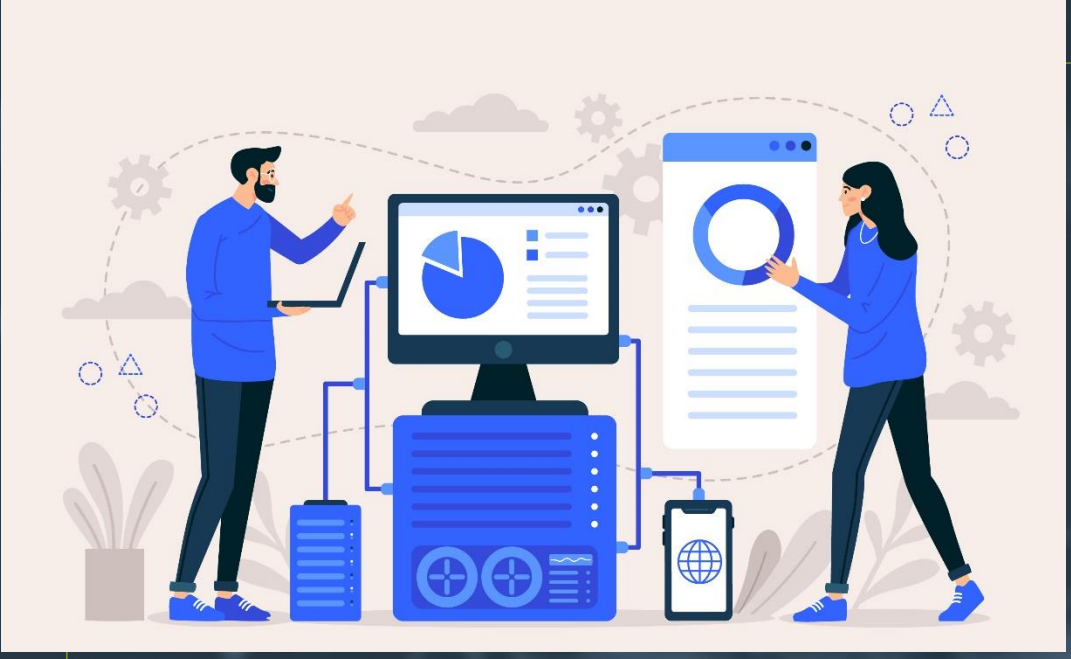
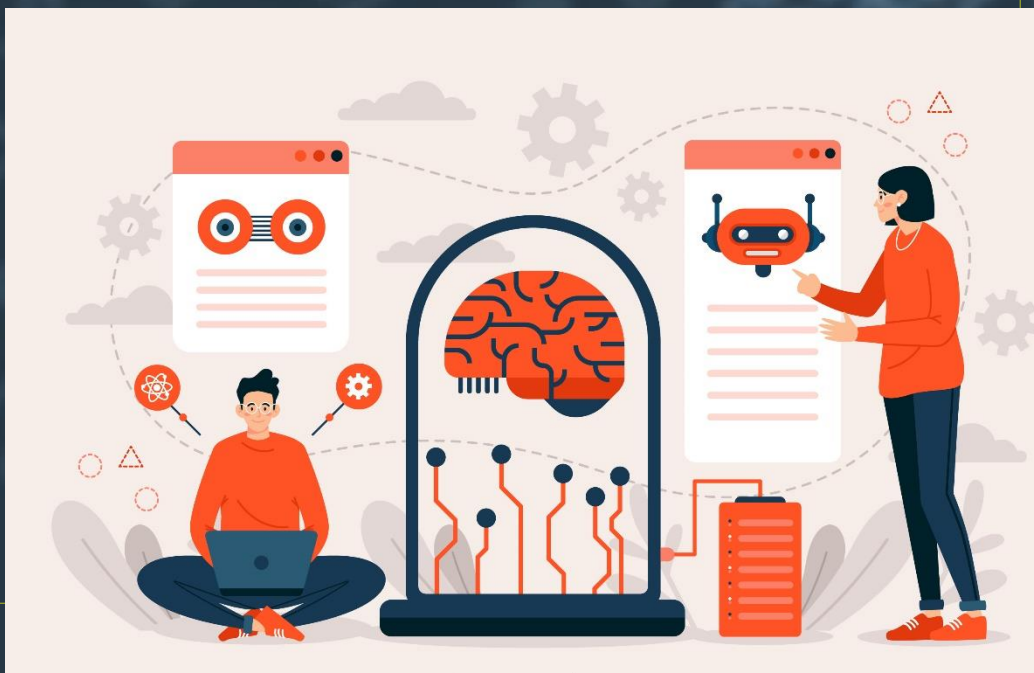


MACHINE LEARNING BOOK WRITING



ML BOOK WRITING

By

KAMLESH NAIK	**1937**
NEHAL PARSEKAR	**1942**
KOKILA PILLAI	**1946**
MRUDUL SHIRODKAR	**1955**
ROCHELLE VAZ	**1960**

under the guidance of

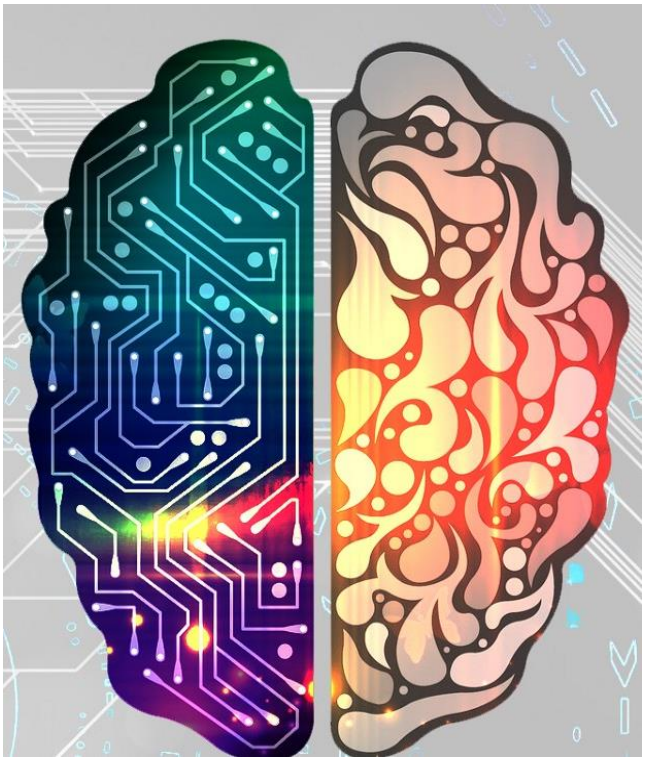
Prof. BASKAR SUNDARRAJAN

Department of Computer Science

GOA UNIVERSITY

Taleigao Plateau

INDEX



MACHINE LEARNING

Introduction

We have seen Machine Learning as a buzzword for the past few years, the reason for this might be the high amount of data production by applications, the increase of computation power in the past few years and the development of better algorithms.

Machine Learning is used anywhere from automating mundane tasks to offering intelligent insights, industries in every sector try to benefit from it. You may already be using a device that utilizes it. For example, a wearable fitness tracker like Fitbit, or an intelligent home assistant like Google Home. But there are much more examples of ML in use.

Prediction: Machine learning can also be used in the prediction systems. Considering the loan example, to compute the probability of a fault, the system will need to classify the available data in groups.

Image recognition: Machine learning can be used for face detection in an image as well. There is a separate category for each person in a database of several people.

Speech Recognition: It is the translation of spoken words into the text. It is used in voice searches and more. Voice user interfaces include voice dialing, call routing, and appliance control. It can also be used a simple data entry and the preparation of structured documents.

Medical diagnoses: ML is trained to recognize cancerous tissues.

Financial industry and trading: Companies use ML in fraud investigations and credit checks.

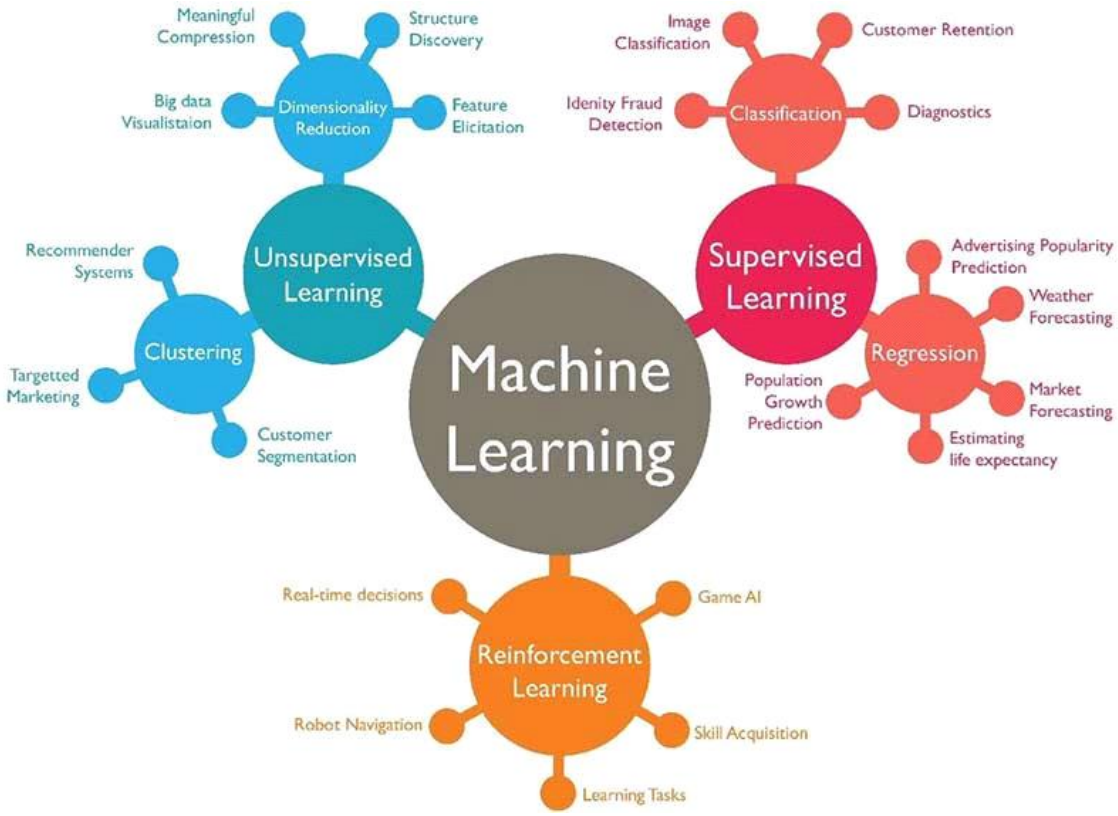
What is Machine Learning?

According to Arthur Samuel, Machine Learning algorithms enable the computers to learn from data, and even improve themselves, without being explicitly programmed.

Machine learning (ML) is a category of an algorithm that allows software applications to become more accurate in predicting outcomes without being explicitly programmed. The basic premise of machine learning is to build algorithms that can receive input data and use statistical analysis to predict an output while updating outputs as new data becomes available.

Types of Machine Learning

- Machine learning can be classified into 3 types of algorithms.
- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning



How does supervised machine learning work?

Supervised machine learning requires the data scientist to train the algorithm with both labeled inputs and desired outputs. Supervised learning algorithms are good for the following tasks:

Binary classification: Dividing data into two categories.

Multi-class classification: Choosing between more than two types of answers.

Regression modeling: Predicting continuous values.

Ensembling: Combining the predictions of multiple machine learning models to produce an accurate prediction.

How does unsupervised machine learning work?

Unsupervised machine learning algorithms do not require data to be labeled. They sift through unlabeled data to look for patterns that can be used to group data points into subsets. Most types of deep learning, including neural networks, are unsupervised algorithms. Unsupervised learning algorithms are good for the following tasks:

Clustering: Splitting the dataset into groups based on similarity.

Anomaly detection: Identifying unusual data points in a data set.

Association mining: Identifying sets of items in a data set that frequently occur together.

Dimensionality reduction: Reducing the number of variables in a data set.

How does semi-supervised learning work?

Semi-supervised learning works by data scientists feeding a small amount of labeled training data to an algorithm. From this, the algorithm learns the dimensions of the data set, which it can then apply to new, unlabeled data. The performance of algorithms typically improves when they train on labeled data sets. But labeling data can be time consuming and expensive. Semi-supervised learning strikes a middle ground between the performance of supervised learning and the efficiency of unsupervised learning. Some areas where semi-supervised learning is used include:

Machine translation: Teaching algorithms to translate language based on less than a full dictionary of words.

Fraud detection: Identifying cases of fraud when you only have a few positive examples.

Labelling data: Algorithms trained on small data sets can learn to apply data labels to larger sets automatically.

Introduction to Artificial Intelligence

Artificial Intelligence is an approach to make a computer, a robot, or a product to think how smart human think. AI is a study of how human brain think, learn, decide and work, when it tries to solve problems. And finally this study outputs intelligent software systems. The aim of AI is to improve computer functions which are related to human knowledge, for example, reasoning, learning, and problem-solving.

The intelligence is intangible. It is composed of:

- Reasoning
- Learning
- Problem Solving
- Perception
- Linguistic Intelligence

The objectives of AI research are reasoning, knowledge representation, planning, learning, natural language processing, realization, and ability to move and manipulate objects. There are long-term goals in the general intelligence sector.

Approaches include statistical methods, computational intelligence, and traditional coding AI. During the AI research related to search and mathematical optimization, artificial neural networks and methods based on statistics, probability, and economics, we use many tools. Computer science attracts AI in the field of science, mathematics, psychology, linguistics, philosophy and so on.

Applications of AI

Gaming: AI plays important role for machine to think of large number of possible positions based on deep knowledge in strategic games. for example, chess, river crossing, N-queens problems and etc.

Natural Language Processing: Interact with the computer that understands natural language spoken by humans.

Expert Systems: Machine or software provide explanation and advice to the users.

Vision Systems: Systems understand, explain, and describe visual input on the computer.

Speech Recognition: There are some AI based speech recognition systems have ability to hear and express as sentences and understand their meanings while a person talks to it. For example, Siri and Google assistant.

Handwriting Recognition: The handwriting recognition software reads the text written on paper and recognize the shapes of the letters and convert it into editable text.

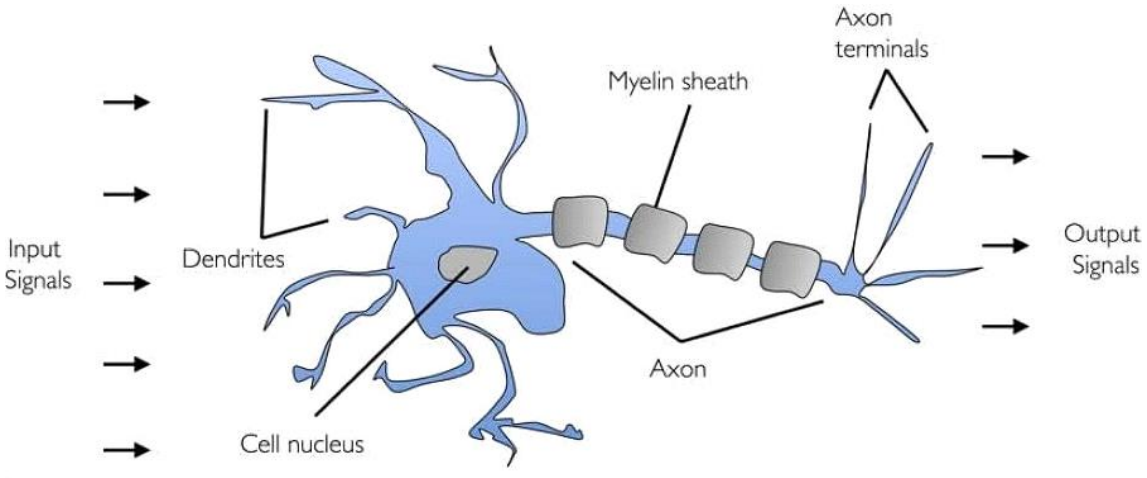
Intelligent Robots: Robots are able to perform the instructions given by a human.

What is Perceptron

A perceptron is a neural network unit (an artificial neuron) that does certain computations to detect features or business intelligence in the input data. And this perceptron tutorial will give you an in-depth knowledge of Perceptron and its activation functions.

Biological Neuron

A human brain has billions of neurons. Neurons are interconnected nerve cells in the human brain that are involved in processing and transmitting chemical and electrical signals. Dendrites are branches that receive information from other neurons.



Cell nucleus or Soma processes the information received from dendrites. Axon is a cable that is used by neurons to send information. Synapse is the connection between an axon and other neuron dendrites.

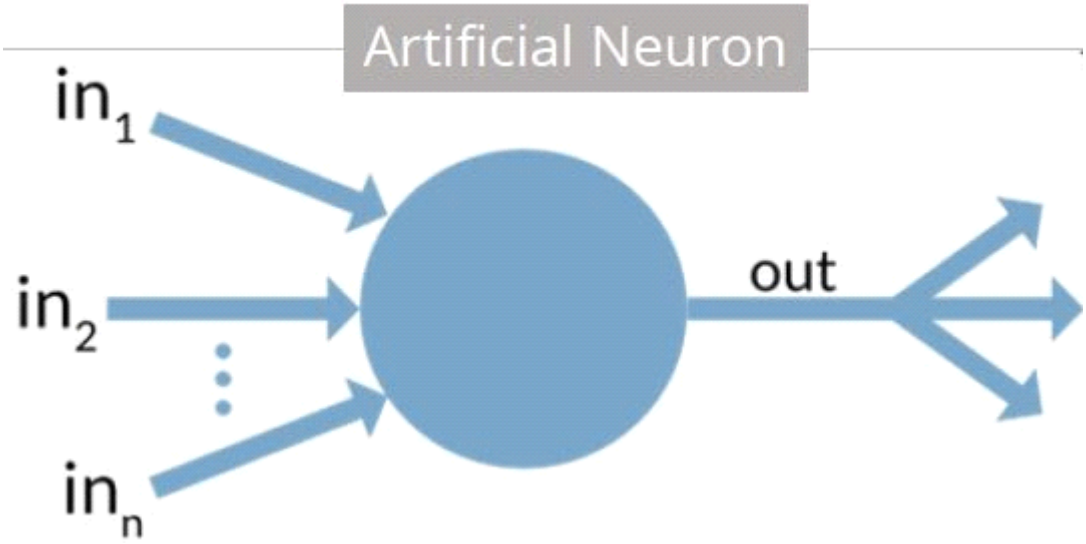
Rise of Artificial Neurons

Researchers Warren McCullock and Walter Pitts published their first concept of simplified brain cell in 1943. This was called McCullock-Pitts (MCP) neuron. They described such a nerve cell as a simple logic gate with binary outputs.

Multiple signals arrive at the dendrites and are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon. In the next section, let us talk about the artificial neuron.

What is Artificial Neuron

An artificial neuron is a mathematical function based on a model of biological neurons, where each neuron takes inputs, weighs them separately, sums them up and passes this sum through a nonlinear function to produce output.



Biological Neuron vs. Artificial Neuron

The biological neuron is analogous to artificial neurons in the following terms:

Biological Neuron	Artificial Neuron
Cell Nucleus (Soma)	Node
Dendrites	Input
Synapse	Weights or interconnections
Axon	Output

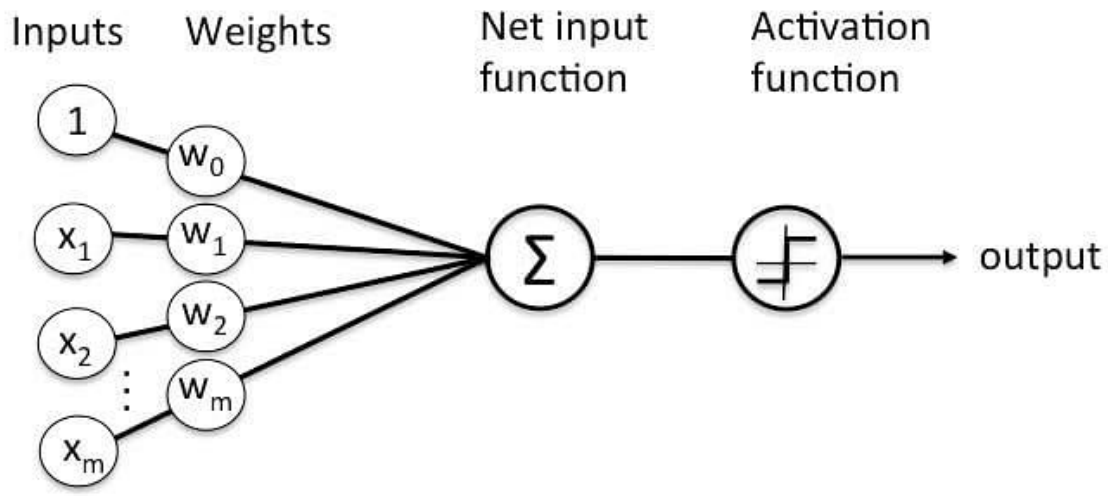
Artificial Neuron

The artificial neuron has the following characteristics:

- A neuron is a mathematical function modeled on the working of biological neurons
- It is an elementary unit in an artificial neural network
- One or more inputs are separately weighted
- Inputs are summed and passed through a nonlinear function to produce output
- Every neuron holds an internal state called activation signal
- Each connection link carries information about the input signal
- Every neuron is connected to another neuron via connection link

Perceptron

Perceptron was introduced by Frank Rosenblatt in 1957. He proposed a Perceptron learning rule based on the original MCP neuron. A Perceptron is an algorithm for supervised learning of binary classifiers. This algorithm enables neurons to learn and processes elements in the training set one at a time.



Types of Perceptron

Single layer: Single layer perceptron can learn only linearly separable patterns

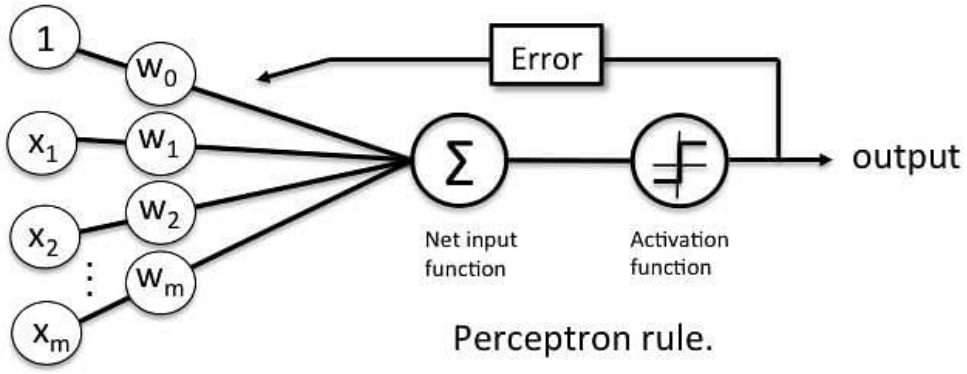
Multilayer: Multilayer perceptron or feedforward neural networks with two or more layers have the greater processing power

The Perceptron algorithm learns the weights for the input signals in order to draw a linear decision boundary.

This enables you to distinguish between the two linearly separable classes +1 and -1.

Perceptron Learning Rule

Perceptron Learning Rule states that the algorithm would automatically learn the optimal weight coefficients. The input features are then multiplied with these weights to determine if a neuron fires or not.



The Perceptron receives multiple input signals, and if the sum of the input signals exceeds a certain threshold, it either outputs a signal or does not return an output. In the context of supervised learning and classification, this can then be used to predict the class of a sample.

Perceptron Function

Perceptron is a function that maps its input "x," which is multiplied with the learned weight coefficient; an output value "f(x)"is generated.

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

In the equation given above:

"w" = vector of real-valued weights

"b" = bias (an element that adjusts the boundary away from origin without any dependence on the input value)

"x" = vector of input x values

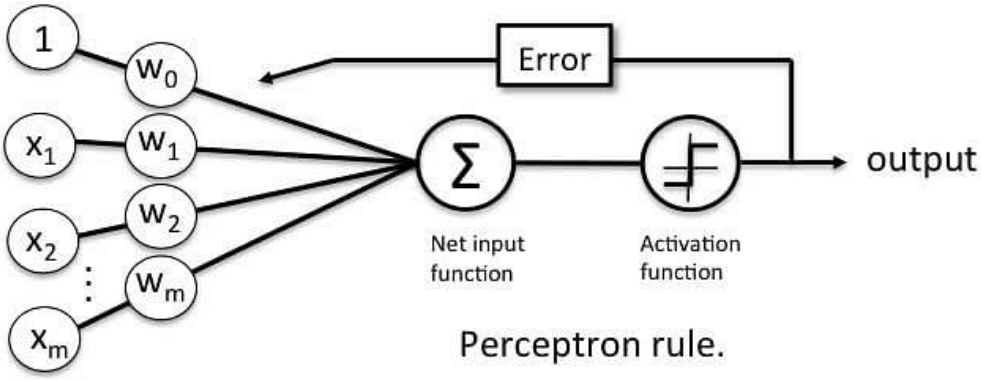
$$\sum_{i=1}^{m} w_i x_i$$

"m" = number of inputs to the Perceptron

The output can be represented as "1" or "0." It can also be represented as "1" or "-1" depending on which activation function is used.

Inputs of a Perceptron

A Perceptron accepts inputs, moderates them with certain weight values, then applies the transformation function to output the final result. The image below shows a Perceptron with a Boolean output.

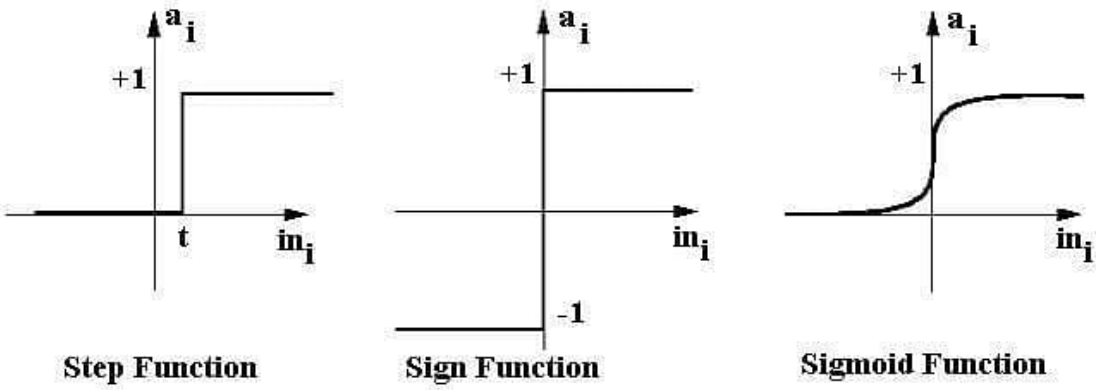


A Boolean output is based on inputs such as salaried, married, age, past credit profile, etc. It has only two values: Yes and No or True and False. The summation function "∑" multiplies all inputs of "x" by weights "w" and then adds them up as follows:

$$w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

Activation Functions of Perceptron

The activation function applies a step rule (convert the numerical output into +1 or -1) to check if the output of the weighting function is greater than zero or not.



for example:

If ∑ wixi> 0 => then final output "o" = 1 (issue bank loan)

Else, final output "o" = -1 (deny bank loan)

Step function gets triggered above a certain value of the neuron output; else it outputs zero. Sign Function outputs +1 or -1 depending on whether neuron output is greater than zero or not. Sigmoid is the S-curve and outputs a value between 0 and 1.

Output of Perceptron

Perceptron with a Boolean output:

Inputs: x1...xn

Output: o(x1....xn)

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

Weights: wi=> contribution of input xi to the Perceptron output;

w0=> bias or threshold

If ∑w.x > 0, output is +1, else -1. The neuron gets triggered only when weighted input reaches a certain threshold value.

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

An output of +1 specifies that the neuron is triggered. An output of -1 specifies that the neuron did not get triggered.

"sgn" stands for sign function with output +1 or -1.

Error in Perceptron

In the Perceptron Learning Rule, the predicted output is compared with the known output. If it does not match, the error is propagated backward to allow weight adjustment to happen.

Perceptron: Decision Function

A decision function ϕ(z) of Perceptron is defined to take a linear combination of x and w vectors.

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

The value z in the decision function is given by:

$$z = w_1 x_1 + \cdots + w_m x_m$$

The decision function is +1 if z is greater than a threshold θ, and it is -1 otherwise.

$$\phi(z) = \begin{cases} 1 & if\ z \geq \theta \\ -1 & otherwise \end{cases}$$

This is the Perceptron algorithm.

Bias Unit

For simplicity, the threshold θ can be brought to the left and represented as w0x0, where w0= -θ and x0= 1.

$$z = w_0 x_0 + w_1 x_1 + \ldots + w_m x_m = \boldsymbol{w}^T \boldsymbol{x}$$

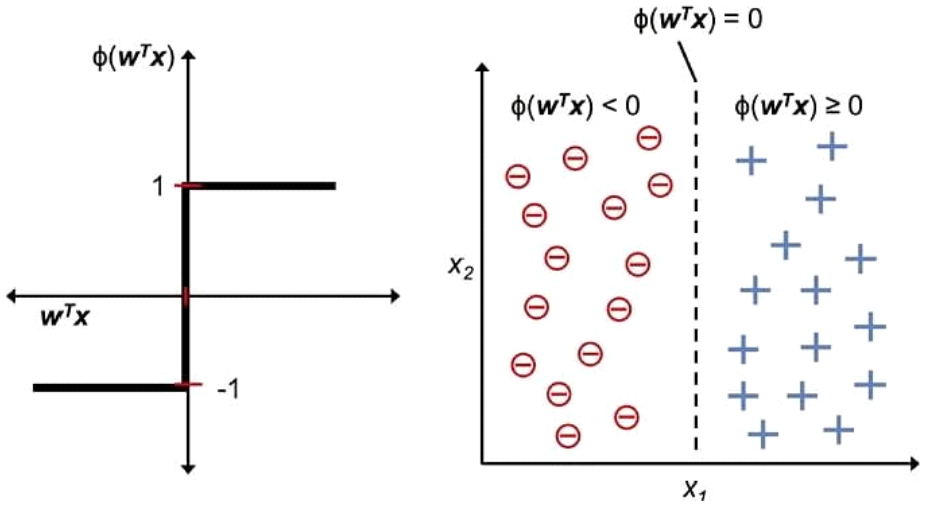
The value w0 is called the bias unit.

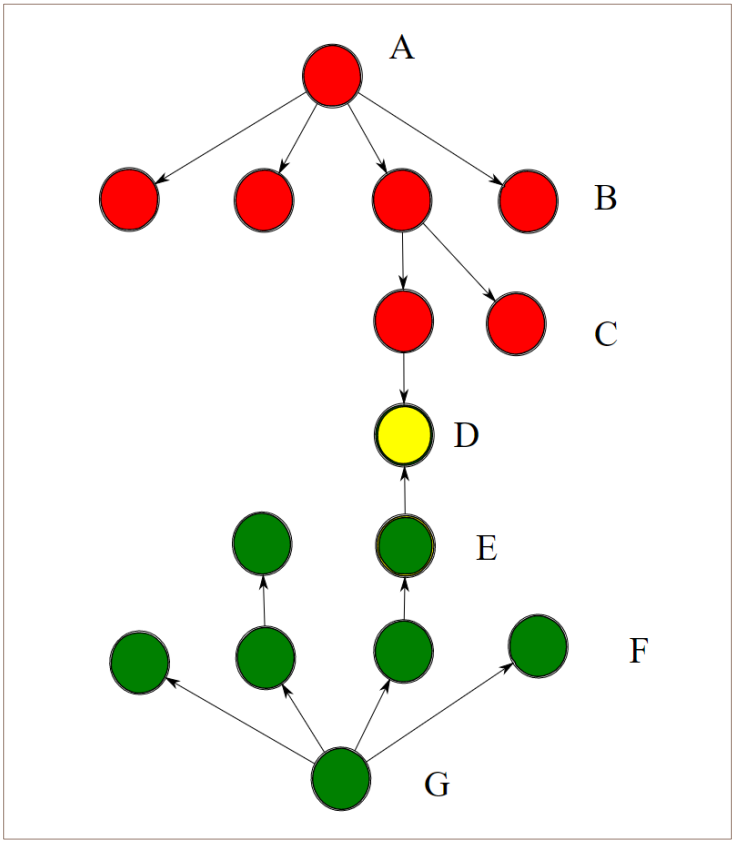
The decision function then becomes:

$$\phi(z) = \begin{cases} 1 & if\ z \geq 0 \\ -1 & otherwise \end{cases}$$

Output:

The figure shows how the decision function squashes wTx to either +1 or -1 and how it can be used to discriminate between two linearly separable classes.





VERSION SPACE

Introduction

A **version space** is a hierarchical representation of knowledge that enables you to keep track of all the useful information supplied by a sequence of learning examples without remembering any of the examples.

The **version space method** is a concept learning process accomplished by managing multiple models within a version space.

Version Space Characteristics

Tentative heuristics are represented using version spaces.

A version space represents all the alternative plausible **descriptions** of a heuristic. A plausible description is one that is applicable to all known positive examples and no known negative example.

A version space description consists of two complementary trees:

1. One that contains nodes connected to overly **general** models, and
2. One that contains nodes connected to overly **specific** models.

Node values/attributes are **discrete**.

Fundamental Assumptions

1. The data is correct; there are no erroneous instances.
2. A correct description is a conjunction of some of the attributes with values.

Diagrammatical Guidelines

There is a **generalization** tree and a **specialization** tree.

Each **node** is connected to a **model**.

Nodes in the generalization tree are connected to a model that matches everything in its subtree.

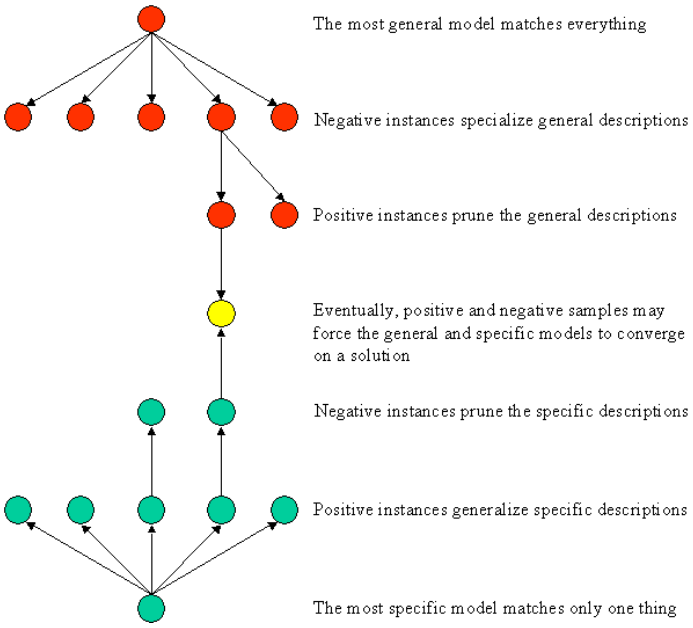
Nodes in the specialization tree are connected to a model that matches only one thing in its subtree.

Links between nodes and their models denote

- generalization relations in a generalization tree, and
- specialization relations in a specialization tree.

Diagram of a Version Space

In the diagram below, the specialization tree is colored **red**, and the generalization tree is colored **green**.



Generalization and Specialization Leads to Version Space Convergence

The key idea in version space learning is that specialization of the general models and generalization of the specific models may ultimately lead to just one correct model that matches all observed positive examples and does not match any negative examples.

That is, each time a negative example is used to specialilize the general models, those specific models that match the negative example are eliminated and each time a positive example is used to generalize the specific models, those general models that fail to match the positive example are eliminated. Eventually, the positive and negative examples may be such that only one general model and one identical specific model survive.

Version Space Method Learning Algorithm: Candidate-Elimination

Let's say that we have the weather data on the days our friend Aldo enjoys (or doesn't enjoy) his favorite water sport:

Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
Sunny	Warm	Normal	Strong	Warm	Same	Yes
Sunny	Warm	High	Strong	Warm	Same	Yes
Rainy	Cold	High	Strong	Warm	Change	No
Sunny	Warm	High	Strong	Cool	Change	Yes

We want a boolean function that would be *true* for all the examples for which *Enjoy*= Yes. Each boolean function defined over the space of the tuples of <*Sky, AirTemp, Humidity, Wind, Water, forecast*> is a candidate hypothesis for our target concept "days when Aldo enjoys his favorite water sport". Since that's an infinite space to search, we restrict it by choosing an appropriate hypothesis representation.

For example, we may consider only the hypotheses that are conjunctions of the terms A = v, where A € {Sky, AirTemp, Humidity, Wind, Water, Forecast} and v is:

- a single value A can take
- or one of two special symbols:
  - ?, which means that any admissible value of A is acceptable
  - ∅, which means that A should have no value

For instance, the hypothesis $(Sky = Rainy) \wedge (AirTemp = Normal)$ can be represented as follows:

$$< Rainy, Normal, ?, ?, ?? >$$

and would correspond to the boolean function:

$$f(x) = \begin{cases} true, & (x.Sky = Rainy) \wedge (x.AirTemp = Normal) \\ false, & \text{otherwise} \end{cases}$$

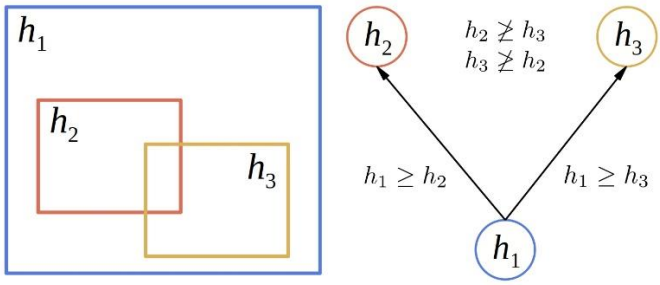
We call the hypothesis space **H** the set of all candidate hypotheses that the chosen representation can express.

There may be multiple hypotheses that fully capture the positive objects in our data. But, we're interested in those also consistent with the negative ones. All the functions consistent with positive and negative objects (which means they classify them correctly) constitute the version space of the target concept. The Candidate Elimination Algorithm (CEA) relies on a partial ordering of hypotheses to find the version space.

The ordering CEA uses is induced by relation "more general than or equally general as", which we'll denote as ≥. We say that hypothesis h_1 is more general than or equally general as hypothesis h_2 if all the examples h_2 labels as positive are also classified as such by h_1:

$$h_1 \geq h2 \equiv (\forall x)(h_2(x) = true \implies h_1(x) = true)$$

This relation has a nice visual interpretation:



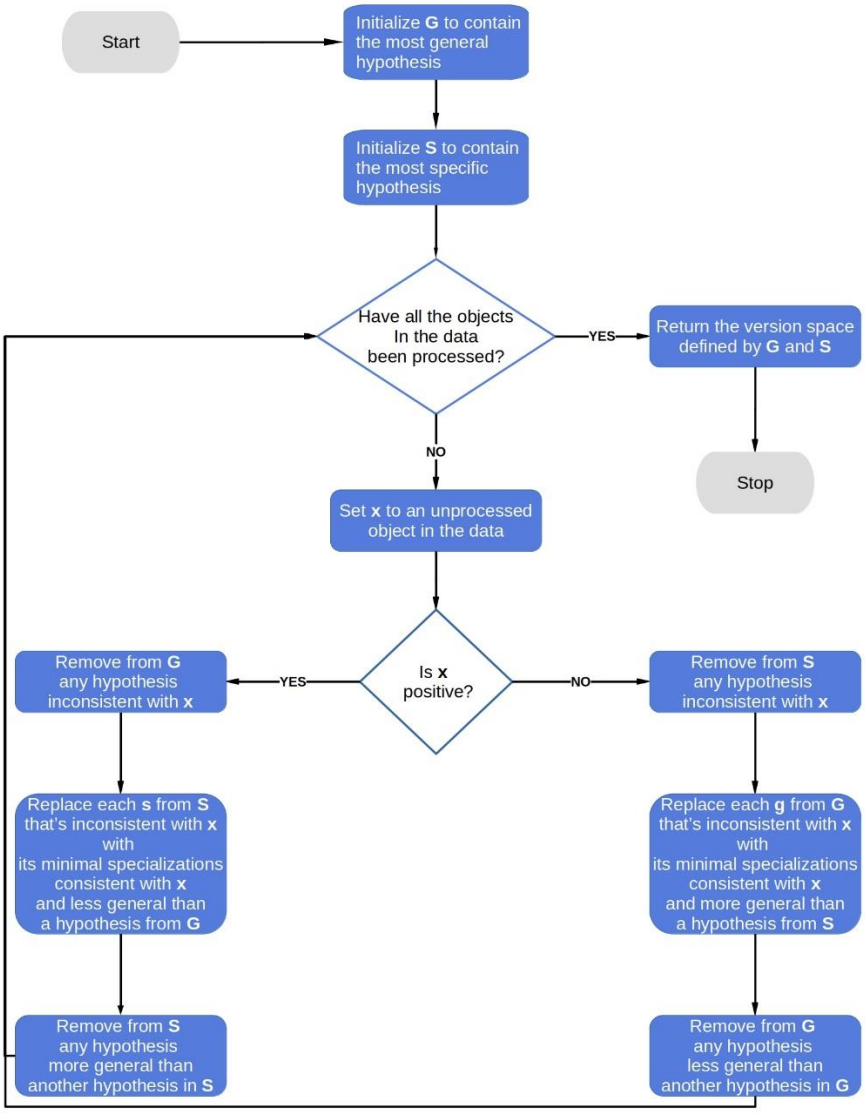
The rectangles are sets of objects that are classified as positive by these three hypotheses.

The subset relation between the rectangles determines the ≥ ordering between the hypotheses.

The most general hypothesis is the one that labels all examples as positive. Similarly, the most specific is the one that always returns false. The accompanying relations: > (strictly more general), < (strictly less general, strictly more specific), and ≤ are analogously defined. For brevity, we'll refer to ≥ as "more general than" from now on.

CEA finds the version space V by identifying its general and specific boundaries, G and S. G contains the most general hypotheses in V, whereas the most specific ones are in S. We can obtain all the hypotheses in V by specializing the ones from G until we reach S or by generalizing those from S until we get G.

Let's denote as V(G, S) the version space whose general and specific boundaries are G and S. At the start of CEA, G contains only the most general hypothesis in H. Similarly, S contains only the most specific hypothesis in H. CEA processes the labeled objects one by one. If necessary, it specializes G and generalizes S so that the V(G, S) correctly classifies all the objects processed so far:



Comments on the Version Space Method

The version space method is still a trial and error method. The program does not base its choice of examples, or its learned heuristics, on an *analysis* of what works or why it works, but rather on the simple assumption that what works will probably work again.

Unlike the decision tree ID3 algorithm,

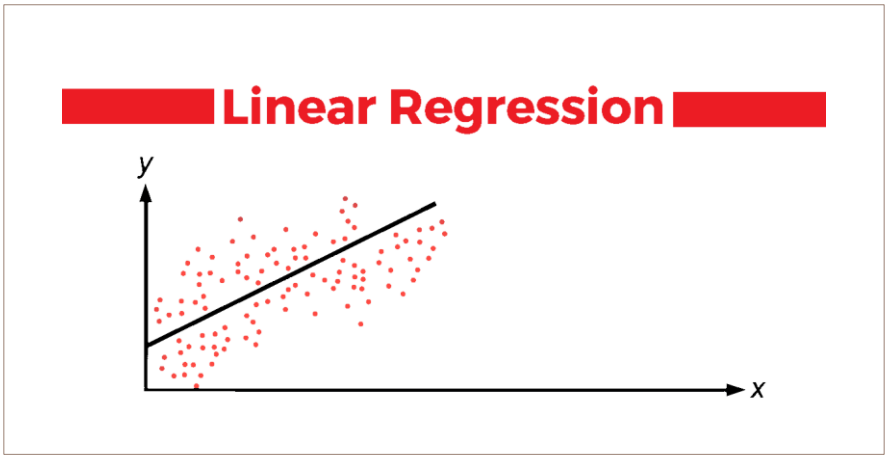
- Candidate-elimination searches an *incomplete* set of hypotheses (ie. only a subset of the potentially teachable concepts are included in the hypothesis space).
- Candidate-elimination finds every hypothesis that is consistent with the training data, meaning it searches the hypothesis space *completely*.
- Candidate-elimination's inductive bias is a consequence of how well it can represent the subset of possible hypotheses it will search. In other words, the bias is a product of its *search space*.
- No additional bias is introduced through Candidate-eliminations' *search strategy*.

Advantages of the version space method:

- Can describe all the possible hypotheses in the language consistent with the data.
- Fast (close to linear).

Disadvantages of the version space method:

- Inconsistent data (noise) may cause the target concept to be pruned.
- Learning disjunctive concepts is challenging.



LINEAR REGRESSION

Introduction

Regression is a method of modelling a target value based on independent predictors. This method is mostly used for forecasting and finding out cause and effect relationship between variables. Regression techniques mostly differ based on the number of independent variables and the type of relationship between the independent and dependent variables.

Simple linear regression is a type of regression analysis where the number of independent variables is one and there is a linear relationship between the independent(x) and dependent(y) variable. The red line in the above graph is referred to as the best fit straight line. Based on the given data points, we try to plot a line that models the points the best. The line can be modelled based on the linear equation shown below.

$$y = a_0 + a_1 * x$$

The motive of the linear regression algorithm is to find the best values for a_0 and a_1. Before moving on to the algorithm, let's have a look at two important concepts you must know to better understand linear regression.

Cost Function

The cost function helps us to figure out the best possible values for a_0 and a_1 which would provide the best fit line for the data points. Since we want the best values for a_0 and a_1, we convert this search problem into a minimization problem where we would like to minimize the error between the predicted value and the actual value.

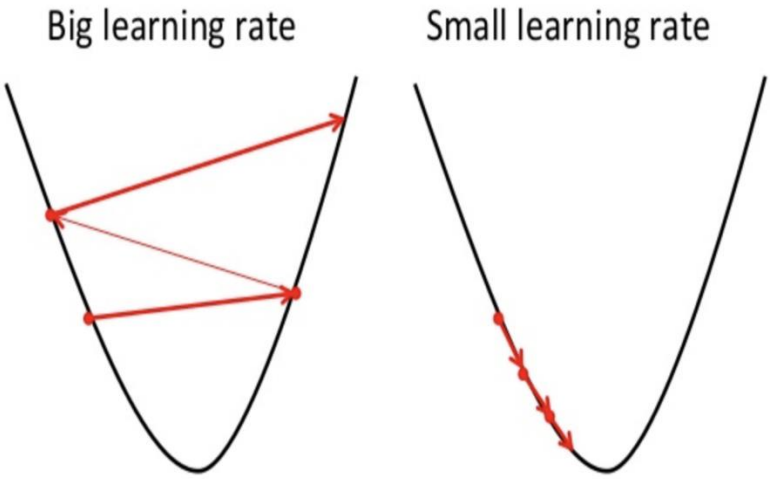
$$minimize \frac{1}{n} \sum_{i=1}^{n} (pred_i - y_i)^2$$

$$J = \frac{1}{n} \sum_{i=1}^{n} (pred_i - y_i)^2$$

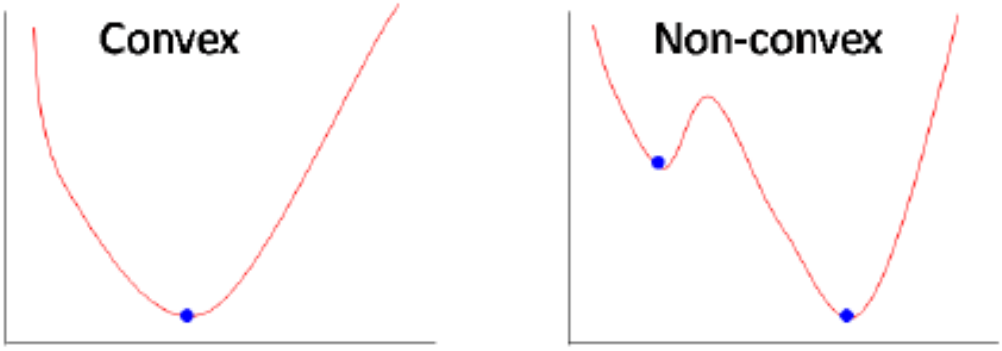
We choose the above function to minimize. The difference between the predicted values and ground truth measures the error difference. We square the error difference and sum over all data points and divide that value by the total number of data points. This provides the average squared error over all the data points. Therefore, this cost function is also known as the Mean Squared Error(MSE) function. Now, using this MSE function we are going to change the values of a_0 and a_1 such that the MSE value settles at the minima.

Gradient Descent

The next important concept needed to understand linear regression is gradient descent. Gradient descent is a method of updating a_0 and a_1 to reduce the cost function(MSE). The idea is that we start with some values for a_0 and a_1 and then we change these values iteratively to reduce the cost. Gradient descent helps us on how to change the values.



To draw an analogy, imagine a pit in the shape of U and you are standing at the topmost point in the pit and your objective is to reach the bottom of the pit. There is a catch, you can only take a discrete number of steps to reach the bottom. If you decide to take one step at a time you would eventually reach the bottom of the pit but this would take a longer time. If you choose to take longer steps each time, you would reach sooner but, there is a chance that you could overshoot the bottom of the pit and not exactly at the bottom. In the gradient descent algorithm, the number of steps you take is the learning rate. This decides on how fast the algorithm converges to the minima.



Sometimes the cost function can be a non-convex function where you could settle at a local minimum but for linear regression, it is always a convex function.

You may be wondering how to use gradient descent to update a_0 and a_1. To update a_0 and a_1, we take gradients from the cost function. To find these gradients, we take partial derivatives with respect to a_0 and a_1. Now, to understand how the partial derivatives are found below you would require some calculus but if you don't, it is alright. You can take it as it is.

$$J = \frac{1}{n}\sum_{i=1}^{n}(pred_i - y_i)^2$$

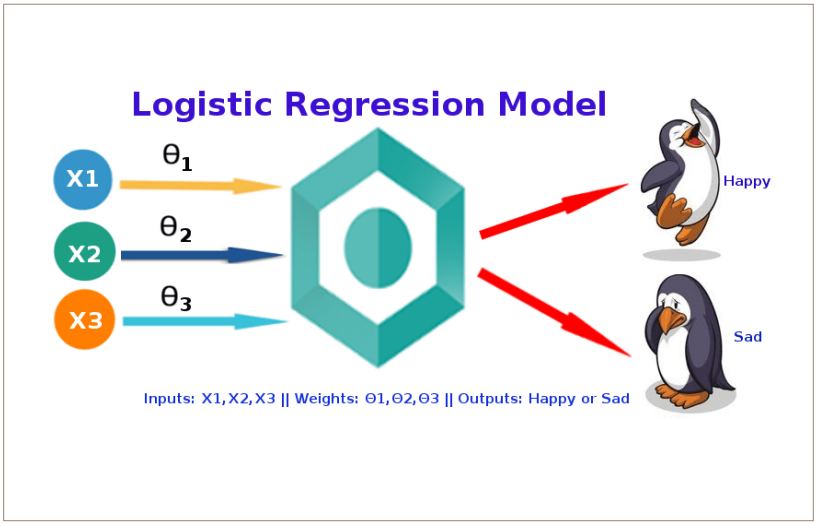
$$J = \frac{1}{n}\sum_{i=1}^{n}(a_0 + a_1 \cdot x_i - y_i)^2$$

$$\frac{\partial J}{\partial a_0} = \frac{2}{n}\sum_{i=1}^{n}(a_0 + a_1 \cdot x_i - y_i) \implies \frac{\partial J}{\partial a_0} = \frac{2}{n}\sum_{i=1}^{n}(pred_i - y_i)$$

$$\frac{\partial J}{\partial a_1} = \frac{2}{n}\sum_{i=1}^{n}(a_0 + a_1 \cdot x_i - y_i) \cdot x_i \implies \frac{\partial J}{\partial a_1} = \frac{2}{n}\sum_{i=1}^{n}(pred_i - y_i) \cdot x_i$$

$$a_0 = a_0 - \alpha \cdot \frac{2}{n}\sum_{i=1}^{n}(pred_i - y_i)$$

$$a_1 = a_1 - \alpha \cdot \frac{2}{n}\sum_{i=1}^{n}(pred_i - y_i) \cdot x_i$$



LOGISTIC REGRESSION

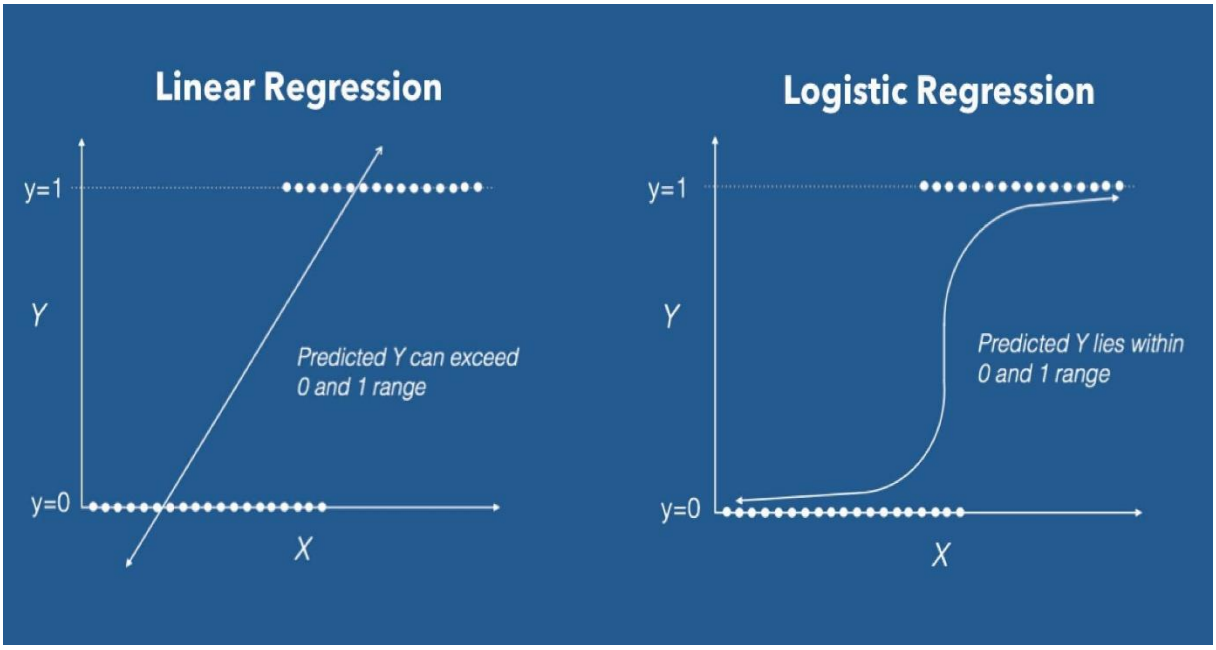
Introduction

Logistic regression is a classification algorithm used to assign observations to a discrete set of classes. Some of the examples of classification problems are Email spam or not spam, Online transactions Fraud or not Fraud, Tumor Malignant or Benign. Logistic regression transforms its output using the logistic sigmoid function to return a probability value.

Types of logistic regression

- Binary (eg. Tumor Malignant or Benign)
- Multi-linear functions (eg. Cats, dogs or Sheep's)

Logistic Regression is a Machine Learning algorithm which is used for the classification problems, it is a predictive analysis algorithm and based on the concept of probability.



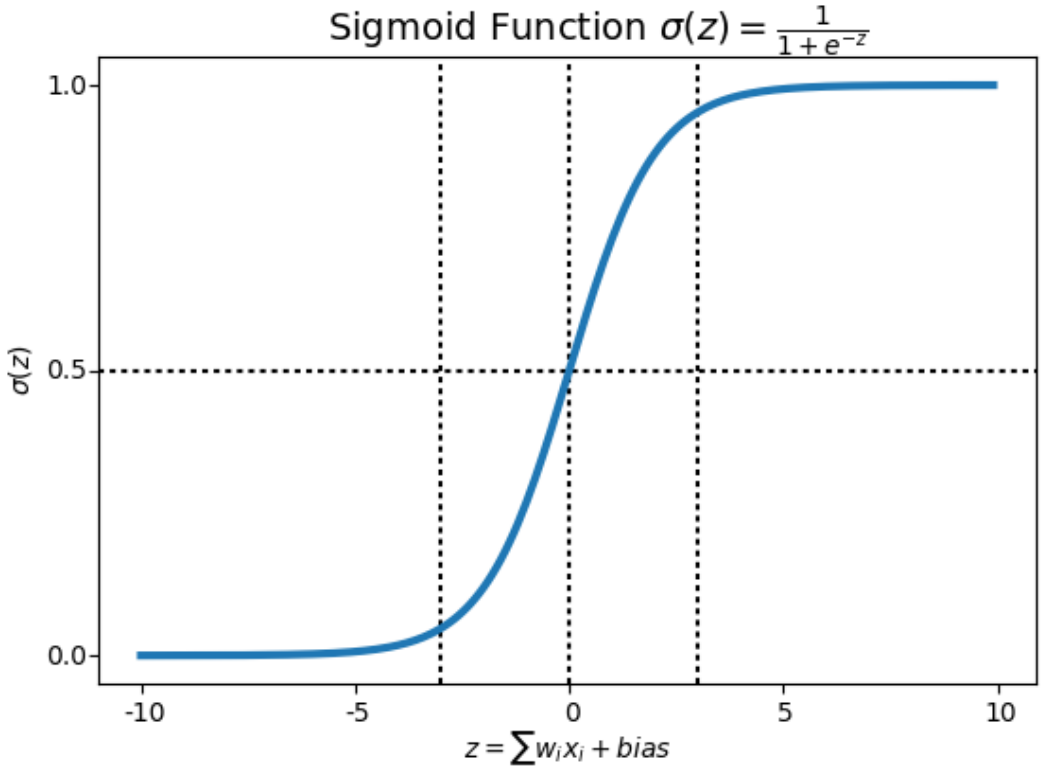
We can call a Logistic Regression a Linear Regression model but the Logistic Regression uses a more complex cost function, this cost function can be defined as the 'Sigmoid function' or also known as the 'logistic function' instead of a linear function.

The hypothesis of logistic regression tends it to limit the cost function between 0 and 1. Therefore linear functions fail to represent it as it can have a value greater than 1 or less than 0 which is not possible as per the hypothesis of logistic regression.

$$0 \leq h_\theta(x) \leq 1$$

Sigmoid Function

In order to map predicted values to probabilities, we use the Sigmoid function. The function maps any real value into another value between 0 and 1. In machine learning, we use sigmoid to map predictions to probabilities.



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

Hypothesis Representation

When using linear regression, we used a formula of the hypothesis i.e.

$$h\Theta(x) = \beta_o + \beta_1 X$$

For logistic regression we are going to modify it a little bit i.e.

$$\sigma(Z) = \sigma(\beta_o + \beta_1 X)$$

We have expected that our hypothesis will give values between 0 and 1.

$$Z = \beta_o + \beta_1 X$$

$$h\Theta(x) = \text{sigmoid}(Z)$$

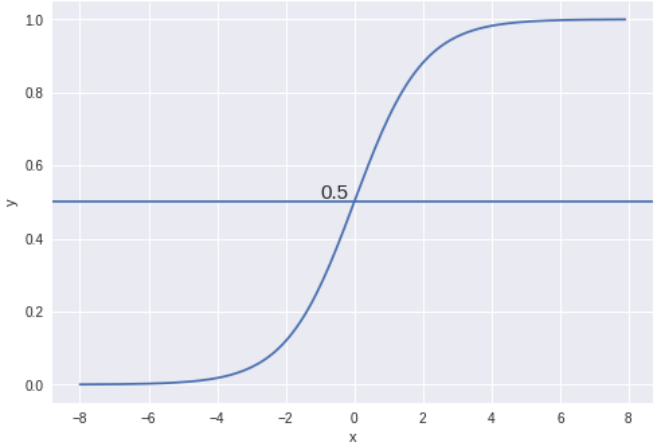
$$\text{i.e. } h\Theta(x) = 1/(1 + e^{\wedge}\text{-}(\beta_o + \beta_1 X)$$

$$h\theta(X) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X)}}$$

Decision Boundary

We expect our classifier to give us a set of outputs or classes based on probability when we pass the inputs through a prediction function and returns a probability score between 0 and 1.

For Example, we have 2 classes, let's take them like cats and dogs (1 — dog, 0 — cats). We basically decide with a threshold value above which we classify values into Class 1 and of the value goes below the threshold then we classify it in Class 2.



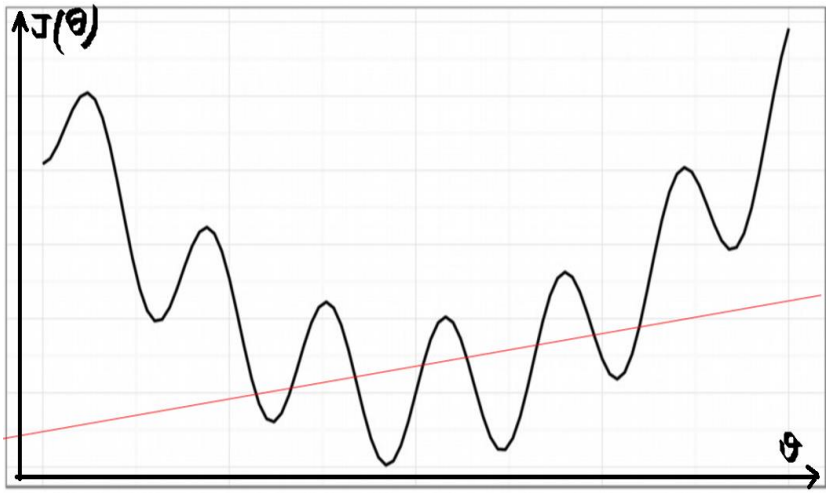
As shown in the above graph we have chosen the threshold as 0.5, if the prediction function returned a value of 0.7 then we would classify this observation as Class 1(DOG). If our prediction returned a value of 0.2 then we would classify the observation as Class 2(CAT).

Cost Function

We learnt about the cost function J(θ) in the Linear regression, the cost function represents optimization objective i.e. we create a cost function and minimize it so that we can develop an accurate model with minimum error.

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2.$$

If we try to use the cost function of the linear regression in 'Logistic Regression' then it would be of no use as it would end up being a non-convex function with many local minimums, in which it would be very difficult to minimize the cost value and find the global minimum.

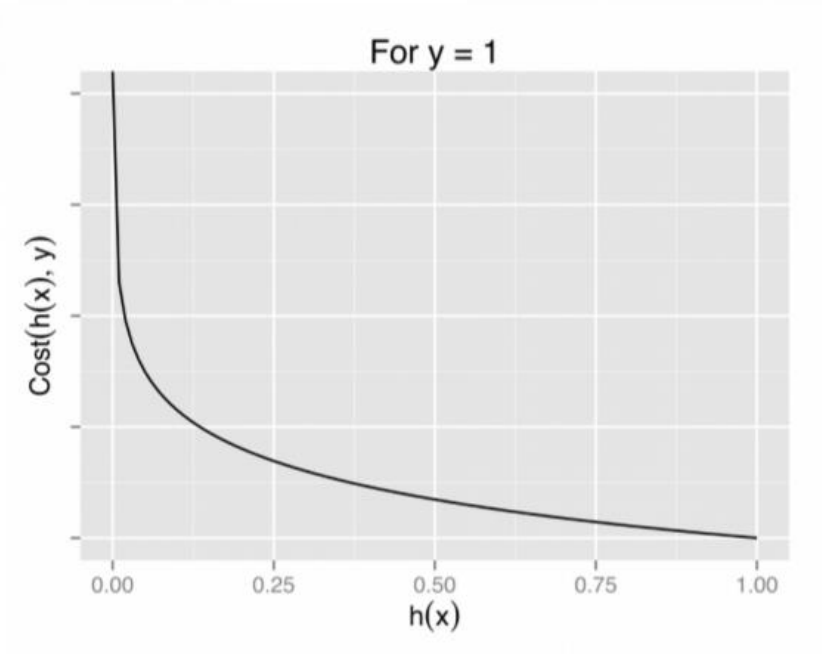


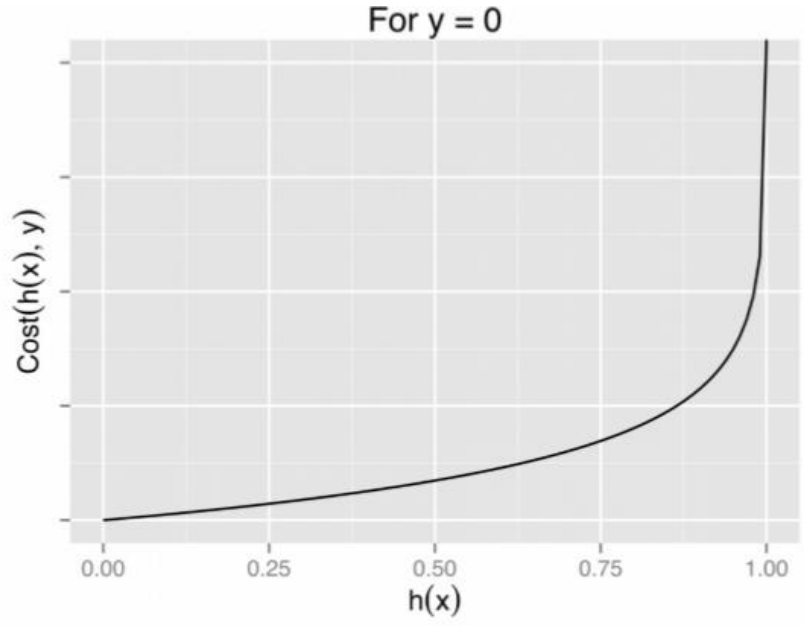
For logistic regression, the Cost function is defined as:

−log(hθ(x)) if y = 1

−log(1−hθ(x)) if y = 0

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





The above two functions can be compressed into a single function i.e.

$$J(\theta) = -\frac{1}{m}\sum\left[y^{(i)}\log(h\theta(x(i))) + \left(1 - y^{(i)}\right)\log(1 - h\theta(x(i)))\right]$$

Gradient Descent

Now the question arises, how do we reduce the cost value. Well, this can be done by using Gradient Descent. The main goal of Gradient descent is to minimize the cost value. i.e. min J(θ).

Now to minimize our cost function we need to run the gradient descent function on each parameter i.e.

$$\theta j := \theta j - \alpha \frac{\partial}{\partial \theta j} J(\theta)$$

Want $\min_\theta J(\theta)$:

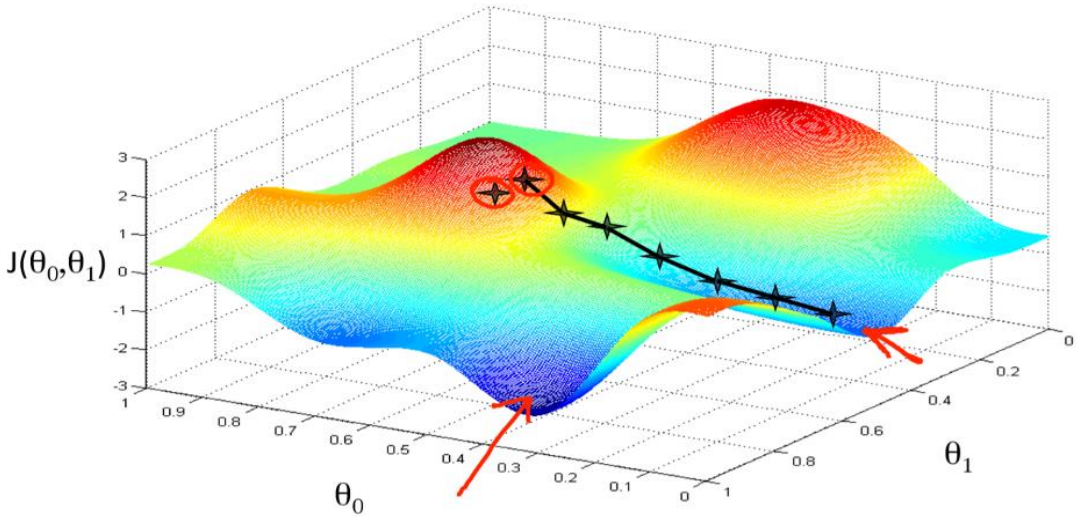
Repeat $\{$

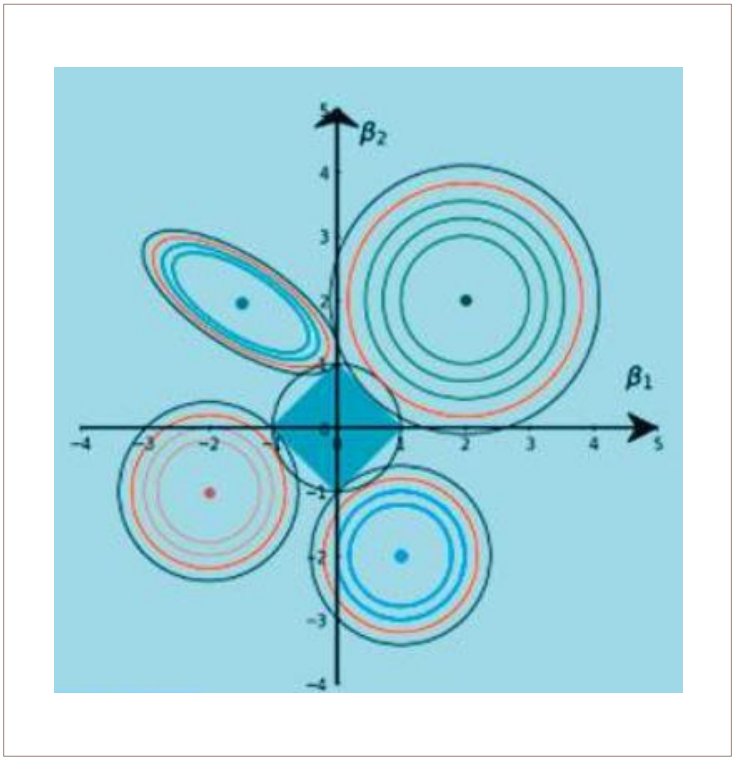
$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

(simultaneously update all θ_j)

$\}$

Gradient descent has an analogy in which we have to imagine ourselves at the top of a mountain valley and left stranded and blindfolded, our objective is to reach the bottom of the hill. Feeling the slope of the terrain around you is what everyone would do. Well, this action is analogous to calculating the gradient descent, and taking a step is analogous to one iteration of the update to the parameters.





REGULARIZATION IN ML

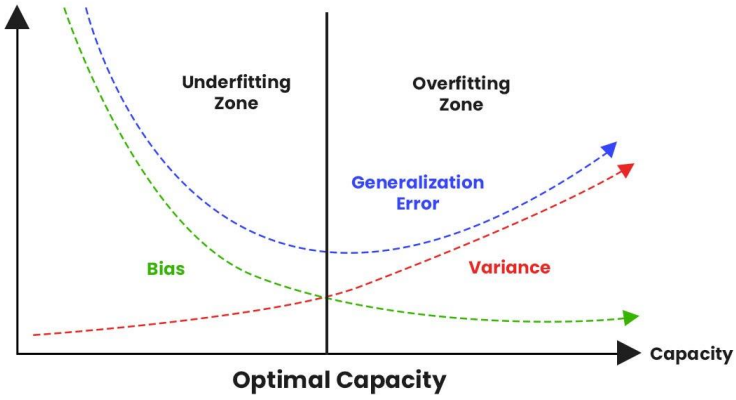
Introduction

In supervised machine learning, the ML models get trained training data and there are the possibilities that the model performs accurately on training data but fails to perform well on test data and also produces high error due to several factors such as collinearity, bias-variance impact and over modeling on train data.
For example, when the model learns signals as well as noises in the training data but couldn't perform appropriately on new data upon which the model wasn't trained, the condition/problem of overfitting takes place.

Bias-Variance Tradeoff

In order to understand how the deviation of the function is varied, bias and variance can be adopted. Bias is the measurement of deviation or error from the real value of function, variance is the measurement of deviation in the response variable function while estimating it over a different training sample of the dataset.

The underlying association between bias and variance is closely related to the overfitting, under fitting and capacity in machine learning such that while calculating the generalization error (where bias and variance are crucial elements) increase in the model capacity can lead to increase in variance and decrease in bias.



Below the conditions of under fitting, exact fit, and overfitting can be observed.

What is regularization

In regression analysis, the features are estimated using coefficients while modelling. Also, if the estimates can be restricted, or shrinked or regularized towards zero, then the impact of insignificant features might be reduced and would prevent models from high variance with a stable fit.

Regularization is the most used technique to penalize complex models in machine learning, it is deployed for reducing overfitting (or, contracting generalization errors) by putting network weights small. Also, it enhances the performance of models for new inputs.

L1 and L2 Regularization

A regression model that uses L1 regularization technique is called **Lasso Regression** and model which uses L2 is called **Ridge Regression**.

Ridge regression adds "squared magnitude" of coefficient as penalty term to the loss function. Here the highlighted part represents L2 regularization element.

$$\sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

Cost function

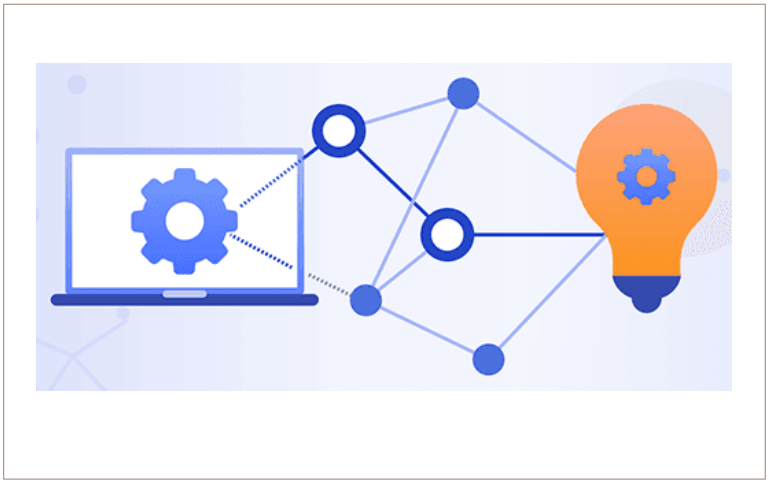
Here, if lambda is zero then you can imagine we get back OLS. However, if lambda is very large then it will add too much weight and it will lead to under-fitting. Having said that it's important how lambda is chosen. This technique works very well to avoid over-fitting issue.

Lasso Regression (Least Absolute Shrinkage and Selection Operator) adds "*absolute value of magnitude*" of coefficient as penalty term to the loss function.

$$\sum_{i=1}^{n}(Y_i - \sum_{j=1}^{p} X_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

Cost function

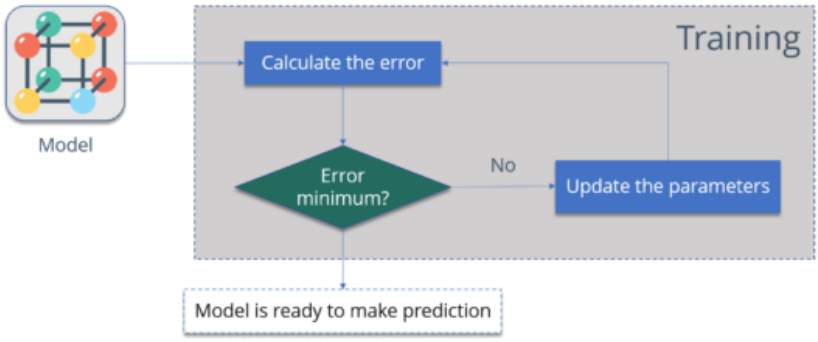
Again, if *lambda* is zero then we will get back OLS whereas very large value will make coefficients zero hence it will under-fit.



BACKPROPAGATION ALGORITHM

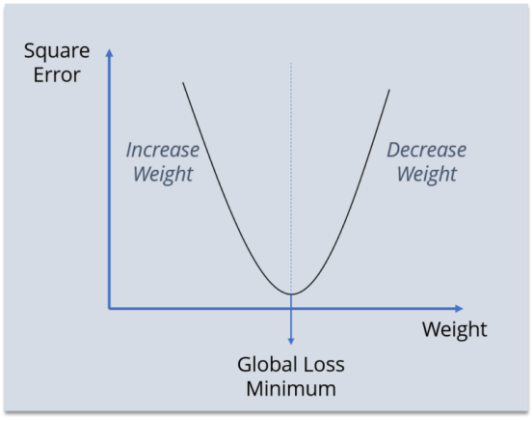
Introduction

The algorithm is used to effectively train a neural network through a method called chain rule. In simple terms, after each forward pass through a network, Backpropagation performs a backward pass while adjusting the model's parameters (weights and biases).



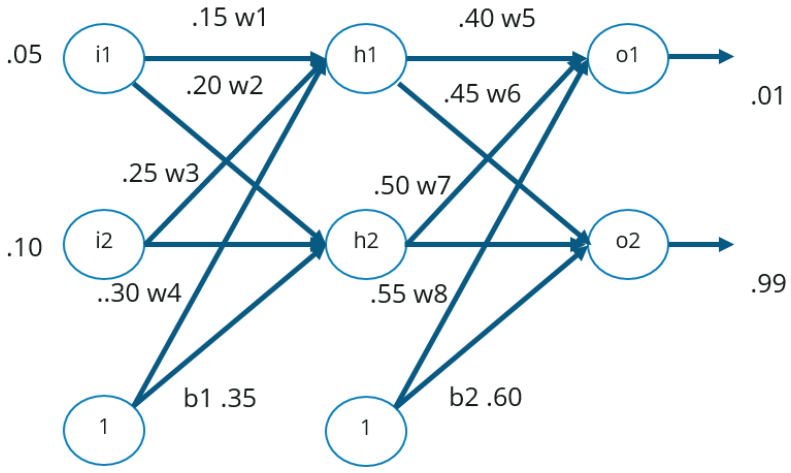
The model is trained and the error is calculated, if the error is minimum then the model is ready to make prediction. Or else if the error is maximum then the weights are adjusted with the help of Backpropagation and again the model is trained and error is calculated. This process continues until the error is minimum.

So, we are trying to get the value of weight such that the error becomes minimum. Basically, we need to figure out whether we need to increase or decrease the weight value. Once we know that, we keep on updating the weight value in that direction until error becomes minimum. You might reach a point, where if you further update the weight, the error will increase. At that time you need to stop, and that is your final weight value.



In the above graph we need to reach the "Global Loss Minimum". This is nothing but Backpropagation.

The working of Backpropagation algorithm

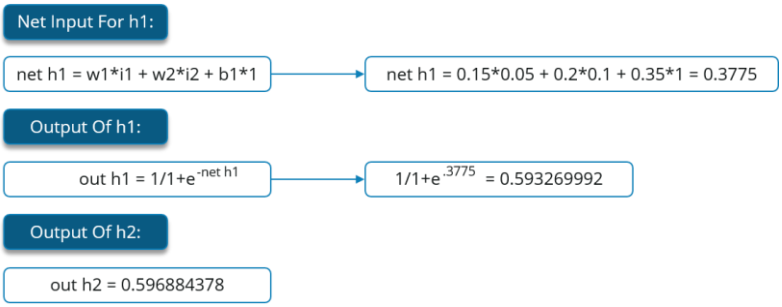


The above network contains the following:

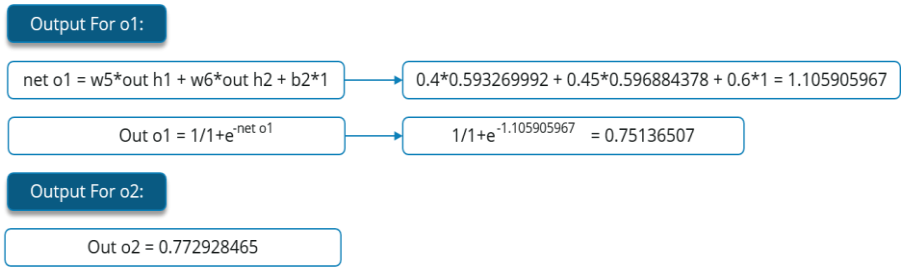
- two inputs
- two hidden neurons
- two output neurons
- two biases

The working of Backpropagation algorithm

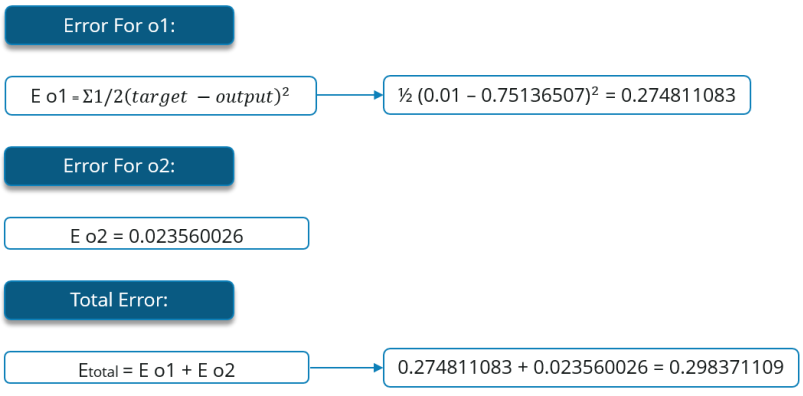
Step – 1: Forward Propagation



Start by propagating forward, and repeat this process for the output layer neuron, using the output from the hidden layer neurons as input.



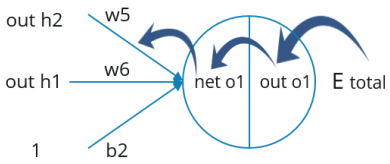
Now we have to see the value of error:



Step – 2: Backward Propagation

Now, we will propagate backwards. This way we will try to reduce the error by changing the values of weights and biases.
Consider W5, we will calculate the rate of change of error w.r.t change in weight W5.

$$\frac{\delta Etotal}{\delta w5} = \frac{\delta Etotal}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w5}$$



Since we are propagating backwards, first thing we need to do is, calculate the change in total errors w.r.t the output O1 and O2.

$$E_{total} = 1/2(\text{target o1} - \text{out o1})^2 + 1/2(\text{target o2} - \text{out o2})^2$$

$$\frac{\delta Etotal}{\delta out\ o1} = -(\text{target o1} - \text{out o1}) = -(0.01 - 0.75136507) = 0.74136507$$

Now, we will propagate further backwards and calculate the change in output O1 w.r.t to its total net input.

$$\text{out o1} = 1/1+e^{-neto1}$$

$$\frac{\delta out\ o1}{\delta net\ o1} = \text{out o1 (1 - out o1)} = 0.75136507\ (1 - 0.75136507) = 0.186815602$$

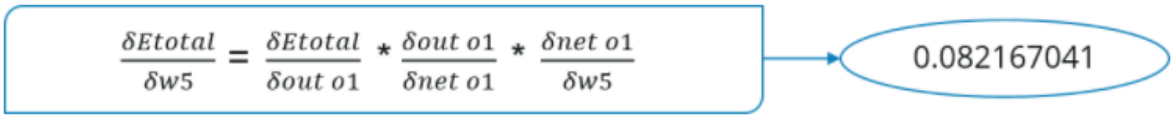
Let's see now how much does the total net input of O1 changes w.r.t W5?

$$\text{net o1} = w5 * \text{out h1} + w6 * \text{out h2} + b2 * 1$$

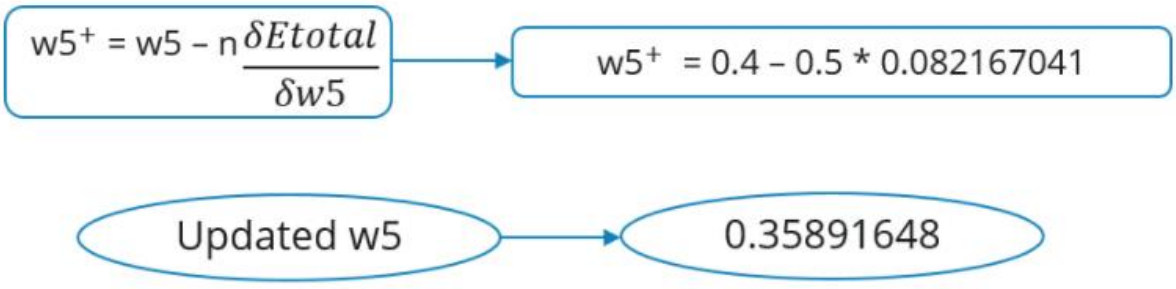
$$\frac{\delta net\ o1}{\delta w5} = 1 * \text{out h1}\ w5^{(1-1)} + 0 + 0 = 0.593269992$$

Step – 3: Putting all the values together and calculating the updated weight value

Let's put all the values together:



Calculate the updated value of W5:



Similarly, we can calculate the other weight values as well.

After that we will again propagate forward and calculate the output. Again, we will calculate the error.

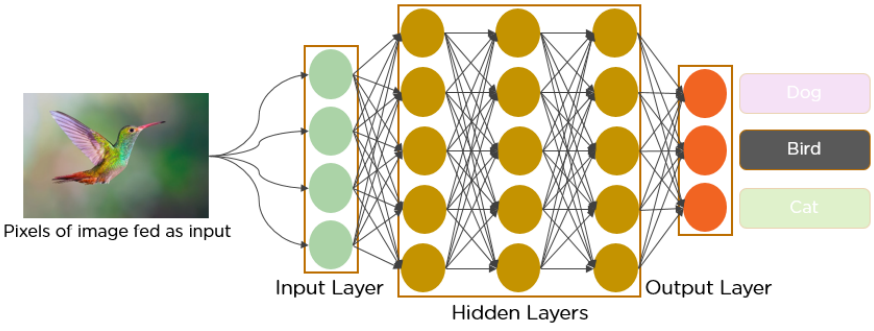
If the error is minimum we will stop right there, else we will again propagate backwards and update the weight values.

This process will keep on repeating until error becomes minimum.

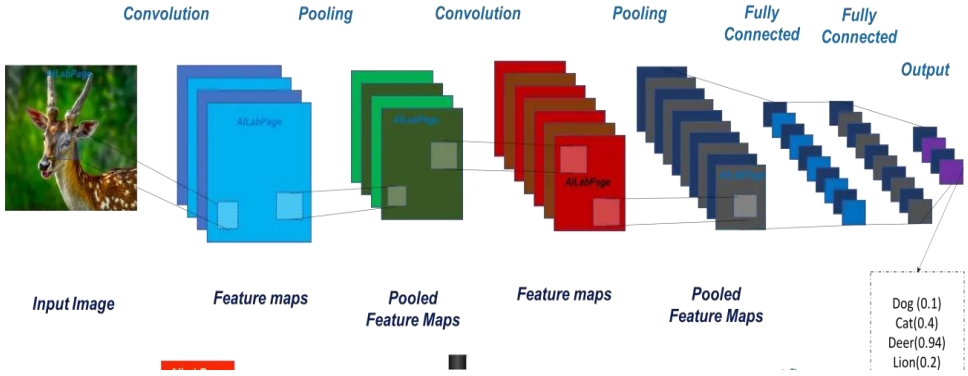
CONVOLUTIONAL NEURAL NETWORKS

Introduction

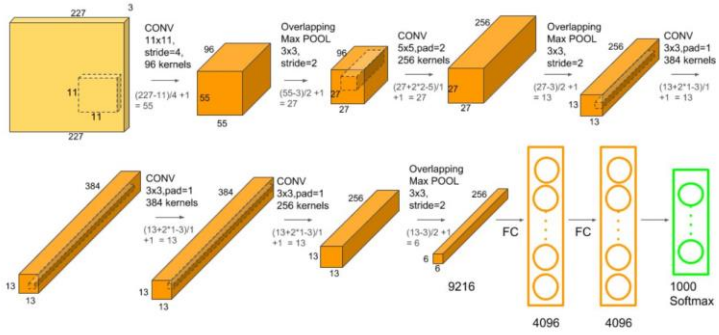
A **convolutional neural network** (**CNN/ConvNet**) is a class of deep neural networks most commonly applied to analyze visual imagery. Now when we think of a neural network we think about matrix multiplications but that is not the case with ConvNet.



It uses a special technique called Convolution. Now in mathematics **convolution** is a mathematical operation on two functions that produces a third function that expresses how the shape of one is modified by the other.



When training a CNN for image classification, a labeled image is used as input for the neural network. A lot of processing is done on this input in multiple layers, such as performing convolutions using kernels of different sizes, passing values through activation functions like ReLu, pooling values to downsize the data, and ultimately, classifying the input image into a category.



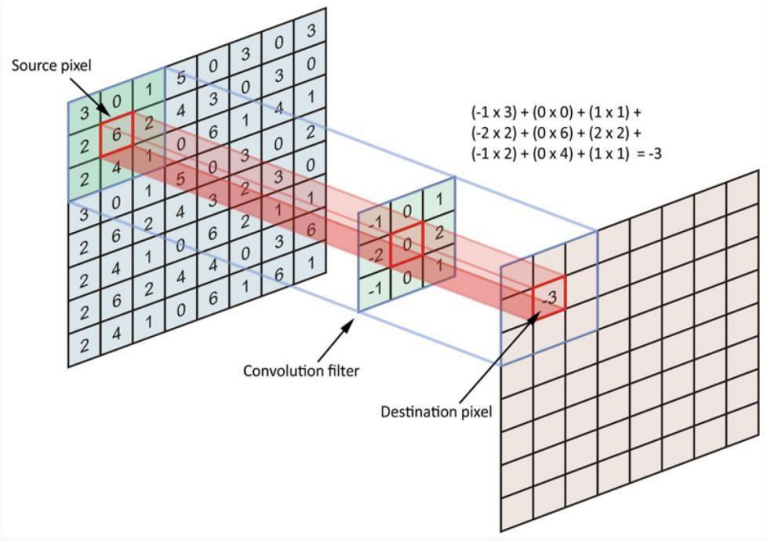
CNN Components

- Convolution layer
- Pooling layer
- Activation function
- Fully connected layer

Convolution layer:

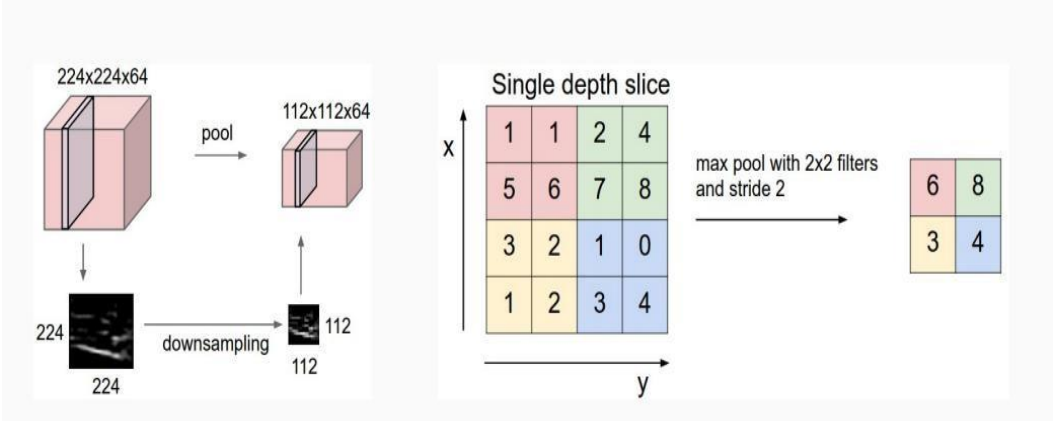
A convolutional layer consists of a set of learnable filters. Every filter is spatially small in width and height, but extends through the full depth of the input volume.

- Learnable filters are applied to local regions
- A weight activation map determines the impact of the filter
- Weights can be shared across activation functions to reduce the number of learnable parameters
- Depth conceptually will learn different features
- Translation equivariant



Pooling layer

- Progressively reduce the spatial size of the representation, to reduce the amount of parameters and computation in the network.
- Using the average or maximum function
- Reducing output vector map-size.

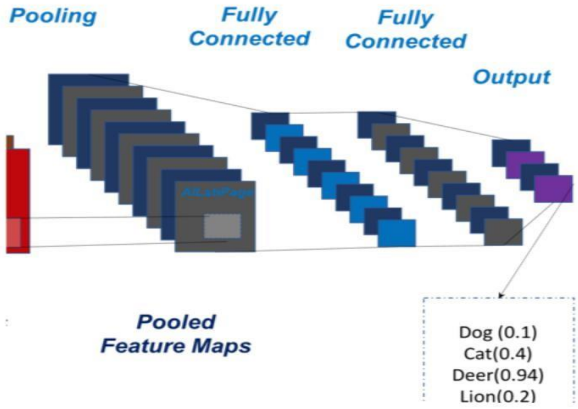


Activation function

- Sigmoid activation function in conventional ML algorithms.
- To introduce non-linearity use Rectified Linear Unit (ReLU).

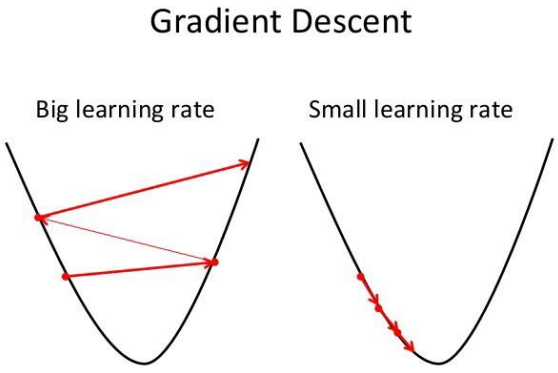
Fully connected network

- Fully connected neurons
- Gradient descent reduces the cost function.
- Used in output layers



Hyper parameters related to Training Algorithm

Learning Rate



The learning rate defines how quickly a network updates its parameters.
Low learning rate slows down the learning process but converges smoothly. **Larger learning rate** speeds up the learning but may not converge.

Number of epochs

Number of epochs is the number of times the whole training data is shown to the network while training.

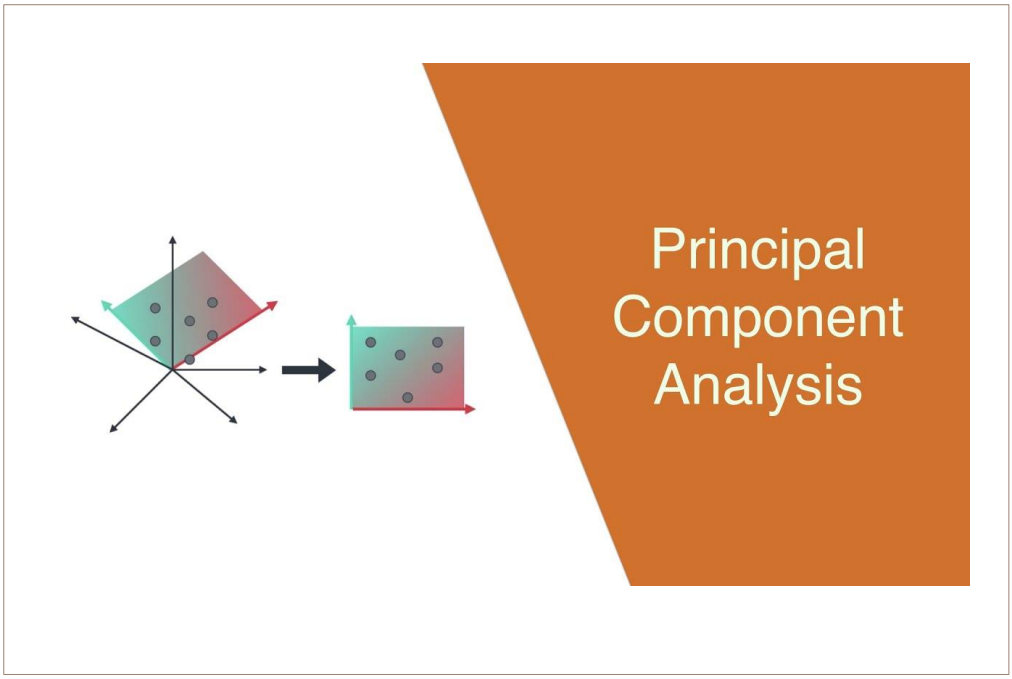
Increase the number of epochs until the validation accuracy starts decreasing even when training accuracy is increasing(overfitting).

What is epoch?

Training the entire training set and expecting the model to give the best fit

Batch size

Mini batch size is the number of sub samples given to the network after which parameter update happens.

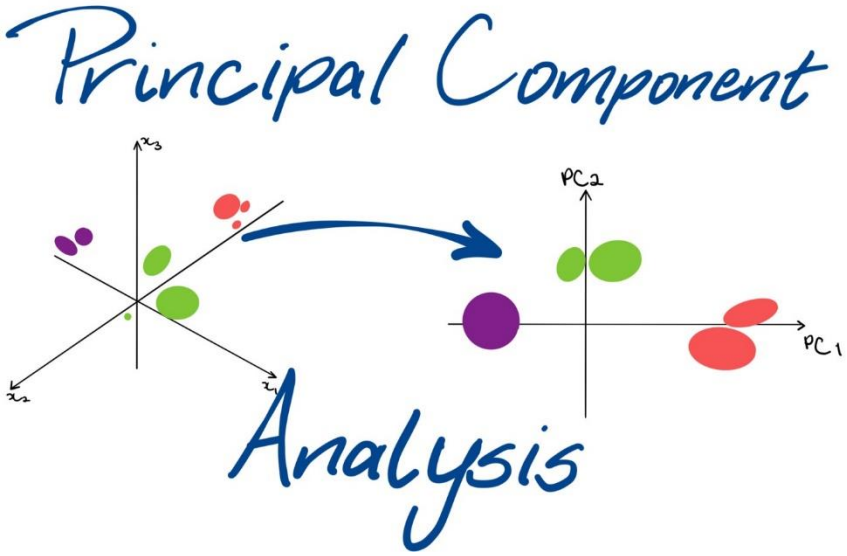


PRINCIPAL COMPONENT ANALYSIS

Introduction

Principal Component Analysis, or PCA, is a dimensionality-reduction method that is often used to reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set.

Reducing the number of variables of a data set naturally comes at the expense of accuracy, but the trick in dimensionality reduction is to trade a little accuracy for simplicity. Because smaller data sets are easier to explore and visualize and make analyzing data much easier and faster for machine learning algorithms without extraneous variables to process.



Steps to perform PCA

- Standardization
- Covariance matrix computation
- Compute the eigenvectors and eigenvalues of the covariance matrix to identify the principal components
- Feature vector
- Recast the data along the principal components axes

Step 1: Standardization

The aim of this step is to standardize the range of the continuous initial variables so that each one of them contributes equally to the analysis.

More specifically, the reason why it is critical to perform standardization prior to PCA, is that the latter is quite sensitive regarding the variances of the initial variables. That is, if there are large differences between the ranges of initial variables, those variables with larger ranges will dominate over those with small ranges (For example, a variable that ranges between 0 and 100 will dominate over a variable that ranges between 0 and 1), which will lead to biased results. So, transforming the data to comparable scales can prevent this problem.

Mathematically, this can be done by subtracting the mean and dividing by the standard deviation for each value of each variable. Once the standardization is done, all the variables will be transformed to the same scale.

$$z = \frac{value - mean}{standard\ deviation}$$

Step 2: Covariance matrix computation

The aim of this step is to understand how the variables of the input data set are varying from the mean with respect to each other, or in other words, to see if there is any relationship between them. Because sometimes, variables are highly correlated in such a way that they contain redundant information. So, in order to identify these correlations, we compute the covariance matrix.

The covariance matrix is a $p \times p$ symmetric matrix (where p is the number of dimensions) that has as entries the covariance's associated with all possible pairs of the initial variables. For example, for a 3-dimensional data set with 3 variables x, y, and z, the covariance matrix is a 3×3 matrix of this from:

$$\begin{bmatrix} Cov(x,x) & Cov(x,y) & Cov(x,z) \\ Cov(y,x) & Cov(y,y) & Cov(y,z) \\ Cov(z,x) & Cov(z,y) & Cov(z,z) \end{bmatrix}$$

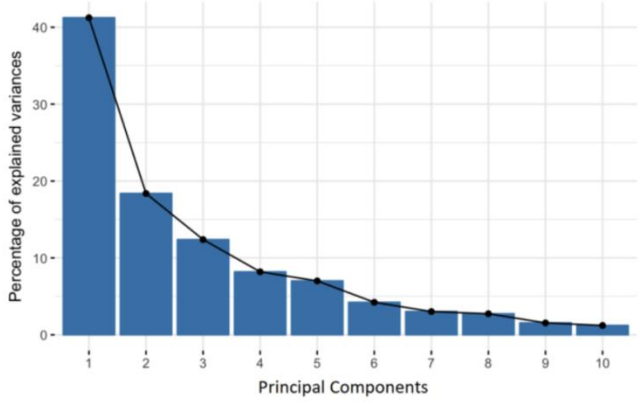
Covariance Matrix for 3-Dimensional Data

Since the covariance of a variable with itself is its variance (Cov(a,a)=Var(a)), in the main diagonal (Top left to bottom right) we actually have the variances of each initial variable. And since the covariance is commutative (Cov(a,b)=Cov(b,a)), the entries of the covariance matrix are symmetric with respect to the main diagonal, which means that the upper and the lower triangular portions are equal.

Step 3: Compute the eigenvectors and eigenvalues of the covariance matrix to identify the principal components

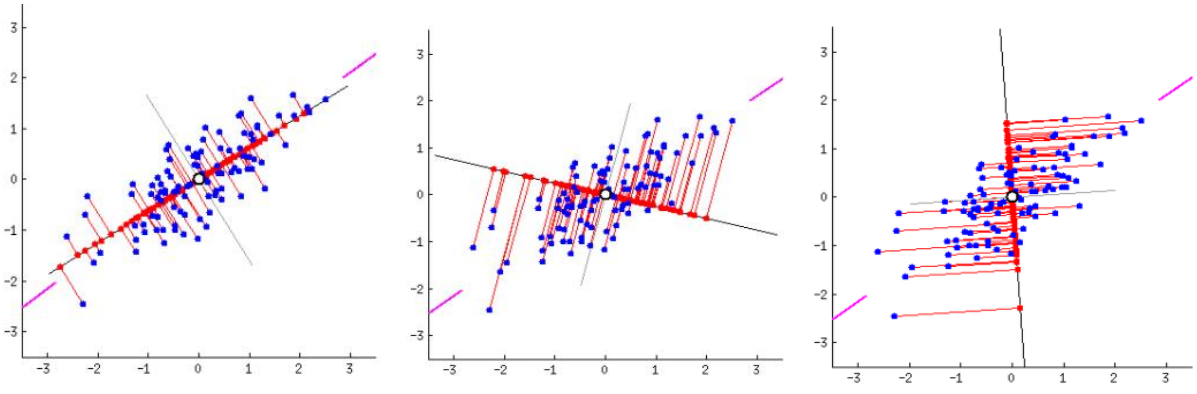
Eigenvectors and eigenvalues are the linear algebra concepts that we need to compute from the covariance matrix in order to determine the *principal components* of the data. Before getting to the explanation of these concepts, let's first understand what do we mean by principal components.

Principal components are new variables that are constructed as linear combinations or mixtures of the initial variables. These combinations are done in such a way that the new variables (i.e., principal components) are uncorrelated and most of the information within the initial variables is squeezed or compressed into the first components. So, the idea is 10-dimensional data gives you 10 principal components, but PCA tries to put maximum possible information in the first component, then maximum remaining information in the second and so on, until having something like shown in the scree plot below.



Percentage of Variance (Information) for each by PC

How PCA constructs the principal components



Step 4: Feature vector

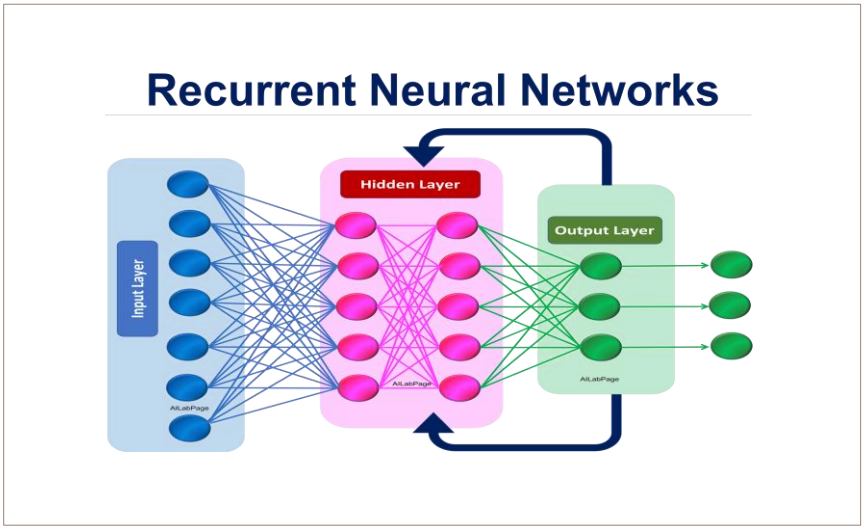
As we saw in the previous step, computing the eigenvectors and ordering them by their eigenvalues in descending order, allow us to find the principal components in order of significance. In this step, what we do is, to choose whether to keep all these components or discard those of lesser significance (of low eigenvalues), and form with the remaining ones a matrix of vectors that we call *Feature vector*.

So, the feature vector is simply a matrix that has as columns the eigenvectors of the components that we decide to keep. This makes it the first step towards dimensionality reduction, because if we choose to keep only *p* eigenvectors (components) out of *n*, the final data set will have only *p* dimensions.

Step 5: Recast the data along the principal components axes

In this step, which is the last one, the aim is to use the feature vector formed using the eigenvectors of the covariance matrix, to reorient the data from the original axes to the ones represented by the principal components (hence the name Principal Components Analysis). This can be done by multiplying the transpose of the original data set by the transpose of the feature vector.

$$FinalDataSet = FeatureVector^T * StandardizedOriginalDataSet^T$$

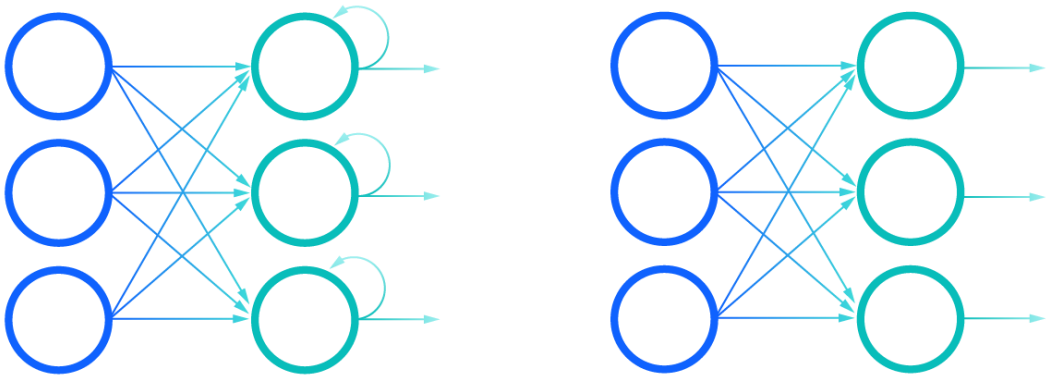


RECURRENT NEURAL NETWORKS

Introduction

A recurrent neural network (RNN) is a type of artificial neural network which uses sequential data or time series data. These deep learning algorithms are commonly used for ordinal or temporal problems, such as language translation, natural language processing (nlp), speech recognition, and image captioning; they are incorporated into popular applications such as Siri, voice search, and Google Translate. Like feedforward and convolutional neural networks (CNNs), recurrent neural networks utilize training data to learn. They are distinguished by their "memory" as they take information from prior inputs to influence the current input and output. While traditional deep neural networks assume that inputs and outputs are independent of each other, the output of recurrent neural networks depend on the prior elements within the sequence. While future events would also be helpful in determining the output of a given sequence, unidirectional recurrent neural networks cannot account for these events in their predictions.

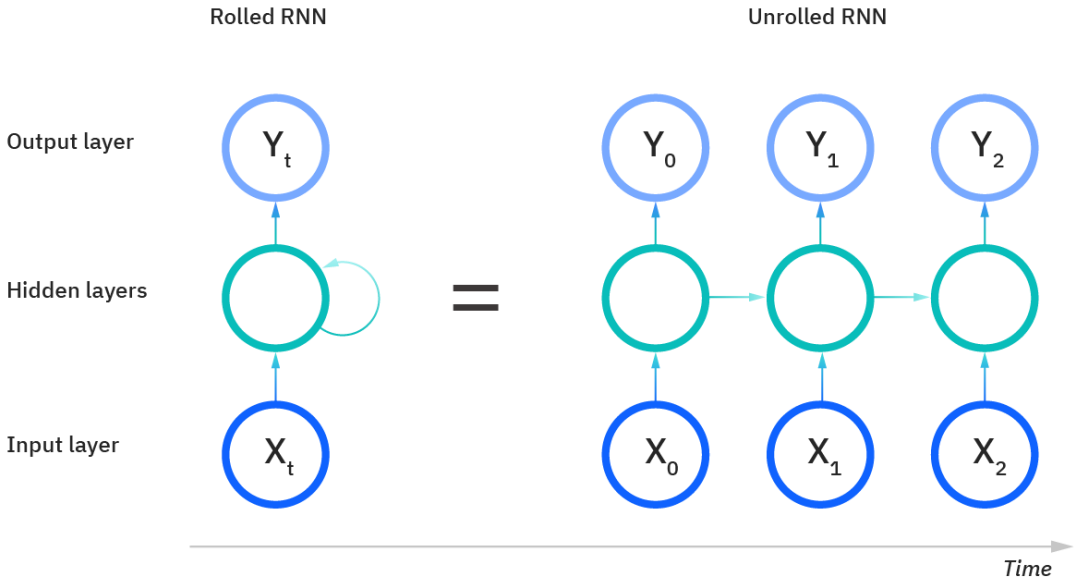
Recurrent Neural Network vs. Feedforward Neural Network



Comparison of Recurrent Neural Networks (on the left) and Feedforward Neural Networks (on the right)

Let's take an idiom, such as "feeling under the weather", which is commonly used when someone is ill, to aid us in the explanation of RNNs. In order for the idiom to make sense, it needs to be expressed in that specific order. As a result, recurrent networks need to account for the position of each word in the idiom and they use that information to predict the next word in the sequence.

Looking at the visual below, the "rolled" visual of the RNN represents the whole neural network, or rather the entire predicted phrase, like "feeling under the weather." The "unrolled" visual represents the individual layers, or time steps, of the neural network. Each layer maps to a single word in that phrase, such as "weather". Prior inputs, such as "feeling" and "under", would be represented as a hidden state in the third timestep to predict the output in the sequence, "the".



Another distinguishing characteristic of recurrent networks is that they share parameters across each layer of the network. While feedforward networks have different weights across each node, recurrent neural networks share the same weight parameter within each layer of the network. That said, these weights are still adjusted in the through the processes of backpropagation and gradient descent to facilitate reinforcement learning.

Recurrent neural networks leverage backpropagation through time (BPTT) algorithm to determine the gradients, which is slightly different from traditional backpropagation as it is specific to sequence data. The principles of BPTT are the same as traditional backpropagation, where the model trains itself by calculating errors from its output layer to its input layer. These calculations allow us to adjust and fit the parameters of the model appropriately. BPTT differs from the traditional approach in that BPTT sums errors at each time step whereas feedforward networks do not need to sum errors as they do not share parameters across each layer.

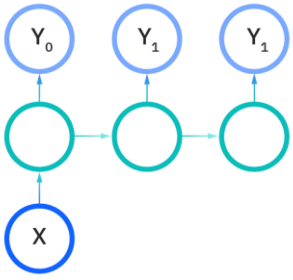
Through this process, RNNs tend to run into two problems, known as exploding gradients and vanishing gradients. These issues are defined by the size of the gradient, which is the slope of the loss function along the error curve. When the gradient is too small, it continues to become smaller, updating the weight parameters until they become insignificant—i.e. 0. When that occurs, the algorithm is no longer learning. Exploding gradients occur when the gradient is too large, creating an unstable model. In this case, the model weights will grow too large, and they will eventually be represented as NaN. One solution to these issues is to reduce the number of hidden layers within the neural network, eliminating some of the complexity in the RNN model.

Types of recurrent neural networks

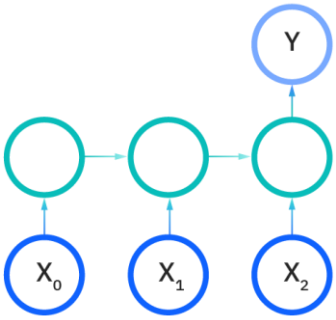
Feedforward networks map one input to one output, and while we've visualized recurrent neural networks in this way in the above diagrams, they do not actually have this constraint. Instead, their inputs and outputs can vary in length, and different types of RNNs are used for different use cases, such as music generation, sentiment classification, and machine translation. Different types of RNNs are usually expressed using the following diagrams:

One-to-one

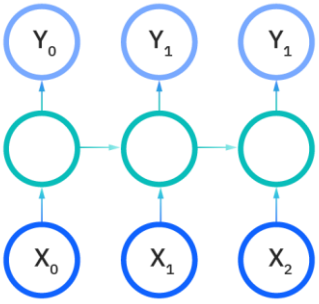
One-to-many:

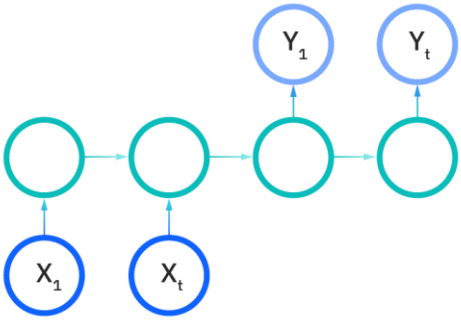


Many-to-one:



Many-to-many:

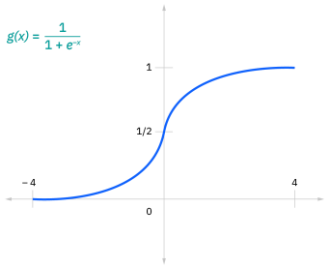




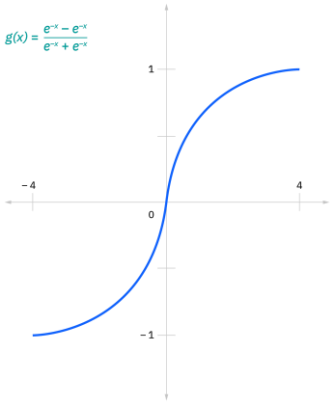
Common activation functions

An activation function determines whether a neuron should be activated. The nonlinear functions typically convert the output of a given neuron to a value between 0 and 1 or -1 and 1. Some of the most commonly used functions are defined as follows:

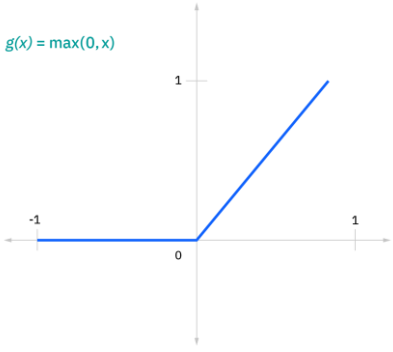
Sigmoid: This is represented with the formula g(x) = 1/(1 + e^-x).

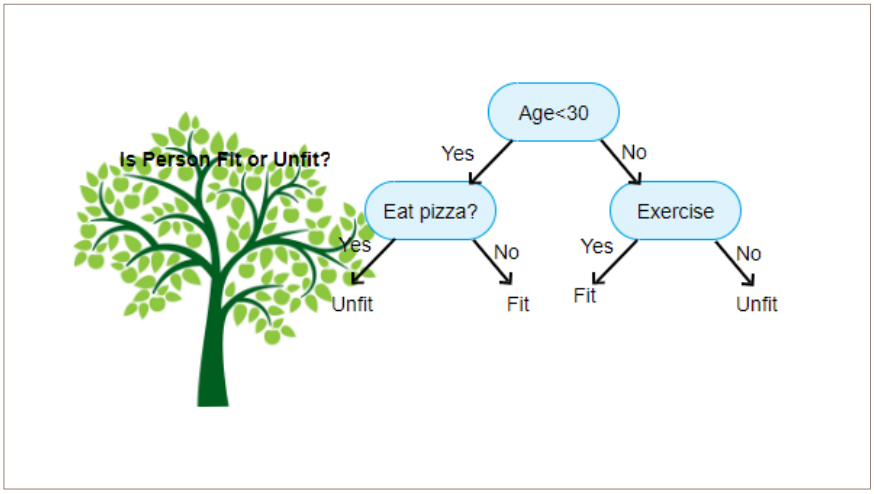


Tanh: This is represented with the formula g(x) = (e^-x - e^-x)/(e^-x + e^-x).



Relu: This is represented with the formula g(x) = max(0 , x)





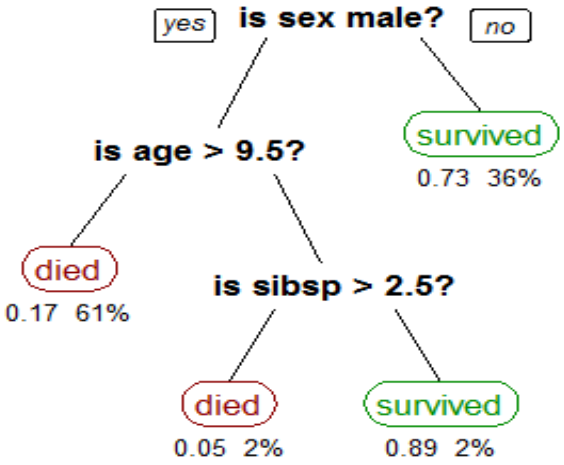
DECISION TREE

Introduction

A tree has many analogies in real life, and turns out that it has influenced a wide area of machine learning, covering both classification and regression. In decision analysis, a decision tree can be used to visually and explicitly represent decisions and decision making. As the name goes, it uses a tree-like model of decisions. Though a commonly used tool in data mining for deriving a strategy to reach a particular goal, its also widely used in machine learning.

How can an algorithm be represented as a tree?

For this let's consider a very basic example that uses titanic data set for predicting whether a passenger will survive or not. Below model uses 3 features/attributes/columns from the data set, namely sex, age and sibsp (number of spouses or children along).



A decision tree is drawn upside down with its root at the top. In the image on the left, the bold text in black represents a condition/internal node, based on which the tree splits into branches/ edges. The end of the branch that doesn't split anymore is the decision/leaf, in this case, whether the passenger died or survived, represented as red and green text respectively.

Although, a real dataset will have a lot more features and this will just be a branch in a much bigger tree, but you can't ignore the simplicity of this algorithm. The feature importance is clear and relations can be viewed easily. This methodology is more commonly known as learning decision tree from data and above tree is called Classification tree as the target is to classify passenger as survived or died. Regression trees are represented in the same manner, just they predict continuous values like price of a house. In general, Decision Tree algorithms are referred to as CART or Classification and Regression Trees.

So, what is actually going on in the background? Growing a tree involves deciding on which features to choose and what conditions to use for splitting, along with knowing when to stop. As a tree generally grows arbitrarily, you will need to trim it down for it to look beautiful. Lets start with a common technique used for splitting.

Recursive Binary Splitting

In this procedure all the features are considered and different split points are tried and tested using a cost function. The split with the best cost (or lowest cost) is selected.

Consider the earlier example of tree learned from titanic dataset. In the first split or the root, all attributes/features are considered and the training data is divided into groups based on this split. We have 3 features, so will have 3 candidate splits. Now we will calculate how much accuracy each split will cost us, using a function. The split that costs least is chosen, which in our example is sex of the passenger. This algorithm is recursive in nature as the groups formed can be sub-divided using same strategy. Due to this procedure, this algorithm is also known as the greedy algorithm, as we have an excessive desire of lowering the cost. This makes the root node as best predictor/classifier.

Cost of a split

Let's take a closer look at cost functions used for classification and regression. In both cases the cost functions try to find most homogeneous branches, or branches having groups with similar responses. This makes sense we can be more sure that a test data input will follow a certain path.

Regression : sum(y — prediction)²

Let's say, we are predicting the price of houses. Now the decision tree will start splitting by considering each feature in training data. The mean of responses of the training data inputs of particular group is considered as prediction for that group. The above function is applied to all data points and cost is calculated for all candidate splits. Again the split with lowest cost is chosen.

Classification : G = sum(pk * (1 — pk))

A Gini score gives an idea of how good a split is by how mixed the response classes are in the groups created by the split. Here, pk is proportion of same class inputs present in a particular group. A perfect class purity occurs when a group contains all inputs from the same class, in which case pk is either 1 or 0 and G = 0, where as a node having a 50–50 split of classes in a group has the worst purity, so for a binary classification it will have pk = 0.5 and G = 0.5.

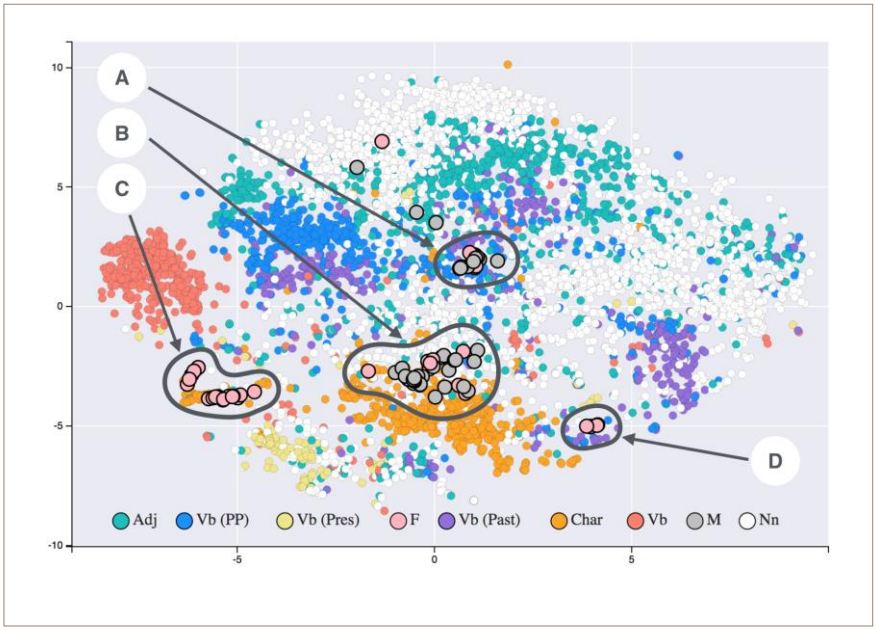
When to stop splitting?

You might ask when to stop growing a tree? As a problem usually has a large set of features, it results in large number of split, which in turn gives a huge tree. Such trees are complex and can lead to overfitting. So, we need to know when to stop? One way of doing this is to set a minimum number of training inputs to use on each leaf. For example, we can use a minimum of 10 passengers to reach a decision (died or survived), and ignore any leaf that takes less than 10 passengers. Another way is to set maximum depth of your model. Maximum depth refers to the length of the longest path from a root to a leaf.

Pruning

The performance of a tree can be further increased by pruning. It involves removing the branches that make use of features having low importance. This way, we reduce the complexity of tree, and thus increasing its predictive power by reducing overfitting.

Pruning can start at either root or the leaves. The simplest method of pruning starts at leaves and removes each node with most popular class in that leaf, this change is kept if it doesn't deteriorate accuracy. It's also called reduced error pruning. More sophisticated pruning methods can be used such as cost complexity pruning where a learning parameter (alpha) is used to weigh whether nodes can be removed based on the size of the sub-tree. This is also known as weakest link pruning.



WORD EMBEDDING

Introduction

Word embedding is one of the most popular representation of document vocabulary. It is capable of capturing context of a word in a document, semantic and syntactic similarity, relation with other words, etc.

What are word embeddings exactly? Loosely speaking, they are vector representations of a particular word. Having said this, what follows is how do we generate them? More importantly, how do they capture the context?

Word2Vec is one of the most popular technique to learn word embeddings using shallow neural network.

Why do we need them?

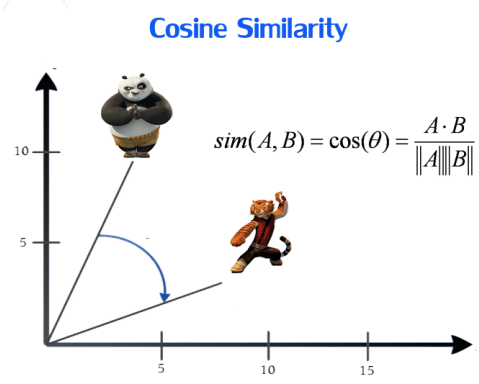
Consider the following similar sentences: Have a good day and Have a great day. They hardly have different meaning. If we construct an exhaustive vocabulary (let's call it V), it would have V = {Have, a, good, great, day}.

Now, let us create a one-hot encoded vector for each of these words in V. Length of our one-hot encoded vector would be equal to the size of V (=5). We would have a vector of zeros except for the element at the index representing the corresponding word in the vocabulary. That particular element would be one. The encodings below would explain this better.

Have = [1,0,0,0,0] `; a= [0,1,0,0,0]` ; good=[0,0,1,0,0]` ; great=[0,0,0,1,0]` ; day=[0,0,0,0,1]` (` represents transpose)

If we try to visualize these encodings, we can think of a 5 dimensional space, where each word occupies one of the dimensions and has nothing to do with the rest (no projection along the other dimensions). This means 'good' and 'great' are as different as 'day' and 'have', which is not true.

Our objective is to have words with similar context occupy close spatial positions. Mathematically, the cosine of the angle between such vectors should be close to 1, i.e. angle close to 0.



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. In one hot encoding representations, all the words are independent of each other, as mentioned earlier.

How does Word2Vec work?

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 (CBOW)

CBOW Model

This method takes the context of each word as the input and tries to predict the word corresponding to the context. Consider our example: Have a great day.

Let the input to the Neural Network be the word, great. Notice that here we are trying to predict a target word (day) using a single context input word great. More specifically, we use the one hot encoding of the input word and measure the output error compared to one hot encoding of the target word (day). In the process of predicting the target word, we learn the vector representation of the target word.

Let us look deeper into the actual architecture.

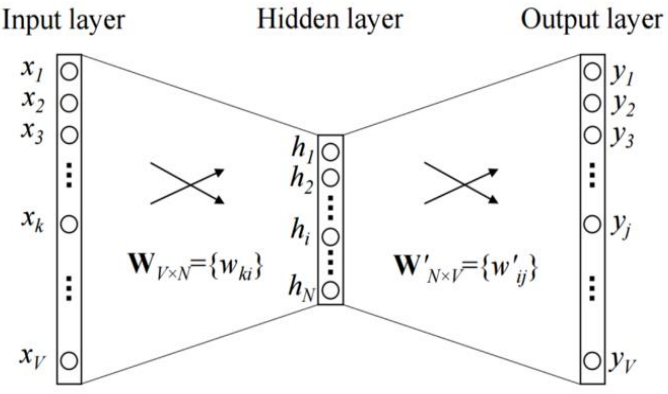


Figure 1: A simple CBOW model with only one word in the context

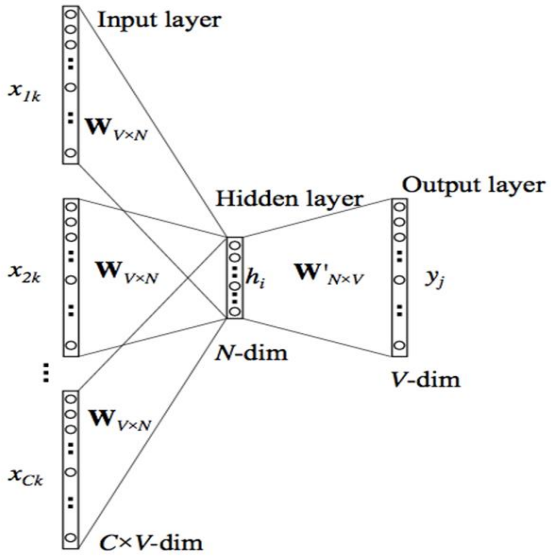
The input or the context word is a one hot encoded vector of size V. The hidden layer contains N neurons and the output is again a V length vector with the elements being the softmax values.

Let's get the terms in the picture right:

- Wvn is the weight matrix that maps the input x to the hidden layer (V*N dimensional matrix)
- W`nv is the weight matrix that maps the hidden layer outputs to the final output layer (N*V dimensional matrix)

I won't get into the mathematics. We'll just get an idea of what's going on.

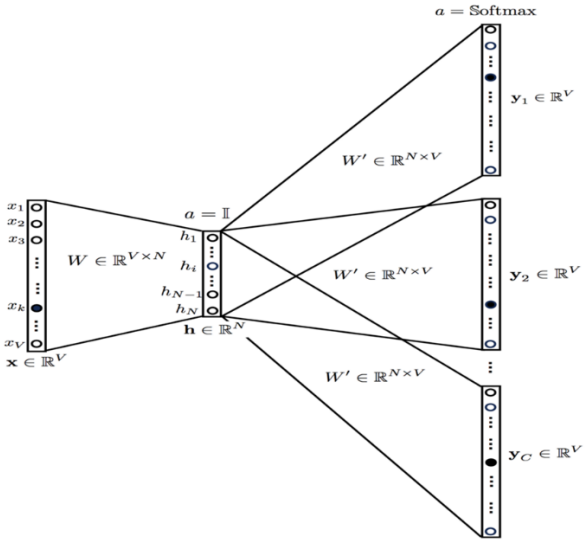
The hidden layer neurons just copy the weighted sum of inputs to the next layer. There is no activation like sigmoid, tanh or ReLU. The only non-linearity is the softmax calculations in the output layer. But, the above model used a single context word to predict the target. We can use multiple context words to do the same.



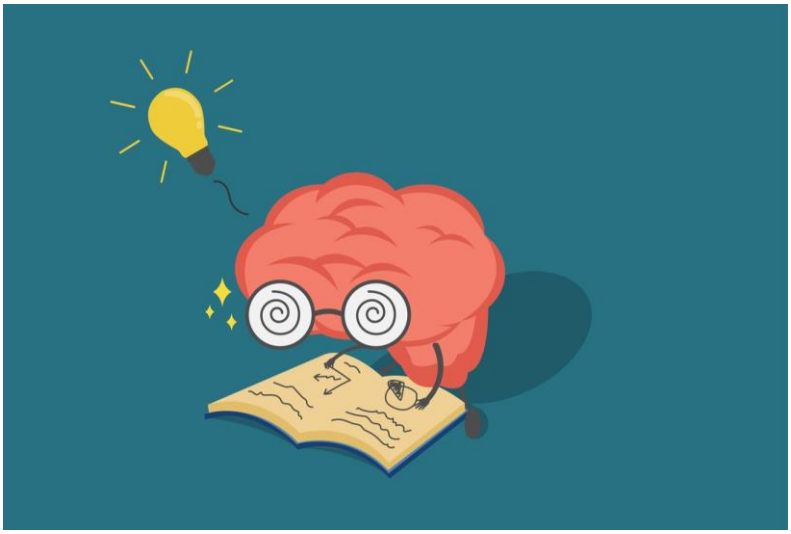
The above model takes C context words. When Wvn is used to calculate hidden layer inputs, we take an average over all these C context word inputs.

So, we have seen how word representations are generated using the context words. But there's one more way we can do the same. We can use the target word (whose representation we want to generate) to predict the context and in the process, we produce the representations. Another variant, called Skip Gram model does this.

Skip-Gram model



This looks like multiple-context CBOW model just got flipped. To some extent that is true. We input the target word into the network. The model outputs C probability distributions. What does this mean? For each context position, we get C probability distributions of V probabilities, one for each word. In both the cases, the network uses back-propagation to learn.



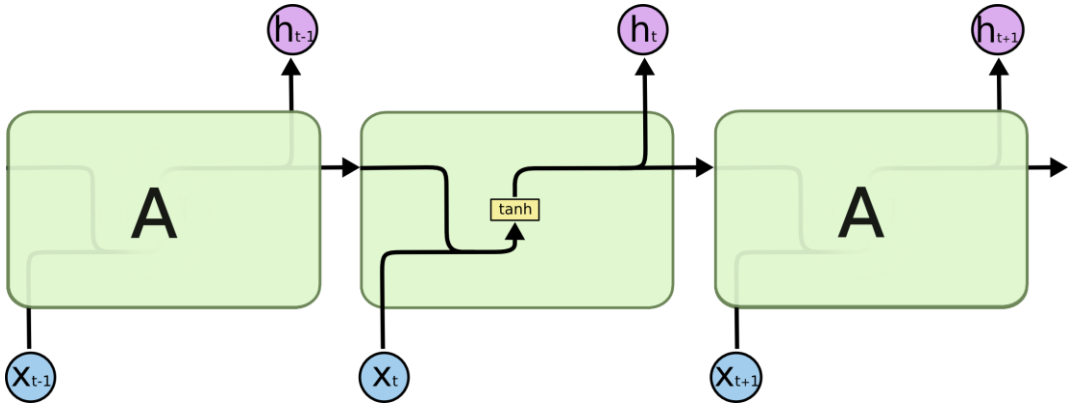
LSTM

Introduction

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter & Schmidhuber (1997), and were refined and popularized by many people in following work.1 They work tremendously well on a large variety of problems, and are now widely used.

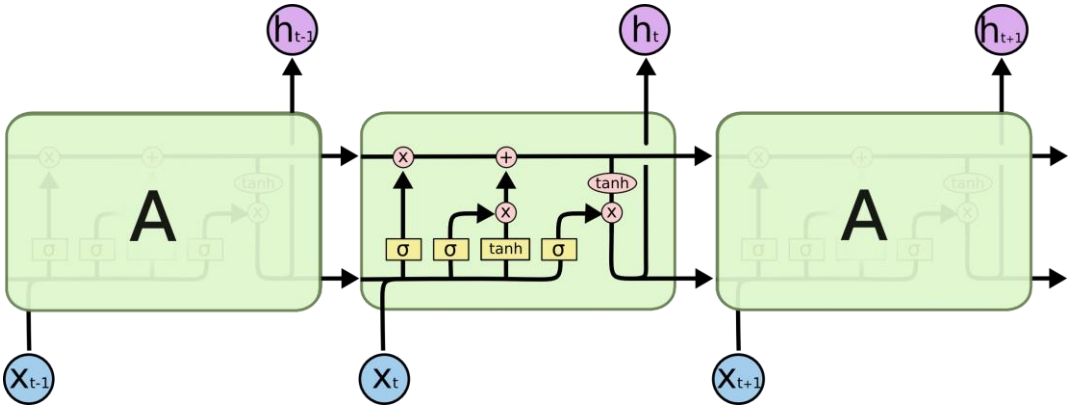
LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.



The repeating module in a standard RNN contains a single layer.

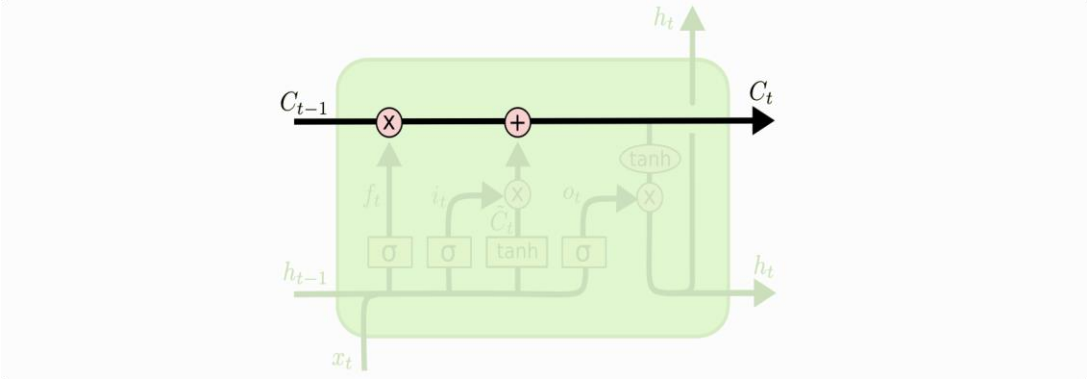
LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.



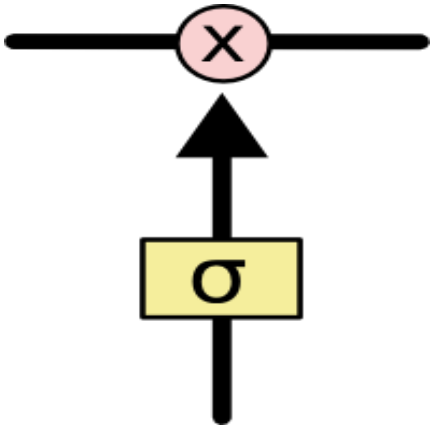
The repeating module in an LSTM contains four interacting layers.

The Core Idea Behind LSTMs

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



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 net layer and a pointwise multiplication operation.

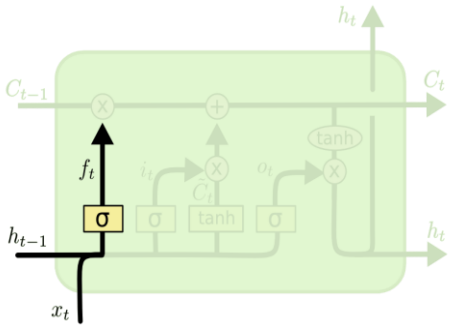


The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of zero means "let nothing through," while a value of one means "let everything through!". An LSTM has three of these gates, to protect and control the cell state.

Step-by-Step LSTM Walk Through

The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the "forget gate layer." It looks at ht−1ht−1 and xtxt, and outputs a number between 00 and 11 for each number in the cell state Ct−1Ct−1. A 11 represents "completely keep this" while a 00 represents "completely get rid of this."

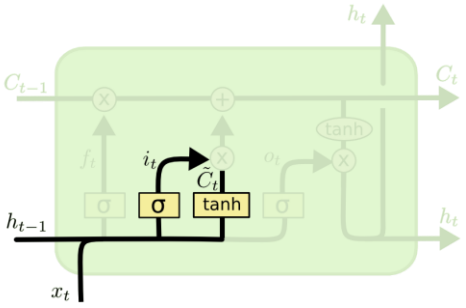
Let's go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the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



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

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the "input gate layer" decides which values we'll update. Next, a tanh layer creates a vector of new candidate values, C~tC~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.



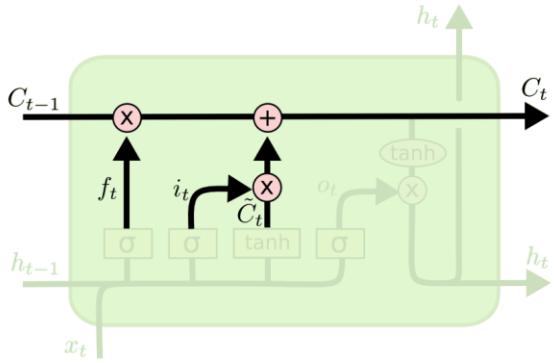
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

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

It's now time to update the old cell state, Ct−1Ct−1, into the new cell state CtCt. The previous steps already decided what to do, we just need to actually do it.

We multiply the old state by ftft, forgetting the things we decided to forget earlier. Then we add it∗C~tit∗C~t. This is the new candidate values, scaled by how much we decided to update each state value.

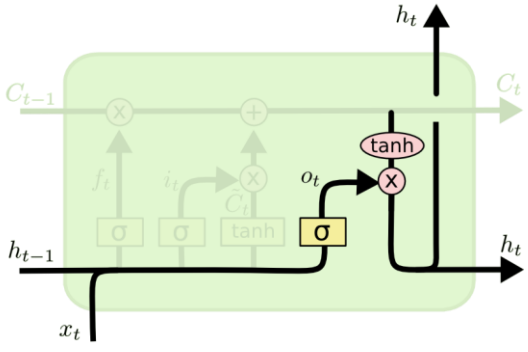
In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps.



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

Finally, we need to decide what we're going to output. 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 tanhtanh (to push the values to be between −1−1 and 11) 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.



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

$$h_t = o_t * \tanh\left(C_t\right)$$

RANDOM FOREST ALGORITHM

Introduction

Random forest is a Supervised Machine Learning Algorithm that is used widely in Classification and Regression problems. It builds decision trees on different samples and takes their majority vote for classification and average in case of regression.

One of the most important features of the Random Forest Algorithm is that it can handle the data set containing continuous variables as in the case of regression and categorical variables as in the case of classification. It performs better results for classification problems.

Real Life Analogy

Let's dive into a real-life analogy to understand this concept further. A student named X wants to choose a course after his 10+2, and he is confused about the choice of course based on his skill set. So he decides to consult various people like his cousins, teachers, parents, degree students, and working people. He asks them varied questions like why he should choose, job opportunities with that course, course fee, etc. Finally, after consulting various people about the course he decides to take the course suggested by most of the people.

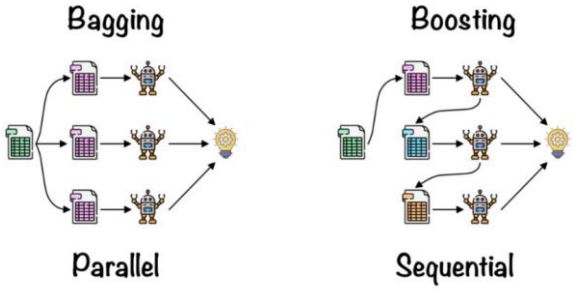
Working of Random Forest Algorithm

Before understanding the working of the random forest we must look into the ensemble technique. Ensemble simply means combining multiple models. Thus a collection of models is used to make predictions rather than an individual model.

Ensemble uses two types of methods:

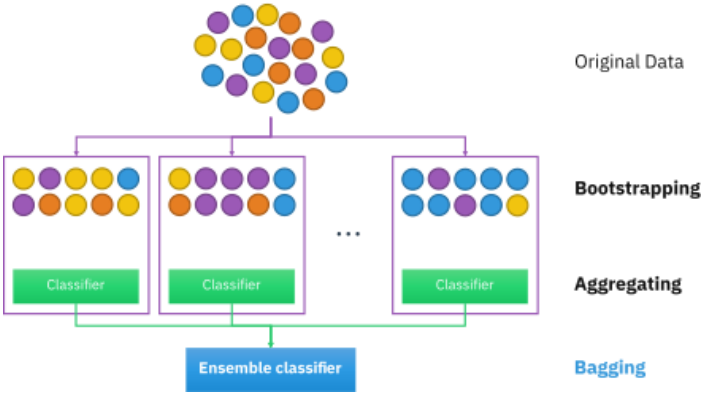
1. **Bagging**– It creates a different training subset from sample training data with replacement & the final output is based on majority voting. For example, Random Forest.

2. **Boosting**– It combines weak learners into strong learners by creating sequential models such that the final model has the highest accuracy. For example, ADA BOOST, XG BOOST

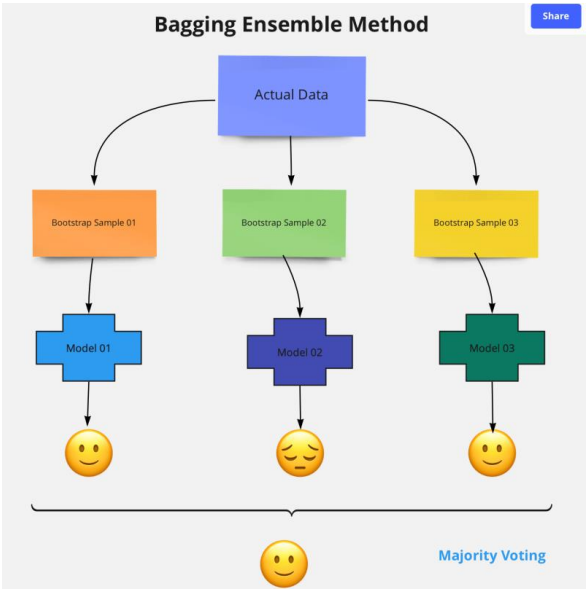


Bagging

Bagging, also known as Bootstrap Aggregation is the ensemble technique used by random forest. Bagging chooses a random sample from the data set. Hence each model is generated from the samples (Bootstrap Samples) provided by the Original Data with replacement known as row sampling. This step of row sampling with replacement is called bootstrap. Now each model is trained independently which generates results. The final output is based on majority voting after combining the results of all models. This step which involves combining all the results and generating output based on majority voting is known as aggregation.

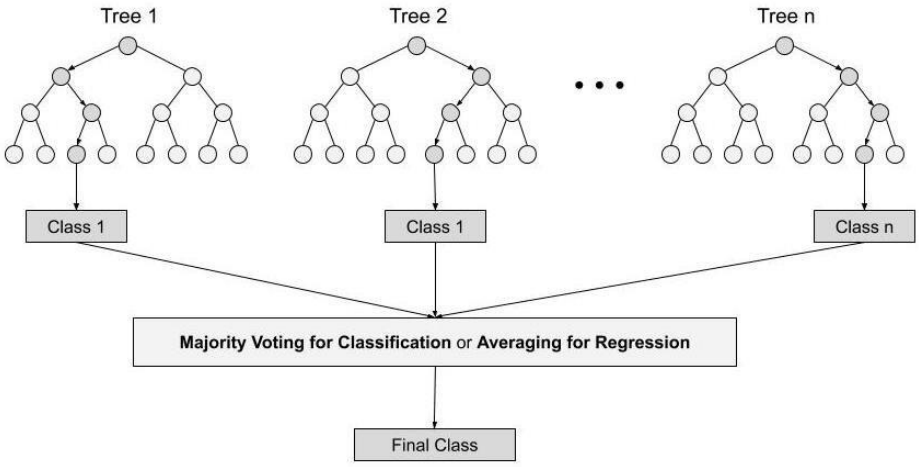


Now let's look at an example by breaking it down with the help of the following figure. Here the bootstrap sample is taken from actual data (Bootstrap sample 01, Bootstrap sample 02, and Bootstrap sample 03) with a replacement which means there is a high possibility that each sample won't contain unique data. Now the model (Model 01, Model 02, and Model 03) obtained from this bootstrap sample is trained independently. Each model generates results as shown. Now Happy emoji is having a majority when compared to sad emoji. Thus based on majority voting final output is obtained as Happy emoji.



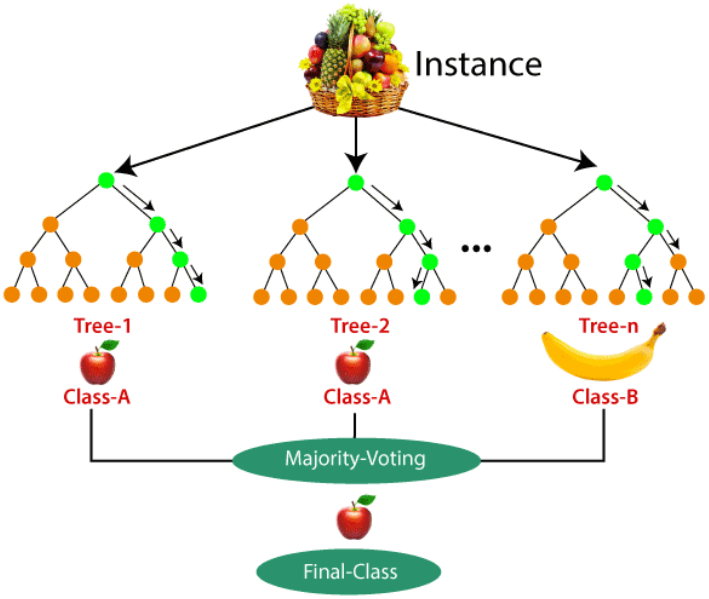
Steps involved in random forest algorithm

- Step 1: In Random forest n number of random records are taken from the data set having k number of records.
- Step 2: Individual decision trees are constructed for each sample.
- Step 3: Each decision tree will generate an output.
- Step 4: Final output is considered based on Majority Voting or Averaging for Classification and regression respectively.



For example:

Consider the fruit basket as the data as shown in the figure below. Now n number of samples are taken from the fruit basket and an individual decision tree is constructed for each sample. Each decision tree will generate an output as shown in the figure. The final output is considered based on majority voting. In the below figure you can see that the majority decision tree gives output as an apple when compared to a banana, so the final output is taken as an apple.



Important Features of Random Forest

1. Diversity- Not all attributes/variables/features are considered while making an individual tree, each tree is different.

2. Immune to the curse of dimensionality- Since each tree does not consider all the features, the feature space is reduced.

3. Parallelization-Each tree is created independently out of different data and attributes. This means that we can make full use of the CPU to build random forests.

4. Train-Test split- In a random forest we don't have to segregate the data for train and test as there will always be 30% of the data which is not seen by the decision tree.

5. Stability- Stability arises because the result is based on majority voting/ averaging.

Difference Between Decision Tree & Random Forest

Random forest is a collection of decision trees; still, there are a lot of differences in their behavior.

Decision trees	Random Forest
1. Decision trees normally suffer from the problem of overfitting if it's allowed to grow without any control.	1. Random forests are created from subsets of data and the final output is based on average or majority ranking and hence the problem of overfitting is taken care of.
2. A single decision tree is faster in computation.	2. It is comparatively slower.
3. When a data set with features is taken as input by a decision tree it will formulate some set of rules to do prediction.	3. Random forest randomly selects observations, builds a decision tree and the average result is taken. It doesn't use any set of formulas.

Thus random forests are much more successful than decision trees only if the trees are diverse and acceptable.

Important Hyper parameters

Hyperparameters are used in random forests to either enhance the performance and predictive power of models or to make the model faster.

Following hyperparameters increases the predictive power:

1**. n_estimators**– number of trees the algorithm builds before averaging the predictions.

2. **max_feature**s– maximum number of features random forest considers splitting a node.

3. **mini_sample_leaf**– determines the minimum number of leaves required to split an internal node.

Following hyperparameters increases the speed:

1. **n_jobs**– it tells the engine how many processors it is allowed to use. If the value is 1, it can use only one processor but if the value is -1 there is no limit.

2. **random_state**– controls randomness of the sample. The model will always produce the same results if it has a definite value of random state and if it has been given the same hyperparameters and the same training data.

3. **oob_score** – OOB means out of the bag. It is a random forest cross-validation method. In this one-third of the sample is not used to train the data instead used to evaluate its performance. These samples are called out of bag samples.

Use Cases

This algorithm is widely used in E-commerce, banking, medicine, the stock market, etc. For example: In the Banking industry it can be used to find which customer will default on the loan.

Advantages and Disadvantages of Random Forest Algorithm

Advantages

1. It can be used in classification and regression problems.

2. It solves the problem of overfitting as output is based on majority voting or averaging.

3. It performs well even if the data contains null/missing values.

4. Each decision tree created is independent of the other thus it shows the property of parallelization.

5. It is highly stable as the average answers given by a large number of trees are taken.

6. It maintains diversity as all the attributes are not considered while making each decision tree though it is not true in all cases.

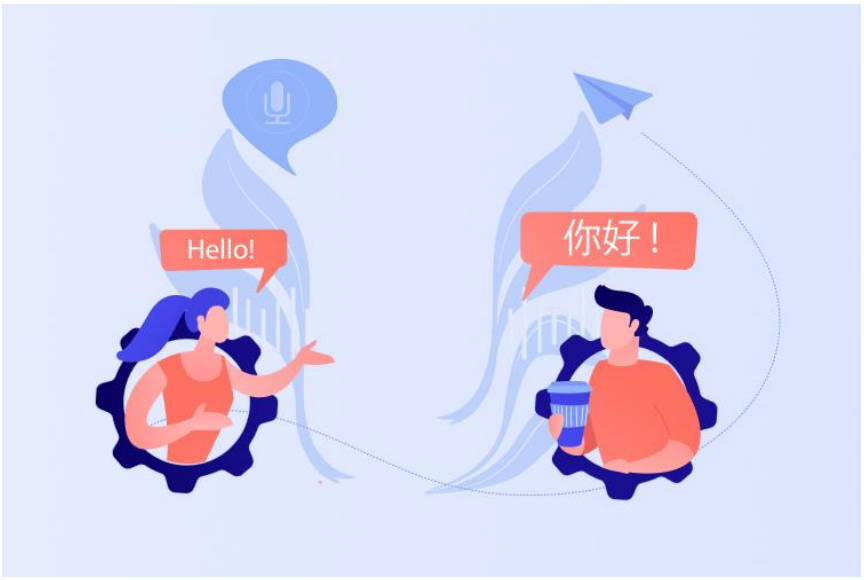
7. It is immune to the curse of dimensionality. Since each tree does not consider all the attributes, feature space is reduced.

8. We don't have to segregate data into train and test as there will always be 30% of the data which is not seen by the decision tree made out of bootstrap.

Disadvantages

1. Random forest is highly complex when compared to decision trees where decisions can be made by following the path of the tree.

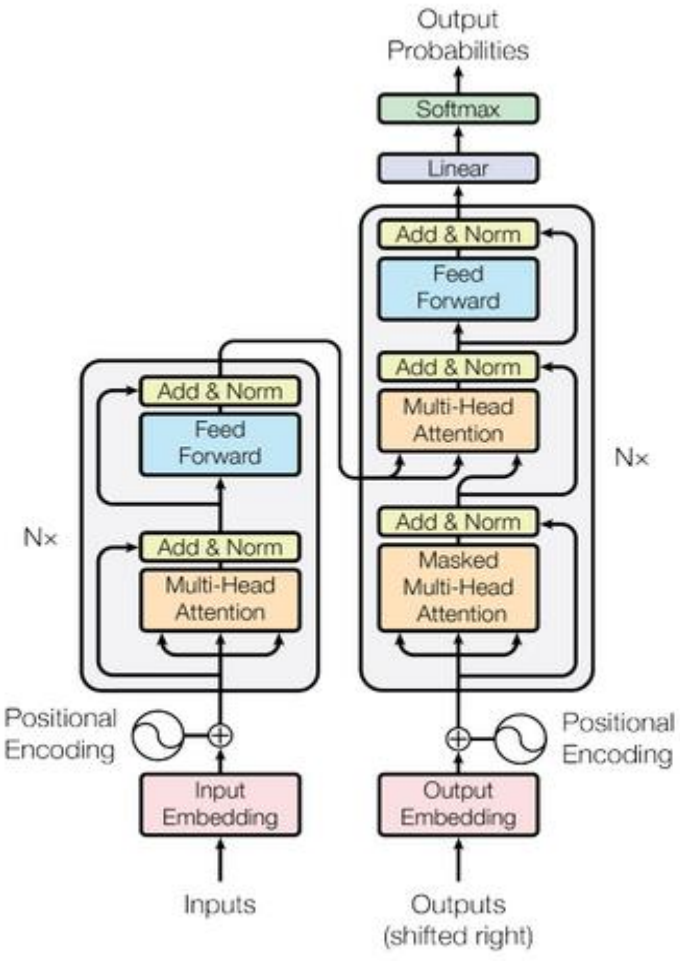
2. Training time is more compared to other models due to its complexity. Whenever it has to make a prediction each decision tree has to generate output for the given input data.



TRANSFORMERS

Introduction

It may seem like a long time since the world of natural language processing (NLP) was transformed by the seminal "Attention is All You Need" paper by Vaswani et al. The relative recency of the introduction of transformer architectures and the ubiquity with which they have upended language tasks speaks to the rapid rate of progress in machine learning and artificial intelligence. There's no better time than now to gain a deep understanding of the inner workings of transformer architectures, especially with transformer models making big inroads into diverse new applications like predicting chemical reactions and reinforcement learning.



What do Transformers do?

Transformers are the current state-of-the-art type of model for dealing with sequences. Perhaps the most prominent application of these models is in text processing tasks, and the most prominent of these is machine translation. In fact, transformers and their conceptual progeny have infiltrated just about every benchmark leaderboard in natural language processing (NLP), from question answering to grammar correction. In many ways, transformer architectures are undergoing a surge in development similar to what we saw with convolutional neural networks following the 2012 ImageNet competition, for better and for worse.

The Transformer represented as a black box. An entire sequence is parsed simultaneously in a feed-forward manner, producing a transformed output tensor. In this diagram, the output sequence is more concise than the input sequence. For practical NLP tasks, word order and sentence length may vary substantially.

Unlike previous state-of-the-art architectures for NLP, such as the many variants of RNNs and LSTMs, there are no recurrent connections and thus no real memory of previous states. Transformers get around this lack of memory by perceiving entire sequences simultaneously. Perhaps a transformer neural network perceives the world a bit like the aliens in the movie Arrival. Strictly speaking, the future elements are usually masked out during training, but other than that, the model is free to learn long-term semantic dependencies throughout the entire sequence.

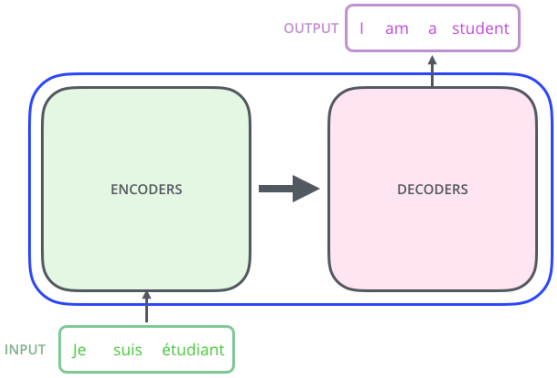
Operating as feed-forward-only models, transformers require a slightly different approach to hardware. Transformers are actually much better suited to run on modern machine learning accelerators because, unlike recurrent networks, there is no sequential processing: the model doesn't have to process a string of elements in order to develop a useful hidden cell state. Transformers can require a lot of memory during training, but running training or inference at reduced precision can help to alleviate memory requirements.

Transfer learning is an important shortcut to state-of-the-art performance on a given text-based task, and, quite frankly, necessary for most practitioners on realistic budgets. Energy and financial costs of training a large modern transformer can easily dwarf an individual researcher's total yearly energy consumption at the cost of thousands of dollars if using cloud compute. Luckily, similar to deep learning for computer vision, the new skills needed for a specialized task can be transferred to large pre-trained transformes/

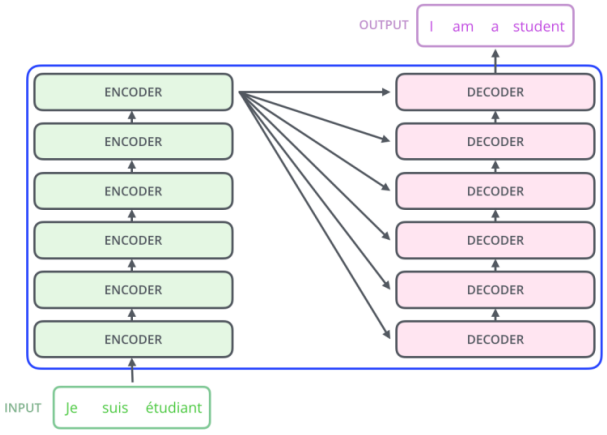
A High-Level Look

Let's begin by looking at the model as a single black box. In a machine translation application, it would take a sentence in one language, and output its translation in another.

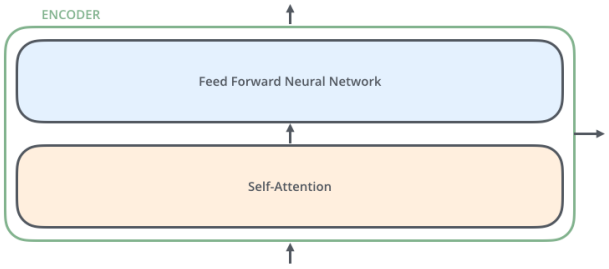
Popping open that Transformer, we see an encoding component, a decoding component, and connections between them.



The encoding component is a stack of encoders. The decoding component is a stack of decoders of the same number.



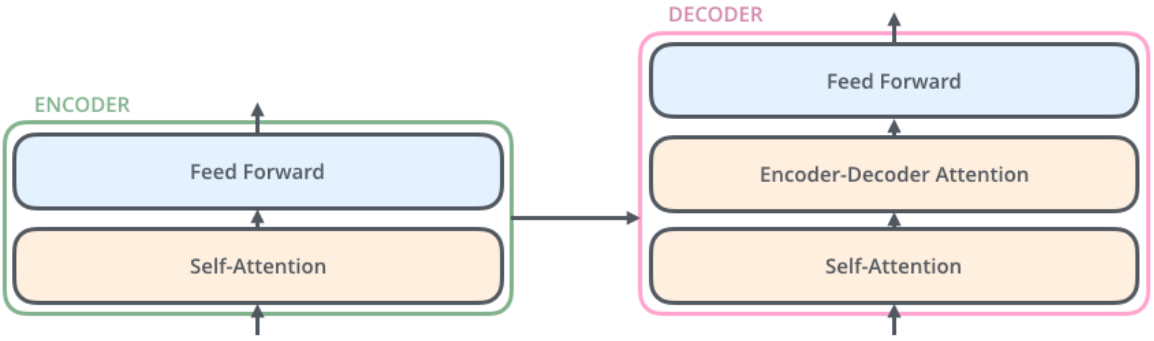
The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sub-layers:



The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. We'll look closer at self-attention later in the post.

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence.



Bringing The Tensors into The Picture

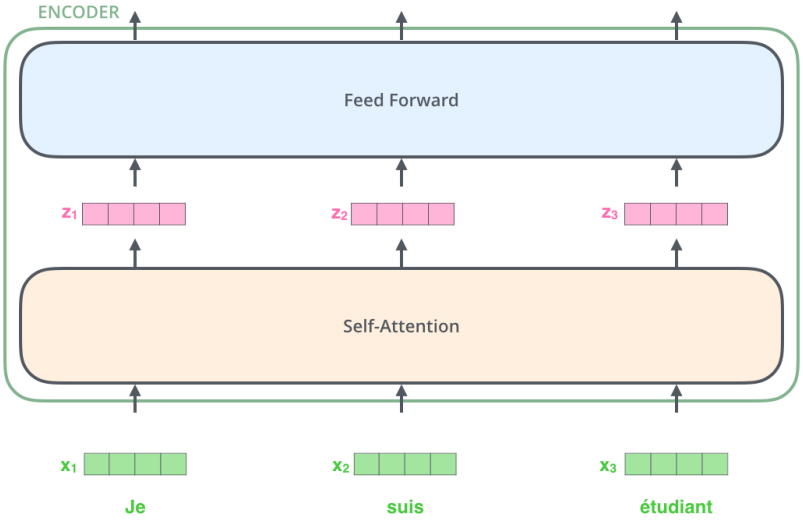
Now that we've seen the major components of the model, let's start to look at the various vectors/tensors and how they flow between these components to turn the input of a trained model into an output.

As is the case in NLP applications in general, we begin by turning each input word into a vector using an embedding algorithm.

Each word is embedded into a vector of size 512. We'll represent those vectors with these simple boxes.

The embedding only happens in the bottom-most encoder. The abstraction that is common to all the encoders is that they receive a list of vectors each of the size 512 – In the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below. The size of this list is hyperparameter we can set – basically it would be the length of the longest sentence in our training dataset.

After embedding the words in our input sequence, each of them flows through each of the two layers of the encoder.

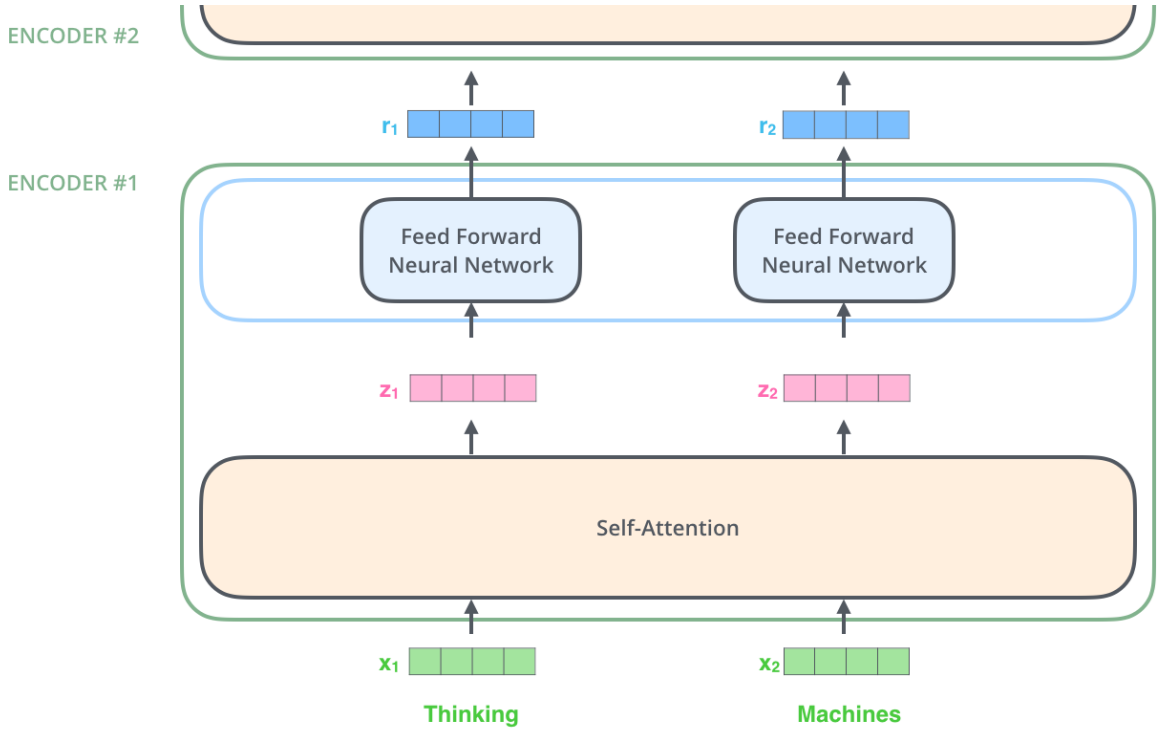


Here we begin to see one key property of the Transformer, which is that the word in each position flows through its own path in the encoder. There are dependencies between these paths in the self-attention layer. The feed-forward layer does not have those dependencies, however, and thus the various paths can be executed in parallel while flowing through the feed-forward layer.

Next, we'll switch up the example to a shorter sentence and we'll look at what happens in each sub-layer of the encoder.

Now We're Encoding!

As we've mentioned already, an encoder receives a list of vectors as input. It processes this list by passing these vectors into a 'self-attention' layer, then into a feed-forward neural network, then sends out the output upwards to the next encoder.



The word at each position passes through a self-attention process. Then, they each pass through a feed-forward neural network -- the exact same network with each vector flowing through it separately.

Self-Attention at a High Level

Say the following sentence is an input sentence we want to translate:

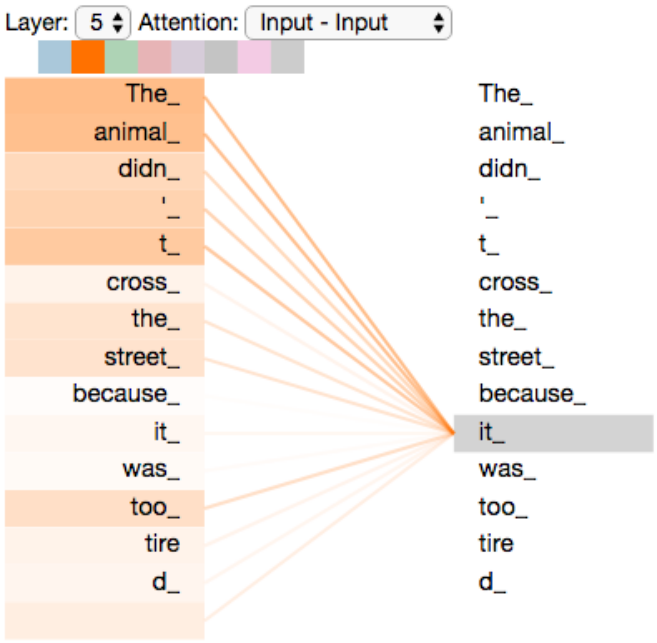
"The animal didn't cross the street because it was too tired"

What does "it" in this sentence refer to? Is it referring to the street or to the animal? It's a simple question to a human, but not as simple to an algorithm.

When the model is processing the word "it", self-attention allows it to associate "it" with "animal".

As the model processes each word (each position in the input sequence), self-attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.

If you're familiar with RNNs, think of how maintaining a hidden state allows an RNN to incorporate its representation of previous words/vectors it has processed with the current one it's processing. Self-attention is the method the Transformer uses to bake the "understanding" of other relevant words into the one we're currently processing.



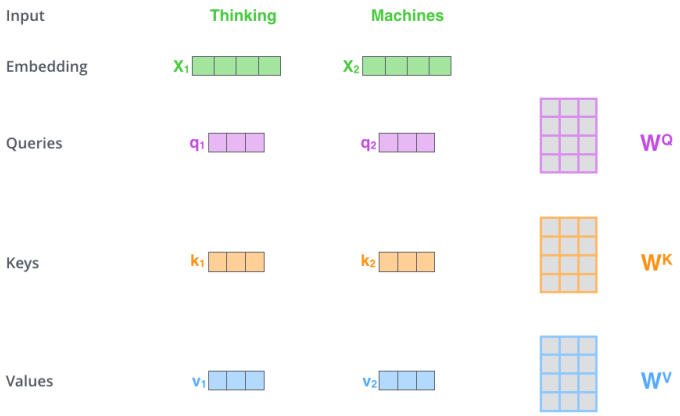
As we are encoding the word "it" in encoder #5 (the top encoder in the stack), part of the attention mechanism was focusing on "The Animal", and baked a part of its representation into the encoding of "it".

Self-Attention in Detail

Let's first look at how to calculate self-attention using vectors, then proceed to look at how it's actually implemented – using matrices.

The first step in calculating self-attention is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word). So for each word, we create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

Notice that these new vectors are smaller in dimension than the embedding vector. Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512. They don't HAVE to be smaller, this is an architecture choice to make the computation of multiheaded attention (mostly) constant.



Multiplying x1 by the WQ weight matrix produces q1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

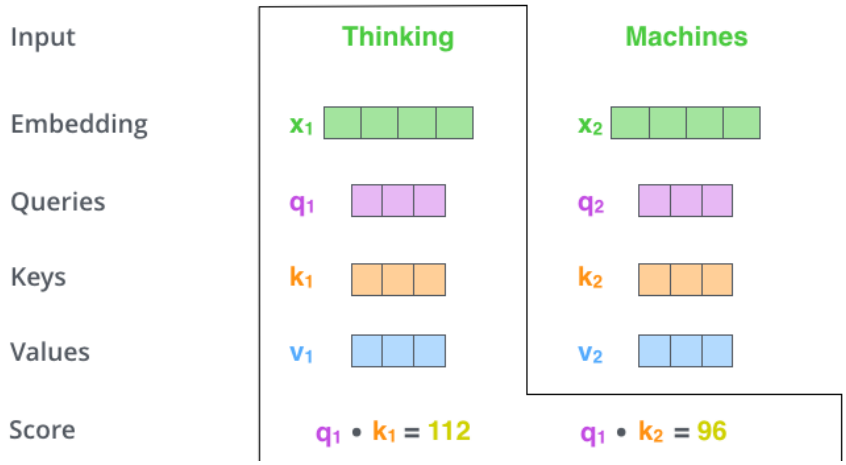
What are the "query", "key", and "value" vectors?

They're abstractions that are useful for calculating and thinking about attention. Once you proceed with reading how attention is calculated below, you'll know pretty much all you need to know about the role each of these vectors plays.

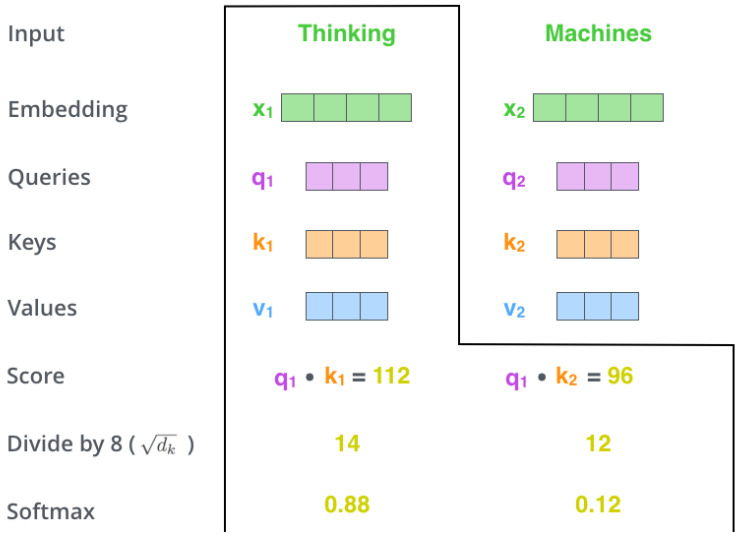
The second step in calculating self-attention is to calculate a score. Say we're calculating the self-attention for the first word in this example, "Thinking". We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring. So if we're processing the self-attention for the word in position #1, the

first score would be the dot product of q1 and k1. The second score would be the dot product of q1 and k2.



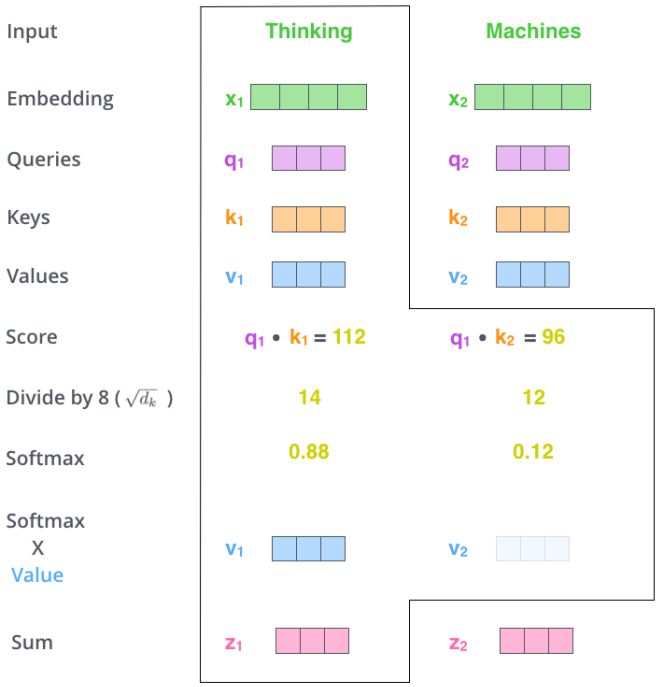
The third and forth steps are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a softmax operation. Softmax normalizes the scores so they're all positive and add up to 1.



This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.

The fifth step is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

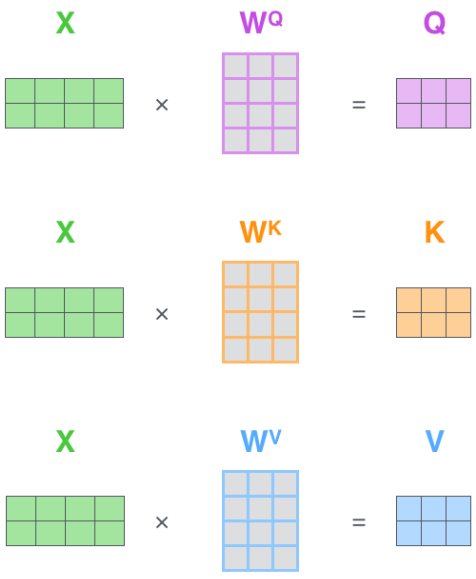
The sixth step is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).



That concludes the self-attention calculation. The resulting vector is one we can send along to the feed-forward neural network. In the actual implementation, however, this calculation is done in matrix form for faster processing. So let's look at that now that we've seen the intuition of the calculation on the word level.

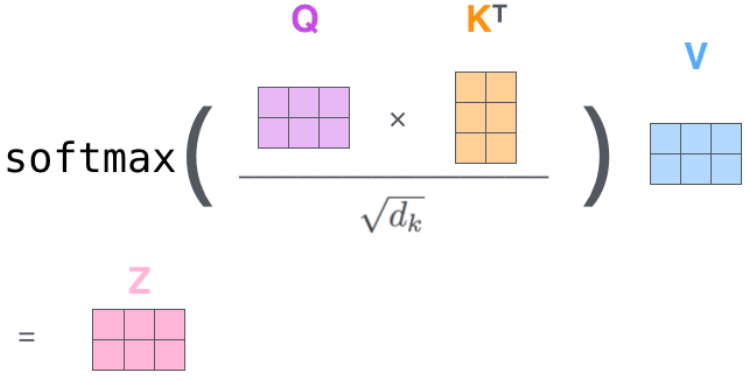
Matrix Calculation of Self-Attention

The first step is to calculate the Query, Key, and Value matrices. We do that by packing our embeddings into a matrix X, and multiplying it by the weight matrices we've trained (WQ, WK, WV).



Every row in the X matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector (512, or 4 boxes in the figure), and the q/k/v vectors (64, or 3 boxes in the figure)

Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.



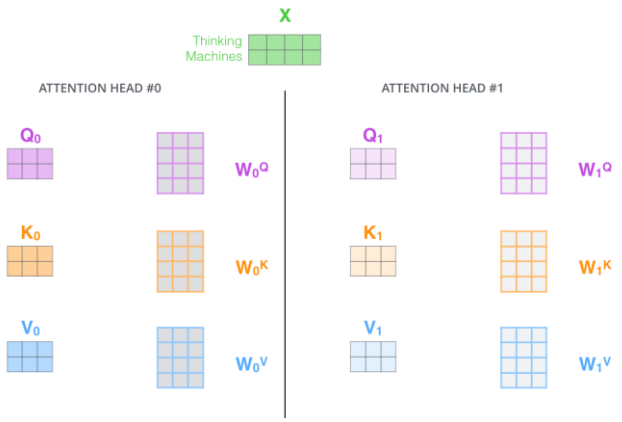
The self-attention calculation in matrix form

Multi headed attention

The paper further refined the self-attention layer by adding a mechanism called "multi-headed" attention. This improves the performance of the attention layer in two ways:

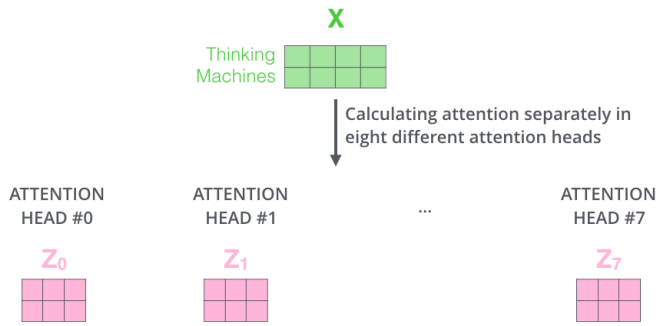
1. It expands the model's ability to focus on different positions. Yes, in the example above, z1 contains a little bit of every other encoding, but it could be dominated by the the actual word itself. It would be useful if we're translating a sentence like "The animal didn't cross the street because it was too tired", we would want to know which word "it" refers to.

2. It gives the attention layer multiple "representation subspaces". As we'll see next, with multi-headed attention we have not only one, but multiple sets of Query/Key/Value weight matrices (the Transformer uses eight attention heads, so we end up with eight sets for each encoder/decoder). Each of these sets is randomly initialized. Then, after training, each set is used to project the input embeddings (or vectors from lower encoders/decoders) into a different representation subspace.



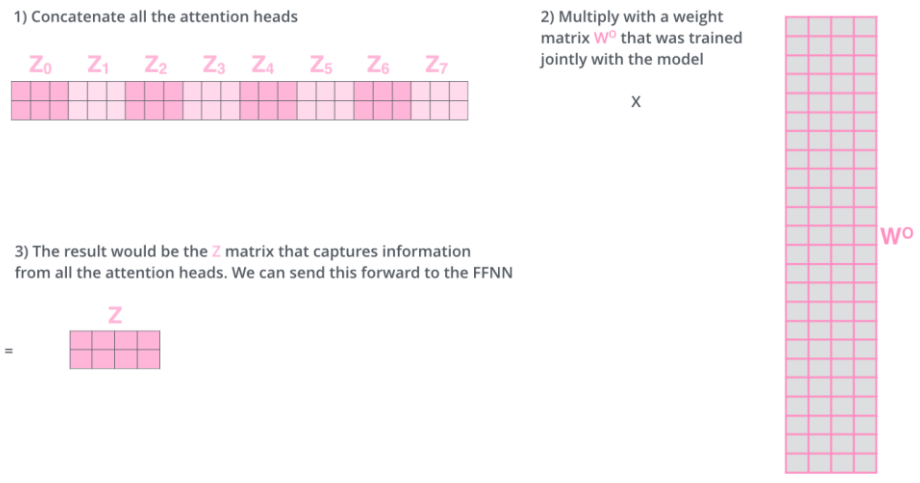
With multi-headed attention, we maintain separate Q/K/V weight matrices for each head resulting in different Q/K/V matrices. As we did before, we multiply X by the WQ/WK/WV matrices to produce Q/K/V matrices.

If we do the same self-attention calculation we outlined above, just eight different times with different weight matrices, we end up with eight different Z matrices

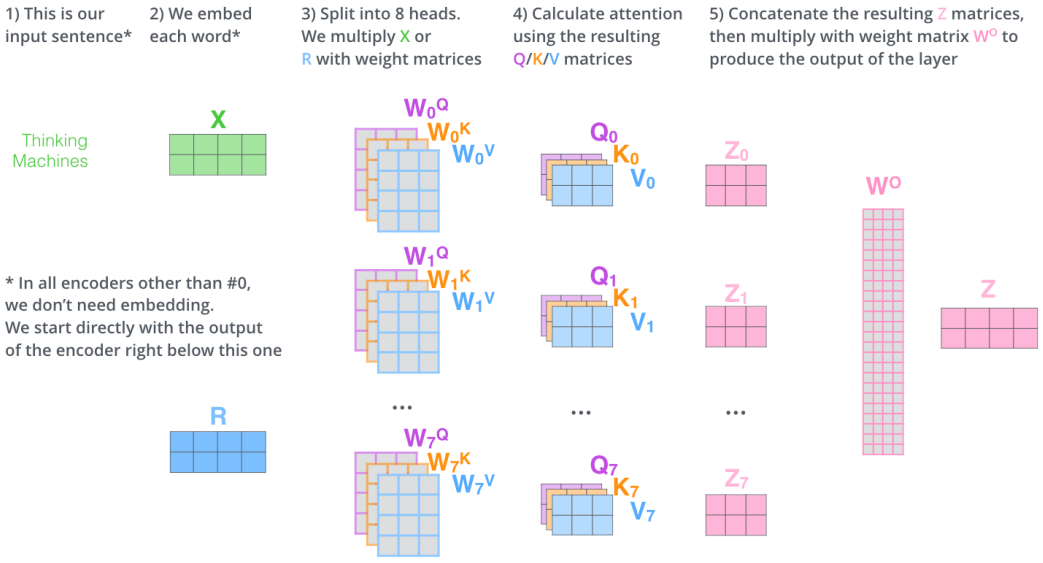


This leaves us with a bit of a challenge. The feed-forward layer is not expecting eight matrices – it's expecting a single matrix (a vector for each word). So we need a way to condense these eight down into a single matrix.

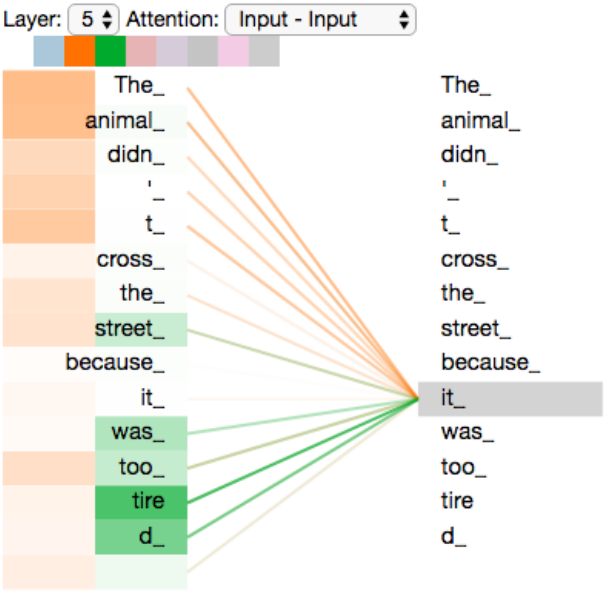
How do we do that? We concat the matrices then multiple them by an additional weights matrix WO.



That's pretty much all there is to multi-headed self-attention. It's quite a handful of matrices, I realize. Let me try to put them all in one visual so we can look at them in one place

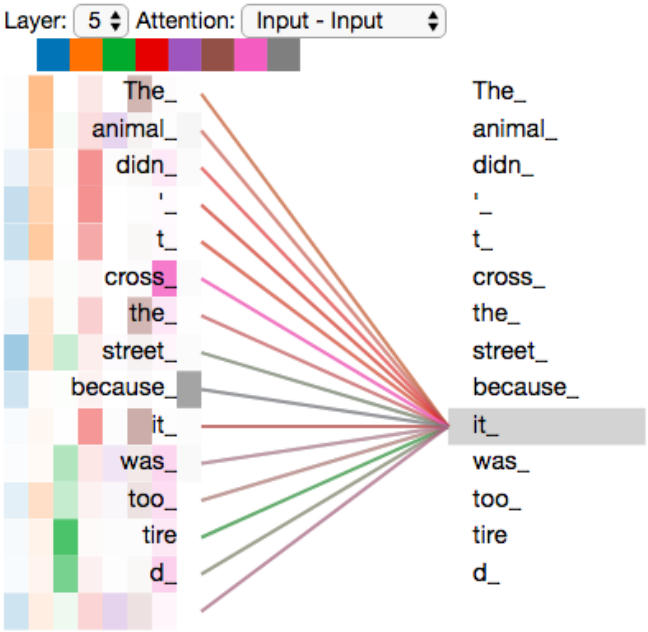


Now that we have touched upon attention heads, let's revisit our example from before to see where the different attention heads are focusing as we encode the word "it" in our example sentence:



As we encode the word "it", one attention head is focusing most on "the animal", while another is focusing on "tired" -- in a sense, the model's representation of the word "it" bakes in some of the representation of both "animal" and "tired".

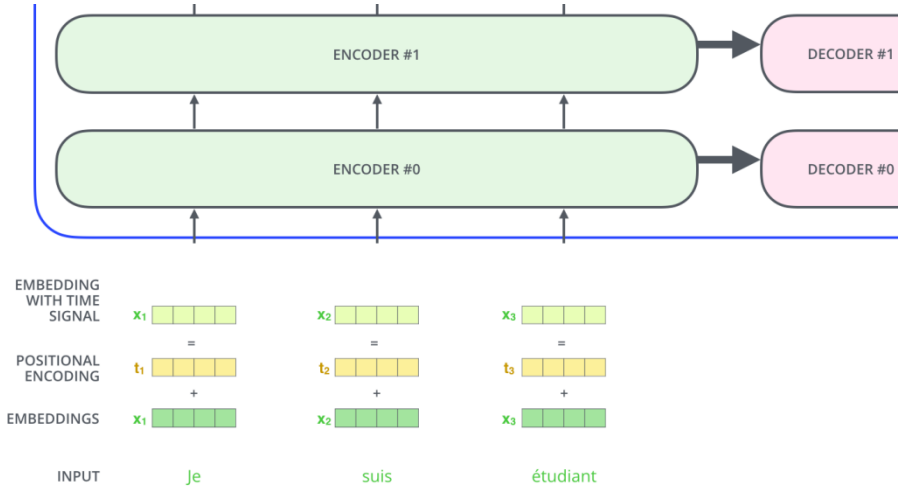
If we add all the attention heads to the picture, however, things can be harder to interpret:



Representing The Order of the Sequence Using Positional Encoding

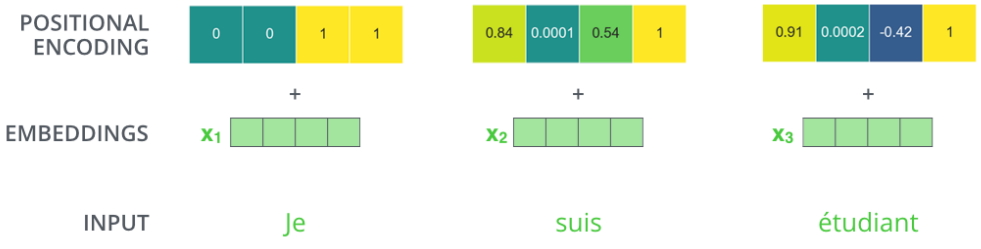
One thing that's missing from the model as we have described it so far is a way to account for the order of the words in the input sequence.

To address this, the transformer adds a vector to each input embedding. These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence. The intuition here is that adding these values to the embeddings provides meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during dot-product attention.



To give the model a sense of the order of the words, we add positional encoding vectors -- the values of which follow a specific pattern.

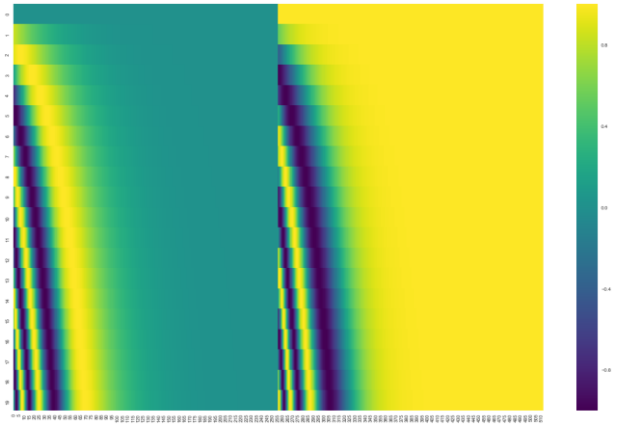
If we assumed the embedding has a dimensionality of 4, the actual positional encodings would look like this:



A real example of positional encoding with a toy embedding size of 4

What might this pattern look like?

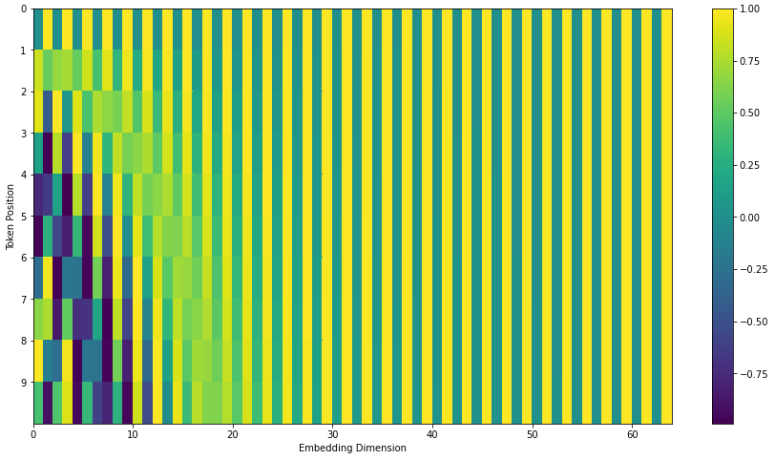
In the following figure, each row corresponds the a positional encoding of a vector. So the first row would be the vector we'd add to the embedding of the first word in an input sequence. Each row contains 512 values – each with a value between 1 and -1. We've color-coded them so the pattern is visible.



A real example of positional encoding for 20 words (rows) with an embedding size of 512 (columns). You can see that it appears split in half down the center. That's because the values of the left half are generated by one function (which uses sine), and the right half is generated by another function (which uses cosine). They're then concatenated to form each of the positional encoding vectors.

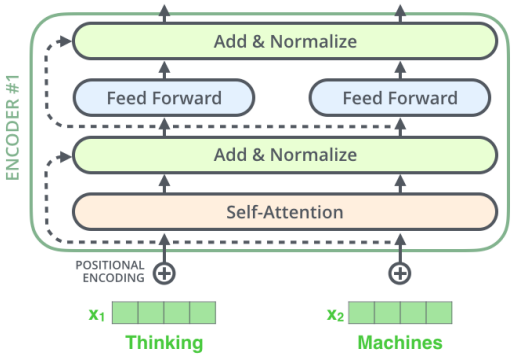
The formula for positional encoding is described in the paper (section 3.5). You can see the code for generating positional encodings in get_timing_signal_1d(). This is not the only possible method for positional encoding. It, however, gives the advantage of being able to scale to unseen lengths of sequences (e.g. if our trained model is asked to translate a sentence longer than any of those in our training set).

July 2020 Update: The positional encoding shown above is from the Tranformer2Transformer implementation of the Transformer. The method shown in the paper is slightly different in that it doesn't directly concatenate, but interweaves the two signals. The following figure shows what that looks like. Here's the code to generate it:

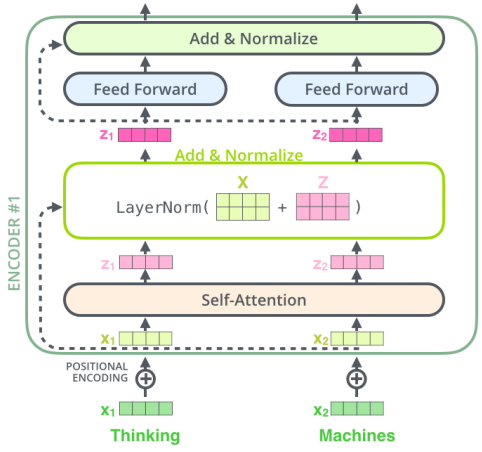


The Residuals

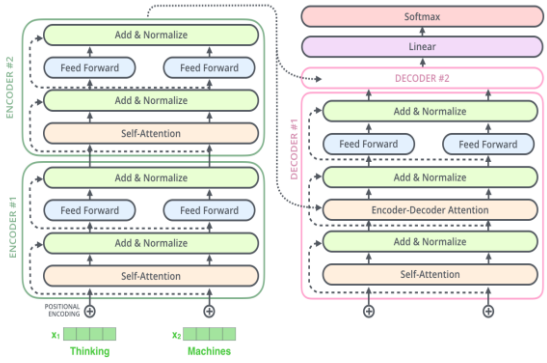
One detail in the architecture of the encoder that we need to mention before moving on, is that each sub-layer (self-attention, ffnn) in each encoder has a residual connection around it, and is followed by a layer-normalization step.



If we're to visualize the vectors and the layer-norm operation associated with self attention, it would look like this:



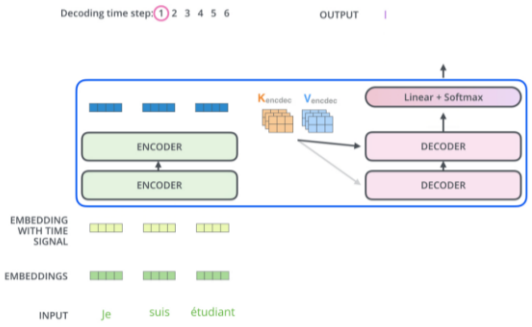
This goes for the sub-layers of the decoder as well. If we're to think of a Transformer of 2 stacked encoders and decoders, it would look something like this:



The Decoder Side

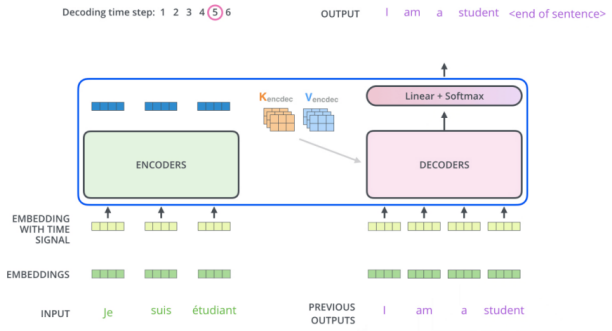
Now that we've covered most of the concepts on the encoder side, we basically know how the components of decoders work as well. But let's take a look at how they work together.

The encoder starts by processing the input sequence. The output of the top encoder is then transformed into a set of attention vectors K and V. These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence:



After finishing the encoding phase, we begin the decoding phase. Each step in the decoding phase outputs an element from the output sequence (the English translation sentence in this case).

The following steps repeat the process until a special symbol is reached indicating the transformer decoder has completed its output. The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did. And just like we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.



The self-attention layers in the decoder operate in a slightly different way than the one in the encoder:

In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to -inf) before the softmax step in the self-attention calculation.

The "Encoder-Decoder Attention" layer works just like multiheaded self-attention, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

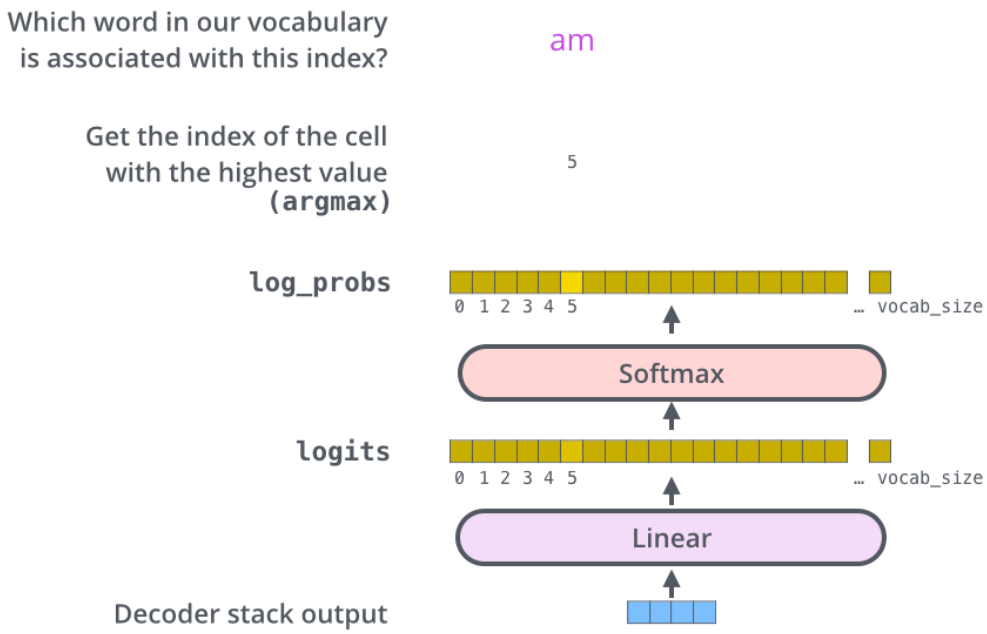
The Final Linear and Softmax Layer

The decoder stack outputs a vector of floats. How do we turn that into a word? That's the job of the final Linear layer which is followed by a Softmax Layer.

The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.

Let's assume that our model knows 10,000 unique English words (our model's "output vocabulary") that it's learned from its training dataset. This would make the logits vector 10,000 cells wide – each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer.

The softmax layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.



This figure starts from the bottom with the vector produced as the output of the decoder stack. It is then turned into an output word.

Recap of Training

Now that we've covered the entire forward-pass process through a trained Transformer, it would be useful to glance at the intuition of training the model.

During training, an untrained model would go through the exact same forward pass. But since we are training it on a labeled training dataset, we can compare its output with the actual correct output.

To visualize this, let's assume our output vocabulary only contains six words("a", "am", "i", "thanks", "student", and "<eos>" (short for 'end of sentence')).

Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

The output vocabulary of our model is created in the preprocessing phase before we even begin training.

Once we define our output vocabulary, we can use a vector of the same width to indicate each word in our vocabulary. This also known as one-hot encoding. So for example, we can indicate the word "am" using the following vector:

Output Vocabulary

WORD	a	am	I	thanks	student	<eos>
INDEX	0	1	2	3	4	5

One-hot encoding of the word "am"

0.0	1.0	0.0	0.0	0.0	0.0

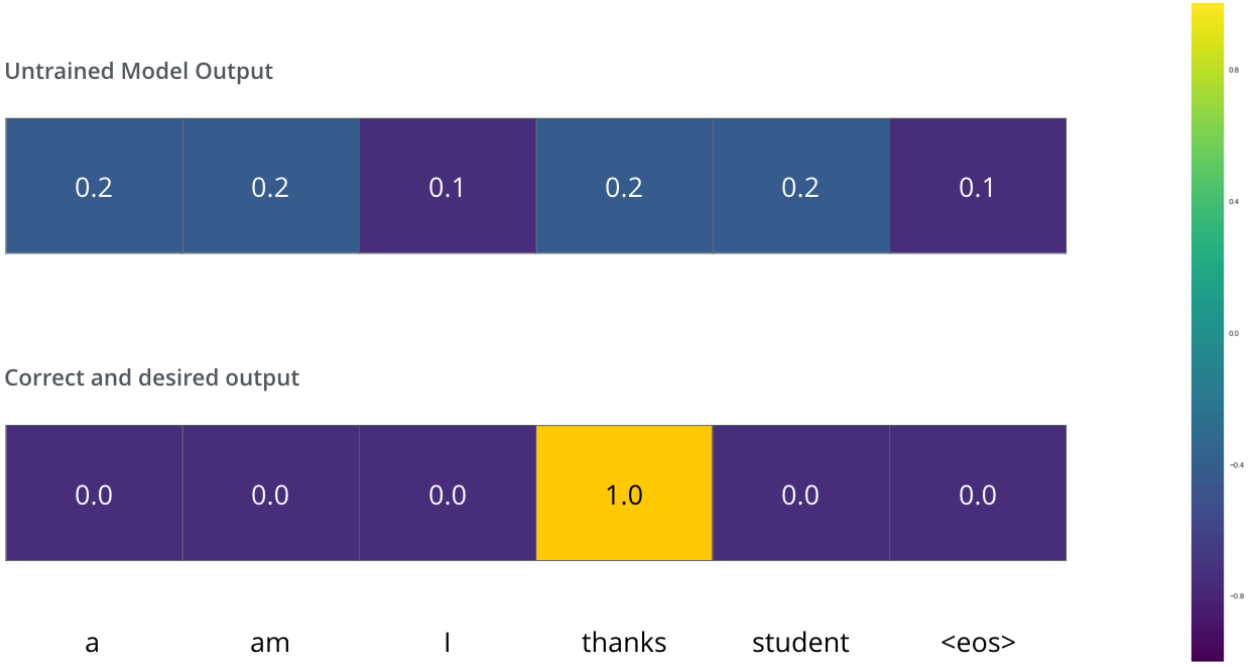
Example: one-hot encoding of our output vocabulary

Following this recap, let's discuss the model's loss function – the metric we are optimizing during the training phase to lead up to a trained and hopefully amazingly accurate model.

The Loss Function

Say we are training our model. Say it's our first step in the training phase, and we're training it on a simple example – translating "merci" into "thanks".

What this means, is that we want the output to be a probability distribution indicating the word "thanks". But since this model is not yet trained, that's unlikely to happen just yet.

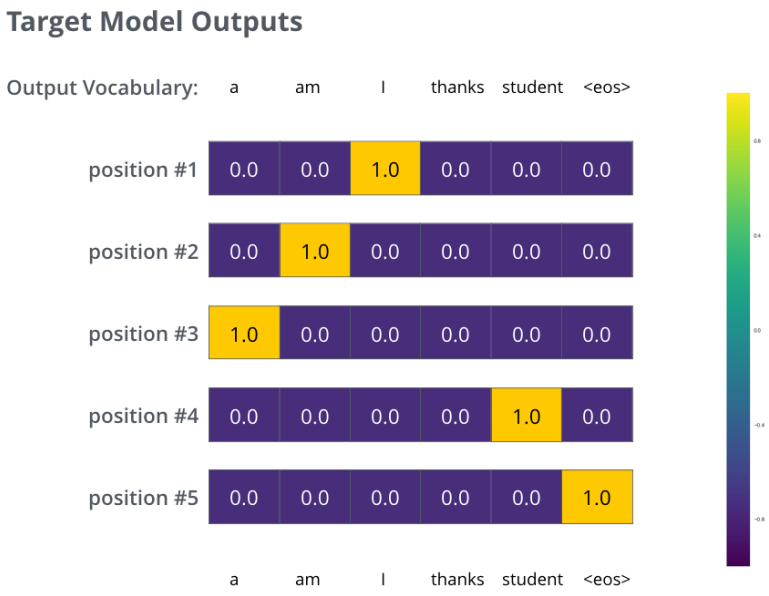


Since the model's parameters (weights) are all initialized randomly, the (untrained) model produces a probability distribution with arbitrary values for each cell/word. We can compare it with the actual output, then tweak all the model's weights using backpropagation to make the output closer to the desired output.

How do you compare two probability distributions? We simply subtract one from the other.

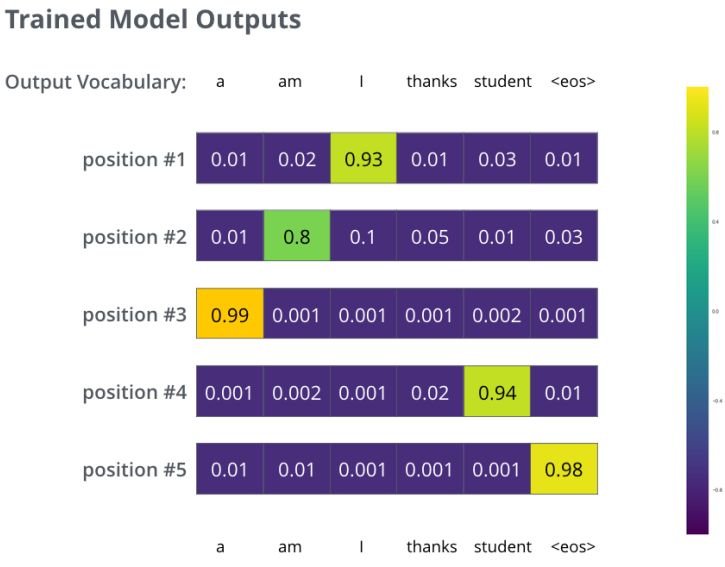
But note that this is an oversimplified example. More realistically, we'll use a sentence longer than one word. For example – input: "je suis étudiant" and expected output: "i am a student". What this really means, is that we want our model to successively output probability distributions where:

- Each probability distribution is represented by a vector of width vocab_size (6 in our toy example, but more realistically a number like 30,000 or 50,000)
- The first probability distribution has the highest probability at the cell associated with the word "i"
- The second probability distribution has the highest probability at the cell associated with the word "am"
- And so on, until the fifth output distribution indicates '<end of sentence>' symbol, which also has a cell associated with it from the 10,000 element vocabulary.



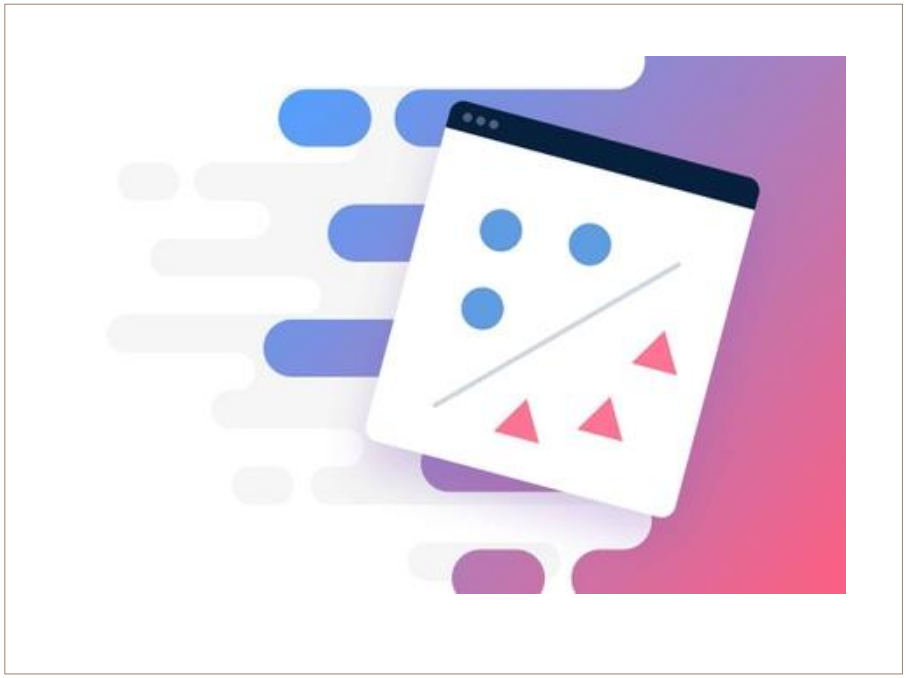
The targeted probability distributions we'll train our model against in the training example for one sample sentence.

After training the model for enough time on a large enough dataset, we would hope the produced probability distributions would look like this:



Hopefully upon training, the model would output the right translation we expect. Of course it's no real indication if this phrase was part of the training dataset (see: cross validation). Notice that every position gets a little bit of probability even if it's unlikely to be the output of that time step -- that's a very useful property of softmax which helps the training process.

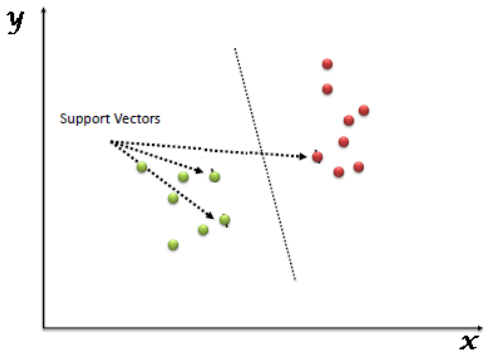
Now, because the model produces the outputs one at a time, we can assume that the model is selecting the word with the highest probability from that probability distribution and throwing away the rest. That's one way to do it (called greedy decoding). Another way to do it would be to hold on to, say, the top two words (say, 'I' and 'a' for example), then in the next step, run the model twice: once assuming the first output position was the word 'I', and another time assuming the first output position was the word 'a', and whichever version produced less error considering both positions #1 and #2 is kept. We repeat this for positions #2 and #3…etc. This method is called "beam search", where in our example, beam_size was two (meaning that at all times, two partial hypotheses (unfinished translations) are kept in memory), and top_beams is also two (meaning we'll return two translations). These are both hyperparameters that you can experiment with.



SUPPORT VECTOR MACHINE

Introduction

"Support Vector Machine" (SVM) is a supervised machine learning algorithm that can be used for both classification or regression challenges. However, it is mostly used in classification problems. In the SVM algorithm, we plot each data item as a point in n-dimensional space (where n is a number of features you have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane that differentiates the two classes very well (look at the below snapshot).



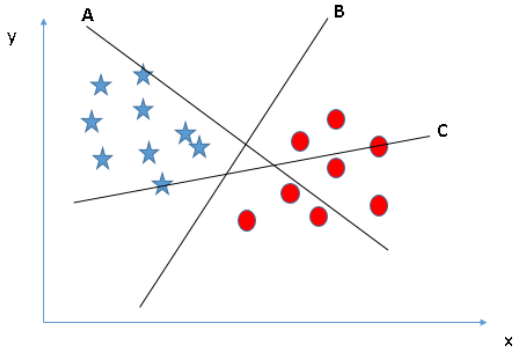
Support Vectors are simply the coordinates of individual observation. The SVM classifier is a frontier that best segregates the two classes (hyper-plane/ line).

How does it work?

Above, we got accustomed to the process of segregating the two classes with a hyper-plane. Now the burning question is "How can we identify the right hyper-plane?". Don't worry, it's not as hard as you think!

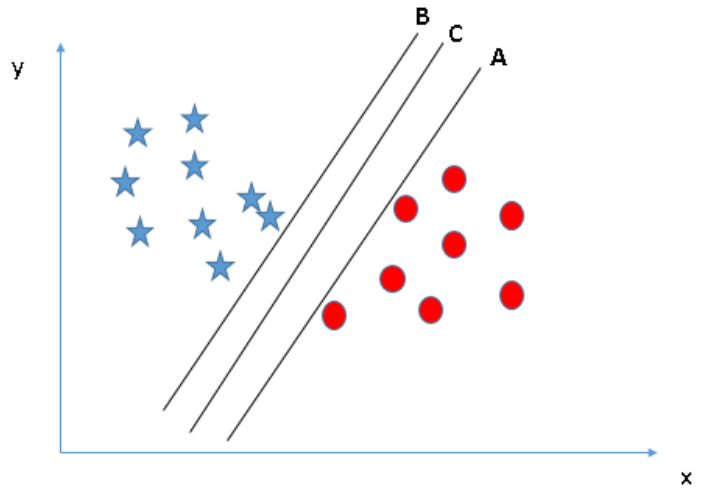
Let's understand:

- Identify the right hyper-plane (Scenario-1): Here, we have three hyper-planes (A, B, and C). Now, identify the right hyper-plane to classify stars and circles.

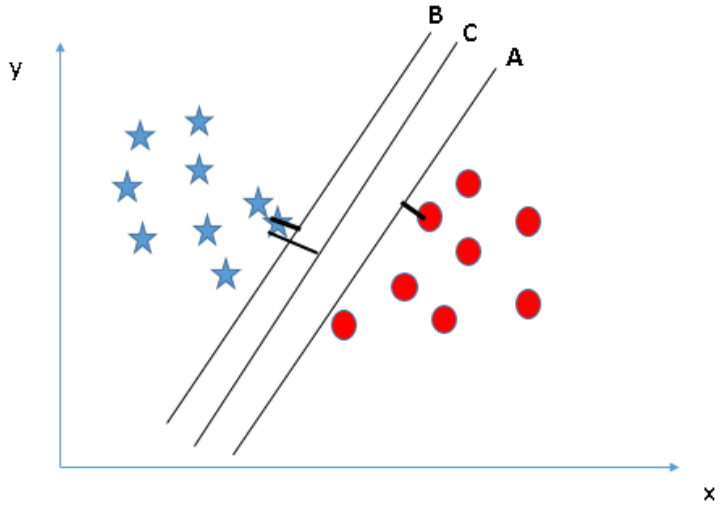


you need to remember a thumb rule to identify the right hyper-plane: "Select the hyper-plane which segregates the two classes better". In this scenario, hyper-plane "B" has excellently performed this job.

- Identify the right hyper-plane (Scenario-2): Here, we have three hyper-planes (A, B, and C) and all are segregating the classes well. Now, How can we identify the right hyper-plane?

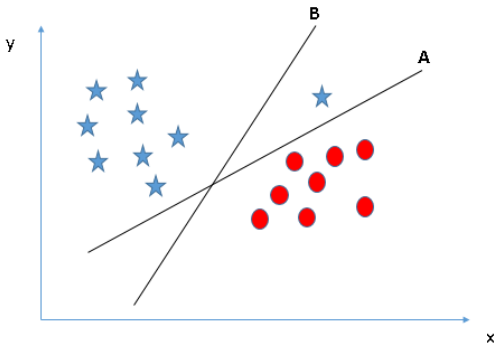


Here, maximizing the distances between nearest data point (either class) and hyper-plane will help us to decide the right hyper-plane. This distance is called as Margin. Let's look at the below snapshot:



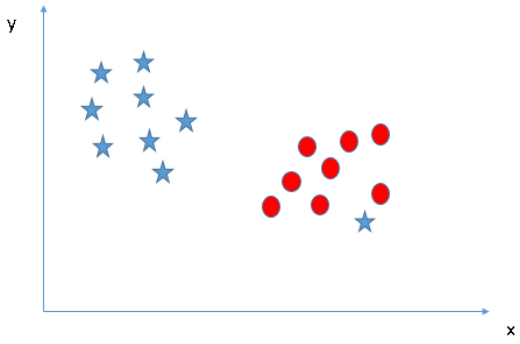
Above, you can see that the margin for hyper-plane C is high as compared to both A and B. Hence, we name the right hyper-plane as C. Another lightning reason for selecting the hyper-plane with higher margin is robustness. If we select a hyper-plane having low margin then there is high chance of miss-classification.

- Identify the right hyper-plane (Scenario-3): Hint: Use the rules as discussed in previous section to identify the right hyper-plane

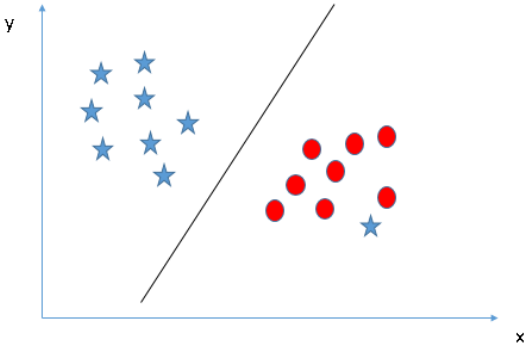


Some of you may have selected the hyper-plane B as it has higher margin compared to A. But, here is the catch, SVM selects the hyper-plane which classifies the classes accurately prior to maximizing margin. Here, hyper-plane B has a classification error and A has classified all correctly. Therefore, the right hyper-plane is A.

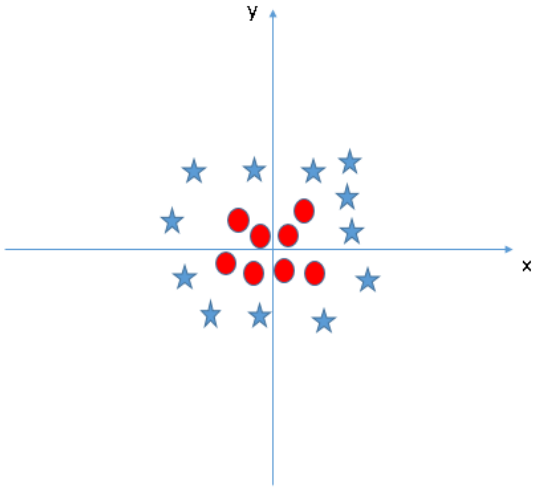
- Can we classify two classes (Scenario-4)?: Below, I am unable to segregate the two classes using a straight line, as one of the stars lies in the territory of other(circle) class as an outlier.



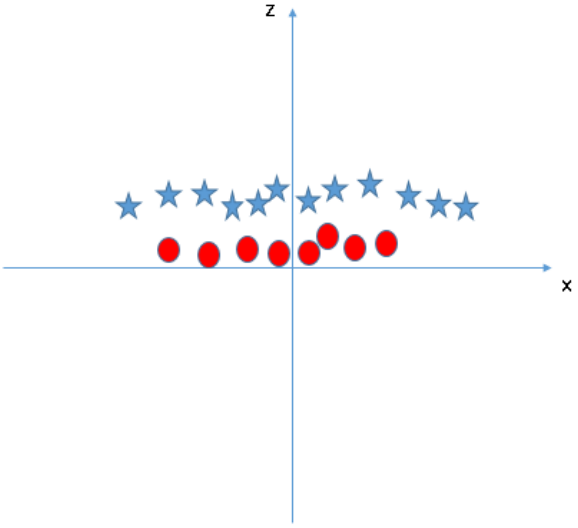
As I have already mentioned, one star at other end is like an outlier for star class. The SVM algorithm has a feature to ignore outliers and find the hyper-plane that has the maximum margin. Hence, we can say, SVM classification is robust to outliers.



- Find the hyper-plane to segregate to classes (Scenario-5): In the scenario below, we can't have linear hyper-plane between the two classes, so how does SVM classify these two classes? Till now, we have only looked at the linear hyper-plane.



SVM can solve this problem. Easily! It solves this problem by introducing additional feature. Here, we will add a new feature z=x^2+y^2. Now, let's plot the data points on axis x and z:

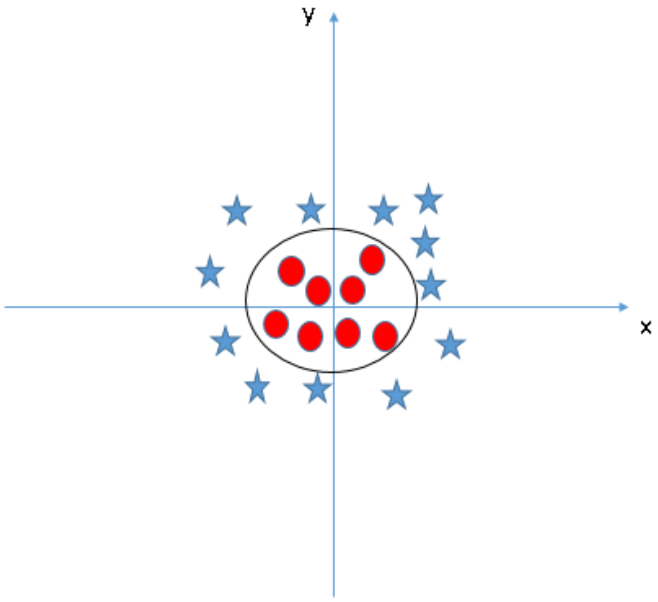


In above plot, points to consider are:

- All values for z would be positive always because z is the squared sum of both x and y
- In the original plot, red circles appear close to the origin of x and y axes, leading to lower value of z and star relatively away from the origin result to higher value of z.

In the SVM classifier, it is easy to have a linear hyper-plane between these two classes. But, another burning question which arises is, should we need to add this feature manually to have a hyper-plane. No, the SVM algorithm has a technique called the kernel trick. The SVM kernel is a function that takes low dimensional input space and transforms it to a higher dimensional space i.e. it converts not separable problem to separable problem. It is mostly useful in non-linear separation problem. Simply put, it does some extremely complex data transformations, then finds out the process to separate the data based on the labels or outputs you've defined.

When we look at the hyper-plane in original input space it looks like a circle:



How to tune Parameters of SVM?

Tuning the parameters' values for machine learning algorithms effectively improves model performance. Let's look at the list of parameters available with SVM.

```
sklearn.svm.SVC(C=1.0, kernel='rbf', degree=3, gamma=0.0, coef0=0.0, shrinking=True, probability=False,tol=0.001, cache_size=200, class_weight=None, verbose=False, max_iter=-1, random_state=None)
```

Here, we have various options available with kernel like, "linear", "rbf","poly" and others (default value is "rbf"). Here "rbf" and "poly" are useful for non-linear hyper-plane. Let's look at the example, where we've used linear kernel on two feature of iris data set to classify their class.

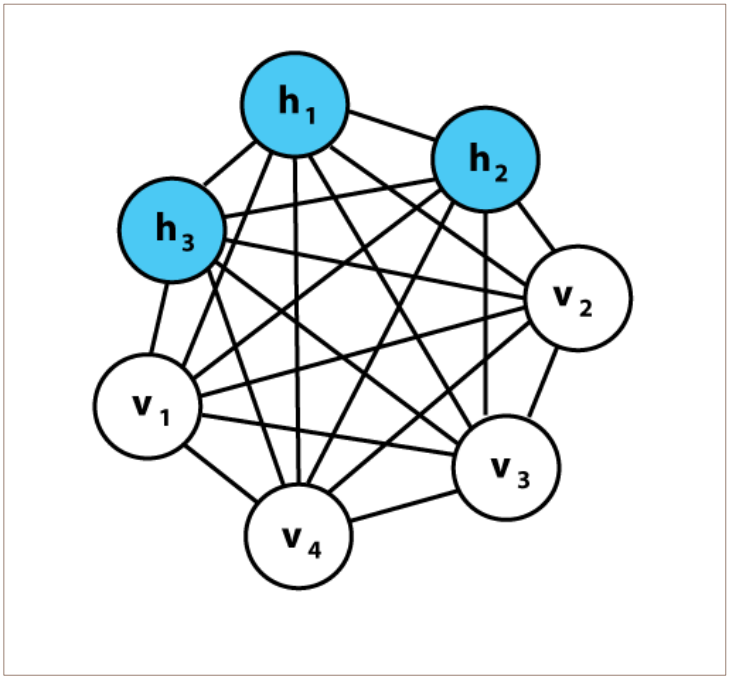
Pros and Cons associated with SVM

Pros:

- It works really well with a clear margin of separation
- It is effective in high dimensional spaces.
- It is effective in cases where the number of dimensions is greater than the number of samples.
- It uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.

Cons:

- It doesn't perform well when we have large data set because the required training time is higher
- It also doesn't perform very well, when the data set has more noise i.e. target classes are overlapping
- SVM doesn't directly provide probability estimates, these are calculated using an expensive five-fold cross-validation. It is included in the related SVC method of Python scikit-learn library.

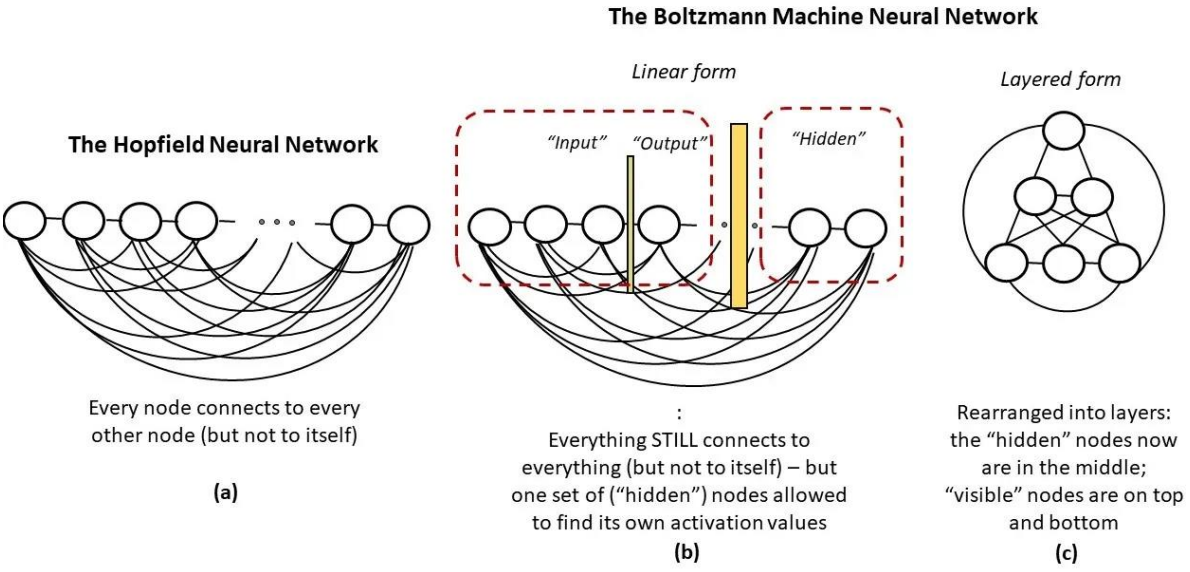


BOLTZMANN MACHINE

Introduction

Deep learning implements structured machine learning algorithms by making use of artificial neural networks. These algorithms help the machine to learn by itself and develop the ability to establish new parameters with which help to make and execute decisions. Deep learning is considered to be a subset of machine learning and utilizes multi-layered artificial neural networks to carry out its processes, which enables it to deliver high accuracy in tasks such as speech recognition, object detection, language translation and other such modern use cases being implemented every day. One of the most intriguing implementations in the domain of artificial intelligence for creating deep learning models has been the Boltzmann Machine.

Boltzmann Machine is a kind of recurrent neural network where the nodes make binary decisions and are present with certain biases. Several Boltzmann machines can be collaborated together to make even more sophisticated systems such as a deep belief network. Coined after the famous Austrian scientist Ludwig Boltzmann, who based the foundation on the idea of Boltzmann distribution in the late 20th century, this type of network was further developed by Stanford scientist Geoffrey Hinton. It derives its idea from the world of thermodynamics to conduct work toward desired states. It consists of a network of symmetrically connected, neuron-like units that make decisions stochastically whether to be active or not.

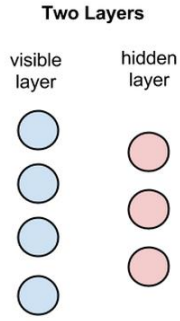


Boltzmann Machines consist of a learning algorithm that helps them to discover interesting features in datasets composed of binary vectors. The learning algorithm is generally slow in networks with many layers of feature detectors but can be made faster by implementing a learning layer of feature detectors. Boltzmann machines are typically used to solve different computational problems such as, for a search problem, the weights present on the connections can be fixed and are used to represent the cost function of the optimization problem. Similarly, for a learning problem, the Boltzmann machine can be presented with a set of binary data vectors from which it must find the weights on the connections so that the data vectors are good solutions to the optimization problem defined by those weights. Boltzmann machines make many small updates to their weights, and each update requires solving many different search problems.

Restricted Boltzmann Machine

Invented by Geoffrey Hinton, a Restricted Boltzmann machine is an algorithm useful for dimensionality reduction, classification, regression, collaborative filtering, feature learning and topic modeling. (For more concrete examples of how neural networks like RBMs can be employed, please see our page on use cases).

RBMs are shallow, two-layer neural nets that constitute the building blocks of deep-belief networks. The first layer of the RBM is called the visible, or input, layer, and the second is the hidden layer.

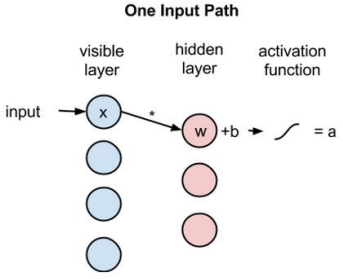


Each circle in the graph above represents a neuron-like unit called a node, and nodes are simply where calculations take place. The nodes are connected to each other across layers, but no two nodes of the same layer are linked.

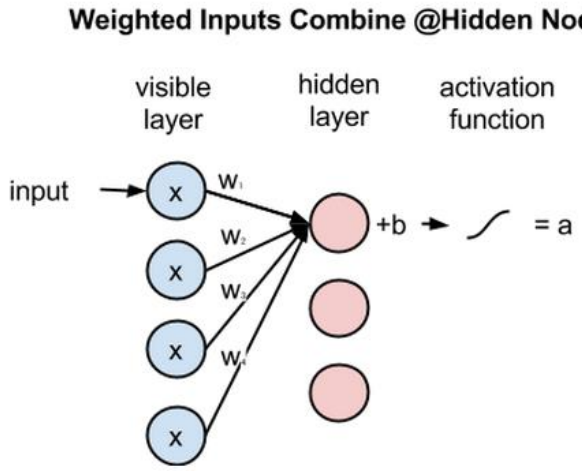
That is, there is no intra-layer communication – this is the restriction in a restricted Boltzmann machine. Each node is a locus of computation that processes input, and begins by making stochastic decisions about whether to transmit that input or not. (Stochastic means "randomly determined", and in this case, the coefficients that modify inputs are randomly initialized.)

ach visible node takes a low-level feature from an item in the dataset to be learned. For example, from a dataset of grayscale images, each visible node would receive one pixel-value for each pixel in one image. (MNIST images have 784 pixels, so neural nets processing them must have 784 input nodes on the visible layer.)

Now let's follow that single pixel value, x, through the two-layer net. At node 1 of the hidden layer, x is multiplied by a weight and added to a so-called bias. The result of those two operations is fed into an activation function, which produces the node's output, or the strength of the signal passing through it, given input x.



Next, let's look at how several inputs would combine at one hidden node. Each x is multiplied by a separate weight, the products are summed, added to a bias, and again the result is passed through an activation function to produce the node's output.

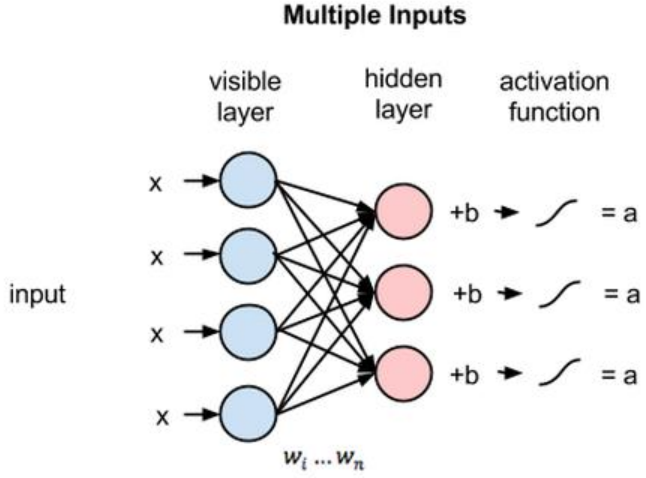


Because inputs from all visible nodes are being passed to all hidden nodes, an RBM can be defined as a symmetrical bipartite graph.

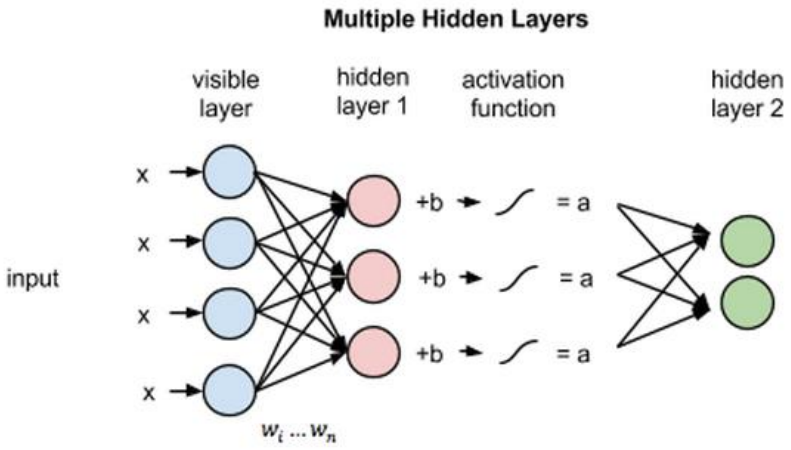
Symmetrical means that each visible node is connected with each hidden node (see below). Bipartite means it has two parts, or layers, and the graph is a mathematical term for a web of nodes.

At each hidden node, each input x is multiplied by its respective weight w. That is, a single input x would have three weights here, making 12 weights altogether (4 input nodes x 3 hidden nodes). The weights between two layers will always form a matrix where the rows are equal to the input nodes, and the columns are equal to the output nodes.

Each hidden node receives the four inputs multiplied by their respective weights. The sum of those products is again added to a bias (which forces at least some activations to happen), and the result is passed through the activation algorithm producing one output for each hidden node.



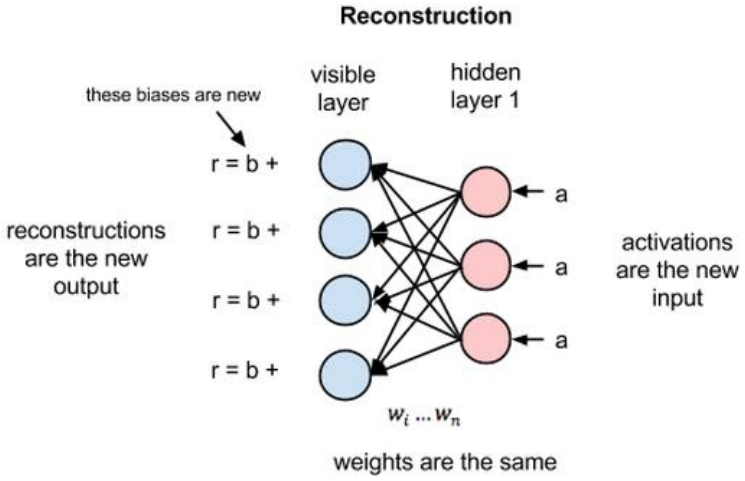
If these two layers were part of a deeper neural network, the outputs of hidden layer no. 1 would be passed as inputs to hidden layer no. 2, and from there through as many hidden layers as you like until they reach a final classifying layer. (For simple feed-forward movements, the RBM nodes function as an autoencoder and nothing more.)



Reconstructions

But in this introduction to restricted Boltzmann machines, we'll focus on how they learn to reconstruct data by themselves in an unsupervised fashion (unsupervised means without ground-truth labels in a test set), making several forward and backward passes between the visible layer and hidden layer no. 1 without involving a deeper network.

In the reconstruction phase, the activations of hidden layer no. 1 become the input in a backward pass. They are multiplied by the same weights, one per internode edge, just as x was weight-adjusted on the forward pass. The sum of those products is added to a visible-layer bias at each visible node, and the output of those operations is a reconstruction; i.e. an approximation of the original input. This can be represented by the following diagram:



Because the weights of the RBM are randomly initialized, the difference between the reconstructions and the original input is often large. You can think of reconstruction error as the difference between the values of **r** and the input values, and that error is then backpropagated against the RBM's weights, again and again, in an iterative learning process until an error minimum is reached.

As you can see, on its forward pass, an RBM uses inputs to make predictions about node activations, or the probability of output given a weighted x: **p(a|x; w)**.

But on its backward pass, when activations are fed in and reconstructions, or guesses about the original data, are spit out, an RBM is attempting to estimate the probability of inputs **x** given activations **a**, which are weighted with the same coefficients as those used on the forward pass. This second phase can be expressed as **p(x|a; w)**.

Together, those two estimates will lead you to the joint probability distribution of inputs x and activations **a**, or **p(x, a)**

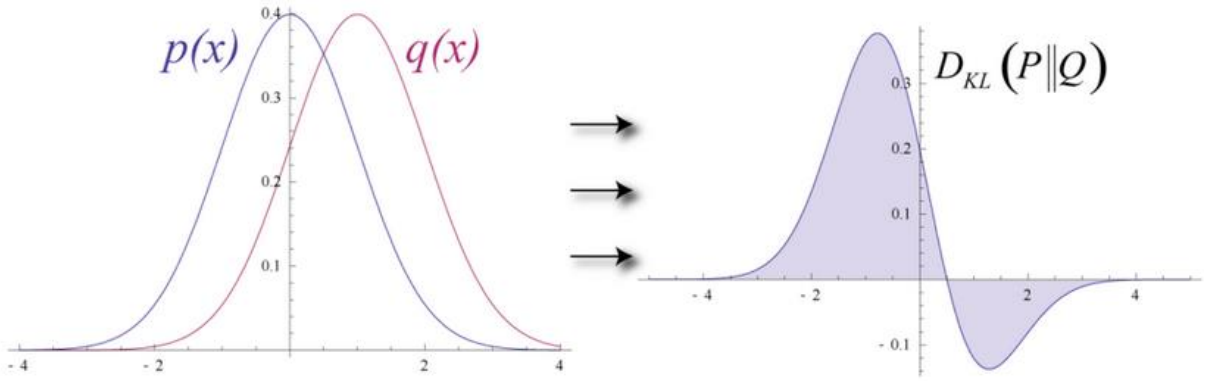
Reconstruction does something different from regression, which estimates a continous value based on many inputs, and different from classification, which makes guesses about which discrete label to apply to a given input example.

Reconstruction is making guesses about the probability distribution of the original input; i.e. the values of many varied points at once. This is known as generative learning, which must be distinguished from the so-called discriminative learning performed by classification, which maps inputs to labels, effectively drawing lines between groups of data points.

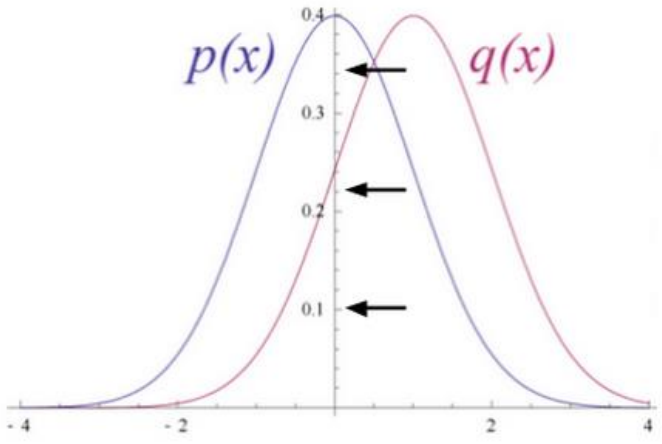
Let's imagine that both the input data and the reconstructions are normal curves of different shapes, which only partially overlap.

To measure the distance between its estimated probability distribution and the ground-truth distribution of the input, RBMs use Kullback Leibler Divergence.

KL-Divergence measures the non-overlapping, or diverging, areas under the two curves, and an RBM's optimization algorithm attempts to minimize those areas so that the shared weights, when multiplied by activations of hidden layer one, produce a close approximation of the original input. On the left is the probability distibution of a set of original input, p, juxtaposed with the reconstructed distribution q; on the right, the integration of their differences.

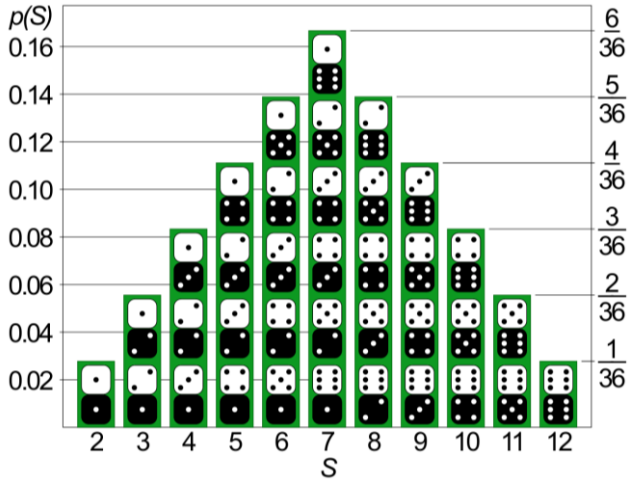


By iteratively adjusting the weights according to the error they produce, an RBM learns to approximate the original data. You could say that the weights slowly come to reflect the structure of the input, which is encoded in the activations of the first hidden layer. The learning process looks like two probability distributions converging, step by step.



Probability Distributions

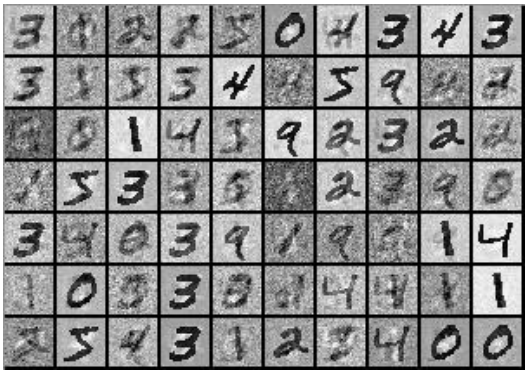
Let's talk about probability distributions for a moment. If you're rolling two dice, the probability distribution for all outcomes looks like this:



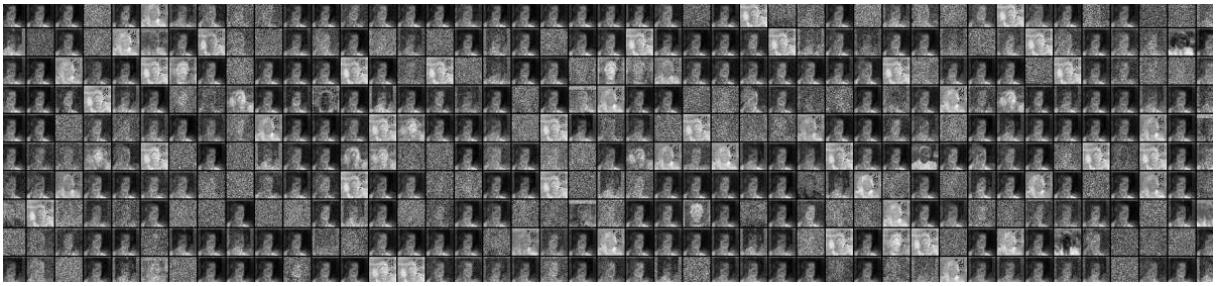
That is, 7s are the most likely because there are more ways to get to 7 (3+4, 1+6, 2+5) than there are ways to arrive at any other sum between 2 and 12. Any formula attempting to predict the outcome of dice rolls needs to take seven's greater frequency into account.

Or take another example: Languages are specific in the probability distribution of their letters, because each language uses certain letters more than others. In English, the letters e, t and a are the most common, while in Icelandic, the most common letters are a, r and n. Attempting to reconstruct Icelandic with a weight set based on English would lead to a large divergence.

In the same way, image datasets have unique probability distributions for their pixel values, depending on the kind of images in the set. Pixels values are distributed differently depending on whether the dataset includes MNIST's handwritten numerals:



or the headshots found in Labeled Faces in the Wild:



Imagine for a second an RBM that was only fed images of elephants and dogs, and which had only two output nodes, one for each animal. The question the RBM is asking itself on the forward pass is: Given these pixels, should my weights send a stronger signal to the elephant node or the dog node? And the question the RBM asks on the backward pass is: Given an elephant, which distribution of pixels should I expect?

That's joint probability: the simultaneous probability of **x** given **a** and of a given **x**, expressed as the shared weights between the two layers of the RBM.

The process of learning reconstructions is, in a sense, learning which groups of pixels tend to co-occur for a given set of images. The activations produced by nodes of hidden layers deep in the network represent significant co-occurrences; e.g. "nonlinear gray tube + big, floppy ears + wrinkles" might be one.

In the two images above, you see reconstructions learned by Deeplearning4j's implemention of an RBM. These reconstructions represent what the RBM's activations "think" the original data looks like. Geoff Hinton refers to this as a sort of machine "dreaming". When rendered during neural net training, such visualizations are extremely useful heuristics to reassure oneself that the RBM is actually learning. If it is not, then its hyperparameters, discussed below, should be adjusted.

One last point: You'll notice that RBMs have two biases. This is one aspect that distinguishes them from other autoencoders. The hidden bias helps the RBM produce the activations on the forward pass (since biases impose a floor so that at least some nodes fire no matter how sparse the data), while the visible layer's biases help the RBM learn the reconstructions on the backward pass.

Multiple Layers

Once this RBM learns the structure of the input data as it relates to the activations of the first hidden layer, then the data is passed one layer down the net. Your first hidden layer takes on the role of visible layer. The activations now effectively become your input, and they are multiplied by weights at the nodes of the second hidden layer, to produce another set of activations.

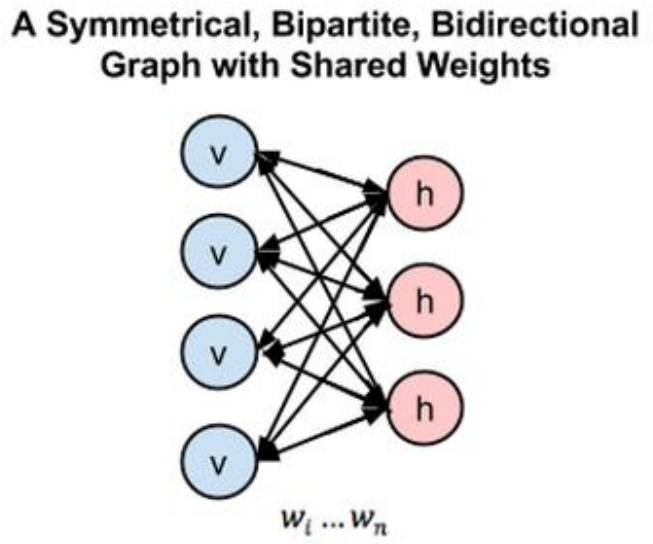
This process of creating sequential sets of activations by grouping features and then grouping groups of features is the basis of a feature hierarchy, by which neural networks learn more complex and abstract representations of data.

With each new hidden layer, the weights are adjusted until that layer is able to approximate the input from the previous layer. This is greedy, layerwise and unsupervised pre-training. It requires no labels to improve the weights of the network, which means you can train on unlabeled data, untouched by human hands, which is the vast majority of data in the world. As a rule, algorithms exposed to more data produce more accurate results, and this is one of the reasons why deep-learning algorithms are kicking butt.

Because those weights already approximate the features of the data, they are well positioned to learn better when, in a second step, you try to classify images with the deep-belief network in a subsequent supervised learning stage.

While RBMs have many uses, proper initialization of weights to facilitate later learning and classification is one of their chief advantages. In a sense, they accomplish something similar to backpropagation: they push weights to model data well. You could say that pre-training and backprop are substitutable means to the same end.

To synthesize restricted Boltzmann machines in one diagram, here is a symmetrical bipartite and bidirectional graph:



Parameters & k

The variable **k** is the number of times you run contrastive divergence. Contrastive divergence is the method used to calculate the gradient (the slope representing the relationship between a network's weights and its error), without which no learning can occur.

Each time contrastive divergence is run, it's a sample of the Markov Chain composing the restricted Boltzmann machine. A typical value is 1.

In the above example, you can see how RBMs can be created as layers with a more general **MultiLayerConfiguration**. After each dot you'll find an additional parameter that affects the structure and performance of a deep neural net. Most of those parameters are defined on this site.

weightInit, or **weightInitialization** represents the starting value of the coefficients that amplify or mute the input signal coming into each node. Proper weight initialization can save you a lot of training time, because training a net is nothing more than adjusting the coefficients to transmit the best signals, which allow the net to classify accurately.

activationFunction refers to one of a set of functions that determine the threshold(s) at each node above which a signal is passed through the node, and below which it is blocked. If a node passes the signal through, it is "activated."

optimizationAlgo refers to the manner by which a neural net minimizes error, or finds a locus of least error, as it adjusts its coefficients step by step. LBFGS, an acronym whose letters each refer to the last names of its multiple inventors, is an optimization algorithm that makes use of second-order derivatives to calculate the slope of gradient along which coefficients are adjusted.

regularization methods such as l2 help fight overfitting in neural nets. Regularization essentially punishes large coefficients, since large coefficients by definition mean the net has learned to pin its results to a few heavily weighted inputs. Overly strong weights can make it difficult to generalize a net's model when exposed to new data.

VisibleUnit/**HiddenUnit** refers to the layers of a neural net. The **VisibleUnit**, or layer, is the layer of nodes where input goes in, and the **HiddenUnit** is the layer where those inputs are recombined in more complex features. Both units have their own so-called transforms, in this case Gaussian for the visible and Rectified Linear for the hidden, which map the signal coming out of their respective layers onto a new space.

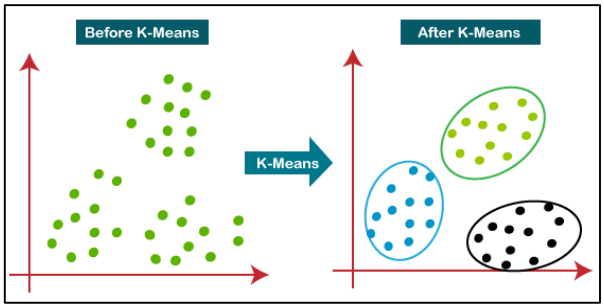
lossFunction is the way you measure error, or the difference between your net's guesses and the correct labels contained in the test set. Here we use **SQUARED_ERROR**, which makes all errors positive so they can be summed and backpropagated.

learningRate, like **momentum**, affects how much the neural net adjusts the coefficients on each iteration as it corrects for error. These two parameters help determine the size of the steps the net takes down the gradient towards a local optimum. A large learning rate will make the net learn fast, and maybe overshoot the optimum. A small learning rate will slow down the learning, which can be inefficient.

K-MEANS CLUSTERING ALGORITHM

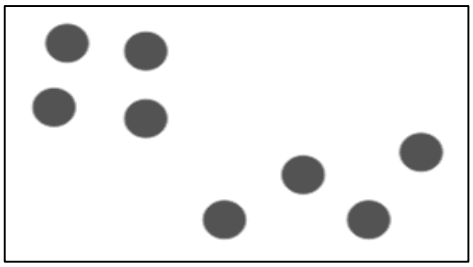
Introduction

K-means Clustering is an unsupervised iterative clustering technique. It partitions the given dataset into k predefined distinct clusters. A cluster is defined as a collection of data points exhibiting certain similarities.
It allows us to cluster the data into different groups and a convenient way to discover the categories of groups in the unlabelled dataset on its own without the need for any training.



It is a centroid-based algorithm, which identifies k number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible.
The main aim of this algorithm is to minimize the sum of distances between the data points and their respective cluster centroid.

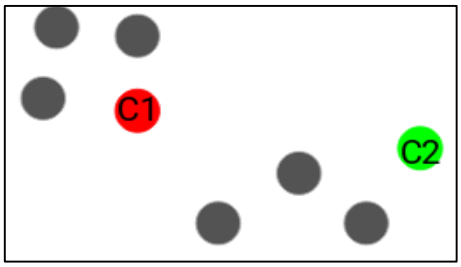
Algorithm with Example



We have these 8 points and we want to apply k-means to create clusters for these points.

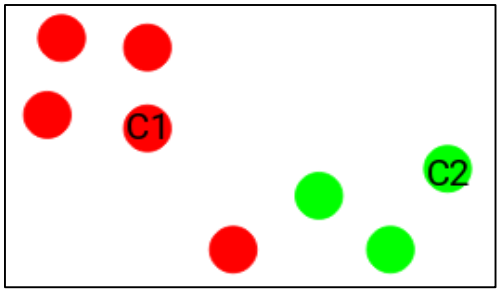
Step 1: Choose the number of clusters k, k=2

Step 2: Randomly select any k data points as cluster centres (or centroids).
Here, the red and green circles represent the centroid for these clusters.



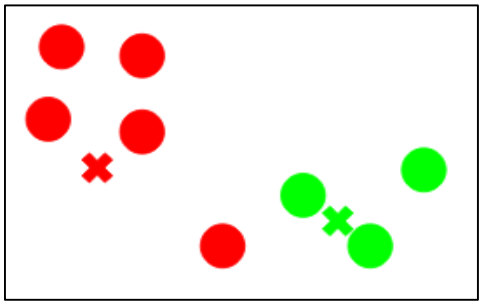
Step 3: Calculate the distance between each point and each cluster centre using given distance function or Euclidean distance formula

Step 4: Assign each data point to some cluster whose centre is nearest to that data point



Here we can see that the points which are closer to the red point are assigned to the red cluster whereas the points which are closer to the green point are assigned to the green cluster.

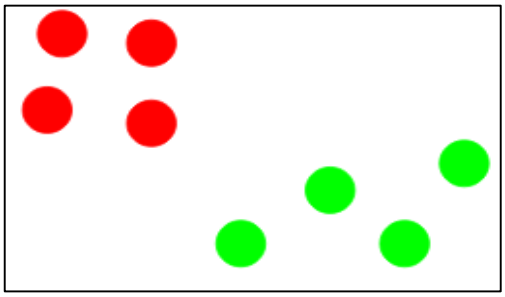
Step 5: Re-compute the centre of newly formed clusters and the centre of a cluster is computed by taking mean of all the data points contained in that cluster



Here, the red and green crosses are the new centroids.

Step 6: Keep repeating the procedure from step 3 to 5 until any of the following stopping criteria is met -

a. Centre of newly formed clusters do not change
b. Data points remain present in the same cluster
c. Maximum number of iterations are reached



Advantages and disadvantages of k-means

Advantages

- It works well with large datasets and it's very easy to implement.
- In clustering, especially in K-means, we have the benefit of having a convergence stage in the final as it's a good indicator of stable clusters. The program stops when the best result comes out.
- We can use numerous examples as data in it. It is a very adaptive type of algorithm.
- It can create clusters of a variety of shapes that gives much broader importance to the data visualization part.
- The clusters of k-means do not overlap with each other.

Disadvantages

- We need to choose the value of 'k' by ourselves. Or we can use a longer method like the elbow point method. But, it would still take time.
- It's hard to cluster for varying density and size.
- Outliers can cause problems for the position of the centroid. If an outlier is in the cluster, it would alter the centroid position
- This isn't suitable for a non-convex shaped cluster.

Applications of k-means

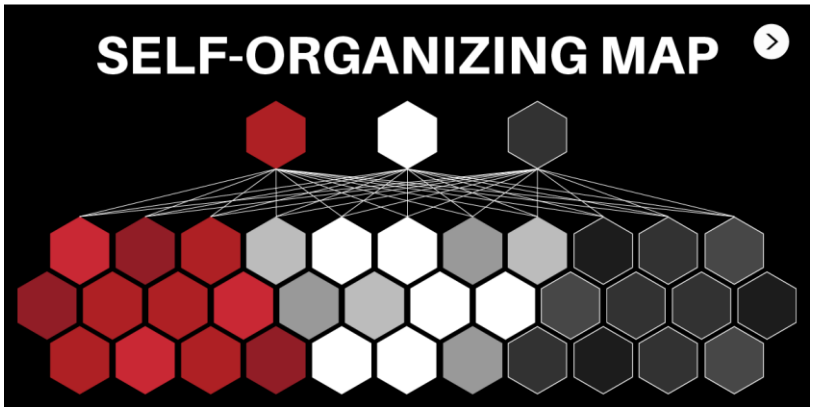
Insurance Fraud Detection
We make use of past data and also we draw various patterns to make sure that if any new case comes, it would definitely incline towards any specific cluster that the algorithms have created. It's an important thing to do as it can really help us to reduce insurance fraud.

Search Engines
The algorithm helps to form various clusters of possible results. If any search happens using the search engine, the result will be from a particular cluster, as search engines have a huge collection of clusters. The query would incline to a particular one prompting the algorithm to give out the accurate result.

Customer Segmentation
Companies find this very beneficial for improving their customer base. They would generally create various clusters of types of customers and would specifically target a few of them to improve their campaign.

Crime Hot-Spot Detection
This is a useful one for any city's police department. Using this model, they would identify specific areas with a high frequency of crimes and can gain a lot of information from this.

Diagnostic Systems
In the medical profession, professionals deal with various cases of ailments, both severe and normal. This clustering algorithm can help them in various ways like detect ailments by taking in data of symptoms. They can also help in being smart support systems and help in making better decisions.



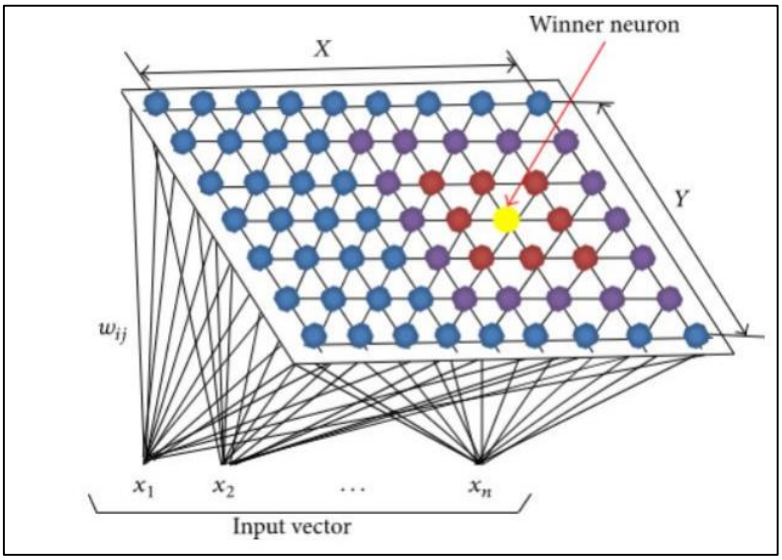
SELF ORGANIZING MAPS

Introduction

Self Organising Maps(SOMs) or Kohonen maps were invented by Prof. Teuvo Kohonen. It is another type of Artificial Neural Networks (ANNs). It is an unsupervised learning and a data visualization technique.
The goal of the technique is to reduce dimensions and detect features. The maps help to visualize high-dimensional data. It represents the multidimensional data in a two- dimensional space using the self-organizing neural networks. The technique is used for data mining, face recognition, pattern recognition, speech analysis, industrial and medical diagnostics, anomalies detection.
SOMs help to preserve the topological relationship of the training dataset (or the input) implying that the underlying properties of the input data are not affected even after the continuous change in the shape or size of the figure. SOMs help to reveal correlations that are not easily identified.

Architecture of SOMs

The SOMs structure is a lattice (or grid) of neurons (or nodes) that accepts and responds to a set of input signals. Each neuron has a location and those that lie close to each other represent clusters with similar properties. Therefore, each neuron represents a cluster learned from the training.
In the lattice, X and Y coordinate, each node has a specific topological position and comprises a vector of weights of the same dimension as that of the input variables.
The input training data is a set of vectors of n dimensions: x1, x2, x3…xn and each node consists of a corresponding weight vector W of n dimensions: w1, w2, w3… wn.

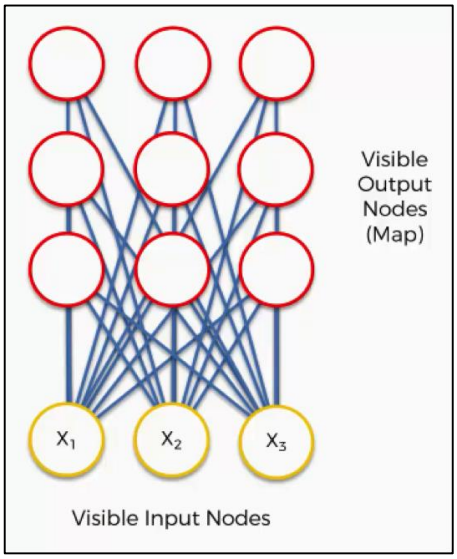


In SOMs, the responses are compared and a 'winning' neuron is selected. This selected neuron is activated together with the neighbourhood neurons. SOMs return a 2D map for any number of indicators as output where there are no lateral connections between output nodes.
Since SOM is an unsupervised learning technique, there is no target variable and hence, we do not calculate the loss function. Therefore there's no backward propagation process needed for SOMs.

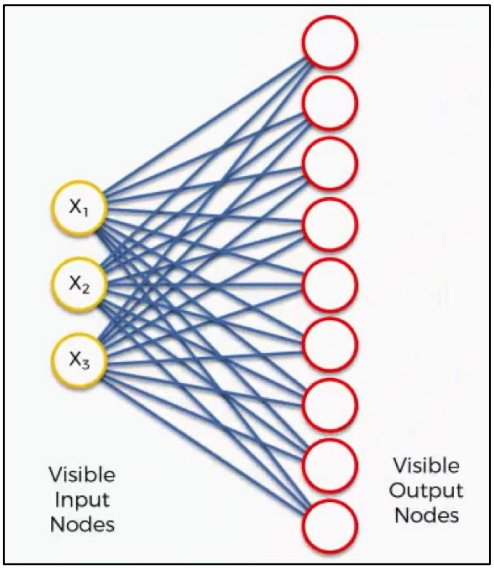
Learning of SOMs

The underlying idea of the SOMs training process is to examine every node and find the one node whose weight is most similar to the input vector. The training is carried out in a few steps and over many iterations.

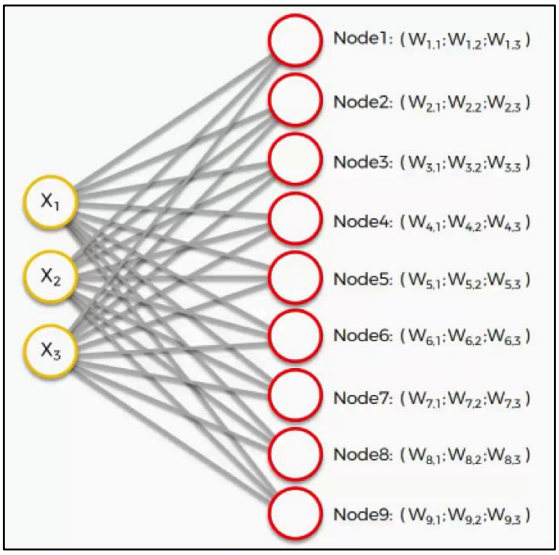
We have three input signals x1, x2, and x3. This input vector is chosen at random from a set of training data and presented to the lattice. So, we have a lattice of neurons in the form of a 3*3 2D array having nine nodes with three rows and three columns as show below:



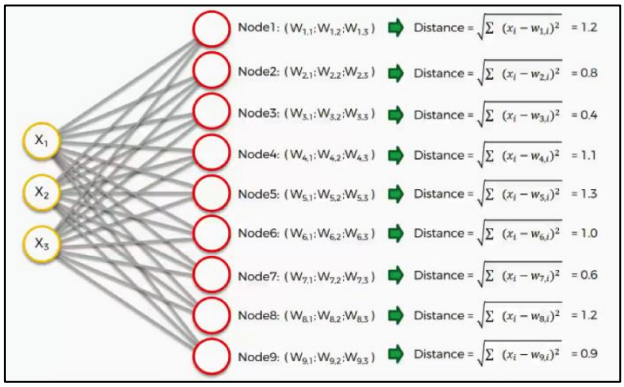
Let's visualize the above 3*3 matrix as below:



Each node has some random values as weights. These weights are not of the neural network as shown as:



From these weights, we can calculate the Euclidean distance as:

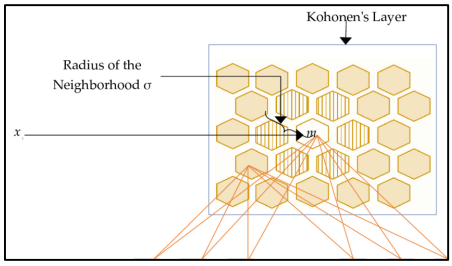


The training process of SOMs

Step 1: Firstly, randomly initialize all the weights.
Step 2: Select an input vector x = [x1, x2, x3, … , xn] from the training set.
Step 3: Compare x with the weights wj by calculating Euclidean distance for each neuron j. The neuron having the least distance is declared as the winner. The winning neuron is known as Best Matching Unit(BMU)
Step 4: Update the neuron weights so that the winner becomes and resembles the input vector x.
Step 5: The weights of the neighbouring neurons are adjusted to make them more like the input vector. The closer a node is to the BMU, the more its weights get altered. The parameters adjusted are learning rate and neighbourhood function.
Step 6: Repeat from step 2 until the map has converged for the given iterations or there are no changes observed in the weights.

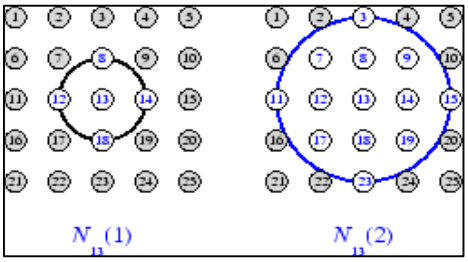
Best Matching Unit (BMU)

As we saw above the distance of each of the nodes (or raw data point) is calculated from all the output nodes. The node or neuron which is identified with the least distance is the Best Matching Unit (BMU) or neighbourhood. It is depicted as (m) in the following figure:



From this BMU, a radius (or sigma) is defined. All the nodes that fall in the radius of the BMU get updated according to their respective distance from the BMU.
So, now the point is that a neuron in the output map can be a part of the radius of many different BMUs. This leads to a constant push and pulls effect on the neuron however, the behaviour of the neuron will be more similar to the BMU near it.

After each iteration, the radius shrinks and the BMU pulls lesser nodes so the process becomes more and more accurate and that shrinks the neighbourhood that reduces the mapping of those segments of those data points which belong to that particular segment. This makes the map more and more specific and ultimately it starts converging and becomes like the training data (or input).

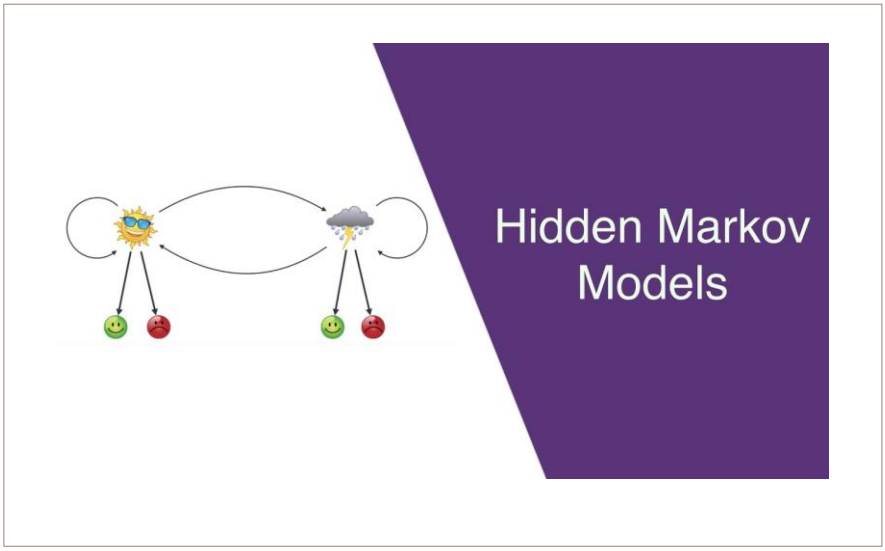


In the above figure, the radius of the neighbourhood of the BMU is calculated. This is a value that starts large as on the right side of the panel, typically set to the radius of the lattice but diminishes each time step. Any nodes found within this radius are deemed to be inside the BMU's neighbourhood.

The Analogy of SOMs to K-means Clustering

Both K-Means clustering and SOMs are a means to convert the higher dimensional data to lower dimensional data. Though one is a machine learning algorithm and another is a deep learning algorithm yet there is an analogy between the two.

In K-Means clustering, when new data points are presented, the centroids are updated and there are methods such as the elbow method to find the value of K whereas in SOMs, the architecture of the neural network is changed to update the weights and the number of neurons is a tuning parameter.



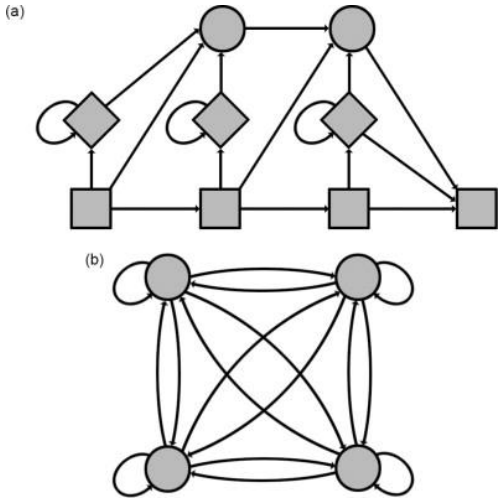
HIDDEN MARKOV MODEL

Introduction

A Hidden Markov Model (HMM) is a statistical model which is also used in machine learning. It can be used to describe the evolution of observable events that depend on internal factors, which are not directly observable. These are a class of probabilistic graphical models that allow us to predict a sequence of unknown variables from a set of observed variables.

The Hidden Markov model is a probabilistic model which is used to explain or derive the probabilistic characteristic of any random process. It basically says that an observed event will not be corresponding to its step-by-step status but related to a set of probability distributions.

The main goal of HMM is to learn about a Markov chain by observing its hidden states. Considering a Markov process X with hidden states Y here the HMM solidifies that for each time stamp the probability distribution of Y must not depend on the history of X according to that time.



Applications of HMM

- Computational finance
- Speed analysis
- Speech recognition
- Speech synthesis
- Part-of-speech tagging
- Document separation in scanning solutions
- Machine translation
- Handwriting recognition
- Time series analysis
- Activity recognition
- Sequence classification
- Transportation forecasting

HMM with an Example

We take an example of two friends, Rahul and Ashok. Now Rahul completes his daily life works according to the weather conditions. Major three activities completed by Rahul are- go jogging, go to the office and cleaning his home. What Rahul is doing today depends on whether and whatever Rahul does he tells Ashok and Ashok has no proper information about the weather but Ashok tries to assume the weather condition according to Rahul work.

Ashok believes that the weather operates as a discrete Markov chain, wherein the chain there are only two states whether the weather is Rainy or it is sunny. The condition of the weather cannot be observed by Ashok, here the conditions of the weather are hidden from Ashok. On each day, there is a certain chance that Bob will perform one activity from the set of the following activities {"jog", "work"," clean"}, which are depending on the weather. Since Rahul tells Ashok that what he has done, those are the observations. The entire system is that of a HMM.

Here, we can say that the parameter of HMM is known to Ashok because he has general information about the weather and he also knows what Rahul likes to do on an average.

Let's consider a day where Rahul called Ashok and told him that he has cleaned his home. In that scenario, Ashok will have a belief that there are more chances of a rainy day and we can say that belief Ashok has is the start probability of HMM.

The states and observation are:

```
states = (
        'Rainy',
        'Sunny'
)
observations = (
        'walk',
        'shop',
        'clean'
)
```

And the start probability is:

```
start_probabaility = {
        'Rainy': 0.6,
        'Sunny':0.4
}
```

Now the distribution of the probability has the weightage more on the rainy day stateside so we can say there will be more chances for a day to being rainy again and the probabilities for next day weather states are as following

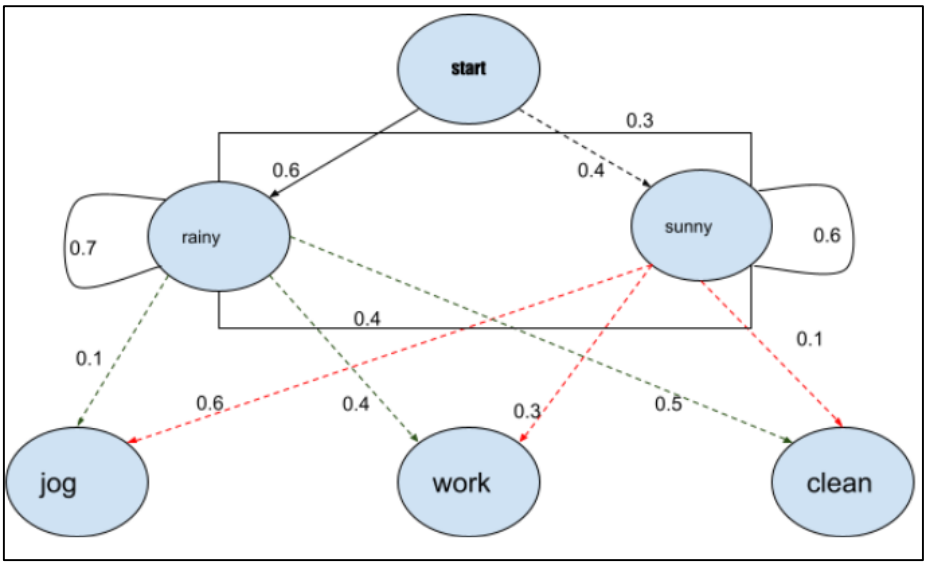
```
transition_probability = {
        'Rainy' : {'Rainy':0.7, 'Sunny':0.3},
        'Sunny' : {'Rainy':0.4, 'Sunny':0.6},
}
```

From above we can say that the changes in the probability for a day is transition probabilities and according to the transition probability the emitted results for the probability of work that Rahul will perform is

```
emission_probability = {
        'Rainy' : {'jog':0.1, 'work':0.4, 'clean':0.5},
        'Sunny' : {'jog':0.6, 'work':0.3, 'clean':0.1},
}
```

This probability can be considered as the emission probability. Using the emission probability Ashok can predict the states of the weather or using the transition probabilities Ashok can predict the work which Rahul is going to perform the next day.

Below image shows the HMM process for making probabilities



So, from the above example we can understand as to how we can use this probabilistic model to make a prediction.

HMMs in NLP

From the above application of HMM, we can understand that the applications where the HMM can be used have sequential data like time series data, audio and video data and text data or NLP data.

POS-tagging

We have learnt that the part of speech indicates the function of any word, like what it means in any sentence. There are commonly nine parts of speeches; noun, pronoun, verb, adverb, article, adjective, preposition, conjunction, interjection and a word need to fit into the proper part of speech to make sense in the sentence.
POS tagging is a very useful part of text pre-processing in NLP as we know that NLP is a task where we make a machine able to communicate with a human or with a different machine. So it becomes compulsory for a machine to understand the part of speech.
Classifying words in their part of speech and providing them labels according to their part of speech is called part of speech tagging or POS tagging. The set of labels/tags is called a tagset.

POS Tagging with HMM

Let's take the sentence "Rahul will eat food" where Rahul is a noun, will is a modal, eat is a verb and food is also a noun. So the probability for a word to be in a particular class of part of speech is called the Emission probability.

Let's take the following sentences:
Mary Jane can see will
The spot will see Mary
Will Jane spot Mary?
Mary will pat Spot

The below table is a counting tableau for the words with their part of speech type

Words	Noun	Modal	Verb
Mary	4	0	0
Jane	2	0	0
Will	1	3	0
Spot	2	0	1
Can	0	1	0
See	0	0	2
Pat	0	0	1

Let's divide each word's appearance by the total number of every part of speech in the set of sentences.

Words	Noun	Modal	Verb
Mary	4/9	0	0
Jane	2/9	0	0
Will	1/9	¾	0
Spot	2/9	0	¼
Can	0	¼	0
See	0	0	2/4

Pat	0	0	¼

Here in the table, we can see the emission probabilities of every word.

We have discussed that the transition probability is the probability of the sequences. We can define a table for the above set of sentences according to the sequence of part of speech.

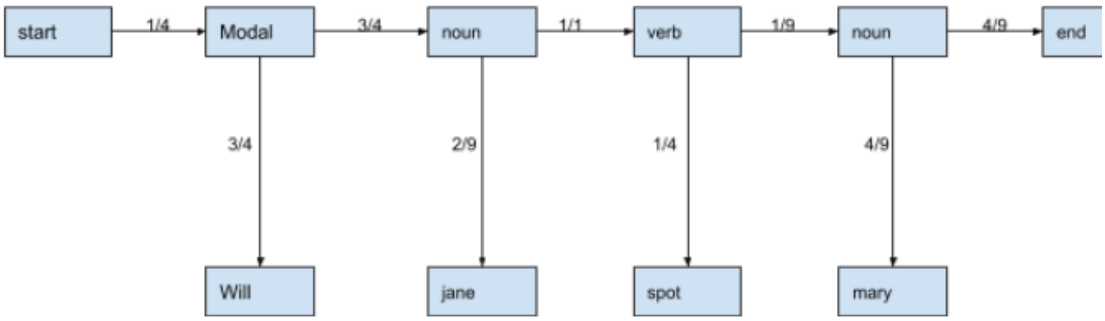
	Noun	Modal	Verb	End
Start	3	1	0	0
Noun	1	3	1	4
Modal	1	0	3	0
Verb	4	0	0	0

Now in the table, we are required to check for the combination of parts of speeches for calculation of the transition probabilities. For example, we can see in the set of sentences modal before a verb has appeared 3 times and 1 time before a noun. This means it has appeared in the set for 4 times and the probability of coming modal before any verb will be ¾ and before a noun will be ¼. Similarly performing this for every entity of the table:

	Noun	Modal	Verb	End
Start	¾	¼	0	0
Noun	1/9	3/9	1/9	4/9
Modal	¼	0	¾	0
Verb	4/4	0	0	0

Here the above values in the table are the respective transition values for a given set of sentences.

Let's take the sentence "Will Jane spot Mary?" out from the set and now we can calculate the probabilities for every part of speech using the above calculations.



In the above diagram, vertical lines indicate the emission probabilities of the words in the sentence and the horizontal lines indicate all the transition probabilities.
The correctness of POS tagging is measured by the product of all these probabilities. The product of probabilities represents the likelihood that the sequence is right.

Let's check for the correctness of the POS tagging.
¼*¾*¾*2/9*1/1*¼*1/9*4/9*4/9 = 0.0001714678

Here, we see that the product is greater than zero which means the POS tagging we have performed is correct and if the result is zero then the tagging performed will be incorrect.

So here we have seen how the HMM algorithm works for providing the POS tagging to the sentences but the example was small where we had only 3 kinds of POS tags but there are 81 different kinds of POS tags available. When it comes to finding the number of combinations from a small data set it can be done with smaller efforts but when it comes to tagging larger sentences and finding the right sequences with all the 81 tags the number of combinations increases exponentially. The computation may cause a larger effect but the more number of POS tags gives more accuracy.

To optimize the implementation of HMM for POS tagging we can use the Viterbi algorithm which is a dynamic programming algorithm.

VITERBI ALGORITHM

Introduction

The Viterbi algorithm is an efficient way to make an inference, or prediction, to the hidden states given the model parameters are optimized, and given the observed data. The operation of Viterbi's algorithm can be visualized by means of a trellis diagram. The Viterbi path is essentially the shortest path through this trellis.

The Viterbi algorithm is used to compute the most probable path (as well as its probability). It requires knowledge of the parameters of the HMM model and a particular output sequence and it finds the state sequence that is most likely to have generated that output sequence. It works by finding a maximum over all possible state sequences.

In this, there are often many state sequences that can produce the same particular output sequence, but with different probabilities. It is possible to calculate the probability for the HMM model to generate that output sequence by doing the summation over all possible state sequences. This also can be done efficiently using the Forward algorithm (or the Backward algorithm), which is also a dynamical programming algorithm.

Viterbi algorithm

The purpose of the Viterbi algorithm is to make an inference based on a trained model and some observed data. It works by asking a question: given the trained parameter matrices and data, what is the choice of states such that the joint probability reaches maximum? In other words, what is the most likely choice given the data and the trained model? This statement can be visualized as the following formula, and obviously, the answer depends on the data!

$$X^*_{0:T} = \underset{X_{0:T}}{\operatorname{argmax}} P[X_{0:T}|Y_{0:T}]$$

this says to find the states that maximize the conditional probability of states given data.

$$\mu(X_k) = \max_{X_{0:k-1}} P[X_{0:k}, Y_{0:k}] = \max_{X_{k-1}} \mu(X_{k-1}) P[X_k|X_{k-1}] P[Y_k|X_k]$$

mu function. It depends on its previous step, the transition and the emission matrices.

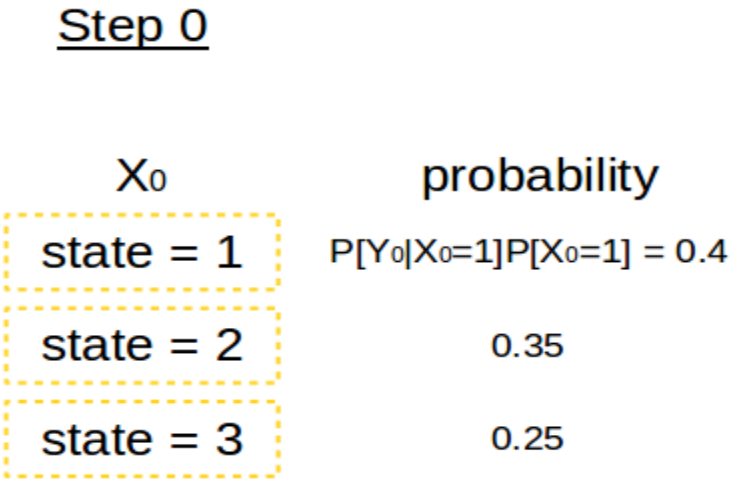
Let's substitute k=1,2,3 so that we can understand this recursion easily.

$$\mu(X_0) = P[Y_0|X_0]P[X_0]$$
$$\mu(X_1) = \max_{X_0} \mu(X_0)P[X_1|X_0]P[Y_1|X_1]$$
$$\mu(X_2) = \max_{X_1} \mu(X_1)P[X_2|X_1]P[Y_2|X_2]$$
$$\mu(X_3) = \max_{X_2} \mu(X_2)P[X_3|X_2]P[Y_3|X_3]$$

The first formula is the starting mu function, and results in a probability distribution for seeing different initial state given the initial observed data. Here, we do not constrain ourselves to any of these initial state, but we determine that in the next formula.

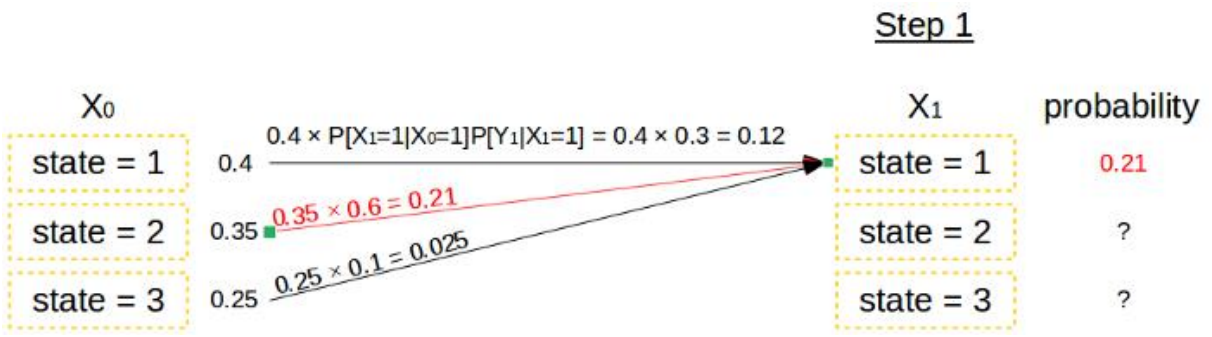
The second formula picks up the best initial state that maximize the product of the terms in the right hand side, and leaving the first state as a free parameter to be determined in the third formula. Similarly, the third formula picks the best first state, then leave the second state for the fourth formula.

Let us visualize these repeated processes in a diagram called trellis. In the diagram, we could see how each state is being chosen based on the rule of maximizing the probability, and to keep the diagram small, I would take an assumption that there are only three possible states for us to choose from at each time step. The same idea applies to any number of states, of course.

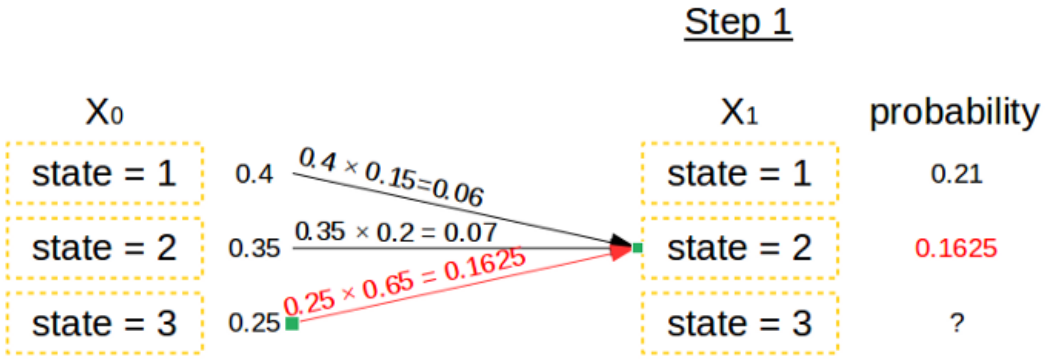


step 0. All three possible states are listed out for the initial state at time 0. The corresponding probability is assumed to be some actual numbers to facilitate the following discussion.

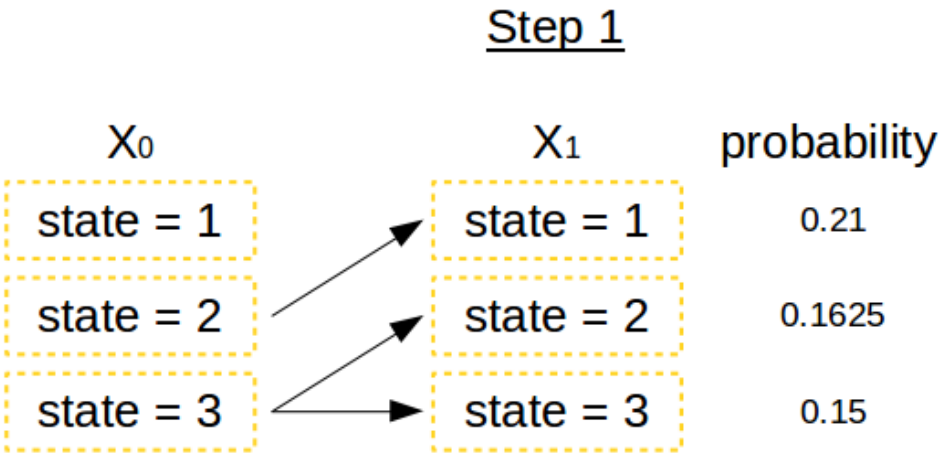
The step 0 is to just list out all possible state at time 0, and their corresponding probability values, and we do not decide which state is chosen at this stage.



step 1–1. For each possible first state, the best possible initial state is chosen. We find that initial state = 2 is most likely to lead to first state = 1.

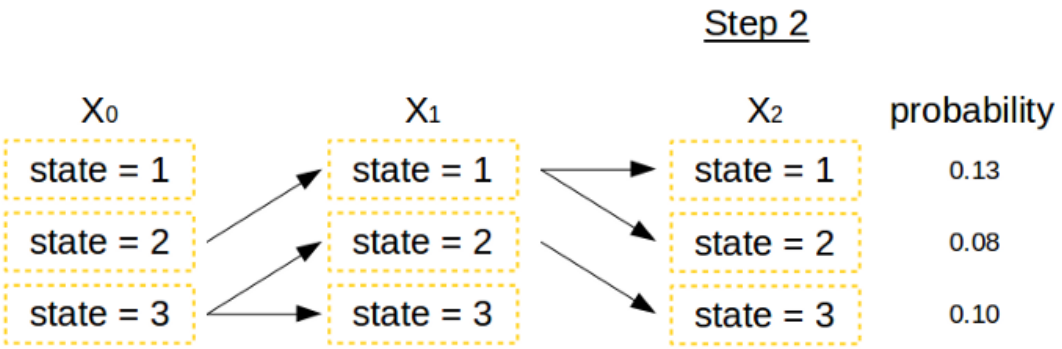


step 1–2. Here we find that initial state = 3 is most likely to lead to first state = 2.



step 1–3. We end Step 1 by finding the initial states that are mostly likely leading to the first state, and remembering their probability values which will be re-used in the next step.

In step 1, we iterate through all possible first state (state at time = 1), and find out their corresponding best initial state (state at time = 0) as illustrated in the above graphs. We then repeat the same procedure and finish step 2.



step 2. Repeating the procedure to go from step 0 to step 1 to get step 2 from step 1.

Now we begin to see some paths. For example, if we end the inference at step 2, then the most likely ending state would be state = 1, and the rest of the previous states could be back-traced through the arrows, which are state 2 at time 0, state 1 at time 1, and state 1 at time 2. The second likely path is 3–2–3, and the least likely path is 2–1–2. It is very unlikely that the path starts with state 1.

The Viterbi algorithm is an efficient way to make an inference, or prediction, to the hidden states given the model parameters are optimized, and given the observed data. It is best visualized by a trellis to find out how the path is select from a time step to the next.

REFERENCE

https://www.digitalocean.com/community/tutorials/an-introduction-to-machine-learning

https://towardsdatascience.com/introduction-to-machine-learning-algorithms-linear-regression-14c4e325882a

http://www2.cs.uregina.ca/~dbd/cs831/notes/ml/vspace/3_vspace.html

https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148 -:~:text=Logistic%20Regression%20is%20a%20Machine,on%20the%20concept%20of%20probability

https://www.hpe.com/in/en/what-is/machine-learning.html

https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c

https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd

https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/
https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

https://builtin.com/data-science/step-step-explanation-principal-component-analysis

https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/

https://www.analyticsvidhya.com/blog/2021/09/beginners-guide-to-anomaly-detection-using-self-organizing-maps/

https://analyticsindiamag.com/a-guide-to-hidden-markov-model-and-its-applications-in-nlp/

https://medium.com/analytics-vidhya/viterbi-algorithm-for-prediction-with-hmm-part-3-of-the-hmm-series-6466ce2f5dc6

https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/

https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/

https://jalammar.github.io/illustrated-transformer/

https://wiki.pathmind.com/restricted-boltzmann-machine

https://colah.github.io/posts/2015-08-Understanding-LSTMs/

https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa

https://towardsdatascience.com/decision-trees-in-machine-learning-641b9c4e8052

https://towardsdatascience.com/understanding-hyperparameters-and-its-optimisation-techniques-f0debba07568