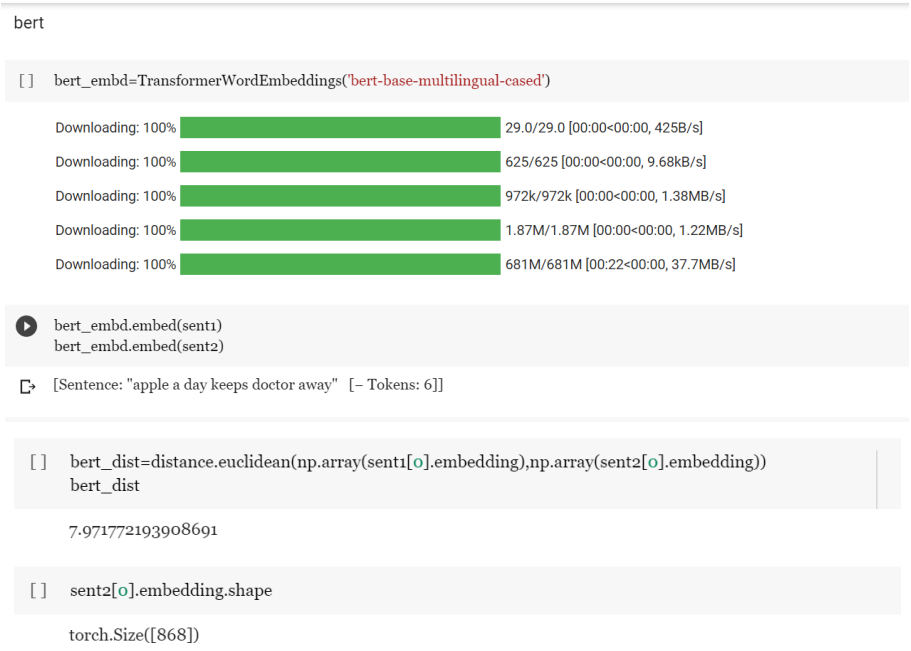
    Sentence B: he bought a gallon of milk .

    Label: IsNextSentence

    Sentence A: the man went to the store .

    Sentence B: penguins are flightless .

    Label: NotNextSentence



Unsupervised Learning Algorithms

In this setting, we are in some sense working blind; the situation is referred to as unsupervised because we lack a response variable that can supervise our analysis.

Principal Components Analysis

Principal Component Analysis, or PCA, is a dimensionality-reduction technique that enables you to identify correlations and patterns in a dataset so that it can be transformed into a dataset of significantly lower dimension without loss of any important information.

Step by step PCA

F	E1	E2	E3	E4	mean
X	4	8	13	7	**8**
Y	11	4	5	14	**8.5**

1. **Standardization**

   Standardization is all about scaling your data in such a way that all the variables and their values lie within a similar range.

2. **Covariance matrix computation**

   A covariance matrix expresses the correlation between the different variables in the dataset.

$$\begin{bmatrix} Cov(a,a) & Cov(a,b) \\ Cov(b,a) & Cov(b,b) \end{bmatrix}$$

The covariance matrix is a p × p symmetric matrix (where p is the number of dimensions) that has as entries the covariances associated with all possible pairs of the initial variables.

If **positive** then : Two variables increase and decrease together(correlated)
If **negative** then : One increases when the other decreases(Inversely correlated)

$$Cov_{x,y} = \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})/N - 1$$

$$Cov(S) = \begin{bmatrix} 14 & -11 \\ -11 & 23 \end{bmatrix}$$

3. **Compute the Eigenvectors and Eigenvalues**

   If your data set is of 5 dimensions, then 5 ***principal components*** *are computed, such that, the first principal component stores the maximum possible information and the second one stores the remaining maximum info and so on, you get the idea.*

   Eigenvectors are those vectors when a linear transformation is performed on them then their direction does not change.
   Eigenvalues simply denote the scalars of the respective eigenvectors.

$$\begin{bmatrix} 14-\lambda & -11 \\ -11 & 23-\lambda \end{bmatrix}$$

$$(14-\lambda)(23-\lambda) - 121 = 0$$

$$\lambda^2 - 37\lambda + 201 = 0$$

λ1 = 30.38486 λ2 = 6.61515

$$e1 = \begin{bmatrix} 0.55739 \\ -0.8303 \end{bmatrix} \qquad e2 = \begin{bmatrix} 0.83042 \\ 0.55714 \end{bmatrix}$$

4. **Computing the principal components**

   Once we have computed the Eigenvectors and eigenvalues, all we have to do is order them in the descending order, where the eigenvector with the highest eigenvalue is the most significant and thus forms the first principal component. The principal components of lesser significance can thus be removed in order to reduce the dimensions of the data.

$$P_{11} = e_1^T \begin{bmatrix} 4-\bar{x} \\ 11-\bar{y} \end{bmatrix} = -4.30531 \qquad P_{12} = e_1^T \begin{bmatrix} 8-\bar{x} \\ 4-\bar{y} \end{bmatrix} = 3.7361$$

$$P_{13} = e_1^T \begin{bmatrix} 13-\bar{x} \\ 5-\bar{y} \end{bmatrix} = 5.6928 \qquad P_{14} = e_1^T \begin{bmatrix} 7-\bar{x} \\ 14-\bar{y} \end{bmatrix} = -5.1238$$

$$\begin{bmatrix} -4.3052 & 3.7361 & 5.6928 & -5.1238 \end{bmatrix}$$

$$P_{21} = e_2^T \begin{bmatrix} 8 - \bar{x} \\ 11 - \bar{y} \end{bmatrix} = -1.9277 \qquad P_{22} = e_2^T \begin{bmatrix} 8 - \bar{x} \\ 4 - \bar{y} \end{bmatrix} = 2.5083$$

$$P_{23} = e_2^T \begin{bmatrix} 13 - \bar{x} \\ 5 - \bar{y} \end{bmatrix} = 6.1024 \qquad P_{24} = e_2^T \begin{bmatrix} 7 - \bar{x} \\ 14 - \bar{y} \end{bmatrix} = 2.2354$$

$$\begin{bmatrix} -1.9277 & 2.5083 & 6.1024 & 2.2354 \end{bmatrix}$$

5. **Deriving the new dataset**

   The last step in performing PCA is to rearrange the original data with the final principal components which represent the maximum and the most significant information of the data set. In order to replace the original data axis with the newly formed Principal Components, you simply multiply the transpose of the original data set by the transpose of the obtained feature vector.

   We shift the x and y axis to the mean variance value of x and y. i.e (8, 8.5)

   Plot e1 and e2 points on shifted axes

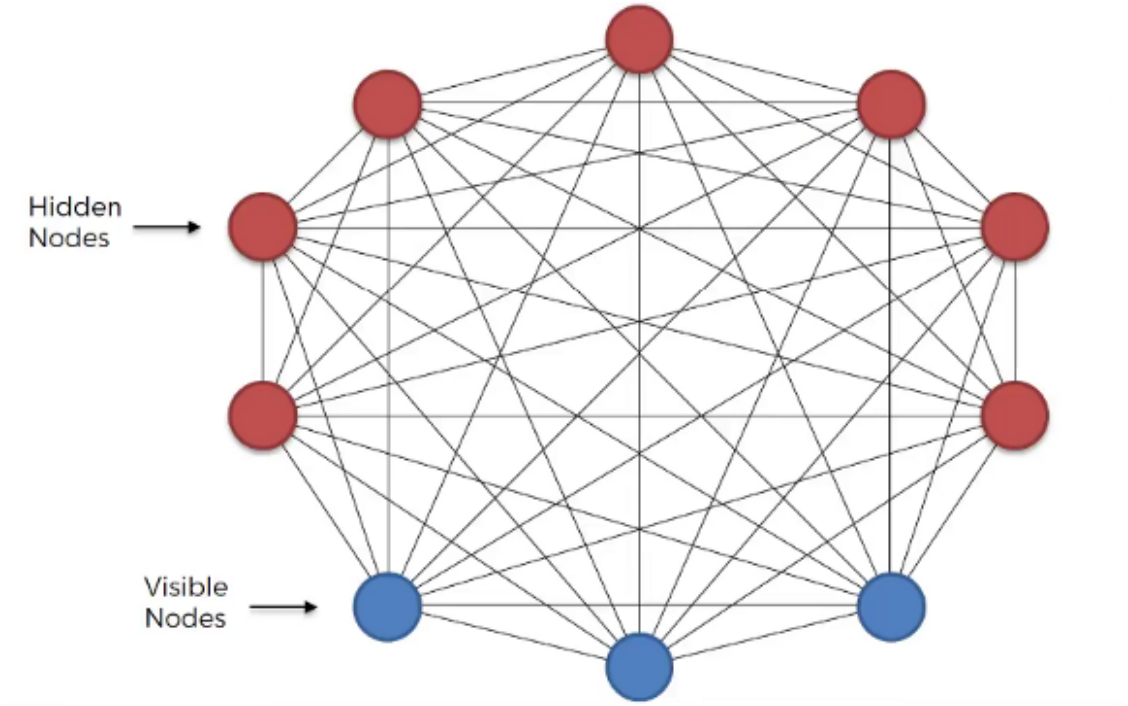
   e1 is PC1 which is the highest eigenvalue. We choose the most valuable eigenvalue (the one having maximum variance).

   e2 is PC2 which is the second highest eigenvalue.

Boltzmann Machine

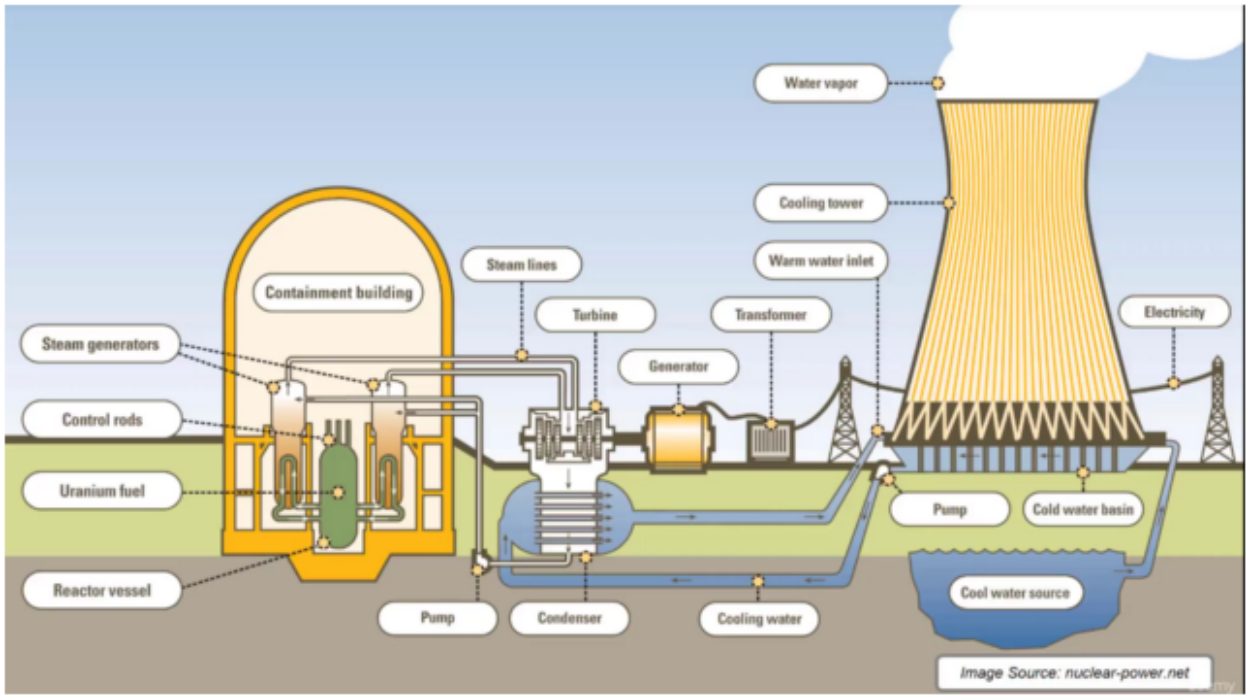
Boltzmann Machine is a generative unsupervised model, which involves learning a probability distribution from an original dataset and using it to make inferences about never before seen data.

Boltzmann Machine has an input layer (also referred to as the visible layer) and one or several hidden layers (also referred to as the hidden layer).



- Boltzmann Machine uses neural networks with neurons that are connected not only to other neurons in other layers but also to neurons within the same layer.
- Everything is connected to everything. Connections are bidirectional, visible neurons connected to each other and hidden neurons also connected to each other
- Boltzmann Machine doesn't expect input data, it generates data. Neurons generate information regardless of whether they are hidden or visible.
- For Boltzmann Machine all neurons are the same, it doesn't discriminate between hidden and visible neurons. For the Boltzmann Machine, the whole thing is the system and its generating the state of the system.

The best way to think about it is through an example nuclear power plant.



- For example we have a nuclear power station and there are certain things we can measure in a nuclear power plant like the temperature of the containment building, how quickly the turbine is spinning, the pressure inside the pump, etc.
- There are lots of things we are not measuring like the *speed of the wind, the moisture of the soil in this specific location, the weather and humidity percentage of that day*, etc.
- All these parameters together form a system, they all work together. All these parameters are binary. So we get a whole bunch of binary numbers that tell us something about the state of the power station.
- What we would like to do is to notice when it is going to go in an unusual state. A state that is not like a normal state which we had seen before. And we don't want to use supervised learning for that. Because we don't want to have any examples of states that cause it to blow up.
- We would rather be able to detect that when it is going into such a state without even having seen such a state before. And we could do that by building a model of a normal state and noticing that this state is different from the normal states.

That's what the Boltzmann Machine represents.

The way this system works is that we use our training data and feed it into the Boltzmann Machine as input to help the system adjust its weights. It resembles our system, not any nuclear power station in the world.

It learns from the input, what are the possible connections between all these parameters, how do they influence each other and therefore it becomes a machine that represents our system.
We can use this Boltzmann Machine to monitor our system
Boltzmann Machine learns how the system works in its normal states through a good example.

Boltzmann Machine consists of a neural network with an input layer and one or several hidden layers. The neurons in the neural network make stochastic decisions about whether to turn on or off based on the data we feed during training and the cost function the Boltzmann Machine is trying to minimize.
By doing so, the Boltzmann Machine discovers interesting features about the data, which help model the complex underlying relationships and patterns present in the data.
This Boltzmann Machine uses neural networks with neurons that are connected not only to other neurons in other layers but also to neurons within the same layer. That makes training an unrestricted Boltzmann machine very inefficient and Boltzmann Machine had very little commercial success.
Boltzmann Machines are primarily divided into two categories: **Energy-based Models (EBMs)** and **Restricted Boltzmann Machines (RBM)**. When these RBMs are stacked on top of each other, they are known as **Deep Belief Networks (DBN)**.

An Energy-Based Model

Energy is a term that may not be associated with deep learning in the first place. Rather, energy is a quantitative property of physics. E.g. Gravitational energy describes the potential energy a body with mass has in relation to another massive object due to gravity. Yet some deep learning architectures use the idea of energy as a metric for the measurement of the model's quality.
One purpose of deep learning models is to encode dependencies between variables. The capturing of dependencies happens through association of scalar energy to each configuration of the variables, which serves as a measure of compatibility. High energy means bad compatibility. An energy-based model always tries to minimize a predefined energy function. The energy function for the RBMs is defined as:

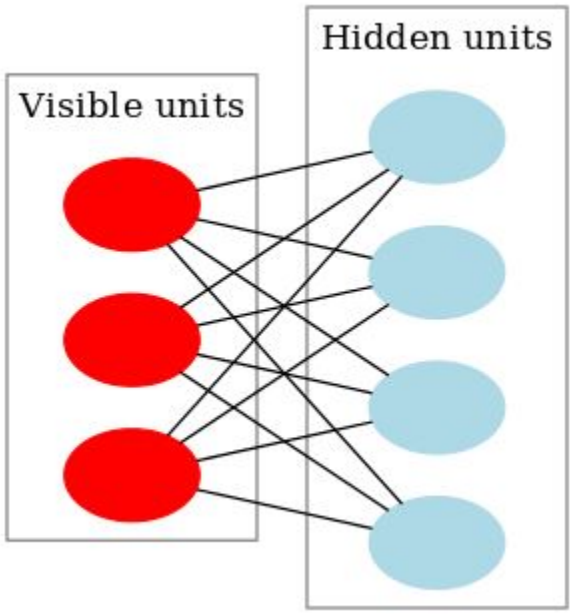
$$E(\mathbf{v}, \mathbf{h}) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_{i,j} v_i h_j w_{ij}$$

As can be noticed the value of the energy function depends on the configurations of visible/input states, hidden states, weights, and biases. The training of RBM consists of the finding of parameters for given input values so that the energy reaches a minimum.

Restricted Boltzmann Machine

What makes RBMs different from Boltzmann machines is that the visible nodes aren't connected to each other, and hidden nodes aren’t connected with each other. Other than that, RBMs are exactly the same as Boltzmann machines.

As you can see below:



- RBM is the neural network that belongs to the energy-based model
- It is a probabilistic, unsupervised, generative deep machine learning algorithm.
- RBM’s objective is to find the joint probability distribution that maximizes the log-likelihood function.
- RBM is undirected and has only two layers, Input layer, and hidden layer
- All visible nodes are connected to all the hidden nodes. RBM has two layers, visible layer or input layer and hidden layer so it is also called an asymmetrical bipartite graph.
- No intralayer connection exists between the visible nodes. There is also no intralayer connection between the hidden nodes. There are connections only between input and hidden nodes.
- The original Boltzmann machine had connections between all the nodes. Since RBM restricts the intralayer connection, it is called a Restricted Boltzmann Machine.

Since RBMs are undirected, they don’t adjust their weights through gradient descent and backpropagation.

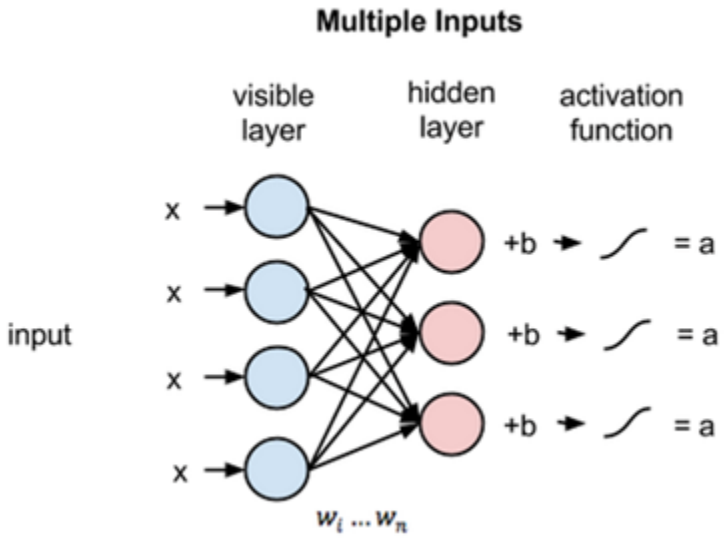
- They adjust their weights through a process called contrastive divergence.

- At the start of this process, weights for the visible nodes are randomly generated and used to generate the hidden nodes.
- These hidden nodes then use the same weights to reconstruct visible nodes.
- The weights used to reconstruct the visible nodes are the same throughout.
- However, the generated nodes are not the same because they aren't connected to each other.

How does the Restricted Boltzmann Machine Work?

In an RBM, we have a symmetric bipartite graph where no two units within the same group are connected. Multiple RBMs can also be stacked and can be fine-tuned through the process of gradient descent and back-propagation. Such a network is called a Deep Belief Network. Although RBMs are occasionally used, most people in the deep-learning community have started replacing their use with General Adversarial Networks or Variational Autoencoders.

RBM is a Stochastic Neural Network which means that each neuron will have some random behavior when activated. There are two other layers of bias units (hidden bias and visible bias) in an RBM. This is what makes RBMs different from autoencoders. The hidden bias RBM produces the activation on the forward pass and the visible bias helps RBM to reconstruct the input during a backward pass. The reconstructed input is always different from the actual input as there are no connections among the visible units and therefore, no way of transferring information among themselves.



The above image shows the first step in training an RBM with multiple inputs. The inputs are multiplied by the weights and then added to the bias. The result is then passed through a sigmoid activation function and the output determines if the hidden state gets activated or not. Weights will be a matrix with the number of input nodes as the number of rows and the

number of hidden nodes as the number of columns. The first hidden node will receive the vector multiplication of the inputs multiplied by the first column of weights before the corresponding bias term is added to it.

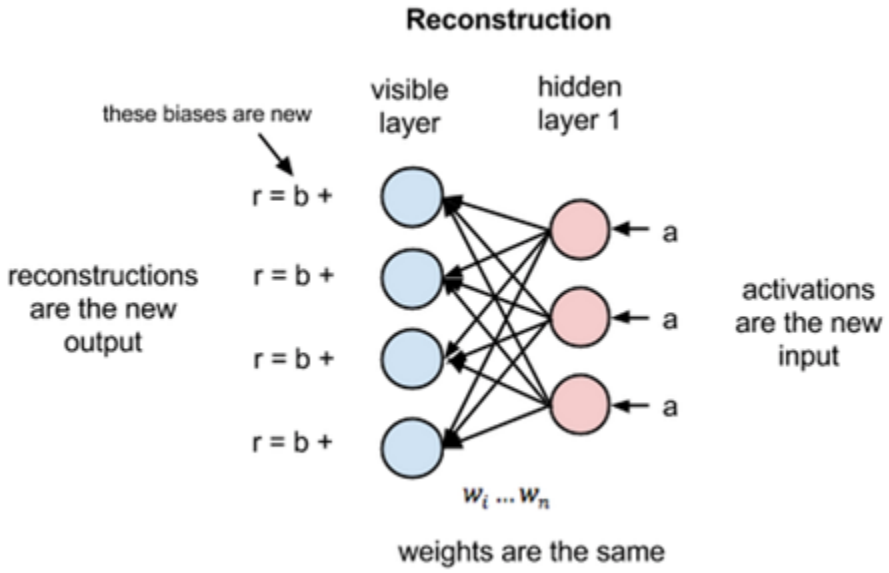
And if you are wondering what a sigmoid function is, here is the formula:

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x}$$

So the equation that we get in this step would be,

$$\mathbf{h}^{(1)} = S(\mathbf{v}^{(0)T}W + \mathbf{a})$$

where h(1) and v(0) are the corresponding vectors (column matrices) for the hidden and the visible layers with the superscript as the iteration v(0) means the input that we provide to the network) and a is the hidden layer bias vector.
(Note that we are dealing with vectors and matrices here and not one-dimensional values.)



Now this image shows the reverse phase or the reconstruction phase. It is similar to the first pass but in the opposite direction. The equation comes out to be:

$$\mathbf{v}^{(1)} = S(\mathbf{h}^{(1)}W^T + \mathbf{b})$$

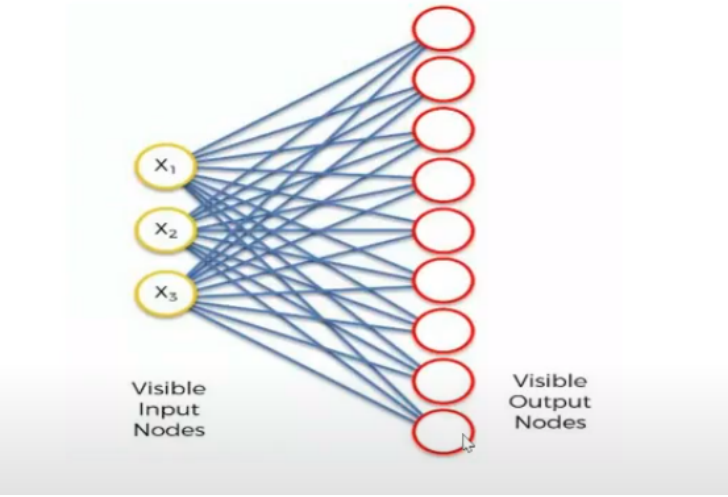
where v(1) and h(1) are the corresponding vectors (column matrices) for the visible and the hidden layers with the superscript as the iteration and b is the visible layer bias vector.

Self-organising Map (SOM)

A Self Organising Map is a neural network based approach under unsupervised learning which reduces the dimensions of data by applying competitive learning instead of error correction learning(such as backpropagation with gradient descent).

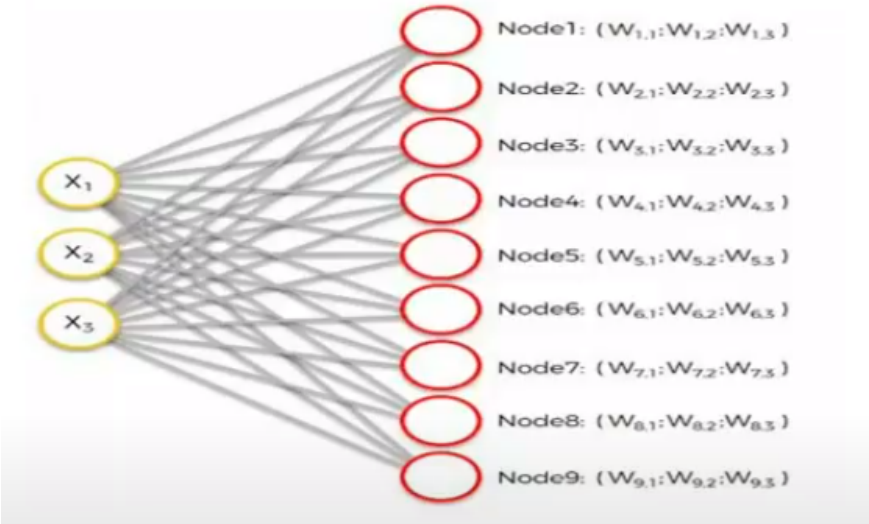
How does SOM learn?

Assume Input nodes x1, x2 and x3 having values **x1 = 0.7**, **x2 = 0.6**, **x3 = 0.9** and there are **9** output nodes as shown in the figure below.



Step 1: Initialize the weights

Assign weight values to all the connections between every output node and the input nodes. In SOM, the weights belong to the output node itself.



Step 2: Calculate the best matching unit

To determine the best matching unit, we iterate through all the nodes and calculate the **Euclidean distance** between each node's weight vector and the current input vector.

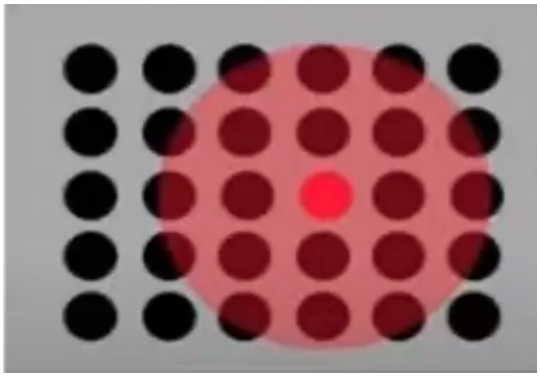
$$\text{Distance} = \sqrt{\sum_{i=0}^{i=n}(X_i - W_i)^2}$$

The output neurons are competing among themselves. Whichever neuron gets the smallest value(equivalent distance) is the winning neuron(BMU).

The new SOM will have to update its weights so that it is even closer to the dataset's first row because our input nodes cannot be updated, whereas we have control over our output nodes. SOM is drawing closer to the data point by stretching the BMU towards it.

Step 3: Calculate the size of neighborhood around the BMU

Calculate what the radius of the neighborhood should be.



Use Pythagoras theorem to determine if each node is within the radial distance of the winning neuron or not so that we can update the weight values of the winning neuron as well as its neighbouring neurons.

The size of the neighbourhood around the BMU is decreasing with an **exponential decay function**. It shrinks on each iteration until reaching just the BMU.
Repeat step 2 and step 3 for every row of data i.e: find the winning neuron, find the neighbourhood and update weights.

Exponential decay function

$$\sigma(t) = \sigma_0 \exp(-\frac{t}{\lambda})$$

σ_0 = *the width of lattice at time zero*

t = *the current time step*

λ = *the time constant*

Where t = 0, 1, 2, 3....

Figure shows how the neighbourhood decreases over time after each iteration

Step 4: Adjust the weights

Every node within the BMU's neighbourhood (including the BMU) has its weight vector adjusted according to the following equation:

New Weights = Old Weights + Learning Rate (Input Vector - Old Weights)

W(t+1) = W(t) + L(t) (V(t) - W(t)) where **t** represents the time step and **L** is the learning rate which decreases with time.

The Influence Rate: It shows the amount of influence a node's distance from the BMU has on its learning (used to calculate the radius of the neighbourhood). Influence Rate = 1 for all the nodes close to the BMU and 0 for others.

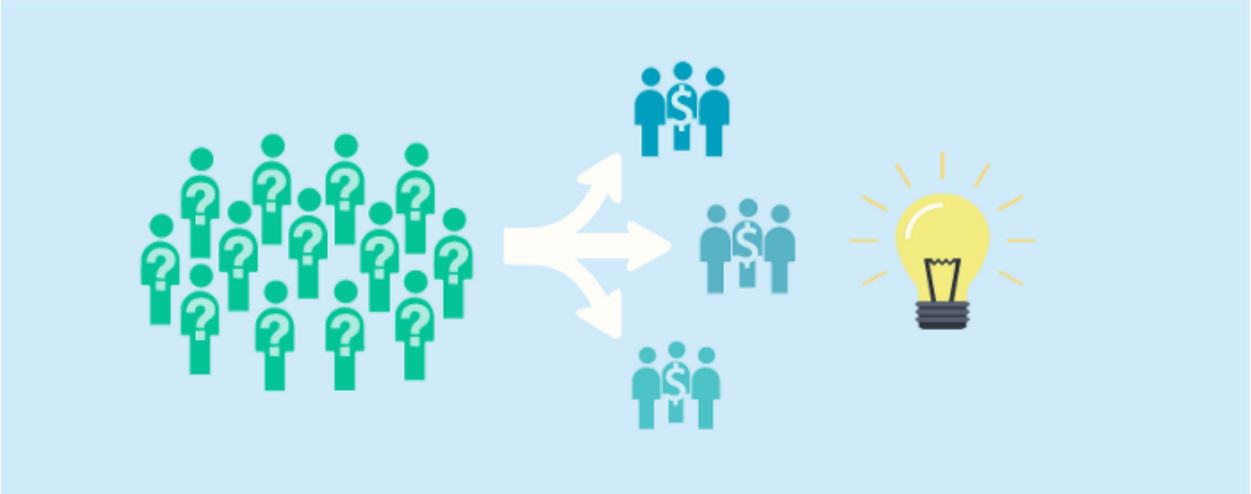
$$\Theta(t) = \exp\left(-\frac{dist^2}{2\sigma^2(t)}\right)$$

$\Theta(t)$ = *Influence rate*

$\sigma(t)$ = *width of the lattice at time t*

Finally, from a random distribution of weights and through many iterations, SOM can arrive at a map of stable zones.

K-means Clustering



What is k-Mean?

K-Means clustering is the most popular unsupervised machine learning algorithm, which is used when you have unlabeled data. The goal of this algorithm is to find groups in the data, with the number of groups represented by the variable K. The algorithm works iteratively to assign each data point to one of K groups based on the features that are provided. Data points are clustered based on feature similarity. The results of the K-means clustering algorithm are:

- The centroids of the K clusters, which can be used to label new data
- Labels for the training data (each data point is assigned to a single cluster)

How does K-Means Clustering work?

K-Means clustering can be represented diagrammatically as follows:

k-means clustering tries to group similar kinds of items in the form of clusters. It finds the similarity between the items and groups them into the clusters. K-means clustering algorithm works in four steps:

Step 1 – First, we need to specify the number of clusters, K, need to be generated by this algorithm.
Step 2 – Next, randomly select K data points and assign each data point to a cluster. In simple words, classify the data based on the number of data points.tep 3 – Now, it will compute the cluster centroids.
Step 4 – Next, keep iterating the following until we find optimal centroid which is the assignment of data points to the clusters that are not changing any more:

1. First, the sum of squared error (SSE) between data points and centroids would be computed.
2. Now, we have to assign each data point to the cluster that is closer than the other cluster (centroid).
3. At last compute the centroids for the clusters by taking the average of all data points of that cluster.

Figures 1 to 4 represent a simple explanation of the K-Means algorithm:

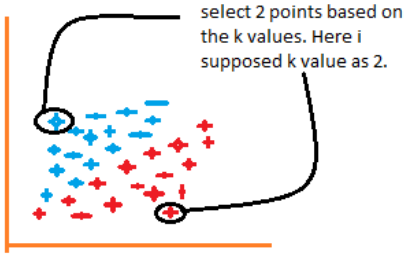


Figure 1

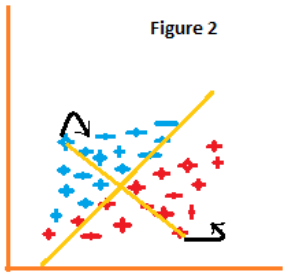
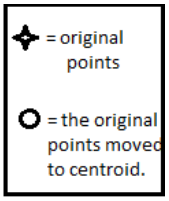
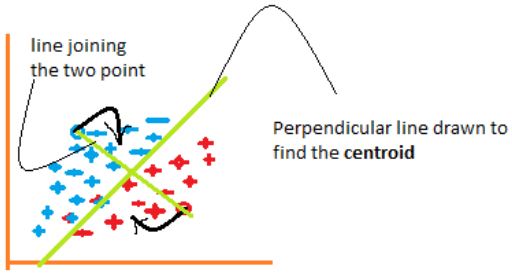


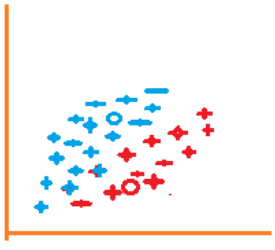
Figure 2



F2: Find the average of all the blue points and red points and move the selected points to **centroid.**

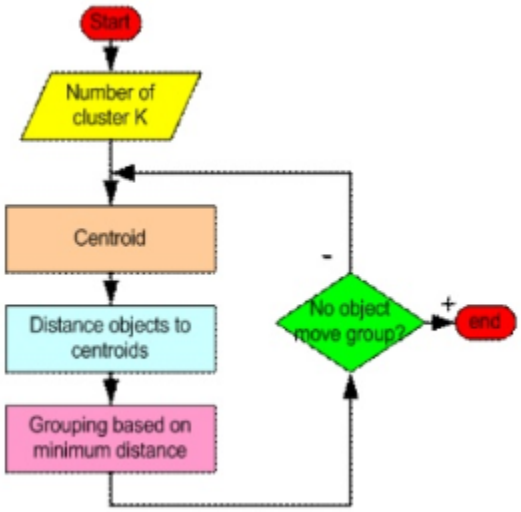


F3: Some of the red points changed to blue points, that means they belong to the group blue now. Again the repeat the same process.



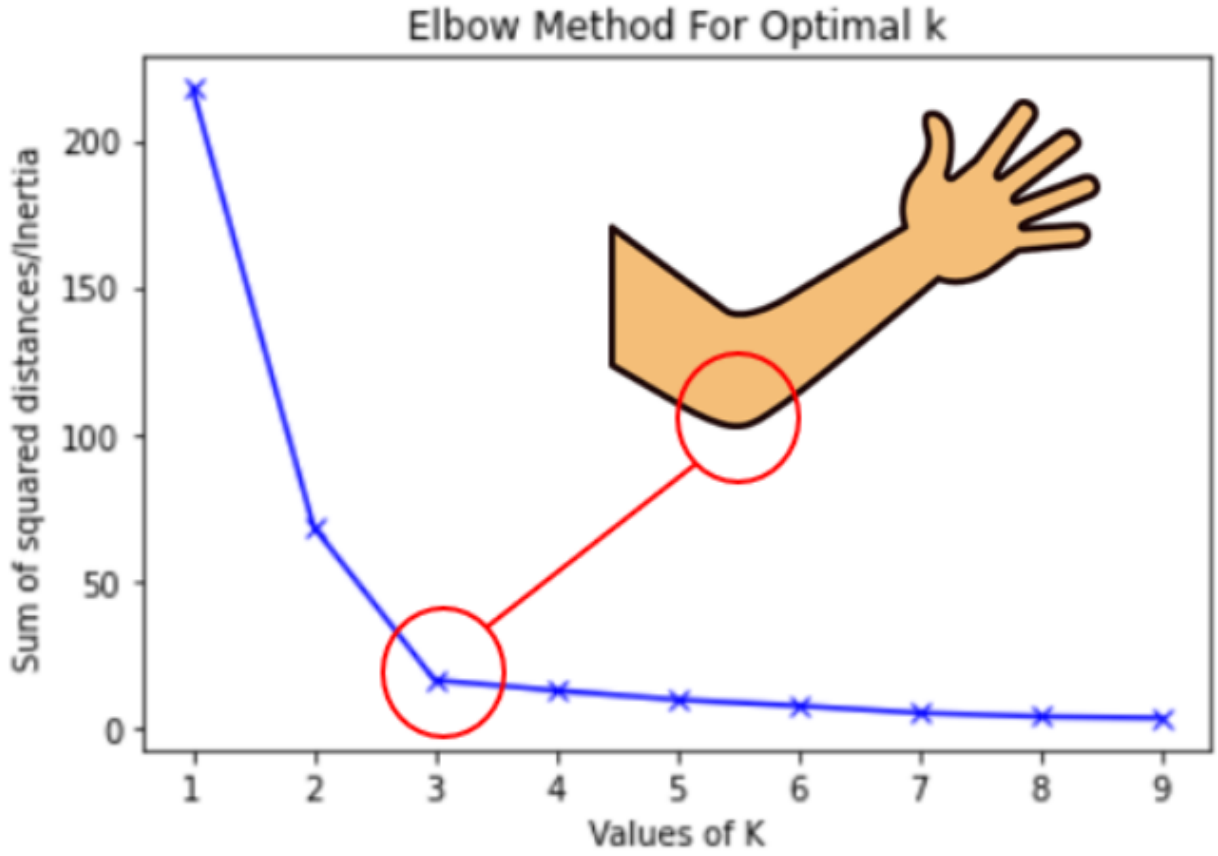
F4: The same process has been applied here. This process will be continued until we get the **two complete different cluster.**

- Figure 1 shows the representation of data of two different items. the first item has shown in blue colour and the second item has shown in red colour. In the first step, the value of K is chosen randomly.
- In figure 2, to find out the centroid, we will draw a perpendicular line to that line. The points will move to their centroid. If you will notice there, then you will see that some of the red points are now moved to the blue points. Now, these points belong to the group of blue colour items.
- The same process will be done in figure 3. The two points are joined and a new perpendicular line is drawn to find the centroid. Again, some of the red points get converted to blue points.
- The same process is happening in figure 4. This process will be continued until and unless we get two completely different clusters of these groups. we will keep iterating the following until we find optimal centroid which is the assignment of data points to the clusters that are not changing any more



How to pick the optimal value of K?

The performance of the K-Means algorithm depends upon the value of K. We should choose the optimal value of K to have the best performance. There are different techniques to find the optimal value of K. The most common technique is the "elbow method" which is described below.



Line plot between K and inertia

Elbow method is an empirical method to find out the best value of k. it picks up a range of values and takes the best among them by running the algorithm multiple times over a loop, with an increasing number of cluster choice and then plotting a clustering score as a function of the number of clusters. The elbow method calculates the sum of the square of the points and calculates the average distance.

Statistical Models

Hidden Markovian Model

The *Hidden Markovian Model* is a model based on a stochastic random process and is useful in case of temporal pattern recognition or whenever time series data is involved (where data changes with time).

Applications

An application of this model could be speech recognition. Another application of this model would be to determine the location of the flight at different time intervals.

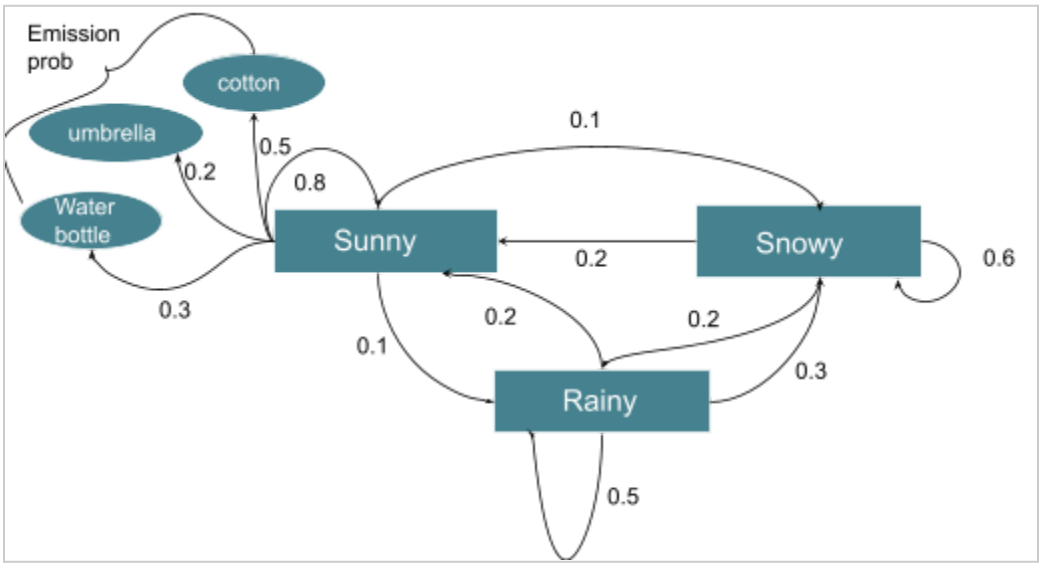
Let's understand the working of the model by taking the example of changes in weather.

Different elements of the HMM Model

Visible/observable states

There are 3 different states: Sunny (s1), rainy (s2) and snowy (s3).

Set of the different states: s={ s1, s2, s3 }



Transitional probability

It is the probability of transition from one state to another. For example: *Given that it is sunny today, what is the probability that it will be rainy tomorrow?*

The summation of all the transition probabilities is equal to 1.

Transition probability of order 1: Determine the future state based on the current state alone.

Transition probability of order 2: Determine the probability from current state as well as previous state.

Transition probability of order 3: Determine the probability from two previous states.

Let's consider transition probability of order 1: An example of this could be *Given that it is sunny today, how likely would it be sunny tomorrow?*

Initial State Probability (π)

It refers to the probabilities of the states at the initial step.

For example: *What is the probability that it is going to be rainy today?*

sunny=0.8, rainy=0.1, snowy=0.1

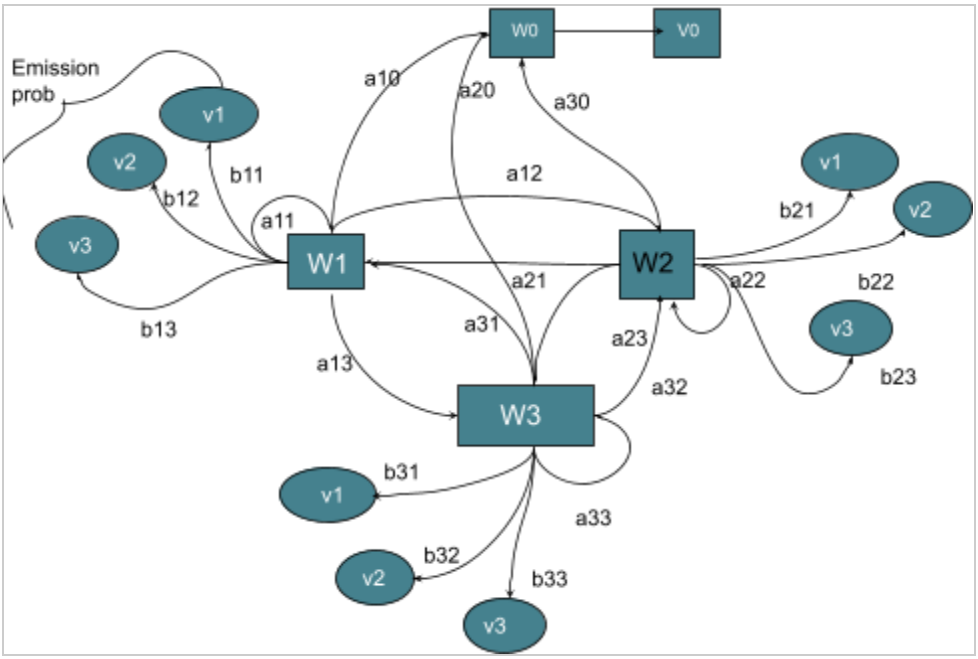
The *Markovian Model* involves observable/visible states whereas *Hidden Markovian Model* includes non-observable states which are also called hidden states.

Emission probabilities

Emission probabilities are the probabilities of emitting an observable state given that the current state is a hidden state.

In the above example of weather, The emission probabilities of people carrying umbrellas and water bottles and wearing cotton clothes are 0.2, 0.3 and 0.5 respectively.

Following is an example of HMM (θ):



- ★ Hidden states/non-observable states: w={w1,w2,w3}
- ★ Visible states v={ v1,v2,v3}
- ★ Transition probabilities (aij)
- ★ Emission probabilities (bjk)
- ★ Final state (w0) - accepting state

- ★ $\sum_j aij = 1, \forall i$

- ★ $\sum_k bjk = 1, \forall j$

Three different issues with Hidden Markovian Model

I. Evaluation problem:

Given the model $\theta = (\omega, v, a_{ij}, b_{jk})$, what is the probability of visible states P (V^t/Θ) , where V^t is a sequence of visible states.

II. Decoding problem (Best state sequence) - Algorithm- Viterbi algorithm:

- Determines what is the sequence of hidden states that generates the visible states.
- *Most likely sequence* ω^t *(set of hidden states to traverse) that leads to generation of sequence of visible states* V^t.
- Deciding the states to visit in order to observe the visible states.

III. Learning problem (re-estimation), algorithm - forward-backward algorithm / Baum Welch algorithm:

Given ω and V, we need to estimate a_{ij} and b_{jk}. It involves learning transition properties and finding out the emission probabilities.

Understanding the evaluation problem with an example:

- ➔ Hidden states: $\omega_0, \omega_1, \omega_2, \omega_3$
- ➔ Visible states: V_0, V_1, V_2, V_3, V_4
- ➔ a_{ij}=

	w0	w1	w2	w3
w0	1	0	0	0
w1	0.2	0.3	0.1	0.4
w2	0.2	0.5	0.2	0.1
w3	0.7	0.1	0.1	0.1

- ➔ $\sum_j aij = 1, \forall i$

	v0	v1	v2	v3	v4
w0	1	0	0	0	0
w1	0	0.3	0.4	0.1	0.2
w2	0	0.1	0.7	0.1	0.1
w3	0	0.5	0.2	0.1	0.3

- ➔ b_{jk}=
- ➔ $\sum_k b_{jk} = 1, \forall j$
- ➔ Given the model $\theta = (\omega, v, a_{ij}, b_{jk})$, what is the probability of generating the visible states P (V^t/Θ) in the sequence V^T={V_1, V_3,V_2,V_0}.

Ways to solve Evaluation problem

1. **Recursive algorithm for solving the evaluation problem using Trellis diagram:**

2. **Backward algorithm ($\beta_j(t)$)**

Recursive algorithm

The *Recursive algorithm* is also known as forward algorithm. The algorithm works as given below:

$\alpha(t) = Probability\ that\ the\ machine(HMM)\ will\ be\ in\ \omega_j\ at\ time\ t\ after\ emitting\ first\ t\ number\ of\ visible\ symbols.$

Assume the sequence of visible states V^T={V_3, V_1, V_2, V_1, V_0}

$\alpha_3(3) = HMM\ reaching\ the\ state\ \omega_3\ after\ generating\ the\ first\ 3\ symbols\ (V_3, V_1, V_2).$

$\alpha_2(4) = HMM\ reaching\ the\ state\ \omega_2\ after\ generating\ the\ first\ 4\ symbols\ (V_3, V_1, V_2, V_1).$

$\alpha_2(4)$ - After generating first 4 symbols, the HMM model reaches state ω_4.

★ $\alpha(t) = \{\ 0\ ,\ t = 0\ and\ j \neq initial\ state$

$1,\ t = 0\ and\ j = initial\ state$

$[\sum_{i=1}^{N} \alpha_i(t-1)\, a_{ij}]\, b_{jk} V(t)$

$\}$

The forward algorithm works as follows:

★ Initialise t=0, a_{ij}, b_{jk}, V^T, $\alpha_j(0)$

★ For t → t+1

$\alpha_j(t)$=$[\sum_{i=1}^{N} \alpha_i(t-1)\, a_{ij}]\, b_{jk} V(t)$ until t=T

Return P (V^t/Θ) for final state $\alpha_0(T)$.

Given the visible state sequence V^T={ V_3, V_1, V_2, V_0 }, we need to determine the probability with which the model θ generates the given visible states P (V^4/Θ).

Consider the trellis diagram given below:

t	0	1	2	3	4
w0	0	0	0	0	**0.00138**
w1	**1**	0.09	0.0052	**0.05592**	0
w2	0	0.01	**0.0217**	0.000543	0
w3	0	**0.2**	0.0057	0.000964	0

P (V^4/Θ) = 0.00138

P (ω^T/V^4) = (ω_3, ω_2, ω_1, ω_0) → best state sequence that needs to be traversed in order to observe visible state sequence.

Viterbi Algorithm for decoding problem:

a_{ij} =

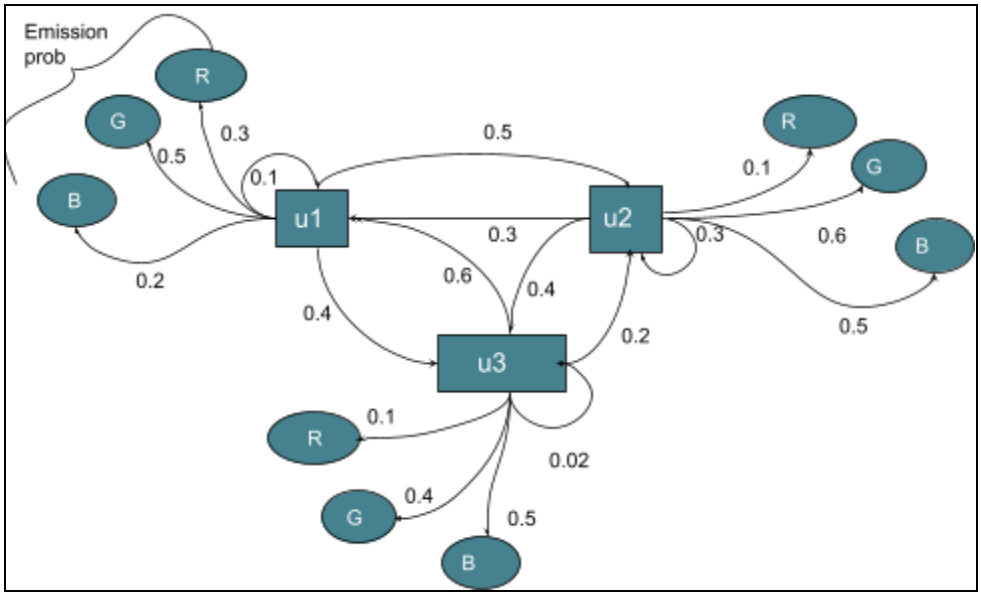
	u1	u2	u3
u1	0.1	0.4	0.5
u2	0.6	0.2	0.2
u3	0.3	0.4	0.3

b_{jk} =

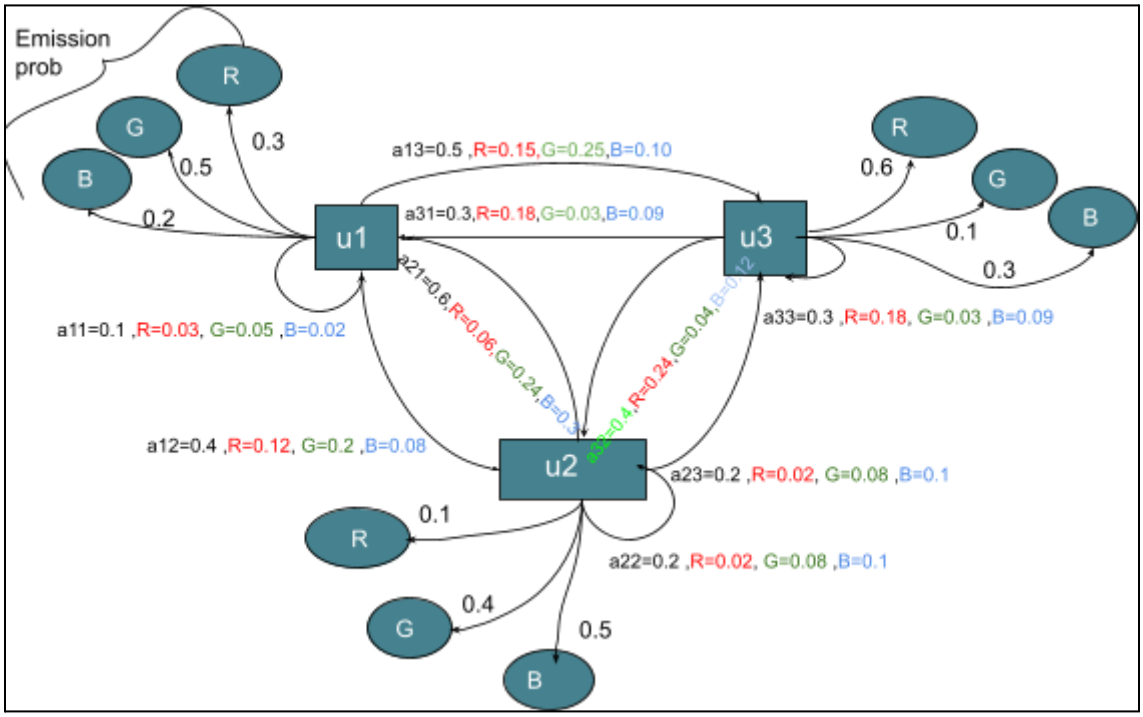
	R	G	B
u1	0.3	0.5	0.2
u2	0.1	0.4	0.5
u3	0.6	0.1	0.3

Observation= V^T= RRG. Decoding problem to determine the best state sequence using viterbi algorithm.

HMM diagram for the above problem:

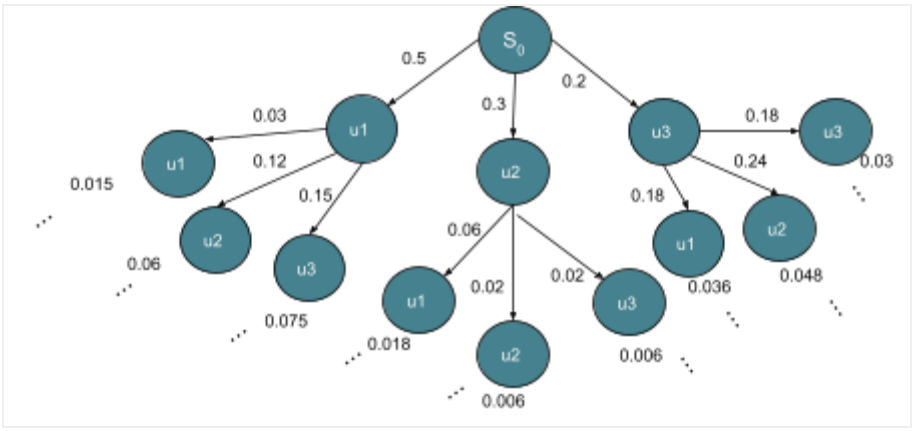


Absorb initial probability into transition probability by multiplying:

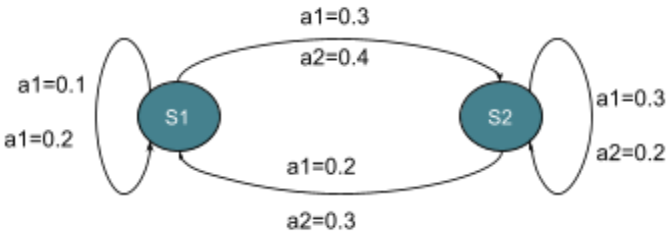


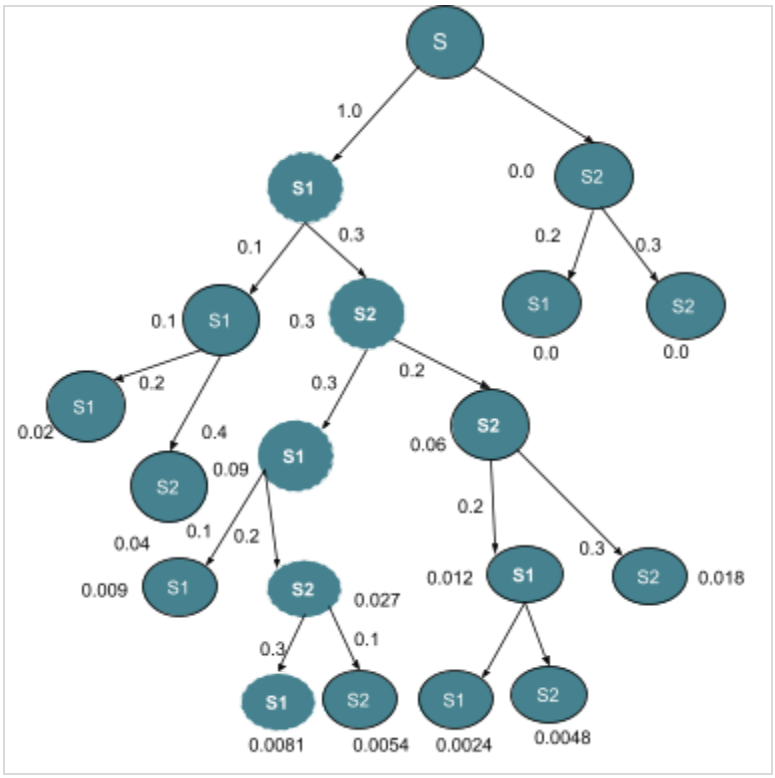
Observation given: RRGGBRGR

We need to find the best hidden state sequence to observe the sequence V^T=RRGGBRGR.



Example





$V^T = V^3$ = a1,a2,a2

Therefore,the best state sequence = S1,S2,S1,S2,S1.

Transformers

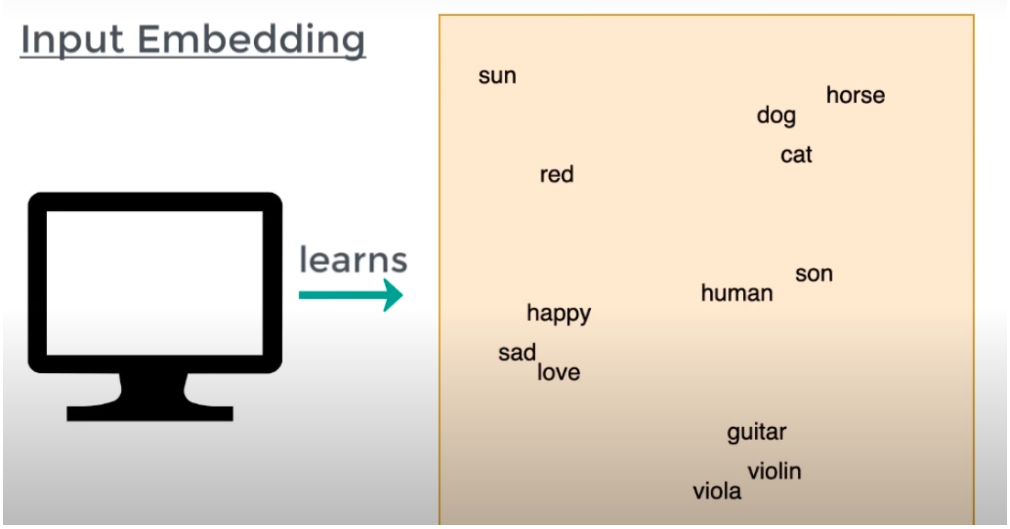
A transformer is a deep learning model that adopts the mechanism of **self-attention**, differentially weighting the significance of each part of the input data. It is used primarily in the field of natural language processing (NLP) and in computer vision (CV).

Like recurrent neural networks (RNNs), transformers are designed to handle sequential input data, such as natural language, for tasks such as translation and text summarization. However, unlike RNNs, transformers do not necessarily process the data in order. Rather, the attention mechanism provides context for any position in the input sequence. This feature allows for more **parallelization** than RNNs and therefore reduces training times.

Transformer Components:

Input embedding:

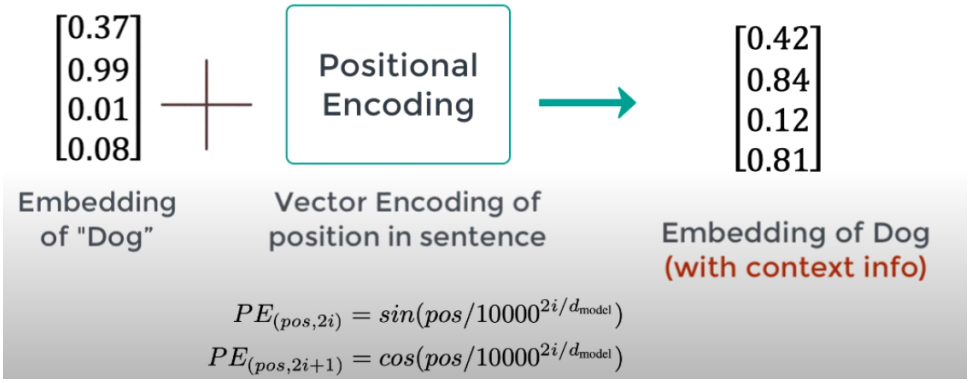
The idea is to map every word to a point in space where similar words in meaning are physically closer to each other.



The space in which they are present is called the embedding space. We can pre-train this embedding space or else use an already pre-trained embedding space like GloVe.

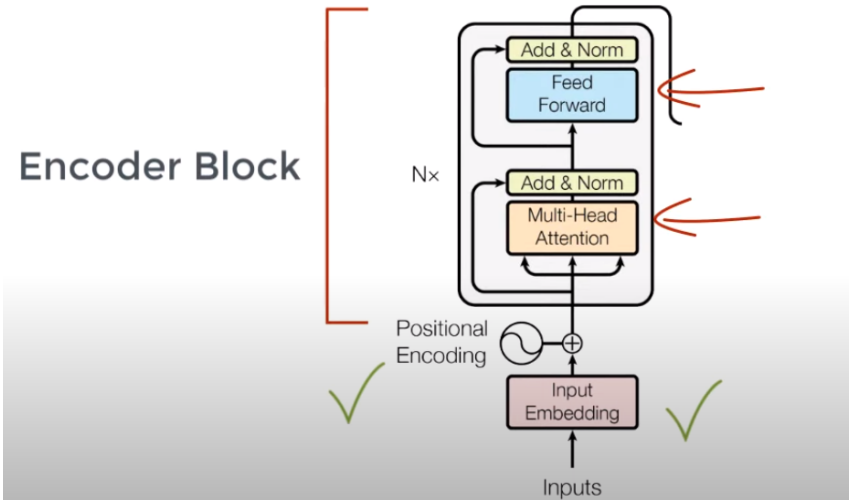
Positional Encoders - It's a vector that has information on distances between words and the sentence. Vector that gives context based on the position of the word in a sentence.

In this discussion we are gonna consider a real life example of Machine Translation of English to Spanish. After passing the English sentence through the input embedding and applying the positional encoding we get word vectors that have positional information (context).



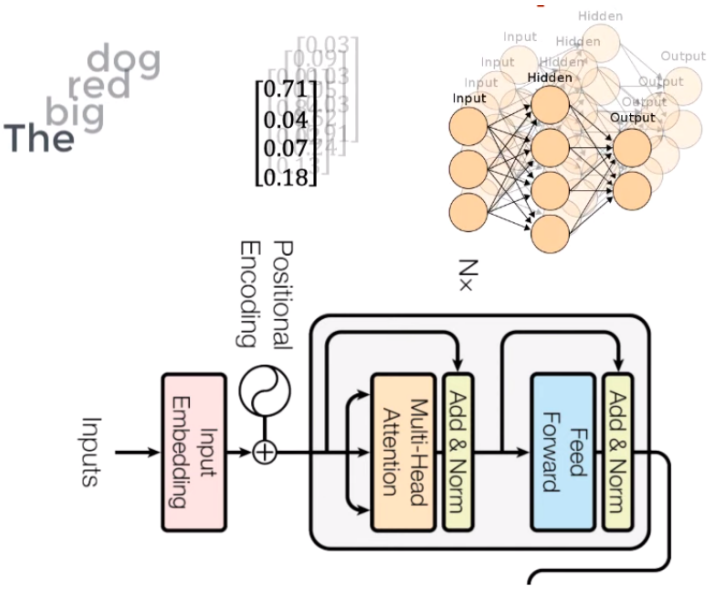
We pass this into the encoder block where it goes through a multi-headed attention layer and a feed forward layer.

Encoder Block:

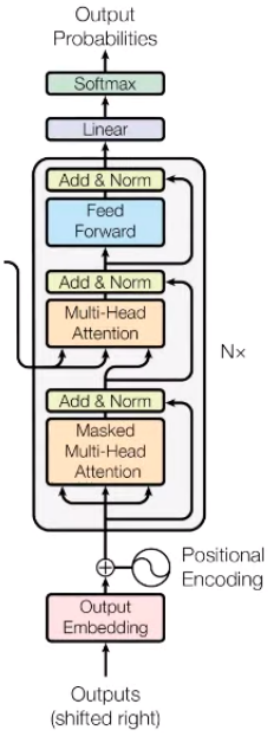


Attention – answers what part of input we should focus on. Considering the translation of English to Spanish using self attention, we need to answer how relevant is the i^{th} word in an English sentence relevant to other words in the same English sentence. This is represented in the i^{th} attention vector and it is computed in the attention block. For every word we can have an attention vector generated which captures contextual relationships between words in the sentence.

Feed Forward Network: Simple feed forward neural network that is applied to every one of the attention vectors. These feed forward networks are used in practice to transform the attention vectors into the form that is digestible by the next encoder/decoder block.

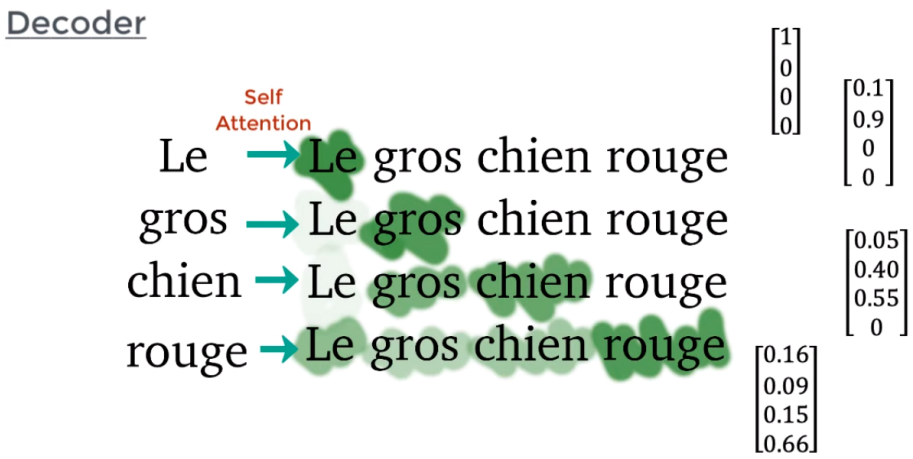


Decoder Block:

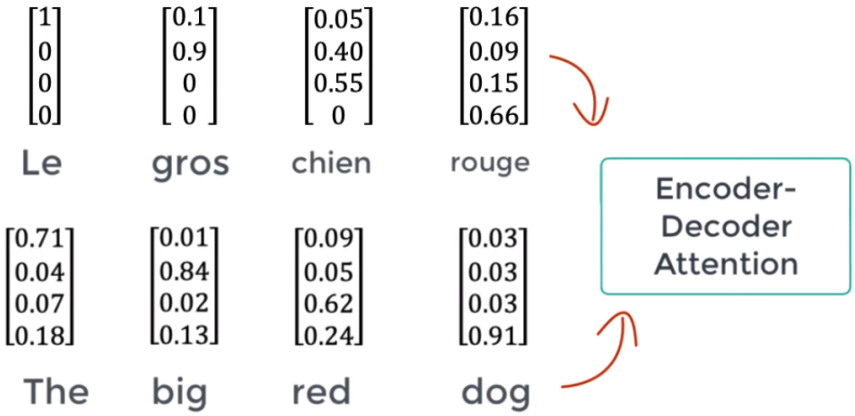


During the training phase for English to Spanish translation, we feed the output Spanish sentence to the decoder and since computers understand numbers/vectors, we process it using the input embedding to get the vector form of the word and then we add a positional vector to get the context of the word in a sentence. Then we pass this vector to the decoder block that has **3 main components:**

Self Attention Block: It generates self attention vectors for every word in the Spanish sentence to represent how much each word is related to every word in the same sentence. (The problem here is that the attention vector may not be too strong. For every word, the attention vector may weigh its relation with itself much higher. So we determine 8 such attention vectors per word and take a weighted average to compute the final attention vector for every word. This is called a **multi-headed attention block**).



These attention vectors and vectors from the encoder are passed into another attention block.



This attention block will determine how related each word vector is with respect to each other and this is where the main English to Spanish word mapping happens.

The output of this block is attention vectors for every word in English and the Spanish sentence. Each vector represents the relationship with other words in both the languages.

Next we pass each attention vector to a feed forward unit, this makes the output vector more digestible by the next decoder block or a linear layer.

Linear Layer: Linear Layer is another feed forward connected layer. It's used to expand the dimensions into the number of words in the Spanish sentence (# Neurons = # words in Spanish).

Softmax Layer: Softmax layer transforms it into a probability distribution which is now human interpretable and the final word is the word corresponding to the highest probability. Overall this decoder predicts the next word.

And we execute this over multiple time steps until the end of sentence token is generated.