Do you ever think of what it's like to build anything like a brain, how these things work, or what they do? Let us look at how nodes communicate with neurons and what are some differences between artificial and biological neural networks.

1. Artificial Neural Network: Artificial Neural Network (ANN) is a type of neural network that is based on a Feed-Forward strategy. It is called this because they pass information through the nodes continuously till it reaches the output node. This is also known as the simplest type of neural network. **Some advantages of ANN :**

- Ability to learn irrespective of the type of data (Linear or Non-Linear).
- ANN is highly volatile and serves best in financial time series forecasting.

Some disadvantages of ANN :

- The simplest architecture makes it difficult to explain the behavior of the network.
- This network is dependent on hardware.

2. Biological Neural Network: Biological Neural Network (BNN) is a structure that consists of Synapse, dendrites, cell body, and axon. In this neural network, the processing is carried out by neurons. Dendrites receive signals from other neurons, Soma sums all the incoming signals and axon transmits the signals to other cells.

Some advantages of BNN :

- The synapses are the input processing element.
- It is able to process highly complex parallel inputs.

Some disadvantages of BNN :

- There is no controlling mechanism.
- Speed of processing is slow being it is complex.

Differences between ANN and BNN :

Biological Neural Networks (BNNs) and Artificial Neural Networks (ANNs) are both composed of similar basic components, but there are some differences between them.

Neurons: In both BNNs and ANNs, neurons are the basic building blocks that process and transmit information. However, BNN neurons are more complex and diverse than ANNs. In BNNs, neurons have multiple **dendrites** that receive input from multiple sources, and the **axons** transmit signals to other neurons, while in ANNs, neurons are simplified and usually only have a single output.

Synapses: In both BNNs and ANNs, synapses are the points of connection between neurons, where information is transmitted. However, in ANNs, the connections between neurons are usually fixed, and the strength of the connections is determined by a set of weights, while in BNNs, the connections between neurons are more flexible, and the strength of the connections can be modified by a variety of factors, including learning and experience.

Neural Pathways: In both BNNs and ANNs, neural pathways are the connections between neurons that allow information to be transmitted throughout the network. However, in BNNs, neural pathways are highly complex and diverse, and the connections between neurons can be modified by experience and learning. In ANNs, neural pathways are usually simpler and predetermined by the architecture of the network.

Parameters	**ANN**	**BNN**
Structure	input	dendrites

	weight	synapse
	output	axon
	hidden layer	cell body
Learning	very precise structures and formatted data	they can tolerate ambiguity
	complex	simple
Processor	high speed	low speed
	one or a few	large number
	separate from a processor	integrated into processor
Memory	localized	distributed
	non-content addressable	content-addressable
	centralized	distributed
Computing	sequential	parallel
	stored programs	self-learning
Reliability	very vulnerable	robust
Expertise	numerical and symbolic	perceptual
	manipulations	problems
Operating Environment	well-defined	poorly defined
	well-constrained	un-constrained
Fault Tolerance	the potential of fault tolerance	performance degraded even on partial damage

Overall, while BNNs and ANNs share many basic components, there are significant differences in their complexity, flexibility, and adaptability. BNNs are highly complex and adaptable systems that can process information in parallel, and their plasticity allows them to learn and adapt over time. In contrast, ANNs are simpler systems that are designed to perform specific tasks, and their connections are usually fixed, with the network architecture determined by the designer.

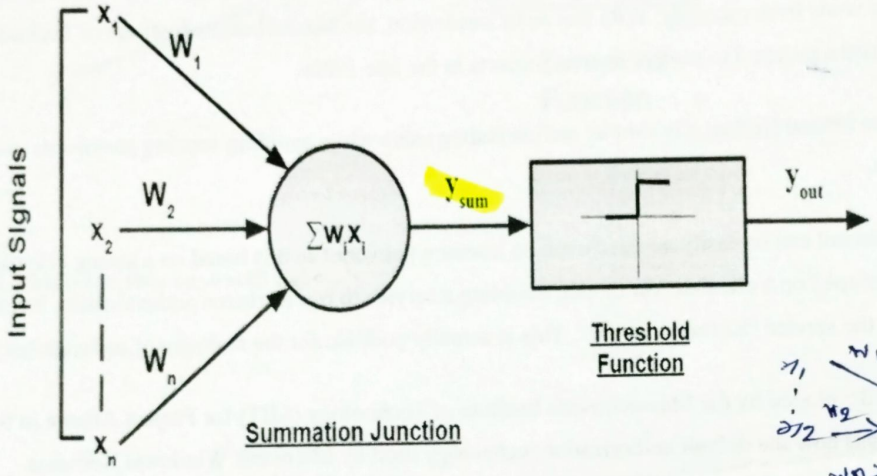
The output of ...

McCulloch-pitts

McCulloch-Pitts Model of Neuron

The McCulloch-Pitts neural model, which was the earliest ANN model, has only two types of inputs — **Excitatory and Inhibitory.** The excitatory inputs have weights of positive magnitude and the inhibitory weights have weights of negative magnitude. The inputs of the McCulloch-Pitts neuron could be either 0 or 1. It has a threshold function as an activation function. So, the output signal y_{out} is 1 if the input y_{sum} is greater than or equal to a given threshold value, else 0. The diagrammatic representation of the model is as follows:

1. Excitatory
2. Inhibitory



McCulloch-Pitts Model

Simple McCulloch-Pitts neurons can be used to design logical operations. For that purpose, the connection weights need to be correctly decided along with the threshold function (rather than the threshold value of the activation function). For better understanding purpose, let me consider an example:

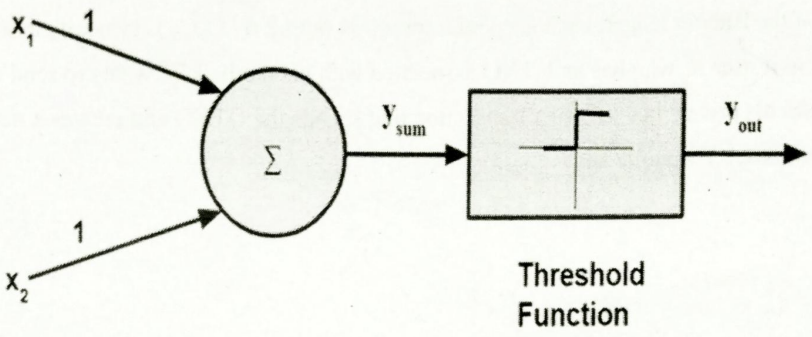
John carries an umbrella if it is sunny or if it is raining. There are four given situations. I need to decide when John will carry the umbrella. The situations are as follows:

- First scenario: It is not raining, nor it is sunny
- Second scenario: It is not raining, but it is sunny
- Third scenario: It is raining, and it is not sunny
- Fourth scenario: It is raining as well as it is sunny

To analyse the situations using the McCulloch-Pitts neural model, I can consider the input signals as follows:

- X_1: Is it raining?
- X_2 : Is it sunny?

So, the value of both scenarios can be either 0 or 1. We can use the value of both weights X_1 and X_2 as 1 and a threshold function as 1. So, the neural network model will look like:



Truth Table for this case will be:

Situation	x_1	x_2	y_{sum}	y_{out}
1	0	0	0	0
2	0	1	1	1
3	1	0	1	1
4	1	1	2	1

So, I can say that,

The truth table built with respect to the problem is depicted above. From the truth table, I can conclude that in the situations where the value of y_{out} is 1, John needs to carry an umbrella. Hence, he will need to carry an umbrella in scenarios 2, 3 and 4.

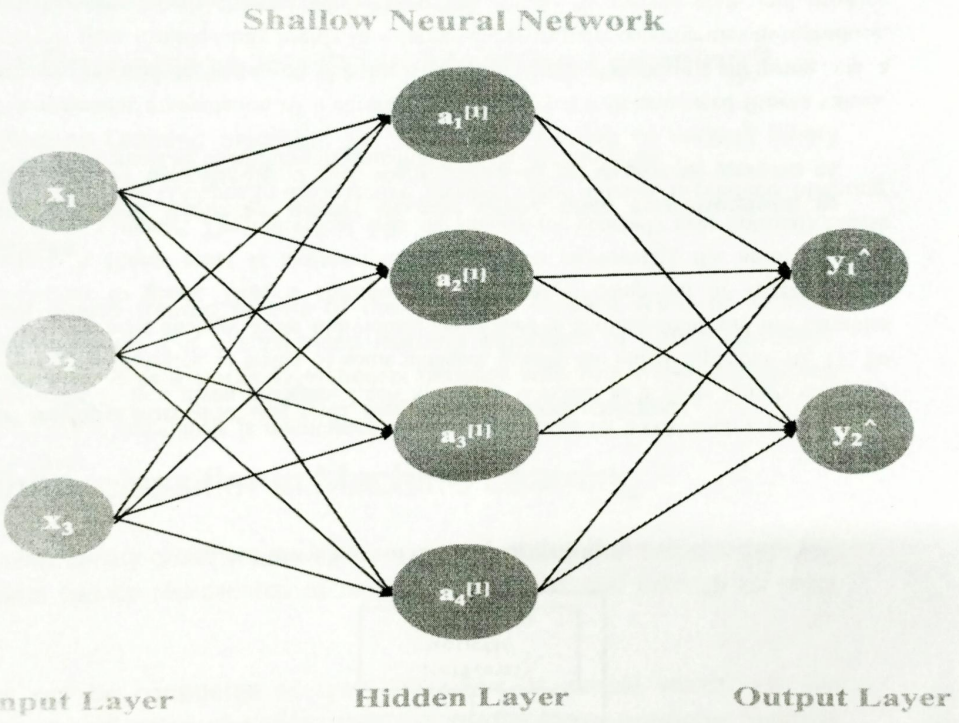
The output of step 1 is now passed through an *activation function*. The activation function g is a mathematical function that lets you transform the outputs to a desired non-linear format before it is sent to the next layer. It maps the summation result to a desired range. This helps in identifying whether the neuron needs to be fired.

For example, a sigmoid function maps values to the range [0,1], which is useful if you want your system to predict probabilities. Doing so lets you model complex non-linear decision boundaries.

Shallow neural network

In the previous section, you saw the calculations that happen within each perceptron. Now, you'll see how these perceptrons fit inside the network and how the flow is completed.

In its most basic form, a neural network contains three layers: *input layer*, *hidden layer*, and *output layer*. As shown in the following figure, a network with just one hidden layer is termed a *shallow neural network*.



The computations discussed in the previous sections happen for all neurons in a neural network including the output layer, and one such pass is known as *forward propagation*. After one forward pass is completed, the output layer must compare its results to the actual ground truth labels and adjust the weights based on the differences between the ground truth and the predicted values. This process is a backward pass through the neural network and is known as *backpropagation*. While the mathematics behind back propagation are outside the scope of this article, the basics of the process can be outlined as follows:

- The network works to minimize an objective function, for example, the error incurred across all points in a data sample.
- At the output layer, the network must calculate the total error (difference between actual and predicted values) for all data points and take its derivative with respect to weights at that layer. The derivative of error function with respect to weights is called the *gradient* of that layer.

Perceptron in Machine Learning

In Machine Learning and Artificial Intelligence, Perceptron is the most commonly used term for all folks. It is the primary step to learn Machine Learning and Deep Learning technologies, which consists of a set of weights, input values or scores, and a threshold. ***Perceptron is a building block of an Artificial Neural Network***. Initially, in the mid of 19th century, **Mr. Frank Rosenblatt** invented the Perceptron for performing certain calculations to detect input data capabilities or business intelligence. Perceptron is a linear Machine Learning algorithm used for supervised learning for various binary classifiers. This algorithm enables neurons to learn elements and processes them one by one during preparation.

What is the Perceptron model in Machine Learning?

Perceptron is Machine Learning algorithm for supervised learning of various binary classification tasks. Further, ***Perceptron is also understood as an Artificial Neuron or neural network unit that helps to detect certain input data computations in business intelligence.***

Perceptron model is also treated as one of the best and simplest types of Artificial Neural networks. However, it is a supervised learning algorithm of binary classifiers. Hence, we can consider it as a single-layer neural network with four main parameters, i.e., **input values, weights and Bias, net sum, and an activation function.**

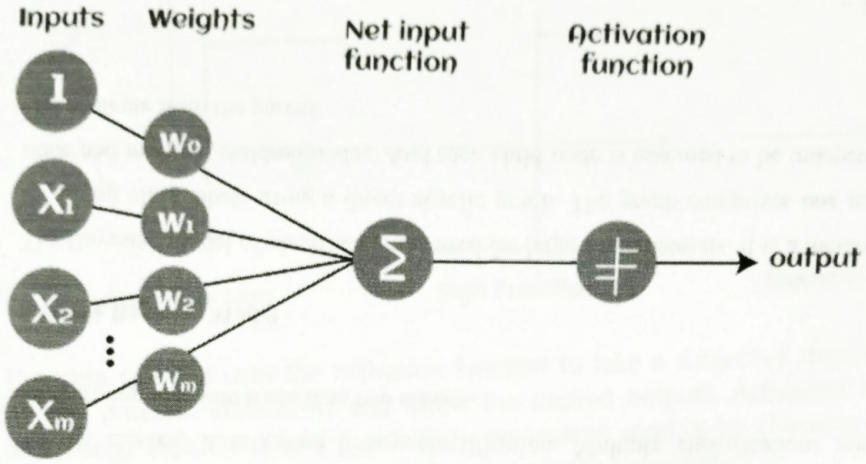
What is Binary classifier in Machine Learning?

In Machine Learning, binary classifiers are defined as the function that helps in deciding whether input data can be represented as vectors of numbers and belongs to some specific class.

Binary classifiers can be considered as linear classifiers. In simple words, we can understand it as a ***classification algorithm that can predict linear predictor function in terms of weight and feature vectors.***

Basic Components of Perceptron

Mr. Frank Rosenblatt invented the perceptron model as a binary classifier which contains three main components. These are as follows:



- **Input Nodes or Input Layer:**

This is the primary component of Perceptron which accepts the initial data into the system for further processing. Each input node contains a real numerical value.

- **Wight and Bias:**

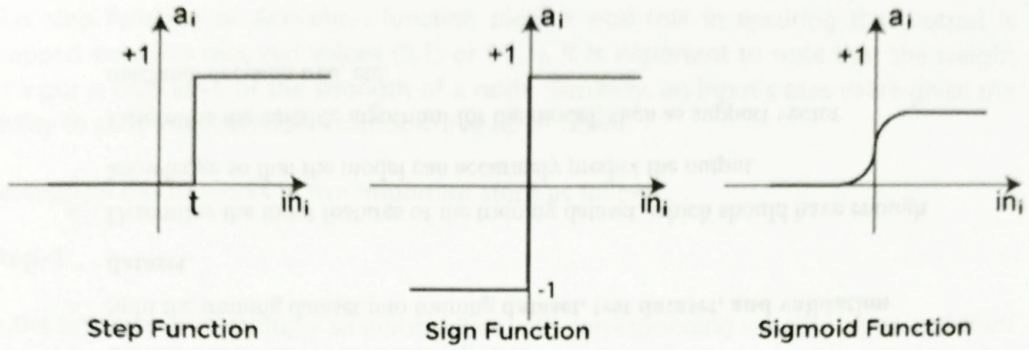
Weight parameter represents the strength of the connection between units. This is another most important parameter of Perceptron components. Weight is directly proportional to the strength of the associated input neuron in deciding the output. Further, Bias can be considered as the line of intercept in a linear equation.

- **Activation Function:**

These are the final and important components that help to determine whether the neuron will fire or not. Activation Function can be considered primarily as a step function.

Types of Activation functions:

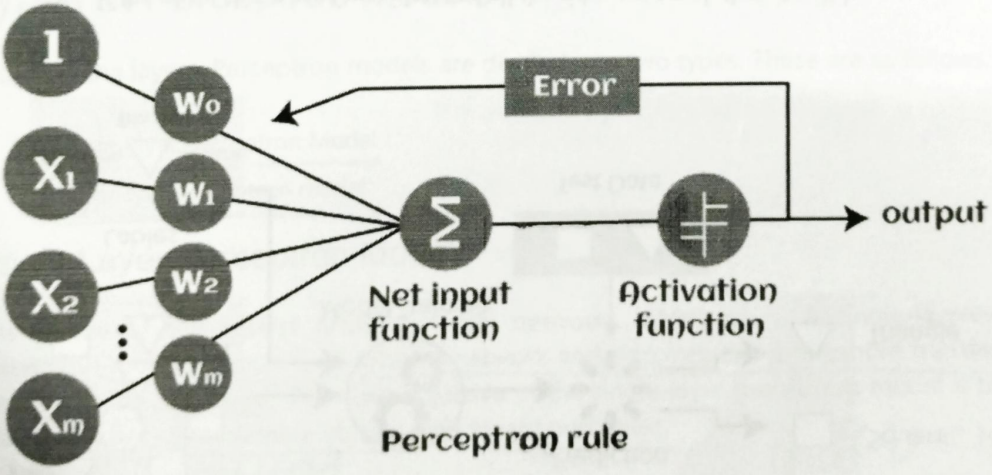
- Sign function
- Step function, and
- Sigmoid function



The data scientist uses the activation function to take a subjective decision based on various problem statements and forms the desired outputs. Activation function may differ (e.g., Sign, Step, and Sigmoid) in perceptron models by checking whether the learning process is slow or has vanishing or exploding gradients.

How does Perceptron work?

In Machine Learning, Perceptron is considered as a single-layer neural network that consists of four main parameters named input values (Input nodes), weights and Bias, net sum, and an activation function. The perceptron model begins with the multiplication of all input values and their weights, then adds these values together to create the weighted sum. Then this weighted sum is applied to the activation function 'f' to obtain the desired output. This activation function is also known as the **step function** and is represented by **'f'**.



This step function or Activation function plays a vital role in ensuring that output is mapped between required values (0,1) or (-1,1). It is important to note that the weight of input is indicative of the strength of a node. Similarly, an input's bias value gives the ability to shift the activation function curve up or down.

Perceptron model works in two important steps as follows:

Step-1

In the first step first, multiply all input values with corresponding weight values and then add them to determine the weighted sum. Mathematically, we can calculate the weighted sum as follows:

$\sum wi*xi = x1*w1 + x2*w2 + ...wn*xn$

Add a special term called **bias 'b'** to this weighted sum to improve the model's performance.

$\sum wi*xi + b$

Step- 2

In the second step, an activation function is applied with the above-mentioned weighted sum, which gives us output either in binary form or a continuous value as follows:

$Y = f(\sum wi*xi + b)$

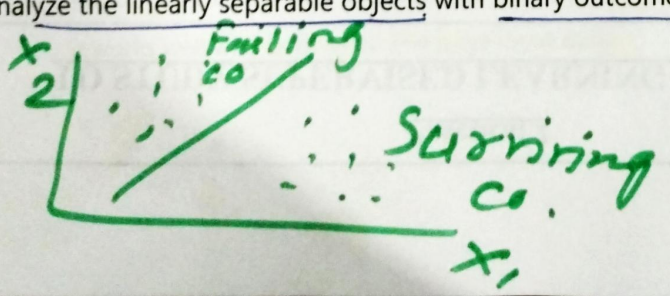
Types of Perceptron Models

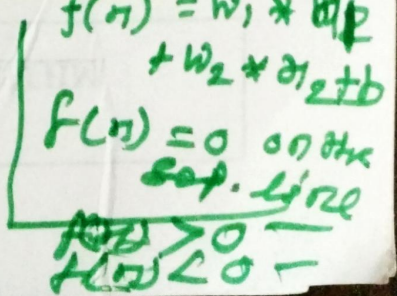
Based on the layers, Perceptron models are divided into two types. These are as follows:

1. Single-layer Perceptron Model
2. Multi-layer Perceptron model

Single Layer Perceptron Model:

This is one of the easiest Artificial neural networks (ANN) types. A single-layered perceptron model consists feed-forward network and also includes a threshold transfer function inside the model. The main objective of the single-layer perceptron model is to analyze the linearly separable objects with binary outcomes.





In a single layer perceptron model, its algorithms do not contain recorded data, so it begins with inconstantly allocated input for weight parameters. Further, it sums up all inputs (weight). After adding all inputs, if the total sum of all inputs is more than a pre-determined value, the model gets activated and shows the output value as +1.

If the outcome is same as pre-determined or threshold value, then the performance of this model is stated as satisfied, and weight demand does not change. However, this model consists of a few discrepancies triggered when multiple weight inputs values are fed into the model. Hence, to find desired output and minimize errors, some changes should be necessary for the weights input.

"*Single-layer perceptron can learn only linearly separable patterns.*"

Multi-Layered Perceptron Model:

Like a single-layer perceptron model, a multi-layer perceptron model also has the same model structure but has a greater number of hidden layers.

The multi-layer perceptron model is also known as the Backpropagation algorithm, which executes in two stages as follows:

- **Forward Stage:** Activation functions start from the input layer in the forward stage and terminate on the output layer.
- **Backward Stage:** In the backward stage, weight and bias values are modified as per the model's requirement. In this stage, the error between actual output and demanded originated backward on the output layer and ended on the input layer.

Hence, a multi-layered perceptron model has considered as multiple artificial neural networks having various layers in which activation function does not remain linear, similar to a single layer perceptron model. Instead of linear, activation function can be executed as sigmoid, TanH, ReLU, etc., for deployment.

A multi-layer perceptron model has greater processing power and can process linear and non-linear patterns. Further, it can also implement logic gates such as AND, OR, XOR, NAND, NOT, XNOR, NOR.

Advantages of Multi-Layer Perceptron:

- A multi-layered perceptron model can be used to solve complex non-linear problems.
- It works well with both small and large input data.

- It helps us to obtain quick predictions after the training.
- It helps to obtain the same accuracy ratio with large as well as small data.

Disadvantages of Multi-Layer Perceptron:

- In Multi-layer perceptron, computations are difficult and time-consuming.
- In multi-layer Perceptron, it is difficult to predict how much the dependent variable affects each independent variable.
- The model functioning depends on the quality of the training.

Perceptron Function

Perceptron function "f(x)" can be achieved as output by multiplying the input 'x' with the learned weight coefficient 'w'.

Mathematically, we can express it as follows:

f(x)=1; if w.x+b>0

otherwise, f(x)=0

- 'w' represents real-valued weights vector
- 'b' represents the bias
- 'x' represents a vector of input x values.

Characteristics of Perceptron

The perceptron model has the following characteristics.

1. Perceptron is a machine learning algorithm for supervised learning of binary classifiers.
2. In Perceptron, the weight coefficient is automatically learned.
3. Initially, weights are multiplied with input features, and the decision is made whether the neuron is fired or not.
4. The activation function applies a step rule to check whether the weight function is greater than zero.

5. The linear decision boundary is drawn, enabling the distinction between the two linearly separable classes +1 and -1.
6. If the added sum of all input values is more than the threshold value, it must have an output signal; otherwise, no output will be shown.

Limitations of Perceptron Model

A perceptron model has limitations as follows:

- The output of a perceptron can only be a binary number (0 or 1) due to the hard limit transfer function.
- Perceptron can only be used to classify the linearly separable sets of input vectors. If input vectors are non-linear, it is not easy to classify them properly.

Future of Perceptron

The future of the Perceptron model is much bright and significant as it helps to interpret data by building intuitive patterns and applying them in the future. Machine learning is a rapidly growing technology of Artificial Intelligence that is continuously evolving and in the developing phase; hence the future of perceptron technology will continue to support and facilitate analytical behavior in machines that will, in turn, add to the efficiency of computers.

The perceptron model is continuously becoming more advanced and working efficiently on complex problems with the help of artificial neurons.

Single Layer Perceptron

The single-layer perceptron was the first neural network model, proposed in 1958 by Frank Rosenbluth. It is one of the earliest models for learning. Our goal is to find a linear decision function measured by the weight vector w and the bias parameter b.

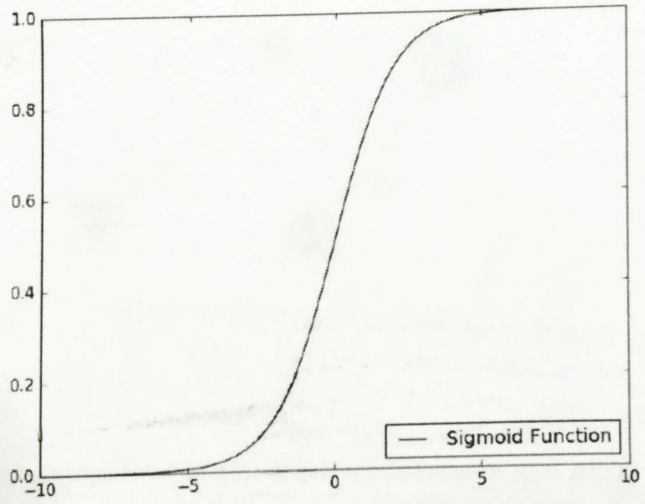
To understand the perceptron layer, it is necessary to comprehend artificial neural networks (ANNs).

The artificial neural network (ANN) is an information processing system, whose mechanism is inspired by the functionality of biological neural circuits. An artificial neural network consists of several processing units that are interconnected.

This is the first proposal when the neural model is built. The content of the neuron's local memory contains a vector of weight.

The single vector perceptron is calculated by calculating the sum of the input vector multiplied by the corresponding element of the vector, with each increasing the amount of the corresponding component of the vector by weight. The value that is displayed in the output is the input of an activation function.

Let us focus on the implementation of a single-layer perceptron for an image classification problem using TensorFlow. The best example of drawing a single-layer perceptron is through the representation of "**logistic regression**."



Now, We have to do the following necessary steps of training logistic regression-

- The weights are initialized with the random values at the origination of each training.
- For each element of the training set, the error is calculated with the difference between the desired output and the actual output. The calculated error is used to adjust the weight.
- The process is repeated until the fault made on the entire training set is less than the specified limit until the maximum number of iterations has been reached.

Feed Forward Process in Deep Neural Network

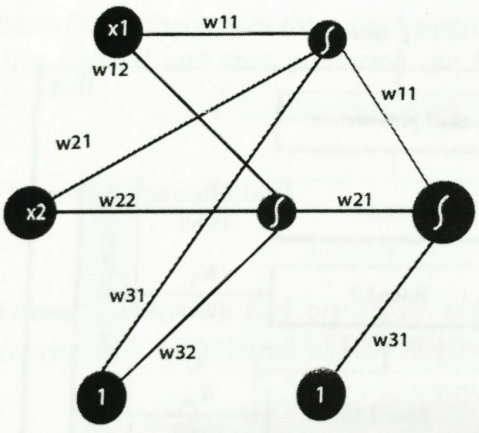
Now, we know how with the combination of lines with different weight and biases can result in non-linear models. How does a neural network know what weight and biased values to have in each layer? It is no different from how we did it for the single based perceptron model.

We are still making use of a gradient descent optimization algorithm which acts to minimize the error of our model by iteratively moving in the direction with the steepest descent, the direction which updates the parameters of our model while ensuring the minimal error. It updates the weight of every model in every single layer. We will talk more about optimization algorithms and backpropagation later.

It is important to recognize the subsequent training of our neural network. Recognition is done by dividing our data samples through some decision boundary.

"The process of receiving an input to produce some kind of output to make some kind of prediction is known as Feed Forward." Feed Forward neural network is the core of many other important neural networks such as convolution neural network.

In the feed-forward neural network, there are not any feedback loops or connections in the network. Here is simply an input layer, a hidden layer, and an output layer.



There can be multiple hidden layers which depend on what kind of data you are dealing with. The number of hidden layers is known as the depth of the neural network. The deep neural network can learn from more functions. Input layer first provides the neural network with data and the output layer then make predictions on that data which is based on a series of functions. ReLU Function is the most commonly used activation function in the deep neural network.

To gain a solid understanding of the feed-forward process, let's see this mathematically.

1) The first input is fed to the network, which is represented as matrix x1, x2, and one where one is the bias value.

$$[x_1 \quad x_2 \quad 1]$$

2) Each input is multiplied by weight with respect to the first and second model to obtain their probability of being in the positive region in each model.

So, we will multiply our inputs by a matrix of weight using matrix multiplication.

$$[x_1 \quad x_2 \quad 1] = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = [\text{score} \quad \text{score}]$$

3) After that, we will take the sigmoid of our scores and gives us the probability of the point being in the positive region in both models.

$$\frac{1}{1+e^{-x}}[\text{score score}] = \text{probability}$$

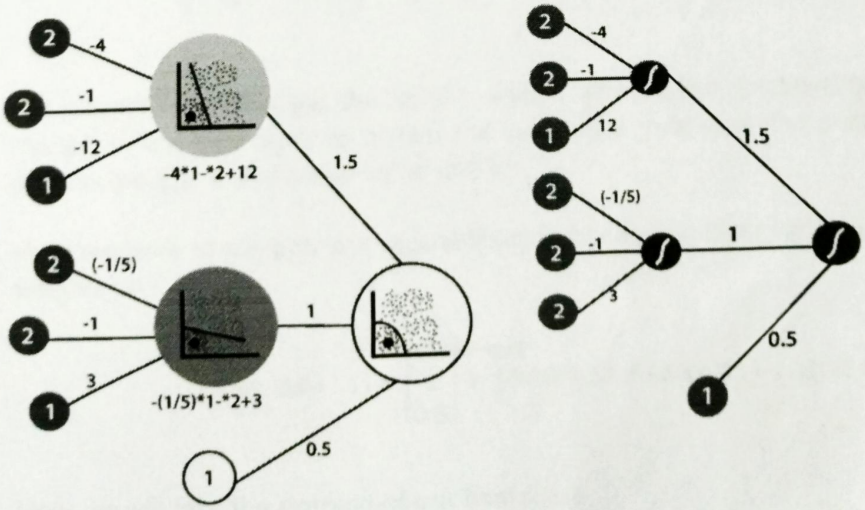
4) We multiply the probability which we have obtained from the previous step with the second set of weights. We always include a bias of one whenever taking a combination of inputs.

$$[\text{probability} \quad \text{probability} \quad 1] \times \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} = [\text{score}]$$

And as we know to obtain the probability of the point being in the positive region of this model, we take the sigmoid and thus producing our final output in a feed-forward process.

$$\frac{1}{1+e^{-x}}[\text{score}] = [\text{probability}]$$

Let takes the neural network which we had previously with the following linear models and the hidden layer which combined to form the non-linear model in the output layer.



So, what we will do we use our non-linear model to produce an output that describes the probability of the point being in the positive region. The point was represented by 2 and 2. Along with bias, we will represent the input as

$$[2 \quad 2 \quad 1]$$

The first linear model in the hidden layer recall and the equation defined it

$$-4x_1 - x_2 + 12$$

Which means in the first layer to obtain the linear combination the inputs are multiplied by -4, -1 and the bias value is multiplied by twelve.

$$[2 \quad 2 \quad 1] \quad \times \begin{bmatrix} -4 & w_{12} \\ -1 & w_{22} \\ 12 & w_{32} \end{bmatrix}$$

The weight of the inputs are multiplied by -1/5, 1, and the bias is multiplied by three to obtain the linear combination of that same point in our second model.

$$[2 \quad 2 \quad 1] \quad \times \begin{bmatrix} -4 & -1/5 \\ -1 & -1 \\ 12 & 3 \end{bmatrix}$$

$$\begin{bmatrix} 2(-4) + 2(-1) + 1(12) \\ 2(-4) + 2(-1) + 1(12) \end{bmatrix}$$

$$[2 \quad 0.6]$$

Now, to obtain the probability of the point is in the positive region relative to both models we apply sigmoid to both points as

$$\left[\frac{1}{1+\frac{1}{e^2}} \quad \frac{1}{1+\frac{1}{e^{0.6}}}\right] = \left[\frac{1}{1+\frac{1}{e^2}} \quad \frac{1}{1+\frac{1}{e^{0.6}}}\right] = [0.88 \quad 0.64]$$

The second layer **contains the** weights which dictated the combination of the linear models in the **first layer** to obtain the non-linear model in the second layer. The weights are **1.5, 1**, and a bias value of 0.5.

Now, **we** have to multiply our probabilities from the first layer with the second set of weights as

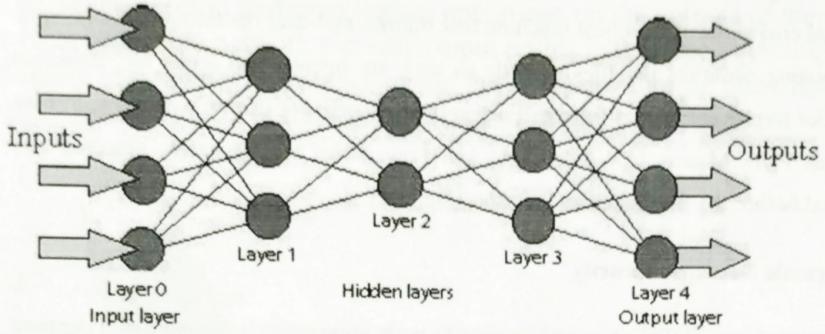
$$[0.88 \quad 0.64 \quad 1] \times \begin{bmatrix} 1.5 \\ 1 \\ 0.5 \end{bmatrix} = [0.88(1.5) + (0.64)(1) + 1(0.5)] = 2.46$$

Now, we will take the sigmoid of our final score

$$\frac{1}{1+\frac{1}{e^{2.46}}}=[0.92]$$

It is complete math behind the feed forward process where the inputs from the input traverse the entire depth of the neural network. In this example, there is only one hidden layer. Whether there is one hidden layer or twenty, the computational processes are the same for all hidden layers.

Feed-Forward networks:



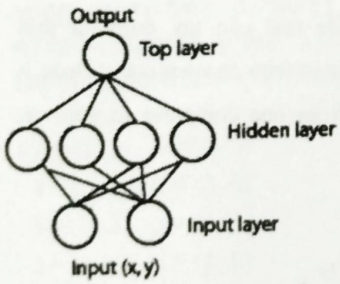
(Fig.1) A feed-forward network.

Feed-forward networks have the following characteristics:

1. Perceptrons are arranged in layers, with the first layer taking in inputs and the last layer producing outputs. The middle layers have no connection with the external world, and hence are called hidden layers.
2. Each perceptron in one layer is connected to every perceptron on the next layer. Hence information is constantly "fed forward" from one layer to the next., and this explains why these networks are called feed-forward networks.
3. There is no connection among perceptrons in the same layer.

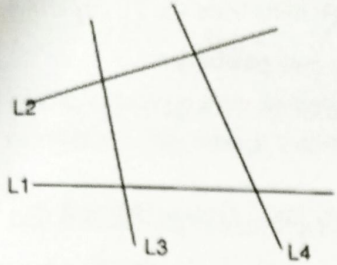
What's so cool about feed-forward networks?

Recall that a single perceptron can classify points into two regions that are linearly separable. Now let us extend the discussion into the separation of points into two regions that are not linearly separable. Consider the following network:



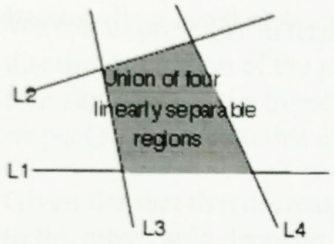
(Fig.2) A feed-forward network with one hidden layer.

The same (x, y) is fed into the network through the perceptrons in the input layer. With four perceptrons that are independent of each other in the hidden layer, the point is classified into 4 pairs of linearly separable regions, each of which has a unique line separating the region.



(Fig.3) 4 lines each dividing the plane into 2 linearly separable regions.

The top perceptron performs logical operations on the outputs of the hidden layers so that the whole network classifies input points in 2 regions that might not be linearly separable. For instance, using the AND operator on these four outputs, one gets the intersection of the 4 regions that forms the center region.



(Fig.4) Intersection of 4 linearly separable regions forms the center region.

By varying the number of nodes in the hidden layer, the number of layers, and the number of input and output nodes, one can classification of points in arbitrary dimension into an arbitrary number of groups. Hence feed-forward networks are commonly used for classification.

Backpropagation -- learning in feed-forward networks:

Learning in feed-forward networks belongs to the realm of supervised learning, in which pairs of input and output values are fed into the network for many cycles, so that the network 'learns' the relationship between the input and output.

We provide the network with a number of training samples, which consists of an input vector i and its desired output o. For instance, in the classification problem, suppose we have points (1, 2) and (1, 3) belonging to group 0, points (2, 3) and (3, 4) belonging to group 1, (5, 6) and (6, 7) belonging to group 2, then for a feed-forward network with 2 input nodes and 2 output nodes, the training set would be:

{ i = (1, 2) , o =(0, 0)
  i = (1, 3) , o = (0, 0)
  i = (2, 3) , o = (1, 0)
  i = (3, 4) , o = (1, 0)
  i = (5, 6) , o = (0, 1)
  i = (6, 7) , o = (0, 1) }

The basic rule for choosing the number of output nodes depends on the number of different regions. It is advisable to use a unary notation to represent the different

regions, i.e. for each output only one node can have value 1. Hence the number of output nodes = number of different regions -1.

In backpropagation learning, every time an input vector of a training sample is presented, the output vector o is compared to the desired value d.

The comparison is done by calculating the squared difference of the two:

$$\mathbf{Err = (d-o)^2}$$

The value of Err tells us how far away we are from the desired value for a particular input. The goal of backpropagation is to minimize the sum of Err for all the training samples, so that the network behaves in the most "desirable" way.

Minimize $\sum \mathbf{Err = (d-o)^2}$

We can express Err in terms of the input vector (i), the weight vectors (w), and the threshold function of the perceptions. Using a continuous function (instead of the step function) as the threshold function, we can express the gradient of Err with respect to the w in terms of w and i.

Given the fact that decreasing the value of w in the direction of the gradient leads to the most rapid decrease in Err, we update the weight vectors every time a sample is presented using the following formula:

$$w_{new} = w_{old} - n \frac{\delta Err}{\delta w}$$

where n is the learning rate (a small number ~ 0.1)

Using this algorithm, the weight vectors are modified so that the value of Err for a particular input sample decreases a little bit every time the sample is presented. When all the samples are presented in turns for many cycles, the sum of Err gradually decreases to a minimum value, which is our goal as mentioned above.

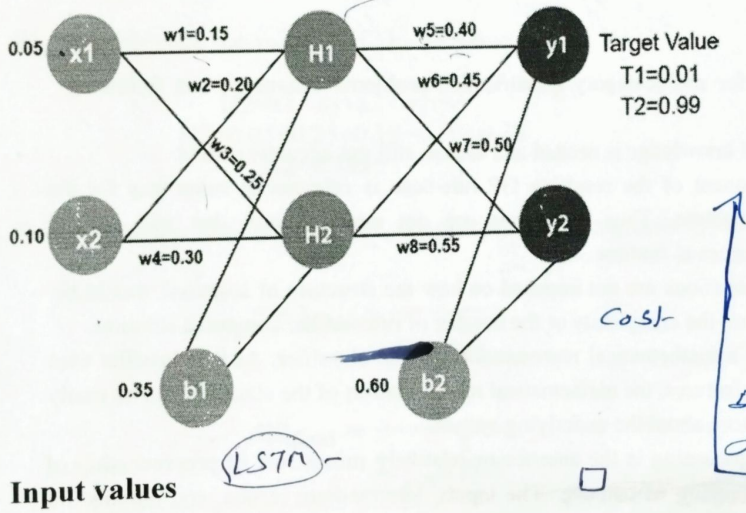
Backpropagation Process in Deep Neural Network

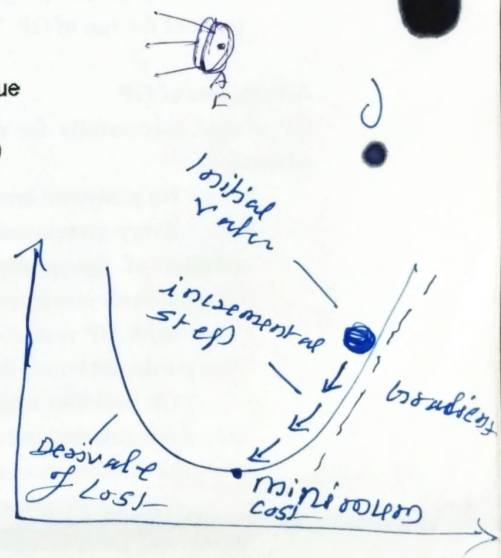
Backpropagation is one of the important concepts of a neural network. Our task is to classify our data best. For this, we have to update the weights of parameter and bias, but how can we do that in a deep neural network? In the linear regression model, we use gradient descent to optimize the parameter. Similarly here we also use gradient descent algorithm using Backpropagation.

For a single training example, **Backpropagation** algorithm calculates the gradient of the **error function**. Backpropagation can be written as a function of the neural network. Backpropagation algorithms are a set of methods used to efficiently train artificial neural networks following a gradient descent approach which exploits the chain rule.

The main features of Backpropagation are the iterative, recursive and efficient method through which it calculates the updated weight to improve the network until it is not able to perform the task for which it is being trained. Derivatives of the activation function to be known at network design time is required to Backpropagation.

Now, how error function is used in Backpropagation and how Backpropagation works? Let start with an example and do it mathematically to understand how exactly updates the weight using Backpropagation.





Input values

X1=0.05
X2=0.10

Initial weight

W1=0.15 w5=0.40
W2=0.20 w6=0.45
W3=0.25 w7=0.50
W4=0.30 w8=0.55

Bias Values

b1=0.35 b2=0.60

Target Values

T1=0.01
T2=0.99

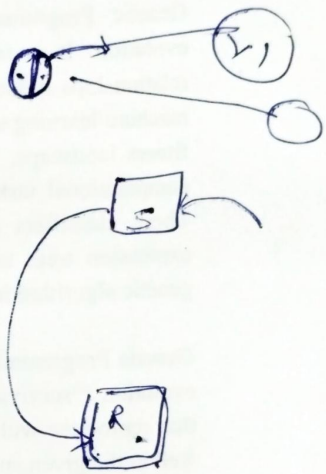
Now, we first calculate the values of H1 and H2 by a forward pass.

Forward Pass

To find the value of H1 we first multiply the input value from the weights as

$$H1=x1\times w_1+x2\times w_2+b1$$
$$H1=0.05\times 0.15+0.10\times 0.20+0.35$$
$$\mathbf{H1=0.3775}$$

To calculate the final result of H1, we performed the sigmoid function as



$$H1_{final}=\frac{1}{1+\frac{1}{e^{H1}}}$$

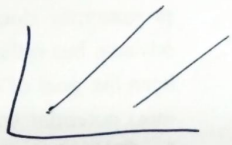
$$H1_{final}=\frac{1}{1+\frac{1}{e^{0.3775}}}$$

$$\mathbf{H1_{final}=0.593269992}$$

We will calculate the value of H2 in the same way as H1

$$H2=x1\times w_3+x2\times w_4+b1$$
$$H2=0.05\times 0.25+0.10\times 0.30+0.35$$
$$\mathbf{H2=0.3925}$$

To calculate the final result of H1, we performed the sigmoid function as



$$H2_{final}=\frac{1}{1+\frac{1}{e^{H2}}}$$

$$H2_{final}=\frac{1}{1+\frac{1}{e^{0.3925}}}$$

$$\mathbf{H2_{final}=0.596884378}$$

Now, we calculate the values of y1 and y2 in the same way as we calculate the H1 and H2.

To find the value of y1, we first multiply the input value i.e., the outcome of H1 and H2 from the weights as

$$y1=H1\times w_5+H2\times w_6+b2$$
$$y1=0.593269992\times 0.40+0.596884378\times 0.45+0.60$$
$$\mathbf{y1=1.10590597}$$

To calculate the final result of y1 we performed the sigmoid function as

$$y1_{final} = \frac{1}{1 + \frac{1}{e^{y1}}}$$

$$y1_{final} = \frac{1}{1 + \frac{1}{e^{1.10590597}}}$$

$$\mathbf{y1_{final} = 0.75136507}$$

We will calculate the value of y2 in the same way as y1

$$y2 = H1 \times w_7 + H2 \times w_8 + b2$$
$$y2 = 0.593269992 \times 0.50 + 0.596884378 \times 0.55 + 0.60$$
$$\mathbf{y2 = 1.2249214}$$

To calculate the final result of H1, we performed the sigmoid function as

$$y2_{final} = \frac{1}{1 + \frac{1}{e^{y2}}}$$

$$y2_{final} = \frac{1}{1 + \frac{1}{e^{1.2249214}}}$$

$$\mathbf{y2_{final} = 0.772928465}$$

Our target values are 0.01 and 0.99. Our y1 and y2 value is not matched with our target values T1 and T2.

Now, we will find the **total error**, which is simply the difference between the outputs from the target outputs. The total error is calculated as

(MSE) (Mean Squared Error)

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

So, the total error is

$$= \frac{1}{2}(t1 - y1_{final})^2 + \frac{1}{2}(T2 - y2_{final})^2$$

$$= \frac{1}{2}(0.01 - 0.75136507)^2 + \frac{1}{2}(0.99 - 0.772928465)^2$$

$$= 0.274811084 + 0.0235600257$$

$$\mathbf{E_{total} = 0.29837111}$$

Now, we will backpropagate this error to update the weights using a backward pass.

Backward pass at the output layer

To update the weight, we calculate the error correspond to each weight with the help of a total error. The error on weight w is calculated by differentiating total error with respect to w.

$$\text{Error}_w = \frac{\partial E_{total}}{\partial w}$$

We perform backward process so first consider the last weight w5 as

$$\text{Error}_{w5} = \frac{\partial E_{total}}{\partial w5} \ldots\ldots\ldots (1)$$

$$E_{total} = \frac{1}{2}(T1 - y1_{final})^2 + \frac{1}{2}(T2 - y2_{final})^2 \ldots\ldots\ldots (2)$$

From equation two, it is clear that we cannot partially differentiate it with respect to w5 because there is no any w5. We split equation one into multiple terms so that we can easily differentiate it with respect to w5 as

$$\frac{\partial E_{total}}{\partial w5} = \frac{\partial E_{total}}{\partial y1_{final}} \times \frac{\partial y1\_final}{\partial y1} \times \frac{\partial y1}{\partial w5} \ldots\ldots\ldots\ldots (3)$$

Now, we calculate each term one by one to differentiate E_{total} with respect to w5 as

$$\frac{\partial E_{total}}{\partial y1_{final}} = \frac{\partial(\frac{1}{2}(T1 - y1_{final})^2 + \frac{1}{2}(T2 - y2_{final})^2)}{\partial y1_{final}}$$

$$= 2 \times \frac{1}{2} \times (T1 - y1_{final})^{2-1} \times (-1) + 0$$

$$= -(T1 - y1_{final})$$

$$= -(0.01 - 0.75136507)$$

$$\frac{\partial E_{total}}{\partial y1_{final}} = 0.74136507 \ldots\ldots\ldots (4)$$

$$y1_{final} = \frac{1}{1 + e^{-y1}} \ldots\ldots\ldots\ldots\ldots (5)$$

$$\frac{\partial y1_{final}}{\partial y1} = \frac{\partial(\frac{1}{1 + e^{-y1}})}{\partial y1}$$

$$= \frac{e^{-y1}}{(1 + e^{-y1})^2}$$

$$= e^{-y1} \times (y1_{final})^2 \ldots\ldots\ldots\ldots\ldots (6)$$

$$y1_{final} = \frac{1}{1 + e^{-y1}}$$

$$e^{-y1} = \frac{1 - y1_{final}}{y1_{final}} \ldots\ldots\ldots\ldots (7)$$

Putting the value of e^{-y} in equation (5)

$$= \frac{1 - y1_{final}}{y1_{final}} \times (y1_{final})^2$$

$$= y1_{final} \times (1 - y1_{final})$$

$$= 0.75136507 \times (1 - 0.75136507)$$

$$\frac{\partial y1_{final}}{\partial y1} = \mathbf{0.186815602} \ldots\ldots\ldots \mathbf{(8)}$$

$$y1 = H1_{final} \times w5 + H2_{final} \times w6 + b2 \ldots\ldots\ldots\ldots\ldots\ldots (9)$$

$$\frac{\partial y1}{\partial w5} = \frac{\partial (H1_{final} \times w5 + H2_{final} \times w6 + b2)}{\partial w5}$$

$$= H1_{final}$$

$$\frac{\partial y1}{\partial w5} = \mathbf{0.596884378} \ldots\ldots\ldots \mathbf{(10)}$$

So, we put the values of $\frac{\partial E_{total}}{\partial y1_{final}}$, $\frac{\partial y1_{final}}{\partial y1}$, and $\frac{\partial y1}{\partial w5}$ in equation no (3) to find the final result.

$$\frac{\partial E_{total}}{\partial w5} = \frac{\partial E_{total}}{\partial y1_{final}} \times \frac{\partial y1_{final}}{\partial y1} \times \frac{\partial y1}{\partial w5}$$

$$= 0.74136507 \times 0.186815602 \times 0.593269992$$

$$\mathbf{Error_{w5}} = \frac{\partial E_{total}}{\partial w5} = \mathbf{0.0821670407} \ldots\ldots\ldots \mathbf{(11)}$$

Now, we will calculate the updated weight $w5_{new}$ with the help of the following formula

$$w5_{new} = w5 - \eta \times \frac{\partial E_{total}}{\partial w5} \quad \text{Here, } \eta = \text{learning rate} = 0.5$$

$$= 0.4 - 0.5 \times 0.0821670407$$

$$\mathbf{w5_{new}} = \mathbf{0.35891648} \ldots\ldots\ldots \mathbf{(12)}$$

In the same way, we calculate $w6_{new}$,$w7_{new}$, and $w8_{new}$ and this will give us the following values

$w5_{new}$=0.35891648
$w6_{new}$=408666186
$w7_{new}$=0.511301270
$w8_{new}$=0.561370121

Backward pass at Hidden layer

Now, we will backpropagate to our hidden layer and update the weight w1, w2, w3, and w4 as we have done with w5, w6, w7, and w8 weights.

We will calculate the error at w1 as

$$Error_{w1} = \frac{\partial E_{total}}{\partial w1}$$

$$E_{total} = \frac{1}{2}(T1 - y1_{final})^2 + \frac{1}{2}(T2 - y2_{final})^2$$

From equation (2), it is clear that we cannot partially differentiate it with respect to w1 because there is no any w1. We split equation (1) into multiple terms so that we can easily differentiate it with respect to w1 as

$$\frac{\partial E_{total}}{\partial w1} = \frac{\partial E_{total}}{\partial H1_{final}} \times \frac{\partial H1\_final}{\partial H1} \times \frac{\partial H1}{\partial w1} \dots\dots\dots\dots (13)$$

Now, we calculate each term one by one to differentiate E_{total} with respect to w1 as

$$\frac{\partial E_{total}}{\partial H1_{final}} = \frac{\partial(\frac{1}{2}(T1 - y1_{final})^2 + \frac{1}{2}(T2 - y2_{final})^2)}{\partial H1} \dots\dots\dots\dots (14)$$

We again split this because there is no any $H1^{final}$ term in E^{toatal} as

$$\frac{\partial E_{total}}{\partial H1_{final}} = \frac{\partial E_1}{\partial H1_{final}} + \frac{\partial E_2}{\partial H1_{final}} \dots\dots\dots (15)$$

$\frac{\partial E_1}{\partial H1_{final}}$ and $\frac{\partial E_2}{\partial H1_{final}}$ will again split because in E1 and E2 there is no H1 term. Splitting is done as

$$\frac{\partial E_1}{\partial H1_{final}} = \frac{\partial E_1}{\partial y1} \times \frac{\partial y1}{\partial H1_{final}} \dots\dots\dots (16)$$

$$\frac{\partial E_2}{\partial H1_{final}} = \frac{\partial E_2}{\partial y2} \times \frac{\partial y2}{\partial H1\_final} \dots\dots\dots (17)$$

We again Split both $\frac{\partial E_1}{\partial y1}$ and $\frac{\partial E_2}{\partial y2}$ because there is no any y1 and y2 term in E1 and E2. We split it as

$$\frac{\partial E_1}{\partial y1} = \frac{\partial E_1}{\partial y1_{fianl}} \times \frac{\partial y1_{final}}{\partial y1} \dots\dots\dots (18)$$

$$\frac{\partial E_2}{\partial y2} = \frac{\partial E_2}{\partial y2_{fianl}} \times \frac{\partial y2_{final}}{\partial y2} \dots\dots\dots (19)$$

Now, we find the value of $\frac{\partial E_1}{\partial y1}$ and $\frac{\partial E_2}{\partial y2}$ by putting values in equation (18) and (19) as

From equation (18)

$$\frac{\partial E_1}{\partial y1} = \frac{\partial E_1}{\partial y1_{fianl}} \times \frac{\partial y1_{final}}{\partial y1}$$

$$= \frac{\partial(\frac{1}{2}(T1 - y1_{final})^2)}{\partial y1_{final}} \times \frac{\partial y1_{final}}{\partial y1}$$

$$= 2 \times \frac{1}{2}(T1 - y1_{final}) \times (-1) \times \frac{\partial y1_{final}}{\partial y1}$$

From equation (8)

$$= 2 \times \frac{1}{2}(0.01 - 0.75136507) \times (-1) \times 0.186815602$$

$$\frac{\partial E_1}{\partial y1} = 0.138498562 \ldots\ldots\ldots (20)$$

From equation (19)

$$\frac{\partial E_2}{\partial y2} = \frac{\partial E_2}{\partial y2_{fianl}} \times \frac{\partial y2_{final}}{\partial y2}$$

$$= \frac{\partial(\frac{1}{2}(T2 - y2_{final})^2)}{\partial y2_{final}} \times \frac{\partial y2_{final}}{\partial y2}$$

$$= 2 \times \frac{1}{2}(T2 - y2_{final}) \times (-1) \times \frac{\partial y2_{final}}{\partial y2} \ldots\ldots\ldots (21)$$

$$y2_{final} = \frac{1}{1 + e^{-y2}} \ldots\ldots\ldots\ldots\ldots (22)$$

$$\frac{\partial y2_{fianl}}{\partial y2} = \frac{\partial(\frac{1}{1 + e^{-y2}})}{\partial y2}$$

$$= \frac{e^{-y2}}{(1 + e^{-y2})^2}$$

$$= e^{-y2} \times (y2_{final})^2 \ldots\ldots\ldots\ldots\ldots (23)$$

$$y2_{final} = \frac{1}{1 + e^{-y2}}$$

$$e^{-y2} = \frac{1 - y2_{final}}{y2_{final}} \ldots\ldots\ldots\ldots (24)$$

Putting the value of e^{-y2} in equation (23)

$$= \frac{1 - y2_{final}}{y2_{final}} \times (y2_{final})^2$$

$$= y2_{final} \times (1 - y2_{final})$$

$$= 0.772928465 \times (1 - 0.772928465)$$

$$\frac{\partial y2_{fianl}}{\partial y2} = 0.175510053 \ldots\ldots\ldots (25)$$

From equation (21)

$$= 2 \times \frac{1}{2}(0.99 - 0.772928465) \times (-1) \times 0.175510053$$

$$\frac{\partial E_1}{\partial y1} = -0.0380982366126414 \ldots\ldots\ldots (26)$$

Now from equation (16) and (17)

$$\frac{\partial E_1}{\partial H1_{final}} = \frac{\partial E_1}{\partial y1} \times \frac{\partial y1}{\partial H1_{final}}$$

$$= 0.138498562 \times \frac{\partial(H1_{final} \times w_5 + H2_{final} \times w_6 + b2)}{\partial H1_{final}}$$

$$= 0.138498562 \times \frac{\partial(H1_{final} \times w_5 + H2_{final} \times w_6 + b2)}{\partial H1_{final}}$$

$$= 0.138498562 \times w5$$

$$= 0.138498562 \times 0.40$$

$$\mathbf{\frac{\partial E_1}{\partial H1_{final}} = 0.0553994248 \ldots\ldots\ldots (27)}$$

$$\frac{\partial E_2}{\partial H1_{final}} = \frac{\partial E_2}{\partial y2} \times \frac{\partial y2}{\partial H1\_final}$$

$$= -0.0380982366126414 \times \frac{\partial(H1_{final} \times w_7 + H2_{final} \times w_8 + b2)}{\partial H1_{final}}$$

$$= -0.0380982366126414 \times w7$$

$$= -0.0380982366126414 \times 0.50$$

$$\mathbf{\frac{\partial E_2}{\partial H1_{final}} = -0.0190491183063207 \ldots\ldots\ldots (28)}$$

Put the value of $\frac{\partial E_1}{\partial H1_{final}}$ and $\frac{\partial E_2}{\partial H1_{final}}$ in equation (15) as

$$\frac{\partial E_{total}}{\partial H1_{final}} = \frac{\partial E_1}{\partial H1_{final}} + \frac{\partial E_2}{\partial H1_{final}}$$

$$= 0.0553994248 + (-0.0190491183063207)$$

$$\mathbf{\frac{\partial E_{total}}{\partial H1_{final}} = 0.0364908241736793 \ldots\ldots\ldots (29)}$$

We have $\frac{\partial E_{total}}{\partial H1_{final}}$, we need to figure out $\frac{\partial H1\_final}{\partial H1}$, $\frac{\partial H1}{\partial w1}$ as

$$\frac{\partial H1_{final}}{\partial H1} = \frac{\partial(\frac{1}{1+e^{-H1}})}{\partial H1}$$

$$= \frac{e^{-H1}}{(1+e^{-H1})^2}$$

$$\mathbf{e^{-H1} \times (H1_{final})^2 \ldots\ldots\ldots (30)}$$

$$\mathbf{H1_{final} = \frac{1}{1+e^{-H1}}}$$

$$\mathbf{e^{-H1} = \frac{1 - H1_{final}}{H1_{final}} \ldots\ldots\ldots\ldots (31)}$$

Putting the value of e^{-H1} in equation (30)

$$= \frac{1 - H1_{final}}{H1_{final}} \times (H1_{final})^2$$

$$= H1_{final} \times (1 - H1_{final})$$

$$= 0.593269992 \times (1 - 0.593269992)$$

$$\frac{\partial H1_{final}}{\partial H1} = 0.2413007085923199$$

We calculate the partial derivative of the total net input to H1 with respect to w1 the same as we did for the output neuron:

$$H1 = H1_{final} \times w5 + H2_{final} \times w6 + b2 \ldots\ldots\ldots\ldots\ldots\ldots (32)$$

$$\frac{\partial y1}{\partial w1} = \frac{\partial(x1 \times w1 + x2 \times w3 + b1 \times 1)}{\partial w1}$$

$$= x1$$

$$\frac{\partial H1}{\partial w1} = 0.05 \ldots\ldots\ldots (33)$$

So, we put the values of $\frac{\partial E_{total}}{\partial H1_{final}}$, $\frac{\partial H1_{final}}{\partial H1}$, and $\frac{\partial H1}{\partial w1}$ in equation (13) to find the final result.

$$\frac{\partial E_{total}}{\partial w1} = \frac{\partial E_{total}}{\partial H1_{final}} \times \frac{\partial H1_{final}}{\partial H1} \times \frac{\partial H1}{\partial w1}$$

$$= 0.0364908241736793 \times 0.2413007085923199 \times 0.05$$

$$Error_{w1} = \frac{\partial E_{total}}{\partial w1} = 0.000438568 \ldots\ldots\ldots (34)$$

Now, we will calculate the updated weight $w1_{new}$ with the help of the following formula

$$w1_{new} = w1 - \eta \times \frac{\partial E_{total}}{\partial w1} \quad \text{Here } \eta = \text{learning rate} = 0.5$$

$$= 0.15 - 0.5 \times 0.000438568$$

$$w1_{new} = 0.149780716 \ldots\ldots\ldots (35)$$

In the same way, we calculate $w2_{new}$,$w3_{new}$, and w4 and this will give us the following values

$w1_{new}$=0.149780716
$w2_{new}$=0.19956143
$w3_{new}$=0.24975114
$w4_{new}$=0.29950229

We have updated all the weights. We found the error 0.298371109 on the network when we fed forward the 0.05 and 0.1 inputs. In the first round of Backpropagation, the total error is down to 0.291027924. After repeating this process 10,000, the total error is down to 0.0000351085. At this point, the outputs neurons generate 0.159121960 and 0.984065734 i.e., nearby our target value when we feed forward the 0.05 and 0.1.

Unit-4

Topic

KIRTI SINGH KS@4MAY

Mathematics behind two important optimization techniques in machine learning

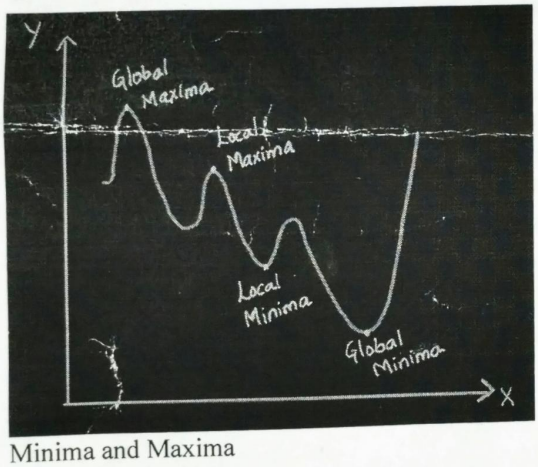
INTRODUCTION

Optimization is the process where we train the model iteratively that results in a maximum and minimum function evaluation. It is one of the most important phenomena in Machine Learning to get better results.

Why do we optimize our machine learning models? We compare the results in every iteration by changing the hyperparameters in each step until we reach the optimum results. We create an accurate model with less error rate. There are different ways using which we can optimize a model. In this article, let's discuss two important Optimization algorithms: **Gradient Descent and Stochastic Gradient Descent Algorithms**; how they are used in Machine Learning Models, and the mathematics behind them.

2. MAXIMA AND MINIMA

Maxima is the largest and Minima is the smallest value of a function within a given range. We represent them as below:



Minima and Maxima

Global Maxima and Minima: It is the maximum value and minimum value respectively on the entire domain of the function

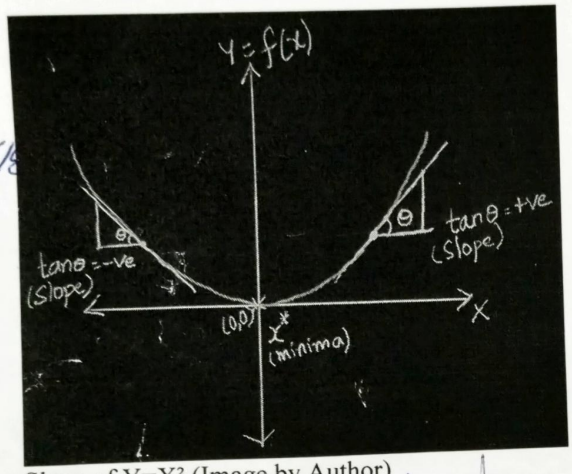
Local Maxima and Minima: It is the maximum value and minimum value respectively of the function within a given range.

There can be only one global minima and maxima but there can be more than one local minima and maxima.

3. GRADIENT DESCENT

Gradient Descent is an optimization algorithm and it finds out the local minima of a differentiable function. It is a minimization algorithm that minimizes a given function.

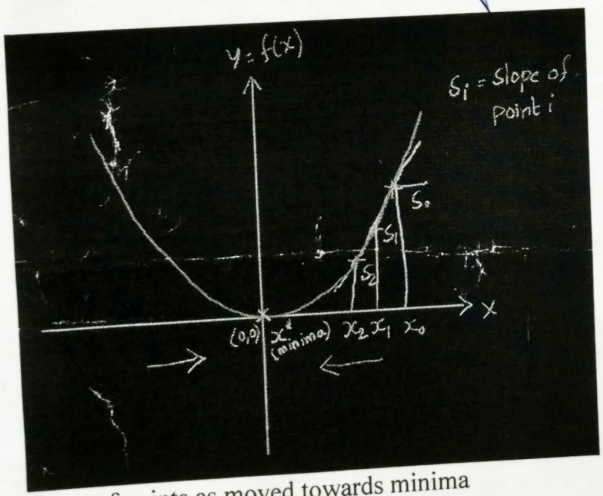
Let's see the geometric intuition of Gradient Descent:



Slope of $Y=X^2$ (Image by Author)

Let's take an example graph of a parabola, $Y=X^2$.

Here, the minima is the origin(0, 0). The slope here is Tanθ. So the slope on the right side is positive as $0<\theta<90$ and its Tanθ is a positive value. The slope on the left side is negative as $90<\theta<180$ and its Tanθ is a negative value.



Slope of points as moved towards minima

One important observation in the graph is that the slope changes its sign from positive to negative at minima. As we move closer to the minima, the slope reduces.

So, how does the Gradient Descent Algorithm work?

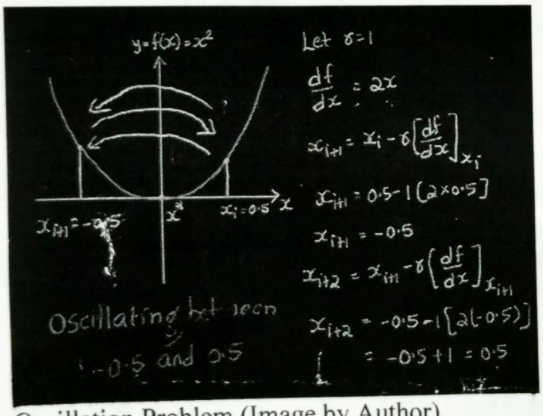
Objective: Calculate X*- local minimum of the function $Y=X^2$.

- Pick an initial point X_0 at random
- Calculate $X_1 = X_0 - r[df/dx]$ at X_0. r is Learning Rate (we'll discuss *r* in Learning Rate Section). Let us take r=1. Here, df/dx is nothing but the *gradient*.
- Calculate $X_2 = X_1 - r[df/dx]$ at X_1.
- Calculate for all the points: $X_1, X_2, X_3, \ldots\ldots, X_{i-1}, X_i$
- General formula for calculating local minima: $X_i = (X_{i-1}) - r[df/dx]$ at X_{i-1}
- When $(X_i — X_{i-1})$ is small, i.e., when X_{i-1}, X_i converge, we stop the iteration and declare $X^* = X_i$

4. LEARNING RATE

Learning Rate is a hyperparameter or tuning parameter that determines the step size at each iteration while moving towards minima in the function. For example, if r = 0.1 in the initial step, it can be taken as r=0.01 in the next step. Likewise it can be reduced exponentially as we iterate further. It is used more effectively in deep learning.

What happens if we keep r value as constant:



Oscillation Problem (Image by Author)

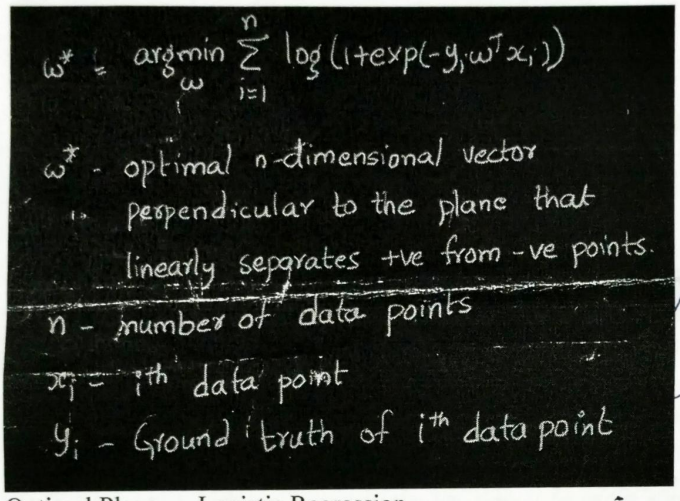
In the above example, we took r=1. As we calculate the points $X_i, X_{i+1}, X_{i+2}, \ldots$to find the local minima, X*, we can see that it is oscillating between X = -0.5 and X = 0.5.

When we keep r as constant, we end up with an *oscillation problem*. So, we have to reduce the "r" value with each iteration. Reduce the *r* value as the iteration step increases.

Important Note: Hyperparameters decide the bias-variance tradeoff. When *r* value is low, it could overfit the model and cause high variance. When *r* value is high, it could underfit the model and cause high bias. We can find the correct *r* value with *Cross Validation* technique. Plot a graph with different learning rates and check for the training loss with each value and choose the one with minimum loss.

5. GRADIENT DESCENT IN LOGISTIC REGRESSION

The formula for the optimal plane in logistic regression after applying sigmoid function is:



Optimal Plane — Logistic Regression

Apply Gradient Descent Algorithm on Logistic Regression:

$$f(w) = \sum_{i=1}^{n} \log\left(1+\exp(-y_i w^T x_i)\right)$$

$$\frac{df}{dw} = \sum_{i=1}^{n} \frac{(-y_i x_i)\exp(y_i w^T x_i)}{1+\exp(-y_i w^T x_i)}$$

$$w_1 = w_0 - \gamma\left[\frac{df}{dw}\right]_{w_0}$$

$$w_1 = w_0 - \gamma\left[\sum_{i=1}^{n} \frac{(-y_i x_i)\exp(-y_i w^T x_i)}{1+\exp(-y_i w^T x_i)}\right]$$

$$w_2 = w_1 - \gamma\left[\sum_{i=1}^{n} \frac{(-y_i x_i)\exp(-y_i w^T x_i)}{1+\exp(-y_i w^T x_i)}\right]$$

and so on...

$$w_i = w_{i-1} - \gamma\left[\sum_{i=1}^{n} \frac{(-y_i x_i)\exp(-y_i w^T x_i)}{1+\exp(-y_i w^T x_i)}\right]$$

Gradient Descent in Logistic Regression (Image by Author)

We'll calculate W_0, W_1, W_2, …., W_{i-1}, W_i to find W*. When (W_{i-1} — W_i) is small i.e., when W_{i-1}, W_i converge, we declare $W^* = W_i$

The **disadvantage** of Gradient Descent:

When n(number of data points) is large, the time it takes for *k* iterations to calculate the optimum vector becomes very large.

Time Complexity: $O(kn^2)$

This problem is solved with Stochastic Gradient Descent and is discussed in the next section.

5. STOCHASTIC GRADIENT DESCENT(SGD)

In SGD, we do not use all the data points but a sample of it to calculate the local minimum of the function. Stochastic basically means Probabilistic. So we select points randomly from the population.

- **SGD in Logistic Regression**

$$\omega_i = \omega_{i-1} - \alpha \left[\sum_{i=1}^{m} \frac{(-y_i x_i) \exp(-y_i \omega^T x_i)}{1 + \exp(-y_i \omega^T x_i)} \right]$$

$$1 < m < n$$

Stochastic Gradient Descent in Logistic Regression (Image by Author)

Here, *m* is the sample of data selected randomly from the population, *n*

Time Complexity: $O(km^2)$. m is significantly lesser than n. So, it takes lesser time to compute when compared to Gradient Descent.

6. CONCLUSION

In this article, we discussed Optimization algorithms like Gradient Descent and Stochastic Gradient Descent and their application in Logistic Regression. SGD is the most important optimization algorithm in Machine Learning. Mostly, it is used in Logistic Regression and Linear Regression. It is extended in Deep Learning as Adam, Adagrad.

Gradient Descent in Machine Learning

Gradient Descent is known as one of the most commonly used optimization algorithms to train machine learning models by means of minimizing errors between actual and expected results. Further, gradient descent is also used to train Neural Networks.

In mathematical terminology, Optimization algorithm refers to the task of minimizing/maximizing an objective function f(x) parameterized by x. Similarly, in machine learning, optimization is the task of minimizing the cost function parameterized by the model's parameters. The main objective of gradient descent is to minimize the convex function using iteration of parameter updates. Once these machine learning models are optimized, these models can be used as powerful tools for Artificial Intelligence and various computer science applications.

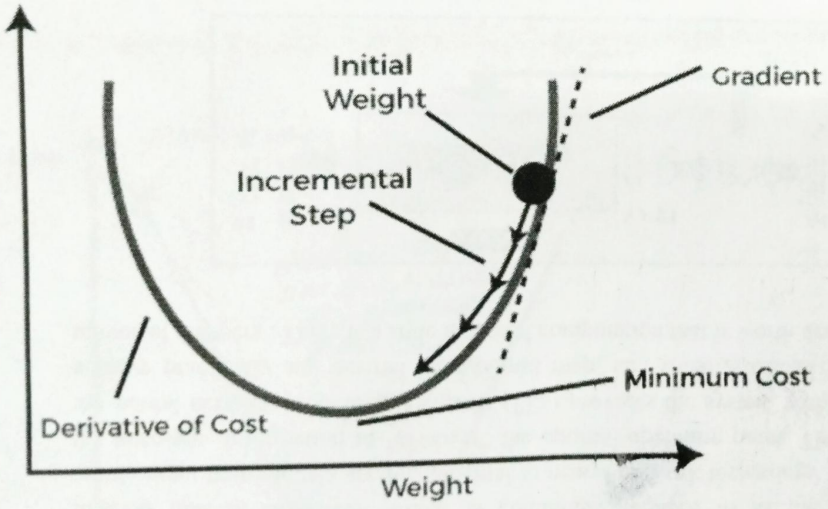
In this tutorial on Gradient Descent in Machine Learning, we will learn in detail about gradient descent, the role of cost functions specifically as a barometer within Machine Learning, types of gradient descents, learning rates, etc.

What is Gradient Descent or Steepest Descent?

Gradient descent was initially discovered by **"Augustin-Louis Cauchy"** in mid of 18th century. ***Gradient Descent is defined as one of the most commonly used iterative optimization algorithms of machine learning to train the machine learning and deep learning models. It helps in finding the local minimum of a function.***

The best way to define the local minimum or local maximum of a function using gradient descent is as follows:

- If we move towards a negative gradient or away from the gradient of the function at the current point, it will give the **local minimum** of that function.
- Whenever we move towards a positive gradient or towards the gradient of the function at the current point, we will get the **local maximum** of that function.



This entire procedure is known as Gradient Ascent, which is also known as steepest descent. ***The main objective of using a gradient descent algorithm is to minimize the cost function using iteration.*** To achieve this goal, it performs two steps iteratively:

- Calculates the first-order derivative of the function to compute the gradient or slope of that function.

- Move away from the direction of the gradient, which means slope increased from the current point by alpha times, where Alpha is defined as Learning Rate. It is a tuning parameter in the optimization process which helps to decide the length of the steps.

What is Cost-function?

The cost function is defined as the measurement of difference or error between actual values and expected values at the current position and present in the form of a single real number. It helps to increase and improve machine learning efficiency by providing feedback to this model so that it can minimize error and find the local or global minimum. Further, it continuously iterates along the direction of the negative gradient until the cost function approaches zero. At this steepest descent point, the model will stop learning further. Although cost function and loss function are considered synonymous, also there is a minor difference between them. The slight difference between the loss function and the cost function is about the error within the training of machine learning models, as loss function refers to the error of one training example, while a cost function calculates the average error across an entire training set.

The cost function is calculated after making a hypothesis with initial parameters and modifying these parameters using gradient descent algorithms over known data to reduce the cost function.

Hypothesis:

Parameters:

Cost function:

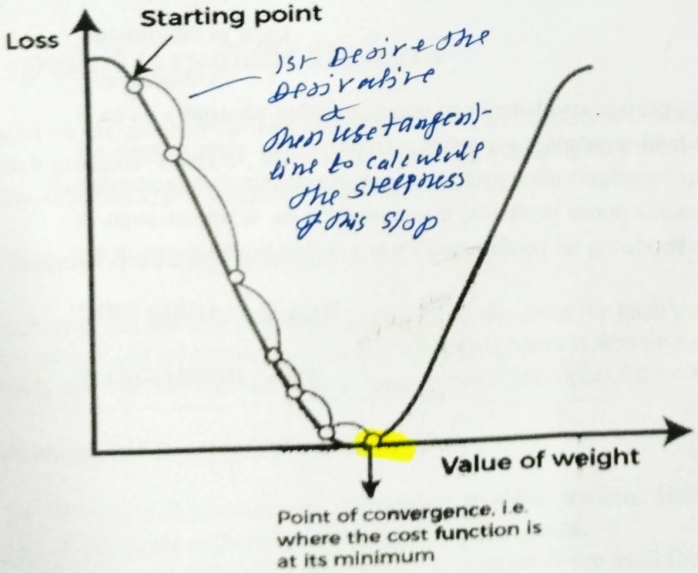
Goal:

How does Gradient Descent work?

Before starting the working principle of gradient descent, we should know some basic concepts to find out the slope of a line from linear regression. The equation for simple linear regression is given as:

1. Y=mX+c

Where 'm' represents the slope of the line, and 'c' represents the intercepts on the y-axis.



The starting point(shown in above fig.) is used to evaluate the performance as it is considered just as an arbitrary point. At this starting point, we will derive the first derivative or slope and then use a tangent line to calculate the steepness of this slope. Further, this slope will inform the updates to the parameters (weights and bias).

The slope becomes steeper at the starting point or arbitrary point, but whenever new parameters are generated, then steepness gradually reduces, and at the lowest point, it approaches the lowest point, which is called **a point of convergence.**

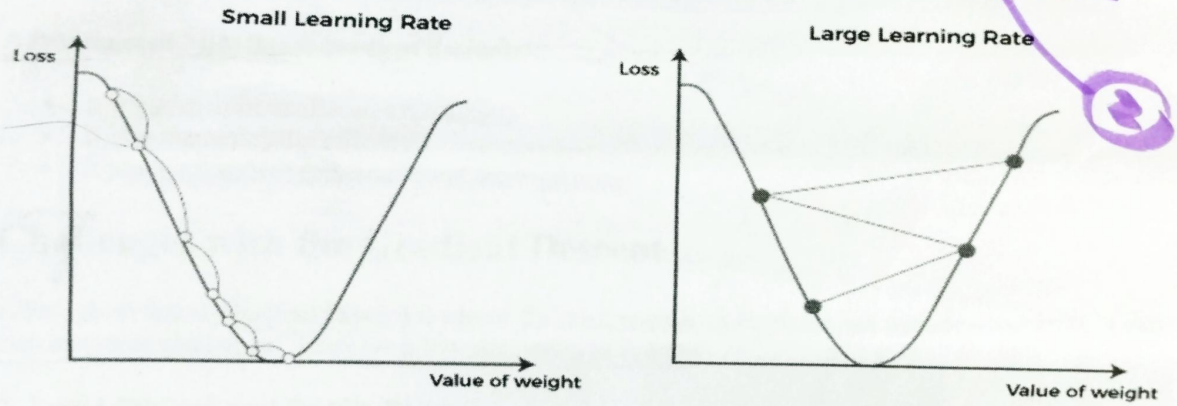
The main objective of gradient descent is to minimize the cost function or the error between expected and actual. To minimize the cost function, two data points are required:

- **Direction & Learning Rate**

These two factors are used to determine the partial derivative calculation of future iteration and allow it to the point of convergence or local minimum or global minimum. Let's discuss learning rate factors in brief;

Learning Rate:

It is defined as the step size taken to reach the minimum or lowest point. This is typically a small value that is evaluated and updated based on the behavior of the cost function. If the learning rate is high, it results in larger steps but also leads to risks of overshooting the minimum. At the same time, a low learning rate shows the small step sizes, which compromises overall efficiency but gives the advantage of more precision.



Types of Gradient Descent

Based on the error in various training models, the Gradient Descent learning algorithm can be divided into **Batch gradient descent, stochastic gradient descent, and mini-batch gradient descent.** Let's understand these different types of gradient descent:

1. Batch Gradient Descent:

Batch gradient descent (BGD) is used to find the error for each point in the training set and update the model after evaluating all training examples. This procedure is known as the training epoch. In simple words, it is a greedy approach where we have to sum over all examples for each update.

Advantages of Batch gradient descent:

- It produces less noise in comparison to other gradient descent.
- It produces stable gradient descent convergence.
- It is Computationally efficient as all resources are used for all training samples.

2. Stochastic gradient descent

Stochastic gradient descent (SGD) is a type of gradient descent that runs one training example per iteration. Or in other words, it processes a training epoch for each example within a dataset and updates each training example's parameters one at a time. As it requires only one training example at a time, hence it is easier to store in allocated memory. However, it shows some computational efficiency losses in comparison to batch gradient systems as it shows frequent updates that require more detail and speed. Further, due to frequent updates, it is also treated as a noisy gradient. However, sometimes it can be helpful in finding the global minimum and also escaping the local minimum.

Advantages of Stochastic gradient descent:

In Stochastic gradient descent (SGD), learning happens on every example, and it consists of a few advantages over other gradient descent.

- It is easier to allocate in desired memory.
- It is relatively fast to compute than batch gradient descent.
- It is more efficient for large datasets.

3. MiniBatch Gradient Descent:

Mini Batch gradient descent is the combination of both batch gradient descent and stochastic gradient descent. It divides the training datasets into small batch sizes then performs the updates on those batches separately. Splitting training datasets into smaller batches make a balance to maintain the computational efficiency of batch gradient descent and speed of stochastic gradient descent. Hence, we can achieve a special type of gradient descent with higher computational efficiency and less noisy gradient descent.

Advantages of Mini Batch gradient descent:

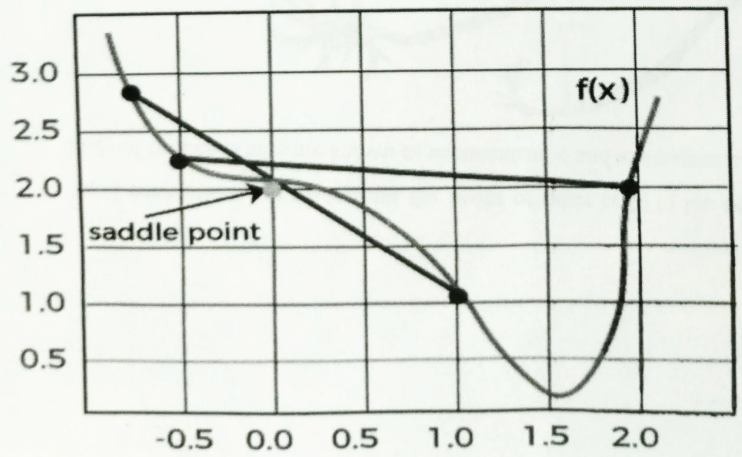
- It is easier to fit in allocated memory.
- It is computationally efficient.
- It produces stable gradient descent convergence.

Challenges with the Gradient Descent

Although we know Gradient Descent is one of the most popular methods for optimization problems, it still also has some challenges. There are a few challenges as follows:

1. Local Minima and Saddle Point:

For convex problems, gradient descent can find the global minimum easily, while for non-convex problems, it is sometimes difficult to find the global minimum, where the machine learning models achieve the best results.



Whenever the slope of the cost function is at zero or just close to zero, this model stops learning further. Apart from the global minimum, there occur some scenarios that can show this slop, which is saddle point and local minimum. Local minima generate the shape similar to the global minimum, where the slope of the cost function increases on both sides of the current points.

In contrast, with saddle points, the negative gradient only occurs on one side of the point, which reaches a local maximum on one side and a local minimum on the other side. The name of a saddle point is taken by that of a horse's saddle.

The name of local minima is because the value of the loss function is minimum at that point in a local region. In contrast, the name of the global minima is given so because the value of the loss function is minimum there, globally across the entire domain the loss function.

2. Vanishing and Exploding Gradient

In a deep neural network, if the model is trained with gradient descent and backpropagation, there can occur two more issues other than local minima and saddle point.

Vanishing Gradients:

Vanishing Gradient occurs when the gradient is smaller than expected. During backpropagation, this gradient becomes smaller that causing the decrease in the learning rate of earlier layers than the later layer of the network. Once this happens, the weight parameters update until they become insignificant.

Exploding Gradient:

Exploding gradient is just opposite to the vanishing gradient as it occurs when the Gradient is too large and creates a stable model. Further, in this scenario, model weight increases, and they will be represented as NaN. This problem can be solved using the dimensionality reduction technique, which helps to minimize complexity within the model.

In the realm of deep learning, the optimization process plays a crucial role in training neural networks. Gradient descent, a fundamental optimization algorithm, can sometimes encounter two common issues: vanishing gradients and exploding gradients. In this article, we will delve into these challenges, providing insights into what they are, why they occur, and how to mitigate them. We will build and train a model, and learn how to face vanishing and exploding problems.

What is Vanishing Gradient?

The vanishing gradient problem is a challenge that emerges during backpropagation when the derivatives or slopes of the activation functions become progressively smaller as we move backward through the layers of a neural network. This phenomenon is particularly prominent in deep networks with many layers, hindering the effective training of the model. The weight updates becomes extremely tiny, or even exponentially small, it can significantly prolong the training time, and in the worst-case scenario, it can halt the training process altogether.

Why the Problem Occurs?

During backpropagation, the gradients propagate back through the layers of the network, they decrease significantly. This means that as they leave the output layer and return to the input layer, the gradients become progressively smaller. As a result, the weights associated with the initial levels, which accommodate these small gradients, are updated little or not at each iteration of the optimization process.

The vanishing gradient problem is particularly associated with the sigmoid and hyperbolic tangent (tanh) activation functions because their derivatives fall within the range of 0 to 0.25 and 0 to 1, respectively. Consequently, extreme weights becomes very small, causing the updated weights to closely resemble the original ones. This persistence of small updates contributes to the vanishing gradient issue.

The sigmoid and tanh functions limit the input values to the ranges [0,1] and [-1,1], so that they saturate at 0 or 1 for sigmoid and -1 or 1 for Tanh. The derivatives at points becomes zero as they are moving. In these regions, especially when inputs are very small or large, the gradients are very close to zero. While this may not be a major concern in shallow networks with a few layers, it is a more pronounced issue in deep networks. When the inputs fall in saturated regions, the gradients approach zero, resulting in little update to the weights of the previous layer. In simple networks this does not pose much of a problem, but as more layers are added, these small gradients, which multiply between layers, decay significantly and consequently the first layer tears very slowly , and hinders overall model performance and can lead to convergence failure.

How can we identify?

Identifying the vanishing gradient problem typically involves monitoring the training dynamics of a deep neural network.

- One key indicator is observing model weights **converging to 0** or stagnation in the improvement of the model's performance metrics over training epochs.
- During training, if the **loss function fails to decrease** significantly, or if there is erratic behavior in the learning curves, it suggests that the gradients may be vanishing.
- Additionally, examining the gradients themselves during backpropagation can provide insights. **Visualization techniques**, such as gradient

histograms or norms, can aid in assessing the distribution of gradients throughout the network.

How can we solve the issue?

- **Batch Normalization** : Batch normalization normalizes the inputs of each layer, reducing internal covariate shift. This can help stabilize and accelerate the training process, allowing for more consistent gradient flow.
- **Activation function**: Activation function like **Rectified Linear Unit (ReLU)** can be used. With **ReLU,** the gradient is 0 for negative and zero input, and it is 1 for positive input, which helps alleviate the vanishing gradient issue. Therefore, ReLU operates by replacing poor enter values with 0, and 1 for fine enter values, it preserves the input unchanged.
- **Skip Connections and Residual Networks (ResNets)**: Skip connections, as seen in ResNets, allow the gradient to bypass certain layers during backpropagation. This facilitates the flow of information through the network, preventing gradients from vanishing.
- **Long Short-Term Memory Networks (LSTMs) and Gated Recurrent Units (GRUs)**: In the context of recurrent neural networks (RNNs), architectures like LSTMs and GRUs are designed to address the vanishing gradient problem in sequences by incorporating gating mechanisms .
- **Gradient Clipping**: Gradient clipping involves imposing a threshold on the gradients during backpropagation. Limit the magnitude of gradients during backpropagation, this can prevent them from becoming too small or exploding, which can also hinder learning.

What is Exploding Gradient?

The exploding gradient problem is a challenge encountered during training deep neural networks. It occurs when the gradients of the network's loss function with respect to the weights (parameters) become excessively large.

Why Exploding Gradient Occurs?

The issue of exploding gradients arises when, during backpropagation, the derivatives or slopes of the neural network's layers grow progressively larger as we move backward. This is essentially the opposite of the vanishing gradient problem.

The root cause of this problem lies in the weights of the network, rather than the choice of activation function. High weight values lead to correspondingly high derivatives, causing significant deviations in new weight values from the previous ones. As a result, the gradient fails to converge and can lead to the network oscillating around local minima, making it challenging to reach the global minimum point.

In summary, exploding gradients occur when weight values lead to excessively large derivatives, making convergence difficult and potentially preventing the neural network from effectively learning and optimizing its parameters.

As we discussed earlier, the update for the weights during backpropagation in a neural network is given by:

Where, is the learning rate.

The exploding gradient problem occurs when the gradients become very large during backpropagation. This is often the result of gradients greater than 1, leading to a rapid increase in values as you propagate them backward through the layers.

Mathematically, the update rule becomes problematic when causing the weights to increase exponentially during training.

How can we identify the problem?

Identifying the presence of exploding gradients in deep neural network requires careful observation and analysis during training. Here are some key indicators:

- The loss function exhibits erratic behavior, oscillating wildly instead of steadily decreasing suggesting that the network weights are being updated excessively by large gradients, preventing smooth convergence.
- The training process encounters "NaN" (Not a Number) values in the loss function or other intermediate calculations..
- If network weights, during training exhibit significant and rapid increases in their values, it suggests the presence of exploding gradients.
- Tools like TensorBoard can be used to visualize the gradients flowing through the network.

How can we solve the issue?

- **Gradient Clipping**: It sets a maximum threshold for the magnitude of gradients during backpropagation. Any gradient exceeding the threshold is clipped to the threshold value, preventing it from growing unbounded.
- **Batch Normalization:** This technique normalizes the activations within each mini-batch, effectively scaling the gradients and reducing their variance. This helps prevent both vanishing and exploding gradients, improving stability and efficiency.

Solution for Exploding Gradient Problem

Below methods can be used to modify the model:

1. Weight Initialization: The weight initialization is changed to 'glorot_uniform,' which is a commonly used initialization for neural networks.
2. Gradient Clipping: The clipnorm parameter in the Adam optimizer is set to 1.0, which performs gradient clipping. This helps prevent exploding gradients.
3. Kernel Constraint: The max_norm constraint is applied to the kernel weights of each layer with a maximum norm of 2.0. This further helps in preventing exploding gradients.

Empirical Risk Minimization (ERM)

In words...

The Empirical Risk Minimization (ERM) principle is a learning paradigm which consists in selecting the model with minimal average error over the training set. This so-called training error can be seen as an estimate of the risk (due to the law of large numbers), hence the alternative name of empirical risk.

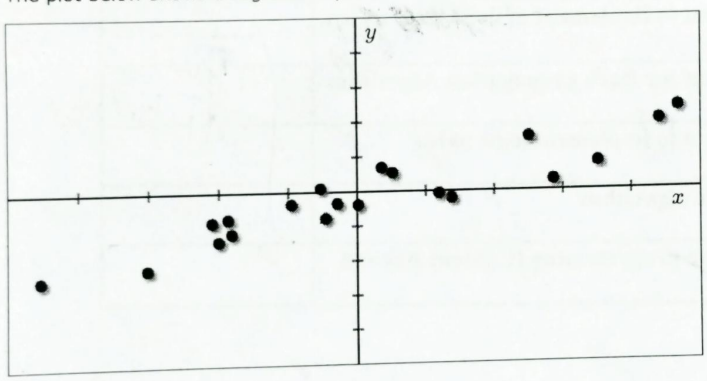
By minimizing the empirical risk, we hope to obtain a model with a low value of the risk. The larger the training set size is, the closer to the true risk the empirical risk is.

If we were to apply the ERM principle without more care, we would end up learning by heart, which we know is bad. This issue is more generally related to the overfitting phenomenon, which can be avoided by restricting the space of possible models when searching for the one with minimal error. The most severe and yet common restriction is encountered in the contexts of linear classification or linear regression. Another approach consists in controlling the complexity of the model by regularization.

In pictures...

The ERM is a nice idea, if used with care

The plot below shows a regression problem with a training set of 15 points.



Next

In maths...

The ERM principle is an inference principle which consists in finding the model $\hat{f}$ by minimizing the empirical risk:

$$\hat{f} = \arg\min_{f:\mathcal{X}\to\mathcal{Y}} R_{emp}(h)$$

where the **empirical risk** is an estimate of the risk computed as the average of the loss function over the **training sample** $D = \{(X_i, Y_i)\}_{i=1}^N$:

$$R_{emp}(f) = \frac{1}{N}\sum_{i=1}^{N} \ell(f(X_i), Y_i)$$

with the loss function ℓ.

Given a **training set**, $\{(x_i, y_i)\}_{i=1}^N$, assumed to be a realization of the training sample, the ERM principle becomes a practical method that learns the model by solving

$$\hat{f} = \arg\min_{f:\mathcal{X}\to\mathcal{Y}} \frac{1}{N}\sum_{i=1}^{N} \ell(f(x_i), y_i)$$

and thus minimizing the training error defined as the avg loss over the training set.

Still under construction

$$f(x) = \begin{cases} y_i & \text{if } x = x_i \text{ for some } i \text{ in the training set} \\ \text{any value } \in \mathcal{Y}, & \text{otherwise,} \end{cases}$$

which all correspond to learning by heart, an extreme form of overfitting.

In order to avoid such issues, we should constrain f to some suitable model space $\mathcal{F}$, i.e., replacing the minimization over the set of functions $f : \mathcal{X} \to \mathcal{Y}$ by a minimization over $f \in \mathcal{F}$.

Notes

The training procedure suggested by the ERM amounts to an optimization problem, as is often the case with learning algorithms.

Written by F. Lauer, 2014.

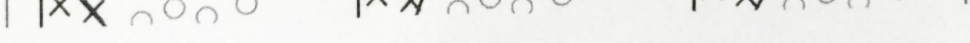
Regularization in Machine Learning

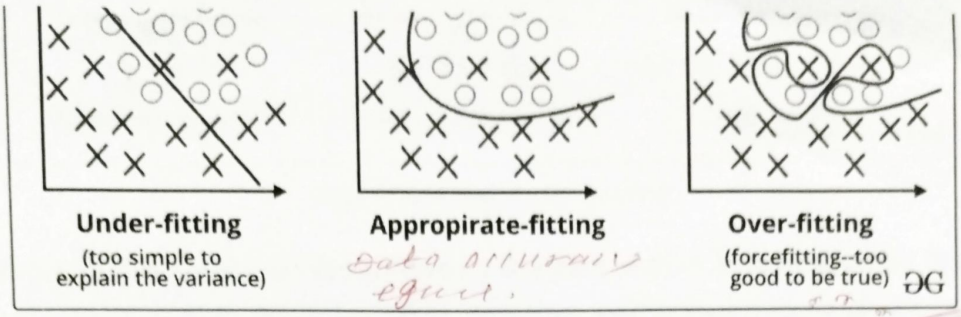
While developing machine learning models you must have encountered a situation in which the training accuracy of the model is low but the validation accuracy or the testing accuracy is too low. This is the case which is popularly known as overfitting in the domain of machine learning also this is the last thing a machine learning practitioner would like to have in his model. In this article, we will learn about a method known as regularization which helps us to solve the problem of overfitting. But before that let's understand what is underfitting and overfitting.

What are Overfitting and Underfitting?

Overfitting is a phenomenon that occurs when a Machine Learning model is constrained to the training set and not able to perform well on unseen data. That is when our model learns the noise in the training data as well. This is the case when our model memorizes the training data instead of learning the patterns in it.

Underfitting on the other hand is the case when our model is not able to learn even the basic patterns available in the dataset. In the case of the underfitting model is unable to perform well even on the training data hence we cannot expect it to perform well on the validation data. This is the case when we are supposed to increase the complexity of the model or add more features to the feature set.



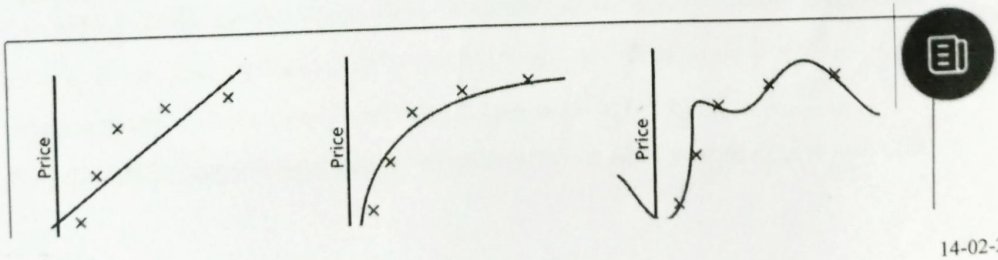


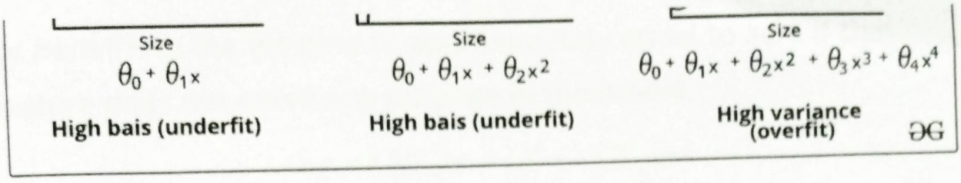
Overfitting and Underfitting in Machine Learning

What are Bias and Variance?

Bias refers to the errors which occur when we try to fit a statistical model on real-world data which does not fit perfectly well on some mathematical model. If we use a way too simplistic a model to fit the data then we are more probably face the situation of **High Bias** which refers to the case when the model is unable to learn the patterns in the data at hand and hence performs poorly.

Variance implies the error value that occurs when we try to make predictions by using data that is not previously seen by the model. There is a situation known as **high variance** that occurs when the model learns noise that is present in the data.





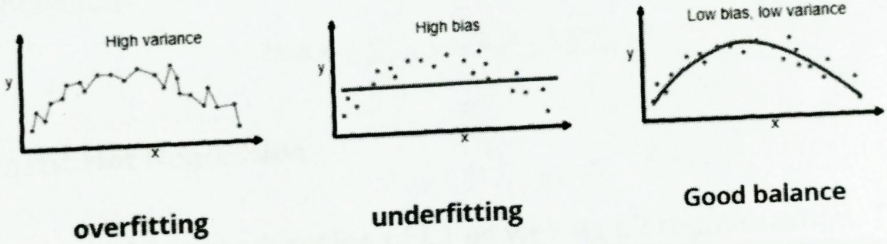
Bias and Variance in Machine Learning

Finding a proper balance between the two that is also known as the Bias-Variance Tradeoff can help us prune the model from getting overfitted to the training data.

Regularization in Machine Learning

Regularization is a technique used to reduce errors by fitting the function appropriately on the given training set and avoiding overfitting. The commonly used regularization techniques are :

1. Lasso Regularization – L1 Regularization
2. Ridge Regularization – L2 Regularization
3. Elastic Net Regularization – L1 and L2 Regularization



Regularized model to avoid underfitting as well as overfitting

Lasso Regression

A regression model which uses the **L1 Regularization** technique is called **LASSO(Least Absolute Shrinkage and Selection Operator)** regression. **Lasso Regression** adds the *"absolute value of magnitude"* of the coefficient as a penalty term to the loss function(L). Lasso regression also helps us achieve feature selection

by penalizing the weights to approximately equal to zero if that feature does not serve any purpose in the model.

$$\text{Cost} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\sum_{i=1}^{m}|w_i|$$

where,

- ***m*** *– Number of Features*
- ***n*** *– Number of Examples*
- ***y_i*** *– Actual Target Value*
- ***y_i(hat)*** *– Predicted Target Value*

Ridge Regression

A regression model that uses the **L2 regularization** technique is called **Ridge regression**. **Ridge regression** adds the "*squared magnitude*" of the coefficient as a penalty term to the loss function(L).

$$\text{Cost} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\sum_{i=1}^{m}w_i^2$$

Elastic Net Regression

This model is a combination of L1 as well as L2 regularization. That implies that we add the absolute norm of the weights as well as the squared measure of the weights. With the help of an extra hyperparameter that controls the ratio of the L1 and L2 regularization.

$$\text{Cost} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \lambda\left((1-\alpha)\sum_{i=1}^{m}|w_i| + \alpha\sum_{i=1}^{m}w_i^2\right)$$

Regularization

The term 'regularization' refers to a set of techniques that regularizes learning from particular features for traditional algorithms or neurons in the case of neural network algorithms.

It normalizes and moderates weights attached to a feature or a neuron so that algorithms do not rely on just a few features or neurons to predict the result. This technique helps to avoid the problem of overfitting.

To understand regularization, let's consider a simple case of linear regression. Mathematically, linear regression is stated as below:

$$y = w_0 + w_1x_1 + w_2x_2 + \ldots\ldots + w_nx_n$$

where y is the value to be predicted;

$x_1, x_2, \ldots, x_n$ are features that decides the value of y;

w_0 is the bias;

$w_1, w_2, \ldots\ldots, w_n$ are the weights attached to $x_1, x_2, \ldots, x_n$ relatively.

Now to build a model that accurately predicts the *y* value, we need to optimize above mentioned bias and weights.

To do so, we need to use a loss function and find optimized parameters using gradient descent algorithms and its variants.

To know more about building a machine learning application and the process, check out below blog:

How to Develop Machine Learning Applications for Business

The loss function called 'the residual sum of square' is mostly used for linear regression. Here's what it looks like :

$$RSS = \sum_{j=1}^{m} \left(Yi - Wo - \sum_{i=1}^{n} Wi\, Xji \right)^2$$

Next, we will learn bias (or intercept) and weights (also identified as parameters and coefficients) using the optimization algorithm (gradient descent) and data. If your dataset does have noise in it, it will face overfitting problem and learned parameters will not generalize well on unseen data.

To avoid this, you will need to regularize or normalize your weights for better learning.

There are three main regularization techniques, namely:

1. Ridge Regression (L2 Norm)
2. Lasso (L1 Norm)
3. Dropout

Ridge and Lasso can be used for any algorithms involving weight parameters, including neural nets. Dropout is primarily used in any kind of neural networks e.g. ANN, DNN, CNN or RNN to moderate the learning. Let's take a closer look at each of the techniques.

Ridge Regression (L2 Regularization)

Ridge regression is also called L2 norm or regularization.

When using this technique, we add the sum of weight's square to a loss function and thus create a new loss function which is denoted thus:

$$\text{Loss} = \sum_{j=1}^{m}\left(Yi - Wo - \sum_{i=1}^{n} Wi\,Xji\right)^2 + \lambda \sum_{i=1}^{n} Wi^2$$

As seen above, the original loss function is modified by adding normalized weights. Here normalized weights are in the form of squares.

You may have noticed parameters λ along with normalized weights. λ is the parameter that needs to be tuned using a cross-validation dataset. When you use λ=0, it returns the residual sum of square as loss function which you chose initially. For a very high value of λ, loss will ignore core loss function and minimize weight's square and will end up taking the parameters' value as zero.

Now the parameters are learned using a modified loss function. To minimize the above function, parameters need to be as small as possible. Thus, L2 norm prevents weights from rising too high.

Lasso Regression (L1 Regularization)

Also called lasso regression and denoted as below:

$$\text{Loss} = \sum_{j=1}^{m}\left(Yi - Wo - \sum_{i=1}^{n} Wi\,Xji\right)^2 + \lambda \sum_{i=1}^{n} |Wi|$$

This technique is different from ridge regression as it uses absolute weight values for normalization. λ is again a tuning parameter and behaves in the same as it does when using ridge regression.

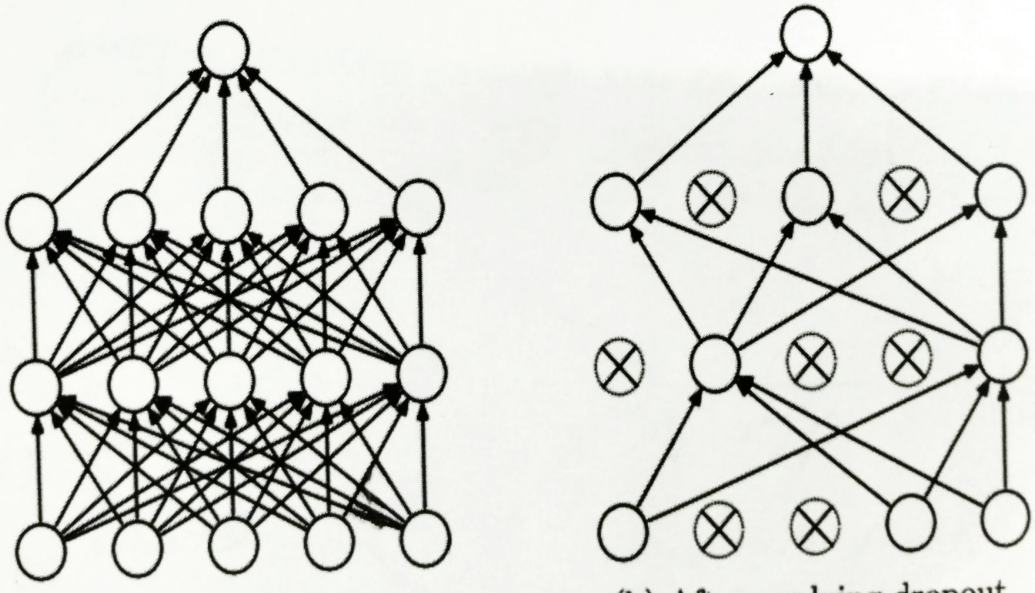
As loss function only considers absolute weights, optimization algorithms penalize higher weight values.

In ridge regression, loss function along with the optimization algorithm brings parameters near to zero but not actually zero, while lasso eliminates less important features and sets respective weight values to zero. Thus, lasso also performs feature selection along with regularization.

Dropout

Dropout is a regularization technique used in neural networks. It prevents complex co-adaptations from other neurons.

In neural nets, fully connected layers are more prone to overfit on training data. Using dropout, you can drop connections with *1-p* probability for each of the specified layers. Where *p* is called **keep probability parameter** and which needs to be tuned.



(a) Standard Neural Net

(b) After applying dropout.

With dropout, you are left with a reduced network as dropped out neurons are left out during that training iteration.

Dropout decreases overfitting by avoiding training all the neurons on the complete training data in one go. It also improves training speed and learns more robust internal functions that generalize better on unseen data. However, it is important to note that Dropout takes more epochs to train compared to training without Dropout (If you have 10000 observations in your training data, then using 10000 examples for training is considered as 1 epoch).

Along with Dropout, neural networks can be regularized also using L1 and L2 norms. Apart from that, if you are working on an image dataset, image augmentation can also be used as a regularization method.

For real-world applications, it is a must that a model performs well on unseen data. The techniques we discussed can help you make your model learn rather than just memorize.

overfit

→ Regularization

→ Dropout

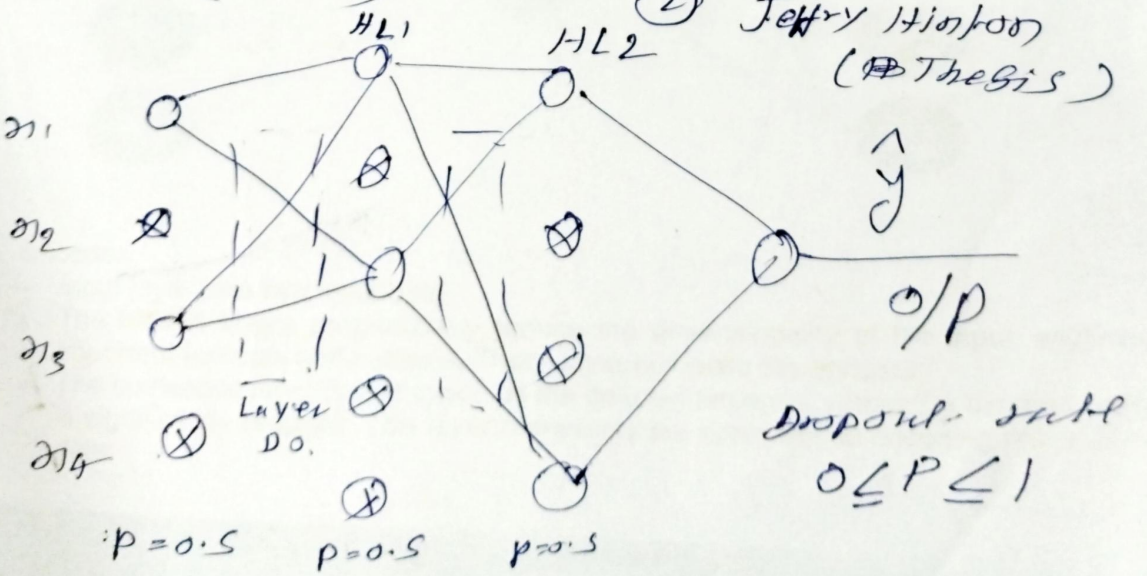
In Deep N.N. (many $\underline{w}$ & $\underline{b}$ ↓ overfit prob.) High variance prob.

In multiple layer Never Underfit prob.

2014 ~~design~~ explain (written)

① Nitesh Shirastar

② Jeffry Hinton (~~PhD~~ Thesis)



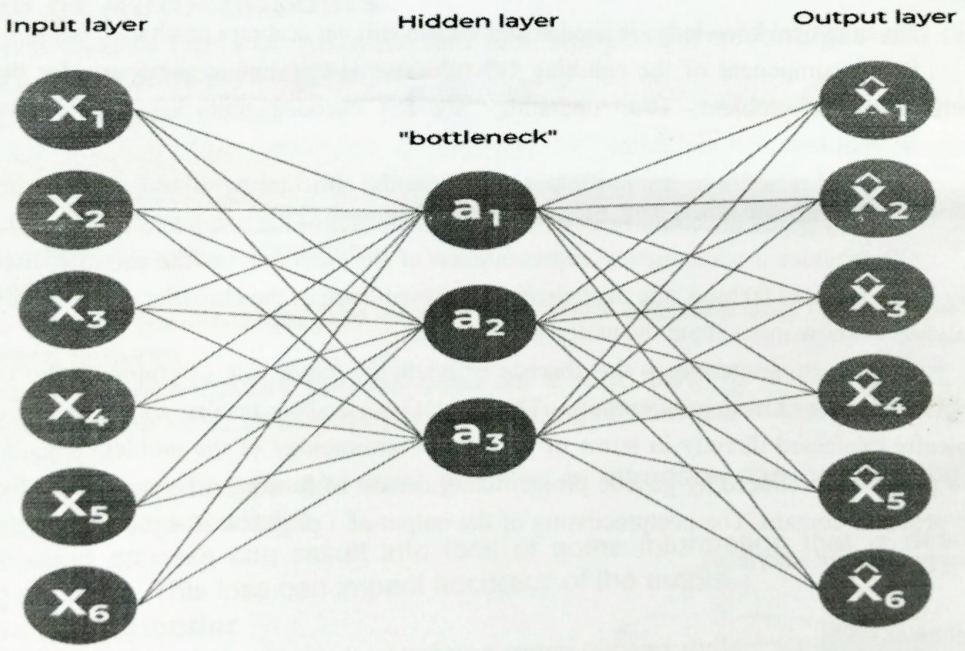
At the heart of **deep learning** lies the neural network, an intricate interconnected system of nodes that mimics the human brain's neural architecture. Neural networks excel at discerning intricate patterns and representations within vast datasets, allowing them to make predictions, classify information, and generate novel insights. **Autoencoders** emerge as a fascinating subset of neural networks, offering a unique approach to unsupervised learning. Autoencoders are an adaptable and strong class of architectures for the dynamic field of deep learning, where neural networks develop constantly to identify complicated patterns and representations. With their ability to learn effective representations of data, these unsupervised learning models have received considerable attention and are useful in a wide variety of areas, from image processing to anomaly detection.

What are Autoencoders?

Autoencoders are a specialized class of algorithms that can learn efficient representations of input data with no need for labels. It is a class of artificial neural networks designed for unsupervised learning. Learning to compress and effectively represent input data without specific labels is the essential principle of an automatic decoder. This is accomplished using a two-fold structure that consists of an encoder and a decoder. The encoder transforms the input data into a reduced-dimensional representation, which is often referred to as "latent space" or "encoding". From that representation, a decoder rebuilds the initial input. For the network to gain meaningful patterns in data, a process of encoding and decoding facilitates the definition of essential features.

Architecture of Autoencoder in Deep Learning

The general architecture of an autoencoder includes an encoder, decoder, and bottleneck layer.



1. Encoder
   - Input layer take raw input data
   - The hidden layers progressively reduce the dimensionality of the input, capturing important features and patterns. These layer compose the encoder.
   - The bottleneck layer (latent space) is the final hidden layer, where the dimensionality is significantly reduced. This layer represents the compressed encoding of the input data.

oder

- The bottleneck layer takes the encoded representation and expands it back to the dimensionality of the original input.
- The hidden layers progressively increase the dimensionality and aim to reconstruct the original input.
- The output layer produces the reconstructed output, which ideally should be as close as possible to the input data.

3. The loss function used during training is typically a reconstruction loss, measuring the difference between the input and the reconstructed output. Common choices include mean squared error (MSE) for continuous data or binary cross-entropy for binary data.
4. During training, the autoencoder learns to minimize the reconstruction loss, forcing the network to capture the most important features of the input data in the bottleneck layer.

After the training process, only the encoder part of the autoencoder is retained to encode a similar type of data used in the training process. The different ways to constrain the network are: –

- **Keep small Hidden Layers:** If the size of each hidden layer is kept as small as possible, then the network will be forced to pick up only the representative features of the data thus encoding the data.
- **Regularization:** In this method, a loss term is added to the cost function which encourages the network to train in ways other than copying the input.
- **Denoising:** Another way of constraining the network is to add noise to the input and teach the network how to remove the noise from the data.
- **Tuning the Activation Functions:** This method involves changing the activation functions of various nodes so that a majority of the nodes are dormant thus, effectively reducing the size of the hidden layers.

Types of Autoencoders

There are diverse types of autoencoders and analyze the advantages and disadvantages associated with different variation:

Denoising Autoencoder

Denoising autoencoder works on a partially corrupted input and trains to recover the original undistorted image. As mentioned above, this method is an effective way to constrain the network from simply copying the input and thus learn the underlying structure and important features of the data.

Advantages

1. This type of autoencoder can extract important features and reduce the noise or the useless features.
2. Denoising autoencoders can be used as a form of data augmentation, the restored images can be used as augmented data thus generating additional training samples.

Disadvantages

1. Selecting the right type and level of noise to introduce can be challenging and may require domain knowledge.
2. Denoising process can result into loss of some information that is needed from the original input. This loss can impact accuracy of the output.

Sparse Autoencoder

This type of autoencoder typically contains more hidden units than the input but only a few are allowed to be active at once. This property is called the sparsity of the network. The sparsity of the network can be controlled by either manually zeroing the required hidden units, tuning the activation functions or by adding a loss term to the cost function.

Advantages

1. The sparsity constraint in sparse autoencoders helps in filtering out noise and irrelevant features during the encoding process.
2. These autoencoders often learn important and meaningful features due to their emphasis on sparse activations.

Disadvantages

choice of hyperparameters play a significant role in the performance of this autoencoder. Different inputs should result in the activation of different nodes of the network.

2. The application of sparsity constraint increases computational complexity.

Variational Autoencoder

Variational autoencoder makes strong assumptions about the distribution of latent variables and uses the **Stochastic Gradient Variational Bayes** estimator in the training process. It assumes that the data is generated by a **Directed Graphical Model** and tries to learn an approximation to to the conditional property where and are the parameters of the encoder and the decoder respectively.

Advantages

1. Variational Autoencoders are used to generate new data points that resemble the original training data. These samples are learned from the latent space.
2. Variational Autoencoder is probabilistic framework that is used to learn a compressed representation of the data that captures its underlying structure and variations, so it is useful in detecting anomalies and data exploration.

Disadvantages

1. Variational Autoencoder use approximations to estimate the true distribution of the latent variables. This approximation introduces some level of error, which can affect the quality of generated samples.
2. The generated samples may only cover a limited subset of the true data distribution. This can result in a lack of diversity in generated samples.

Convolutional Autoencoder

Convolutional autoencoders are a type of autoencoder that use convolutional neural networks (CNNs) as their building blocks. The encoder consists of multiple layers that take a image or a grid as input and pass it through different convolution layers thus forming a compressed representation of the input. The decoder is the mirror image of the encoder it deconvolves the compressed representation and tries to reconstruct the original image.

Advantages

1. Convolutional autoencoder can compress high-dimensional image data into a lower-dimensional data. This improves storage efficiency and transmission of image data.
2. Convolutional autoencoder can reconstruct missing parts of an image. It can also handle images with slight variations in object position or orientation.

Disadvantages

1. These autoencoder are prone to overfitting. Proper regularization techniques should be used to tackle this issue.
2. Compression of data can cause data loss which can result in reconstruction of a lower quality image.

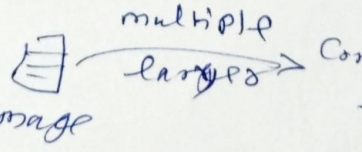
Implementation of Autoencoders

We've created an autoencoder comprising two Dense layers: an encoder responsible for condensing the images into a 64-dimensional latent vector, and a decoder tasked with reconstructing the initial image based on this latent space.

Import necessary libraries

For the implementation, we are going to import matplotlib, numpy, pandas, sklearn and keras.

- Python3




```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf
```

```
klearn.metrics import accuracy_score, precision_score, recall_score
sklearn.model_selection import train_test_split
om keras import layers, losses
rom keras.datasets import mnist
from keras.models import Model
```

Load the MNIST dataset

- Python3

```
# Loading the MNIST dataset and extracting training and testing data
(x_train, _), (x_test, _) = mnist.load_data()

# Normalizing pixel values to the range [0, 1]
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

# Displaying the shapes of the training and testing datasets
print("Shape of the training data:", x_train.shape)
print("Shape of the testing data:", x_test.shape)
```

Output:

```
Shape of the training data: (60000, 28, 28)
Shape of the testing data: (10000, 28, 28)
```

Define a basic Autoencoder

In the following code snippet,

- SimpleAutoencoder class is defined.
- Constructor initializes the autoencoder with specified latent dimensions and data shape
- The encoder and decoder architectures is defined using Sequential model
- The call method defines the forward pass of the autoencoder, where input data is passed through the encoder to obtain encoded data and then through the decoder to obtain the decoded data.

- Python3

```
# Definition of the Autoencoder model as a subclass of the TensorFlow Model class


class SimpleAutoencoder(Model):

    def __init__(self,latent_dimensions , data_shape):

        super(SimpleAutoencoder, self).__init__()

        self.latent_dimensions = latent_dimensions

        self.data_shape = data_shape



        # Encoder architecture using a Sequential model
```

```
        self.encoder = tf.keras.Sequential([
            layers.Flatten(),
            layers.Dense(latent_dimensions, activation='relu'),
        ])

        # Decoder architecture using another Sequential model
        self.decoder = tf.keras.Sequential([
            layers.Dense(tf.math.reduce_prod(data_shape), activation='sigmoid'),
            layers.Reshape(data_shape)
        ])

    # Forward pass method defining the encoding and decoding steps
    def call(self, input_data):
        encoded_data = self.encoder(input_data)
        decoded_data = self.decoder(encoded_data)
        return decoded_data

# Extracting shape information from the testing dataset
input_data_shape = x_test.shape[1:]

# Specifying the dimensionality of the latent space
latent_dimensions = 64

# Creating an instance of the SimpleAutoencoder model
simple_autoencoder = SimpleAutoencoder(latent_dimensions, input_data_shape)
```

Compile and Fit Autoencoder

```
simple_autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())


simple_autoencoder.fit(x_train, x_train,
                epochs=1,
                shuffle=True,
                validation_data=(x_test, x_test))
```

Output:

```
Epoch 1/10
1875/1875 [==============================] - 12s 6ms/step - loss: 0.0243 - val_loss: 0.0091
Epoch 2/10
1875/1875 [==============================] - 16s 9ms/step - loss: 0.0069 - val_loss: 0.0054
Epoch 3/10
1875/1875 [==============================] - 15s 8ms/step - loss: 0.0051 - val_loss: 0.0046
Epoch 4/10
1875/1875 [==============================] - 8s 5ms/step - loss: 0.0045 - val_loss: 0.0043
Epoch 5/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.0043 - val_loss: 0.0041
Epoch 6/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.0042 - val_loss: 0.0041
Epoch 7/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.0041 - val_loss: 0.0040
```

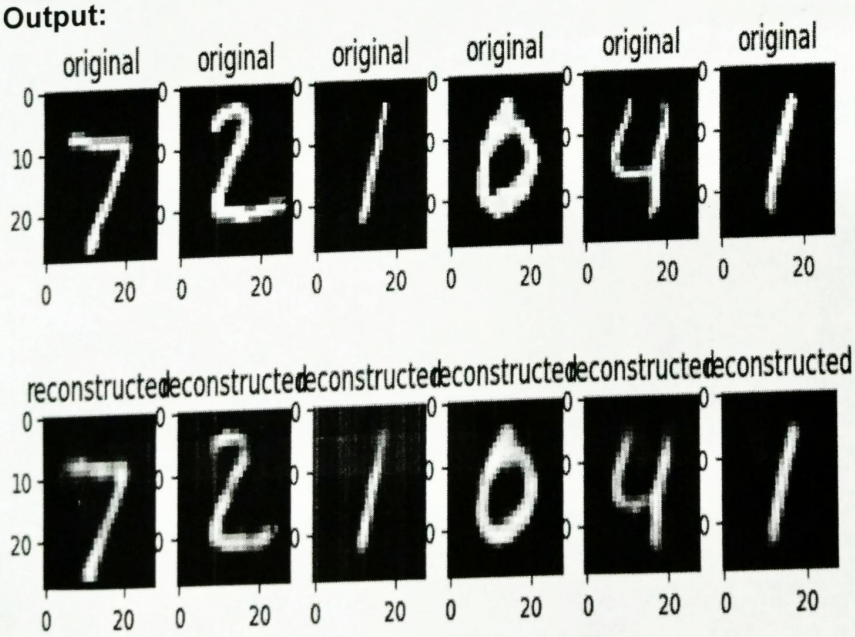
Visualize the original and reconstructed data

- Python3

```
encoded_imgs = simple_autoencoder.encoder(x_test).numpy()

decoded_imgs = simple_autoencoder.decoder(encoded_imgs).numpy()


n = 6
```

```
gure(figsize=(8, 4))
or i in range(n):
 # display original
 ax = plt.subplot(2, n, i + 1)
 plt.imshow(x_test[i])
 plt.title("original")
 plt.gray()

 # display reconstruction
 ax = plt.subplot(2, n, i + 1 + n)
 plt.imshow(decoded_imgs[i])
 plt.title("reconstructed")
 plt.gray()
plt.show()
```

Output:



A **Convolutional Neural Network (CNN)** is a type of Deep Learning neural network architecture commonly used in Computer Vision. Computer vision is a field of Artificial Intelligence that enables a computer to understand and interpret the image or visual data.

When it comes to Machine Learning, Artificial Neural Networks perform really well. Neural Networks are used in various datasets like images, audio, and text. Different types of Neural Networks are used for different purposes, for example for predicting the sequence of words we use **Recurrent Neural Networks** more precisely an LSTM, similarly for image classification we use Convolution Neural networks. In this blog, we are going to build a basic building block for CNN.

In a regular Neural Network there are three types of layers:

1. **Input Layers:** It's the layer in which we give input to our model. The number of neurons in this layer is equal to the total number of features in our data (number of pixels in the case of an image).
2. **Hidden Layer:** The input from the Input layer is then fed into the hidden layer. There can be many hidden layers depending on our model and data size. Each hidden layer can have different numbers of neurons which are generally greater than the number of features. The output from each layer is computed by matrix multiplication of the output of the previous layer with learnable weights of that layer and then by the addition of learnable biases followed by activation function which makes the network nonlinear.
3. **Output Layer:** The output from the hidden layer is then fed into a logistic function like sigmoid or softmax which converts the output of each class into the probability score of each class.

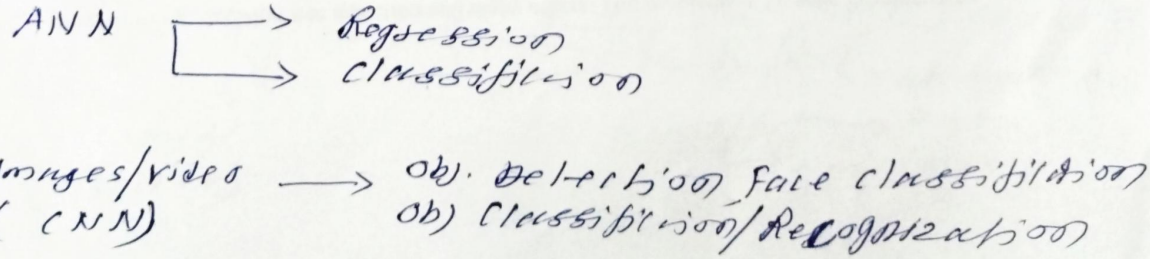
The data is fed into the model and output from each layer is obtained from the above step is called **feedforward**, we then calculate the error using an error function, some common error functions are cross-entropy, square loss error, etc. The error function measures how well the network is performing. After that, we backpropagate into the model by calculating the derivatives. This step is called **Backpropagation** which basically is used to minimize the loss.

Convolution Neural Network

Convolutional Neural Network (CNN) is the extended version of artificial neural networks (ANN) which is predominantly used to extract the feature from the grid-like matrix dataset. For example visual datasets like images or videos where data patterns play an extensive role.

CNN architecture

Convolutional Neural Network consists of multiple layers like the input layer, Convolutional layer, Pooling layer, and fully connected layers.

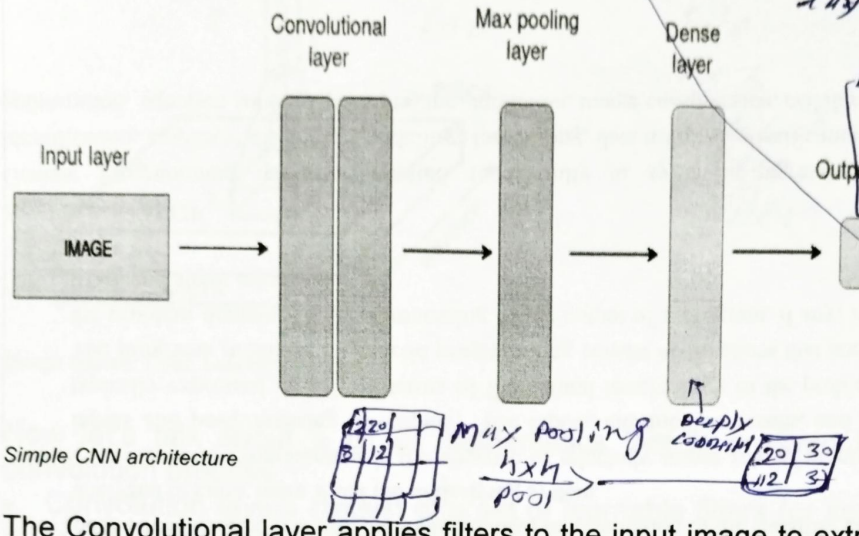


CNN:

cerebral cortex

↓

Visual cortex



Simple CNN architecture

Max Pooling

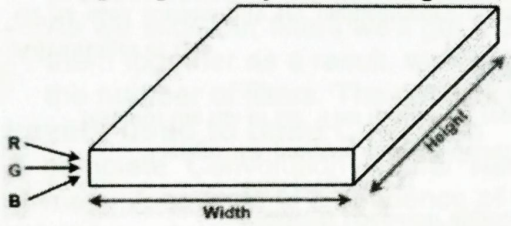
hxh pool

Deeply connected

The Convolutional layer applies filters to the input image to extract features, the Pooling layer downsamples the image to reduce computation, and the fully connected layer makes the final prediction. The network learns the optimal filters through backpropagation and gradient descent.

How Convolutional Layers works

Convolution Neural Networks or covnets are neural networks that share their parameters. Imagine you have an image. It can be represented as a cuboid having its length, width (dimension of the image), and height (i.e the channel as images generally have red, green, and blue channels).



Now imagine taking a small patch of this image and running a small neural network, called a filter or kernel on it, with say, K outputs and representing them vertically. Now slide that neural network across the whole image, as a result, we will get another image with different widths, heights, and depths. Instead of just R, G, and B channels now we have more channels but lesser width and height. This operation is called **Convolution**. If the patch size is the same as that of the image it will be a regular neural network. Because of this small patch, we have fewer weights.

Pixel range: 0 - 255

Gray scale Image (1 pixel)

Channel - 3

6 x 6 x 3

normalize: 0 - 1

min/max

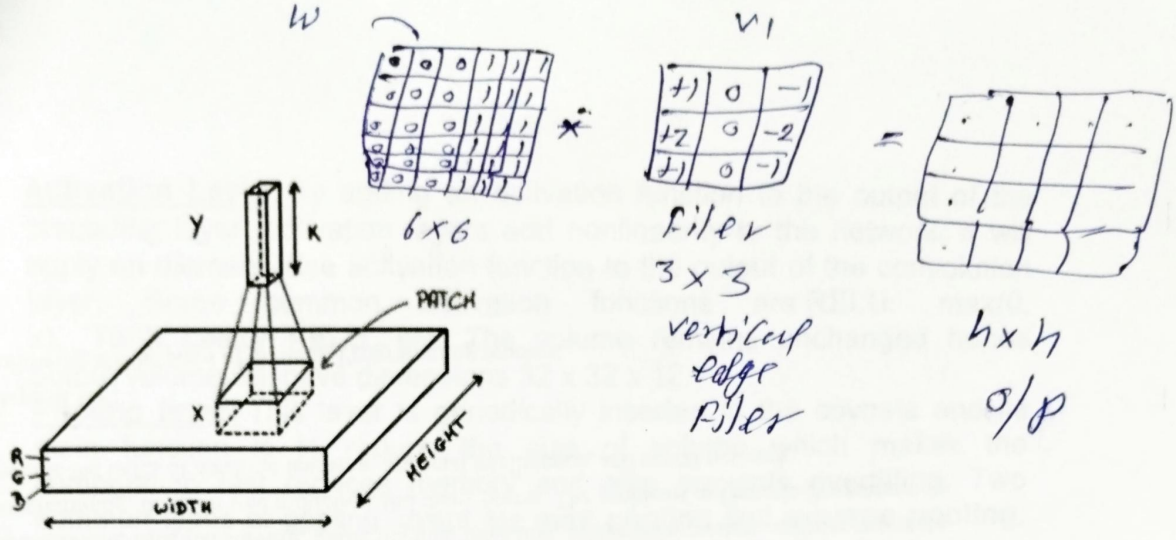


Image source: Deep Learning Udacity

Now let's talk about a bit of mathematics that is involved in the whole convolution process.

- Convolution layers consist of a set of learnable filters (or kernels) having small widths and heights and the same depth as that of input volume (3 if the input layer is image input).
- For example, if we have to run convolution on an image with dimensions 34x34x3. The possible size of filters can be axax3, where 'a' can be anything like 3, 5, or 7 but smaller as compared to the image dimension.
- During the forward pass, we slide each filter across the whole input volume step by step where each step is called **stride** (which can have a value of 2, 3, or even 4 for high-dimensional images) and compute the dot product between the kernel weights and patch from input volume.
- As we slide our filters we'll get a 2-D output for each filter and we'll stack them together as a result, we'll get output volume having a depth equal to the number of filters. The network will learn all the filters.

Layers used to build ConvNets

A complete Convolution Neural Networks architecture is also known as covnets. A covnets is a sequence of layers, and every layer transforms one volume to another through a differentiable function.

Types of layers: datasets

Let's take an example by running a covnets on of image of dimension 32 x 32 x 3.

- **Input Layers:** It's the layer in which we give input to our model. In CNN, Generally, the input will be an image or a sequence of images. This layer holds the raw input of the image with width 32, height 32, and depth 3.
- **Convolutional Layers:** This is the layer, which is used to extract the feature from the input dataset. It applies a set of learnable filters known as the kernels to the input images. The filters/kernels are smaller matrices usually 2×2, 3×3, or 5×5 shape. it slides over the input image data and computes the dot product between kernel weight and the corresponding input image patch. The output of this layer is referred ad feature maps. Suppose we use a total of 12 filters for this layer we'll get an output volume of dimension 32 x 32 x 12.

- **Activation Layer:** By adding an activation function to the output of the preceding layer, activation layers add nonlinearity to the network. it will apply an element-wise activation function to the output of the convolution layer. Some common activation functions are **RELU**: max(0, x), **Tanh**, **Leaky RELU**, etc. The volume remains unchanged hence output volume will have dimensions 32 x 32 x 12.
- **Pooling layer:** This layer is periodically inserted in the covnets and its main function is to reduce the size of volume which makes the computation fast reduces memory and also prevents overfitting. Two common types of pooling layers are **max pooling** and **average pooling**. If we use a max pool with 2 x 2 filters and stride 2, the resultant volume will be of dimension 16x16x12.

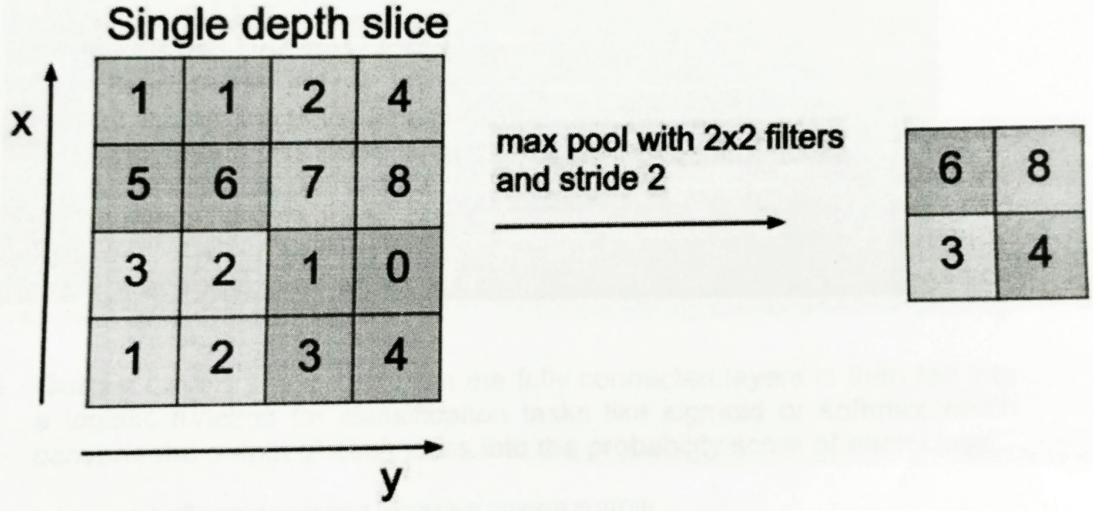


Image source: cs231n.stanford.edu

- **Flattening:** The resulting feature maps are flattened into a one-dimensional vector after the convolution and pooling layers so they can be passed into a completely linked layer for categorization or regression.
- **Fully Connected Layers:** It takes the input from the previous layer and computes the final classification or regression task.

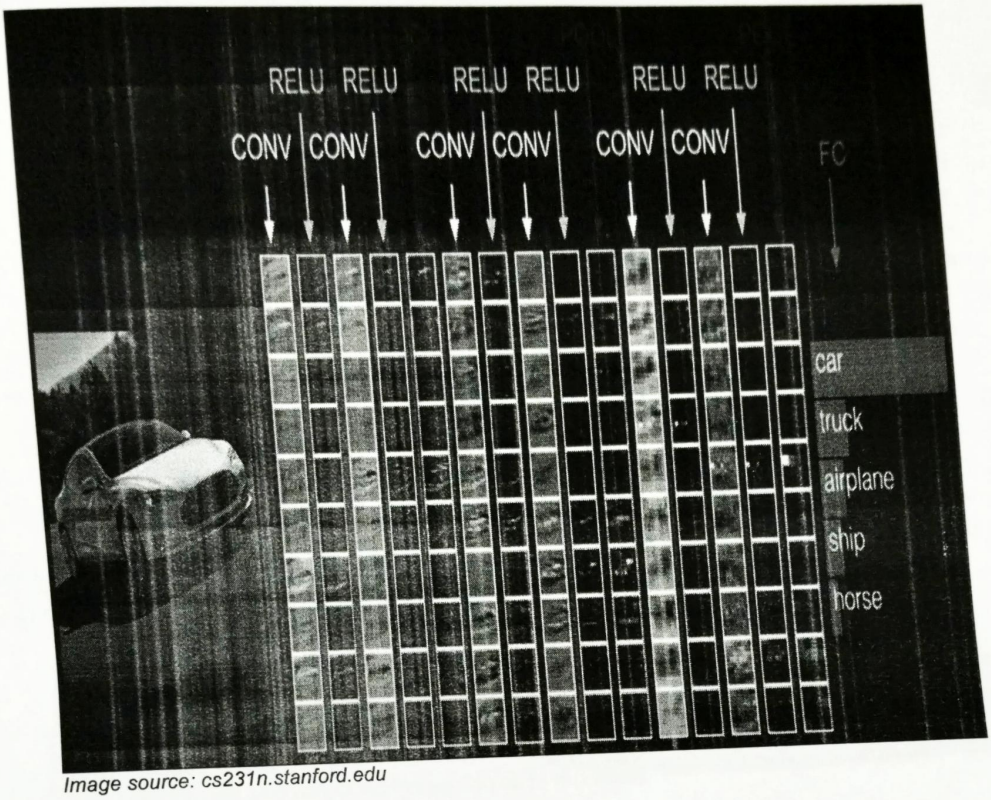


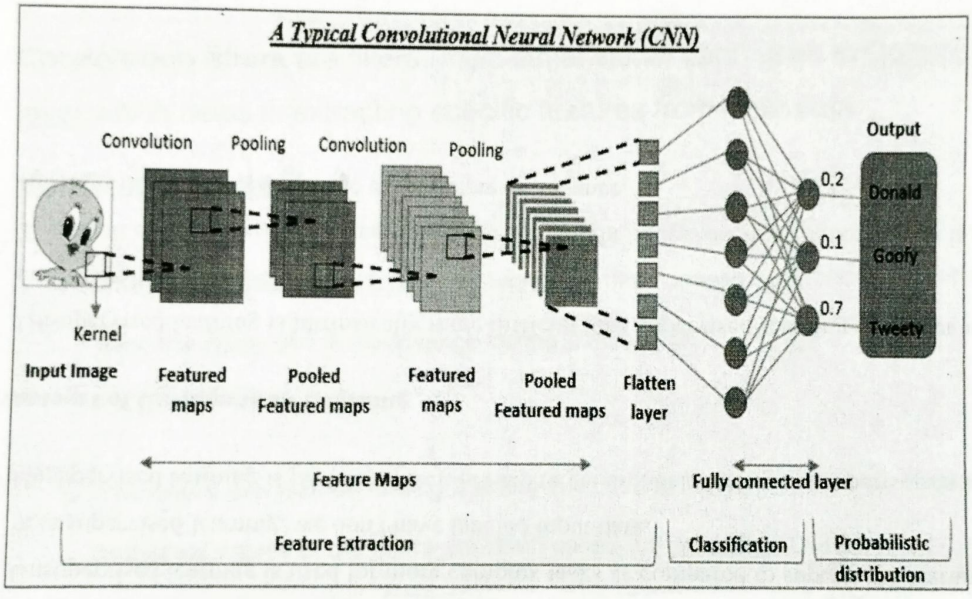
Image source: cs231n.stanford.edu

- **Output Layer:** The output from the fully connected layers is then fed into a logistic function for classification tasks like sigmoid or softmax which converts the output of each class into the probability score of each class.

Convolutional Neural Network: An Overview

S
Saily Shah15 Mar, 2022 • 9 min read

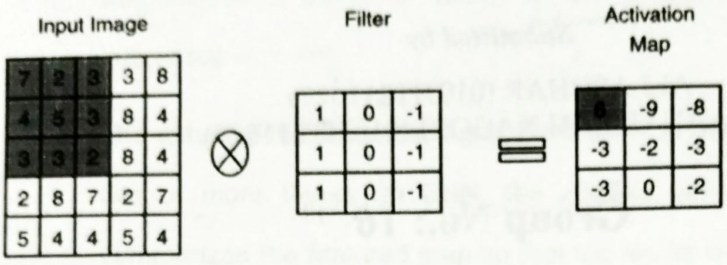
This article was published as a part of the Data Science Blogathon.



Which makes your image sharper or removes part of a photo that you wanted to remove?

If Yes, then you have implied convolution operation to your pl oto.

A convolutional layer is the main building block of a CNN. It contai s a set of filters (or kernels), parameters of which are to be learned throughout the training. The size of the filters is usually smaller than the actual image.



Convolution filters are filters (multi-dimensional data) used in Convolution layer which helps in extracting specific features from input data.

What is Stride?

- The stride indicates the pace by which the filter moves horizontally & vertically over the pixels of the input image during convolution.

Some important terms

1. The filters are learned during training (i.e. during backpropagation). Hence, the individual values of the filters are often called the **weights of CNN**.
2. A **neuron** is a filter whose weights are learned during training. E.g., a (3,3,3) filter (or neuron) has 27 units. Each neuron looks at a particular region in the output (i.e. its 'receptive field')
3. A **feature map** is a collection of multiple neurons, each looking at different inputs with the same weights. All neurons in a feature map extract the same feature (but from other input regions). It is called a 'feature map' because it maps where a particular part is found in the image.

What is Pooling?

- A pooling layer is another essential building block of CNN. It tries to figure out whether a particular region in the image has the feature we are interested in or not.
- The actual dictionary meaning of pooling is the act of sharing or combining two or more things. In CNN, the pooling layer does a similar job. It summarizes the featured map so that the model will not need to be trained on precisely positioned features, making a model more reliable and robust.
- The pooling layer looks at more significant regions (having multiple patches) of the image & captures aggregate statistics (min, max, average & global). In other words, it makes the network invariant to local transformations.
- The two most popular aggregate functions used in pooling are 'max' & 'average':

1.

    1. Max pooling – If any of the patches say something firmly about the presence of a particular feature, then the pooling layer counts that feature as 'detected'.
    2. Average pooling – If one patch says something very firmly, but the other ones disagree, the average pooling takes the average to find out.

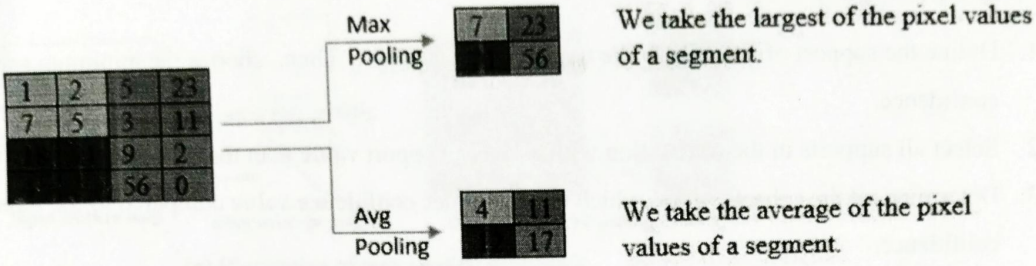


Fig: Max & Avg Pooling

- Pooling has the advantage of making the representation more compact by reducing the spatial size of the feature maps, thereby reducing the number of parameters to be learnt. Pooling reduces only the height & width of the feature map, not the (i.e. number of channels). For example, if we have 'm' feature maps each of size (c,c), the pooling operation will produce 'm' outputs (c/2,c/2).
- On the other hand, pooling also loses a lot of information, which is often considered a potential disadvantage.
- The pooling layer has 'NO PARAMETERS' i.e. 'ZERO TRAINABLE PARAMETERS'. The pooling layer computes the aggregate of the input. E.g. in max pooling, it takes a maximum over a group of pixels. We do not need any adjustments in any parameters.
- The following image explains when a MAX pooling works and when AVG pooling works:

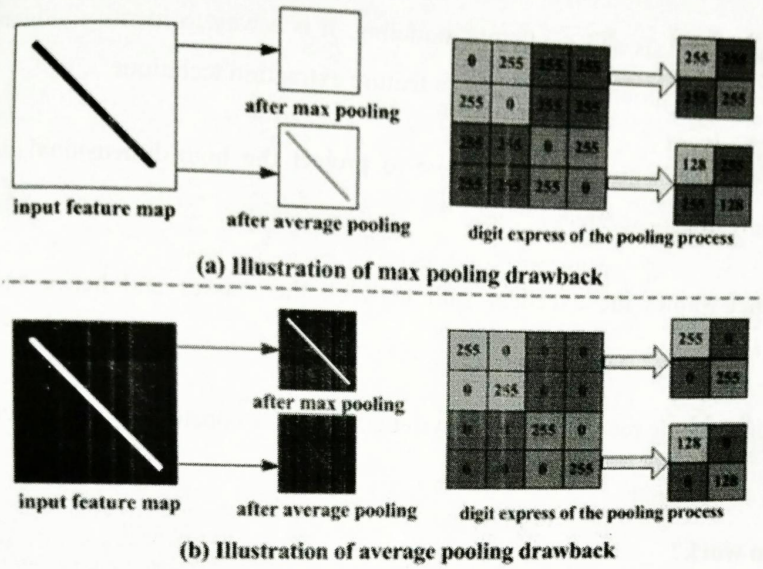


Fig. Toy example illustrating the drawbacks of max pooling and average pooling.

Padding

As described above, one tricky issue when applying convolutional layers is that we tend to lose pixels on the perimeter of our image. Consider Fig. 7.3.1 that depicts the pixel utilization as a function of the convolution kernel size and the position within the image. The pixels in the corners are hardly used at all.

Fig. 7.3.1 Pixel utilization for convolutions of size 1×1, 2×2, and 3×3 respectively.

Since we typically use small kernels, for any given convolution we might only lose a few pixels but this can add up as we apply many successive convolutional layers. One straightforward solution to this problem is to add extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image. Typically, we set the values of the extra pixels to zero. In Fig. 7.3.2, we pad a 3×3 input, increasing its size to 5×5. The corresponding output then increases to a 4×4 matrix. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: 0×0+0×1+0×2+0×3=0.

Fig. 7.3.2 Two-dimensional cross-correlation with padding.

The pooling operation involves sliding a two-dimensional filter over each channel of feature map and summarising the features lying within the region covered by the filter.
For a feature map having dimensions $n_h \times n_w \times n_c$, the dimensions of output obtained after a pooling layer is

$$(n_h - f + 1) / s \times (n_w - f + 1) / s \times n_c$$

where,

-> n_h - height of feature map
-> n_w - width of feature map
-> n_c - number of channels in the feature map
-> f - size of filter
-> s - stride length

A common CNN model architecture is to have a number of convolution and pooling layers stacked one after the other.

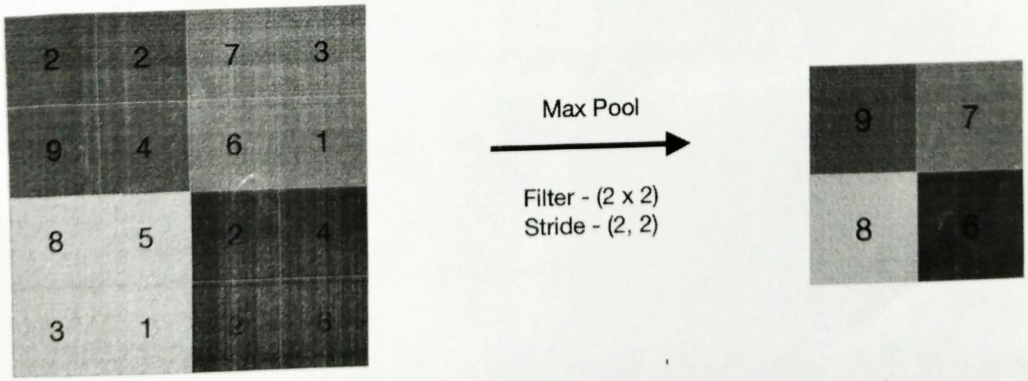
Why to use Pooling Layers?

- Pooling layers are used to reduce the dimensions of the feature maps. Thus, it reduces the number of parameters to learn and the amount of computation performed in the network.
- The pooling layer summarises the features present in a region of the feature map generated by a convolution layer. So, further operations are performed on summarised features instead of precisely positioned features generated by the convolution layer. This makes the model more robust to variations in the position of the features in the input image.

Types of Pooling Layers:

Max Pooling

1. Max pooling is a pooling operation that selects the maximum element from the region of the feature map covered by the filter. Thus, the output after max-pooling layer would be a feature map containing the most prominent features of the previous feature map.



1. This can be achieved using MaxPooling2D layer in keras as follows:
   Code #1 : Performing Max Pooling using keras

- Python3

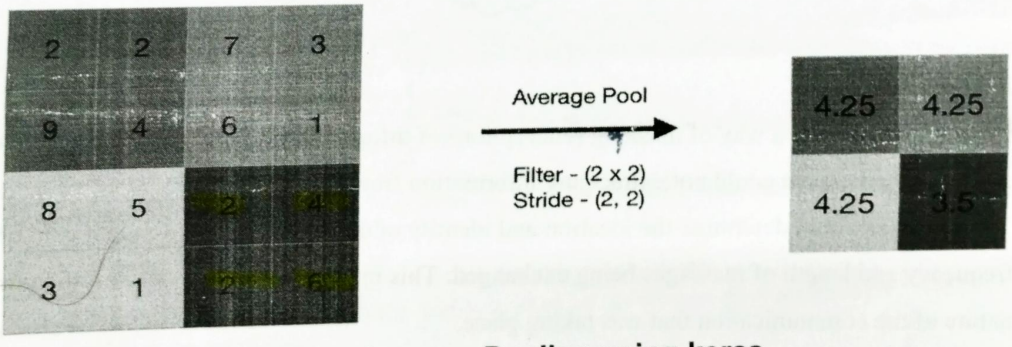
```
import numpy as np
from keras.models import Sequential
from keras.layers import MaxPooling2D

# define input image
image = np.array([[2, 2, 7, 3],
                  [9, 4, 6, 1],
                  [8, 5, 2, 4],
                  [3, 1, 2, 6]])
image = image.reshape(1, 4, 4, 1)

# define model containing just a single max pooling layer
model = Sequential(
    [MaxPooling2D(pool_size = 2, strides = 2)])

# generate pooled output
output = model.predict(image)

# print output image
output = np.squeeze(output)
print(output)
```

1. **Output:**

```
[[9. 7.]
[8. 6.]]
```

Average Pooling

1. Average pooling computes the average of the elements present in the region of feature map covered by the filter. Thus, while max pooling gives the most prominent feature in a particular patch of the feature map, average pooling gives the average of features present in a patch.



1. **Code #2 : Performing Average Pooling using keras**

- Python3

```
import numpy as np
from keras.models import Sequential
from keras.layers import AveragePooling2D

# define input image
image = np.array([[2, 2, 7, 3],
                  [9, 4, 6, 1],
                  [8, 5, 2, 4],
                  [3, 1, 2, 6]])
```

```
image = image.reshape(1, 4, 4, 1)

# define model containing just a single average pooling layer
model = Sequential(
    [AveragePooling2D(pool_size = 2, strides = 2)])

# generate pooled output
output = model.predict(image)

# print output image
output = np.squeeze(output)
print(output)
```

1. **Output:**

```
[[4.25 4.25]
[4.25 3.5 ]]
```

Global Pooling

1. Global pooling reduces each channel in the feature map to a single value. Thus, an n_h **x** n_w **x** n_c feature map is reduced to **1 x 1 x** n_c feature map. This is equivalent to using a filter of dimensions n_h **x** n_w i.e. the dimensions of the feature map.
   Further, it can be either global max pooling or global average pooling.
   Code #3 : Performing Global Pooling using keras

- Python3

```
import numpy as np
from keras.models import Sequential
from keras.layers import GlobalMaxPooling2D
from keras.layers import GlobalAveragePooling2D

# define input image
image = np.array([[2, 2, 7, 3],
                  [9, 4, 6, 1],
                  [8, 5, 2, 4],
                  [3, 1, 2, 6]])
image = image.reshape(1, 4, 4, 1)

# define gm_model containing just a single global-max pooling layer
gm_model = Sequential(
    [GlobalMaxPooling2D()])

# define ga_model containing just a single global-average pooling layer
ga_model = Sequential(
    [GlobalAveragePooling2D()])

# generate pooled output
gm_output = gm_model.predict(image)
ga_output = ga_model.predict(image)

# print output image
gm_output = np.squeeze(gm_output)
ga_output = np.squeeze(ga_output)
print("gm_output: ", gm_output)
print("ga_output: ", ga_output)
```

1. **Output:**

```
gm_output:  9.0
ga_output:  4.0625
```

In convolutional neural networks (CNNs), the pooling layer is a common type of layer that is typically added after convolutional layers. The pooling layer is used to reduce the spatial dimensions (i.e., the width and height) of the feature maps, while preserving the depth (i.e., the number of channels).

1. The pooling layer works by dividing the input feature map into a set of non-overlapping regions, called pooling regions. Each pooling region is then transformed into a single output value, which represents the presence of a particular feature in that region. The most common types of pooling operations are max pooling and average pooling.
2. In max pooling, the output value for each pooling region is simply the maximum value of the input values within that region. This has the effect of preserving the most salient features in each pooling region, while discarding less relevant information. Max pooling is often used in CNNs for object recognition tasks, as it helps to identify the most distinctive features of an object, such as its edges and corners.
3. In average pooling, the output value for each pooling region is the average of the input values within that region. This has the effect of preserving more information than max pooling, but may also dilute the most salient features. Average pooling is often used in CNNs for tasks such as image segmentation and object detection, where a more fine-grained representation of the input is required.

Pooling layers are typically used in conjunction with convolutional layers in a CNN, with each pooling layer reducing the spatial dimensions of the feature maps, while the convolutional layers extract increasingly complex features from the input. The resulting feature maps are then passed to a fully connected layer, which performs the final classification or regression task.

Advantages of Pooling Layer:

1. Dimensionality reduction: The main advantage of pooling layers is that they help in reducing the spatial dimensions of the feature maps. This reduces the computational cost and also helps in avoiding overfitting by reducing the number of parameters in the model.
2. Translation invariance: Pooling layers are also useful in achieving translation invariance in the feature maps. This means that the position of an object in the image does not affect the classification result, as the same features are detected regardless of the position of the object.
3. Feature selection: Pooling layers can also help in selecting the most important features from the input, as max pooling selects the most salient features and average pooling preserves more information.

Disadvantages of Pooling Layer:

1. Information loss: One of the main disadvantages of pooling layers is that they discard some information from the input feature maps, which can be important for the final classification or regression task.
2. Over-smoothing: Pooling layers can also cause over-smoothing of the feature maps, which can result in the loss of some fine-grained details that are important for the final classification or regression task.
3. Hyperparameter tuning: Pooling layers also introduce hyperparameters such as the size of the pooling regions and the stride, which need to be tuned in order to achieve optimal performance. This can be time-consuming and requires some expertise in model building.

Convolution and Pooling as an Infinitely Strong Prior

Weight Prior

Assumptions about weights (before learning) in terms of acceptable values and range are encoded into the **prior distribution** of weights.

S.No.	Prior Type	Variance/Confidence Type
1.	Weak	High Variance, Low Confidence
2.	Strong	**Narrow range** of values about which we are **confident** before learning begins.
3.	Infinitely strong	Demarkates certain values as **forbidden** completely assigning them **zero probability.**

- Convolution imposes an **infinitely strong prior** by making the following **restrictions on weights:**
  - **Adjacent units** must have the **same weight** but shifted in space.
  - Except for a **small spatially connected** region, all **other weights** must be **zero.**
- Features should be **translation invariant.**
- If tasks relies on preserving specific spatial information, then pooling can cause on all features can increase training error.

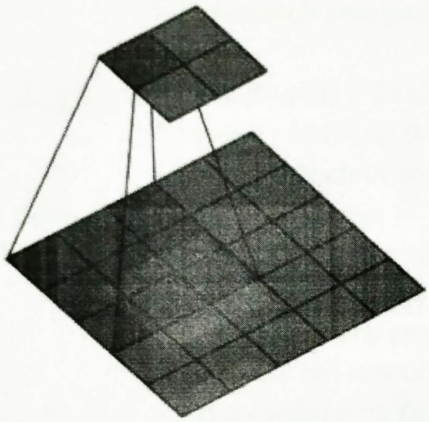
Variants of the Basic Convolution Function

In practical implementations of the convolution operation, certain modifications are made which deviate from standard discrete convolution operation -

- In general a convolution layer consists of application of **several different kernels** to the input. Since, convolution with a **single kernel can extract only one kind of feature.**
- The input is generally not real-valued but instead **vector valued.**
- Multi-channel convolutions are commutative iff **number of output and input channels is the same.**

Effect of Strides

- Stride is the number of pixels shifts over the input matrix.
- In order to allow for calculation of features at a **coarser level** strided convolutions can be used.
- The effect of strided convolution is the same as that of a **convolution followed by a downsampling stage.**
- Strides can be used to **reduce the representation size.**
- Below is an example representing **2-D Convolution, with (3 * 3) Kernel and Stride of 2 units.**

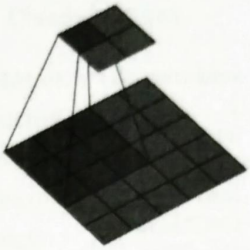
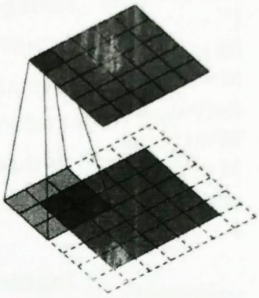
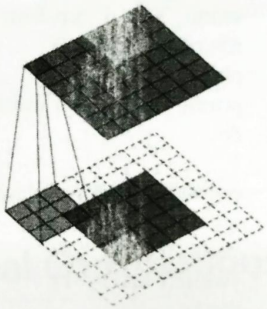


Effect of Zero Padding

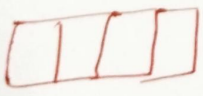
- Convolution networks can implicitly zero pad the input V, to make it wider.
- Without zero padding,the width of representation shrinks by one pixel less than the kernel width at each layer.
- Zero padding the input allows to control kernel width and size of **output** independently.

Zero Padding Strategies

3 common zero padding strategies are:

Zero Padding Type	Properties	Example
Valid Zero-Padding	1. **No zero padding** is used. 2. Output is computed only at places where **entire kernel lies inside the input.** 3. **Shrinkage > 0** 4. **Limits #convolution layers** to be used in network 5. Input's width = m, Kernel's width = k, **Width of Output = m-k+1**	
Same Zero-Padding $n-f+1$ $n+2p-f+1$	1. Just enough **zero padding is added to keep:** 1.a. **Size(Ouput) = Size(Input)** 2. Input is padded by **(k-1) zeros** 3. Since the **#output units connected to border pixels is less** than that for centre pixels, it may **under-represent border pixels.** 4. Can **add as many convolution layers** as hardware can support 5. Input's width = m, Kernel's width = k, **Width of Output = m**	
Strong Zero-Padding	1. The input is padded by enough zeros such that **each input pixel is connected to same #output units.** 2. Allows us to make an **arbitrarily deep NN.** 3. Can **add as many convolution layers** as hardware can support 4. Input's width = m, Kernel's width = k, **Width of Output = m+k-1**	

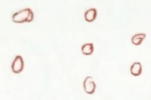
Types of Convolution



$6*6 + 6+3-1 = 111$

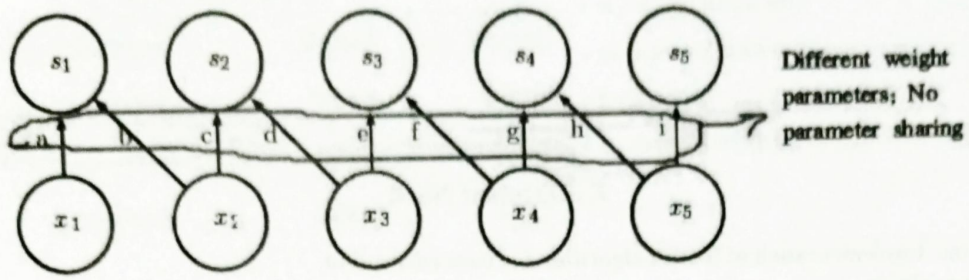
$6 * 3 = 8$

Comparing Unshared, Tiled and Traditional Convolutions

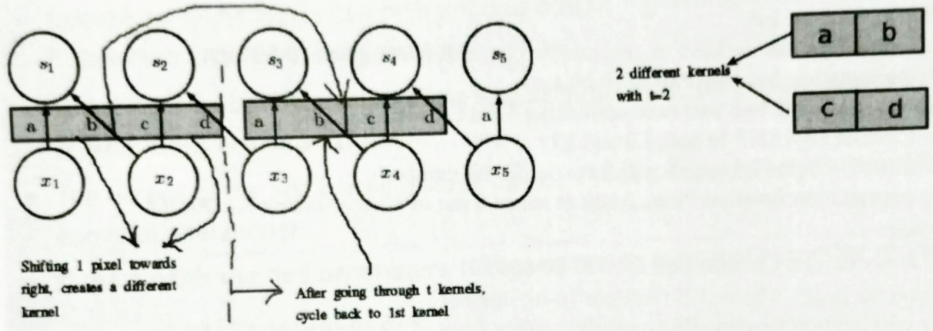
Convolution Type	Properties	Advantages and Disadvantages
Unshared Convolution 	1. **No Parameter sharing**. 2. Each output unit performs a **linear operation on its neighbourhood** but parameters are not shared across **output** units. 3. Captures **local connectivity** while allowing **different features** to be computed at **different spatial locations.**	**Advantages** 1. **Reducing memory** consumption 2. Increasing **statistical efficiency** 3. Reducing the **amount of computation needed** to perform forward and back-propagation. **Disadvantages** 1. requires **much more parameters** than the convolution operation.
Tiled Convolution	1. Offers a **compromise b/w unshared and traditional convoltion.** 2. Learn a **set of kernels and cycle/rotate them** through space. 3. Makes use of **parameter sharing.**	**Advantages** 1. Reduces the #parameters in model.
Traditional Convolution	1. Equivalent to **tiled convolution with t=1.** 2. Has the **same connectivity as unshared convolution**	

Examples of Unshared, Tiled and Traditional Convolutions

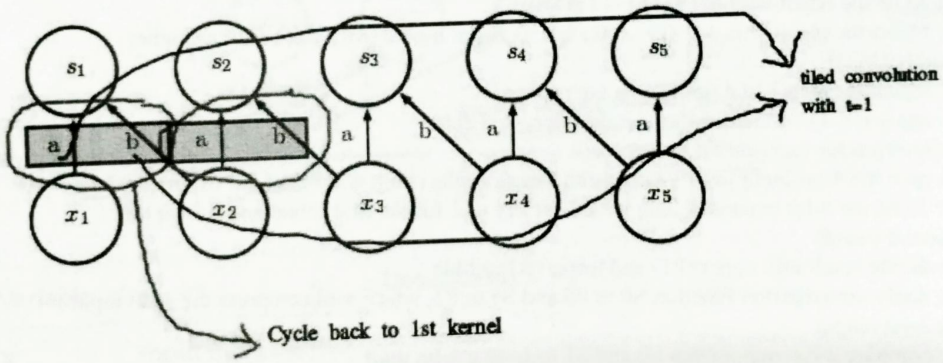
Unshared Convolution



Tiled Convolution



Traditional Convolution



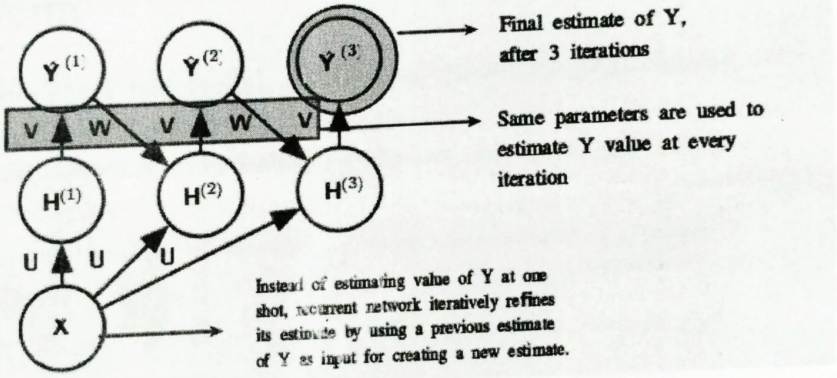
Comparing Computation Times

Type	Computations	Parameters
Fully connected	$O(mn)$	$O(mn)$
Locally connected	$O(kn)$	$O(kn)$
Tiled	$O(kn)$	$O(kl)$
Traditional	$O(kn)$	$O(k)$

- m = number of input units
- n = number of output units
- k = kernel size
- l = number of kernels in the set (for tiled convolution)

Structured Outputs

- Convolutional networks can be trained to output high-dimensional structured output rather than just a classification score.
- To produce an output map as same size as input map, only same-padded convolutions can be stacked.
- The output of the first labelling stage can be refined successively by another convolutional model.
- If the models use tied parameters, this gives rise to a type of recursive model



Variable	Description
X	Input image tensor
Y	Probability distribution over tensor for each pixel
H	Hidden representation
U	Tensor of convolution kernels
V	Tensor of kernels to produce estimation of lables
W	Kernel tensor to convolve over Y to provide input to H

Data Types

> The data used with a convolutional network usually consist of several channels,each channel being the observation of a different quantity at some point in space or time.

- When output is **variable sized, no extra design change** needs to be made.
- When output requires **fixed size** (classification), a **pooling stage** with **kernel size proportional to input size** needs to be used.

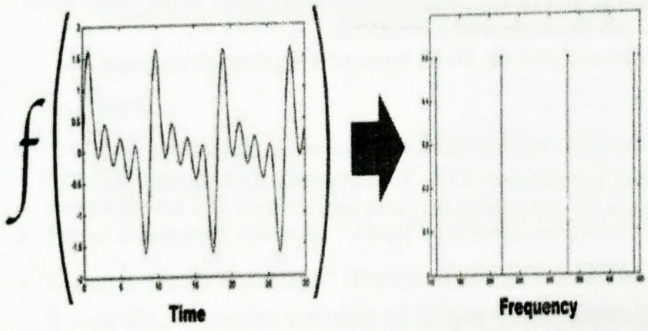
Dimensions	Single channel	Multichannel
1D	**Raw audio** (single amplitude value per time point)	**Skeleton animatin data** (orientation of each joint)
2D	**Audio spectrogram** (one FFT coefficient per time point per frequency)	**Color image** (RGB triplet per (x,y) tuple)
3D	**CT scan** (one value per (x,y,z) tuple)	**Color video** (one RGB triplet per (x,y) tuple per time instant)

Different data types based on the number of spatial dimensions and channels

Efficient Convolution Algorithms

Fourier Transform

The Fourier Transform is a tool that breaks a waveform (a function or signal) into an alternate representation, characterized by sine and cosines.



Separable Kernels

- Convolution is equivalent to converting both **input and kernel** to frequency domain using a **Fourier transform**, performing **point-wise multiplication of two signals**:

$$F\{f \star g\} = F\{f\} \times F\{g\}$$

A * B

- Converting back to **time domain** using an **inverse Fourier transform.**

$$f \star g = F^{-1}\{F\{f \star g\}\} = F^{-1}\{F\{f\} \times F\{g\}\}$$

- When a **d-dimensional kernel** can be expressed as **outer product of d vectors**, one vector per dimension, the kernel is called **separable.**
- The kernel also takes **fewer parameters** to represent as vectors.

Kernel Type	Runtime complexity for *d-dimensional kernel* with *w elements wide*
Traditional Kernel	$O(w^d)$
Separable Kernel	$O(w \times d)$

Random and Unsupervised Features

To reduce the cost of convolutional network training, we have to use features that are not trained in a supervised way:

- **Random Initialization:**
  - Layers consisting of **convolution followed by pooling** naturally become **frequency selective and translation invariant** when assigned random weights.
  - **Randomly initialize** several CNN architectures and just **train the last classification layer.**
  - Once a winner is determined, train that model using a **more expensive approach** (supervised approach).
- **Hand Designed Kernels:** * Used to **detect edges** at a certain orientation or scale.
- **Unsupervised Training:** * Unsupervised pre-training may offer **regularization effect.** * It may also allow for **training of larger CNNs** because of **reduced computation cost.**

Greedy Layer-wise Pre-training

Instead of training an entire convolutional layer at a time, we can **train a model of a small patch:**

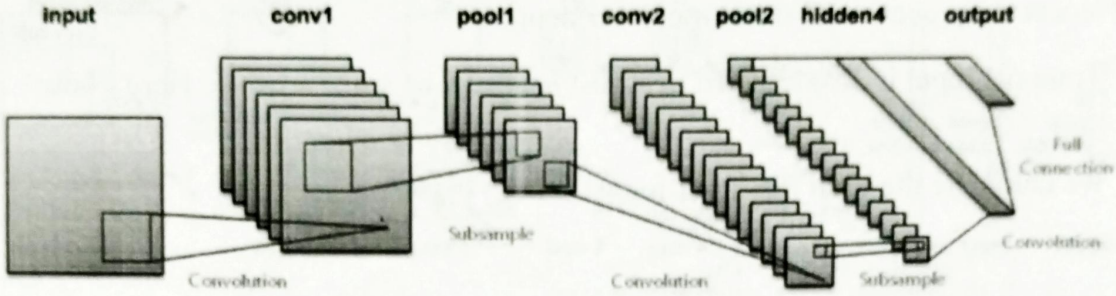
- Train the **first layer in isolation.**
- **Extract all features** from the first layer **only once.**

- Once the first layer is trained, its **output is stored** and **used as input** for training the next layer.
- We can **train very large models** and incur a **high computational cost** only at inference time.

Scroll To Top

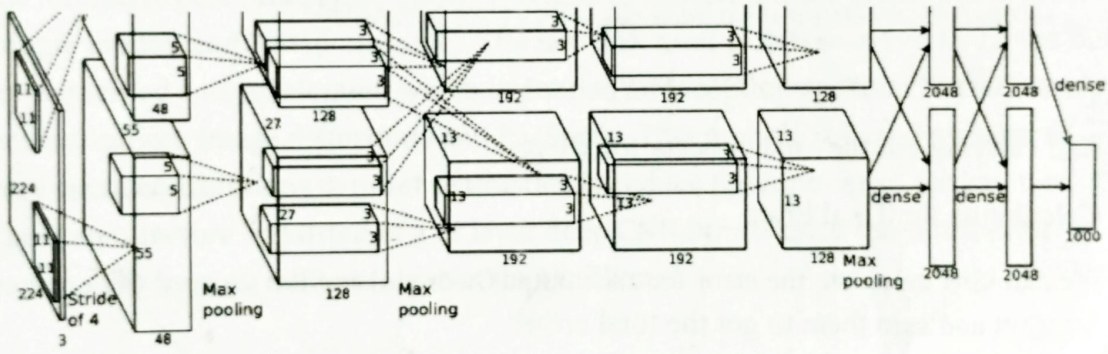
LeNet-5 (1998)

LeNet-5, a pioneering 7-level convolutional network by LeCun et al in 1998, that classifies digits, was applied by several banks to recognise hand-written numbers on checks (cheques) digitized in 32x32 pixel greyscale inputimages. The ability to process higher resolution images requires larger and more convolutional layers, so this technique is constrained by the availability of computing resources.



AlexNet (2012)

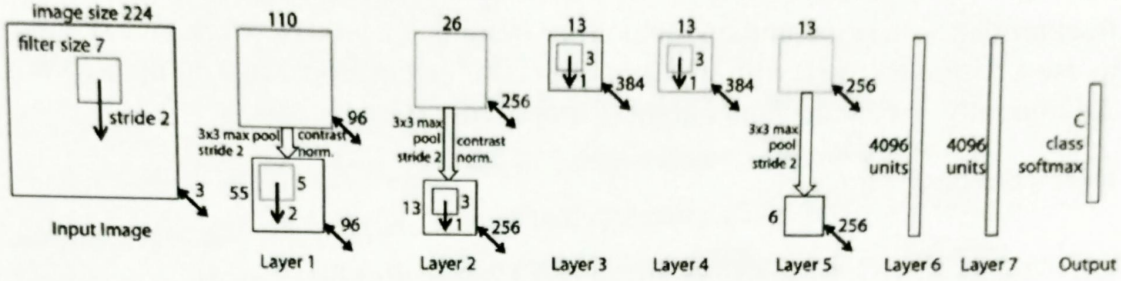
In 2012, AlexNet significantly outperformed all the prior competitors and won the challenge by reducing the top-5 error from 26% to 15.3%. The second place top-5 error rate, which was not a CNN variation, was around 26.2%.



The network had a very similar architecture as LeNet by Yann LeCun et al but was deeper, with more filters per layer, and with stacked convolutional layers. It consisted 11x11, 5x5,3x3, convolutions, max pooling, dropout, data augmentation, ReLU activations, SGD with momentum. It attached ReLU activations after every convolutional and fully-connected layer. AlexNet was trained for 6 days simultaneously on two Nvidia Geforce GTX 580 GPUs which is the reason for why their network is split into two pipelines. AlexNet was designed by the SuperVision group, consisting of Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever.

ZFNet(2013)

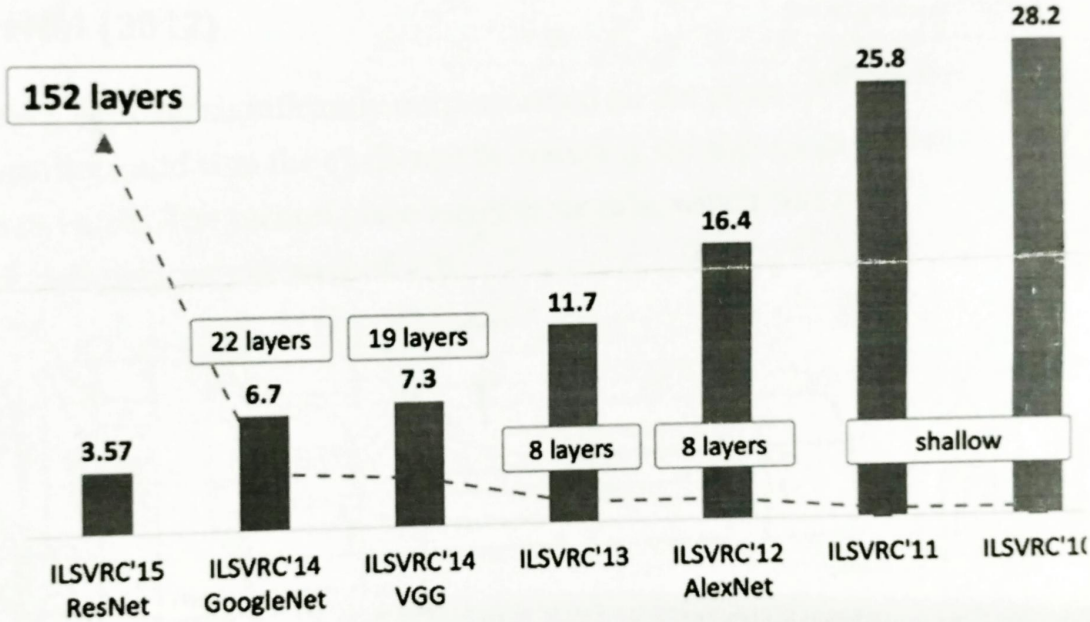
Not surprisingly, the ILSVRC 2013 winner was also a CNN which became known as ZFNet. It achieved a top-5 error rate of 14.8% which is now already half of the prior mentioned non-neural error rate. It was mostly an achievement by tweaking the hyper-parameters of AlexNet while maintaining the same structure with additional Deep Learning elements as discussed earlier in this essay.



GoogLeNet/Inception(2014)

The winner of the ILSVRC 2014 competition was GoogLeNet(a.k.a. Inception V1) from Google. It achieved a top-5 error rate of 6.67%! This was very close to human level performance which the organisers of the challenge were now forced to evaluate. As it turns out, this was actually rather hard to do and required some human training in order to beat GoogLeNets accuracy. After a few days of training, the human expert (Andrej Karpathy) was able to achieve a top-5 error rate of 5.1% (single model) and 3.6%(ensemble). The network used a CNN inspired by LeNet but implemented a novel element which is dubbed an inception module. It used batch normalization, image distortions and RMSprop. This module is based on several very small convolutions in order to drastically reduce the number of parameters. Their architecture consisted of a 22 layer deep CNN but reduced the number of parameters from 60 million (AlexNet) to 4 million.

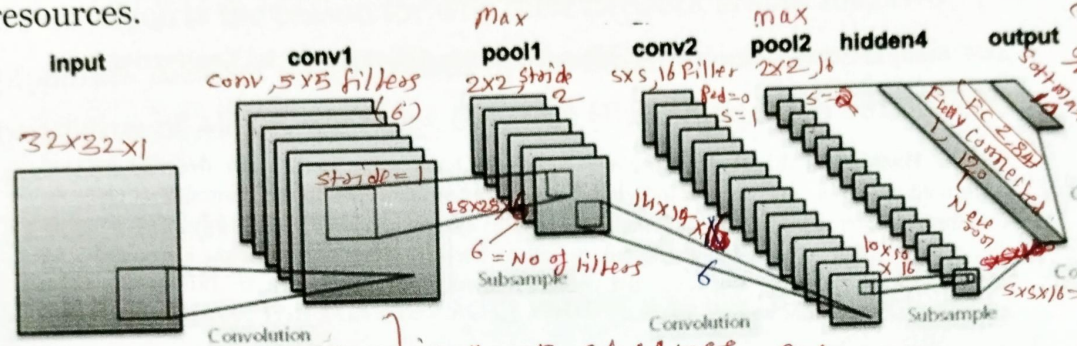
A Convolutional Neural Network (**CNN**, or **ConvNet**) are a special kind of multi-layer neural networks, designed to recognize visual patterns directly from pixel images with minimal preprocessing.. The **ImageNet** project is a large visual database designed for use in visual object recognition software research. The ImageNet project runs an annual software contest, the **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)**, where software programs compete to correctly classify and detect objects and scenes. Here I will talk about CNN architectures of ILSVRC top competitors .



LeNet-5 (1998)

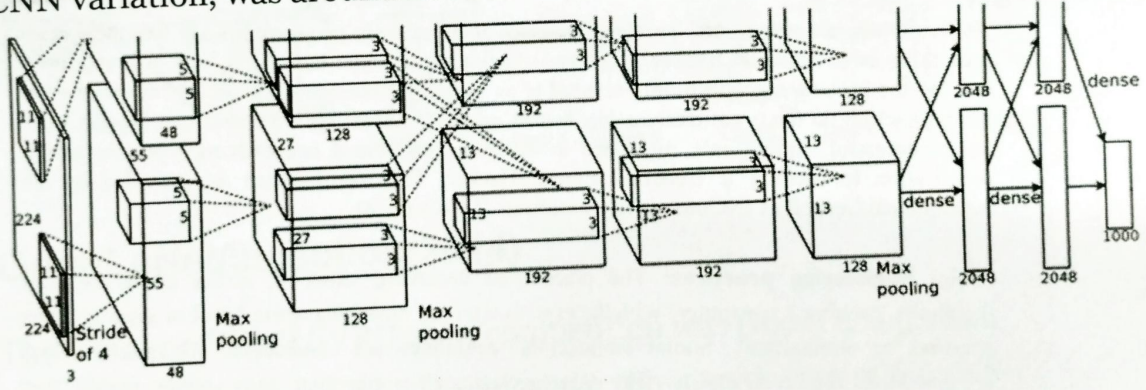
LeNet-5, a pioneering 7-level convolutional network by LeCun et al in 1998, that classifies digits, was applied by several banks to recognise hand-written numbers on checks (cheques) digitized in

32x32 pixel greyscale inputimages. The ability to process higher resolution images requires larger and more convolutional layers, so this technique is constrained by the availability of computing resources.



AlexNet (2012)

In 2012, AlexNet significantly outperformed all the prior competitors and won the challenge by reducing the top-5 error from 26% to 15.3%. The second place top-5 error rate, which was not a CNN variation, was around 26.2%.

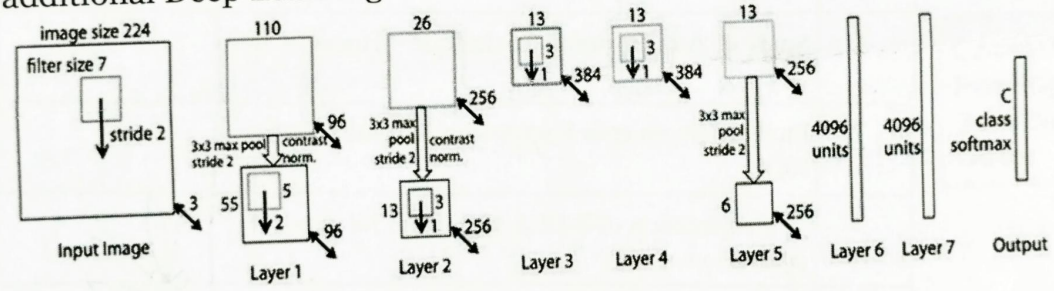


The network had a very similar architecture as LeNet by Yann LeCun et al but was deeper, with more filters per layer, and with stacked convolutional layers. It consisted 11x11, 5x5,3x3, convolutions, max pooling, dropout, data augmentation, ReLU

activations, SGD with momentum. It attached ReLU activations after every convolutional and fully-connected layer. AlexNet was trained for 6 days simultaneously on two Nvidia Geforce GTX 580 GPUs which is the reason for why their network is split into two pipelines. AlexNet was designed by the SuperVision group, consisting of Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever.

ZFNet(2013)

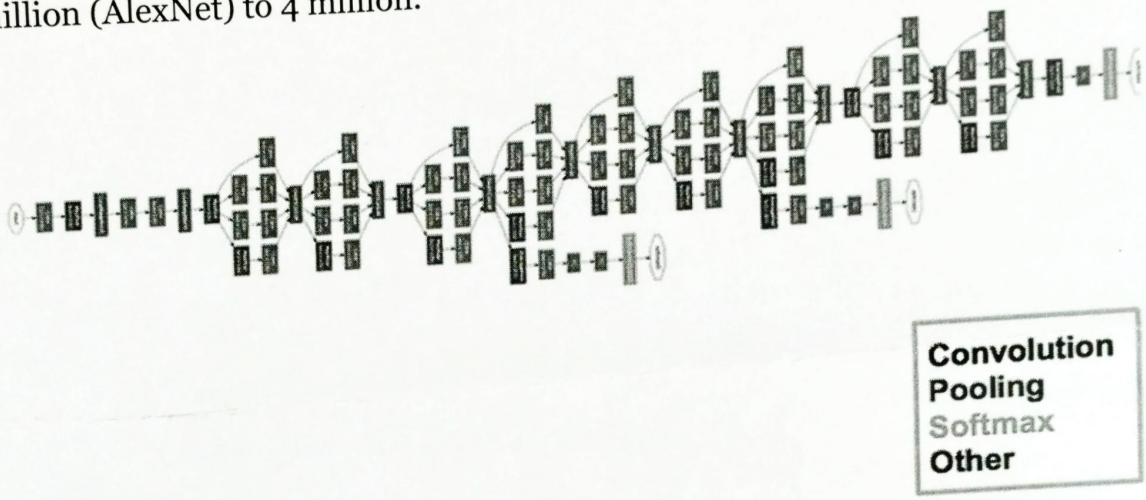
Not surprisingly, the ILSVRC 2013 winner was also a CNN which became known as ZFNet. It achieved a top-5 error rate of 14.8% which is now already half of the prior mentioned non-neural error rate. It was mostly an achievement by tweaking the hyper-parameters of AlexNet while maintaining the same structure with additional Deep Learning elements as discussed earlier in this essay.



GoogLeNet/Inception(2014)

The winner of the ILSVRC 2014 competition was GoogLeNet(a.k.a. Inception V1) from Google. It achieved a top-5 error rate of 6.67%! This was very close to human level performance which the organisers of the challenge were now forced to evaluate. As it turns out, this was actually rather hard to do and required some human training in order to beat GoogLeNets accuracy. After a few days of

training, the human expert (Andrej Karpathy) was able to achieve a top-5 error rate of 5.1%(single model) and 3.6%(ensemble). The network used a CNN inspired by LeNet but implemented a novel element which is dubbed an inception module. It used batch normalization, image distortions and RMSprop. This module is based on several very small convolutions in order to drastically reduce the number of parameters. Their architecture consisted of a 22 layer deep CNN but reduced the number of parameters from 60 million (AlexNet) to 4 million.



VGGNet (2014)

The runner-up at the ILSVRC 2014 competition is dubbed VGGNet by the community and was developed by Simonyan and Zisserman. VGGNet consists of 16 convolutional layers and is very appealing because of its very uniform architecture. Similar to AlexNet, only 3x3 convolutions, but lots of filters. Trained on 4 GPUs for 2–3 weeks. It is currently the most preferred choice in the community for extracting features from images. The weight

A Brief Overview of Recurrent Neural Networks (RNN)

Debasish Kalita – Updated On August 3rd, 2023

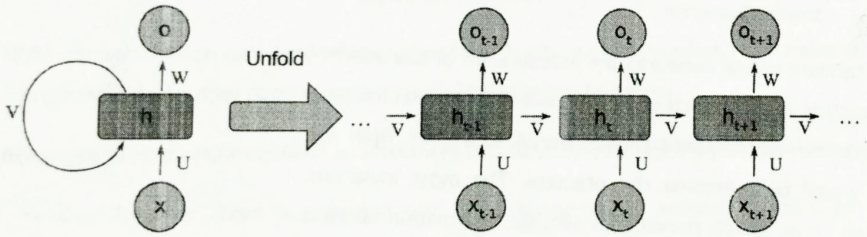
Beginner Deep Learning Python

This article was published as a part of the Data Science Blogathon.

Apple's Siri and Google's voice search both use Recurrent Neural Networks (RNNs), which are the state-of-the-art method for sequential data. It's the first algorithm with an internal memory that remembers its input, making it perfect for problems involving sequential data in machine learning. It's one of the algorithms responsible for the incredible advances in deep learning over the last few years. In this article, we'll go over the fundamentals of recurrent neural networks, as well as the most pressing difficulties and how to address them.

Introduction on Recurrent Neural Networks

A Deep Learning approach for modelling sequential data is **Recurrent Neural Networks (RNN)**. RNNs were the standard suggestion for working with sequential data before the advent of attention models. Specific parameters for each element of the sequence may be required by a deep feedforward model. It may also be unable to generalize to variable-length sequences.



Source: Medium.com

Recurrent Neural Networks use the same weights for each element of the sequence, decreasing the number of parameters and allowing the model to generalize to sequences of varying lengths. RNNs generalize to structured data other than sequential data, such as geographical or graphical data, because of its design.

Recurrent neural networks, like many other deep learning techniques, are relatively old. They were first developed in the 1980s, but we didn't appreciate their full potential until lately. The advent of long short-term memory (LSTM) in the 1990s, combined with an increase in computational power and the vast amounts of data that we now have to deal

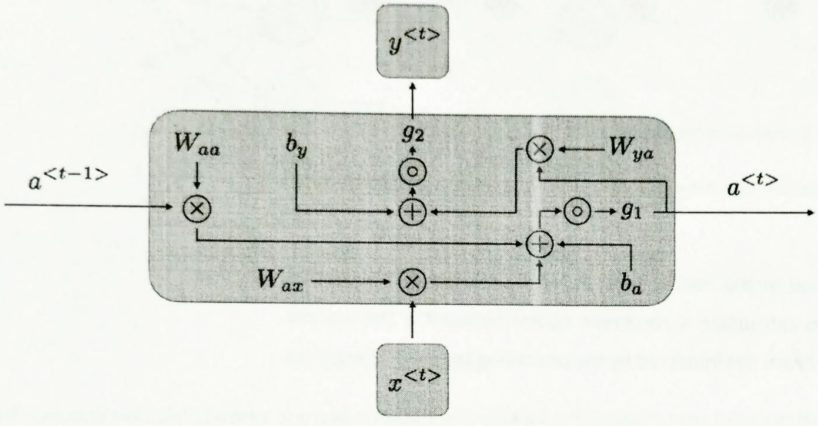
Brief Overview of Recurrent Neural Networks (RNN)

Source: Standford.edu

For each timestep t, the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows:

$$\boxed{a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)} \quad \text{and} \quad \boxed{y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)}$$

where $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$ are coefficients that are shared temporally and g_1, g_2 activation functions.



Source: Standford.edu

RNN architecture can vary depending on the problem you're trying to solve. From those with a single input and output to those with many (with variations between).

Below are some examples of RNN architectures that can help you better understand this.

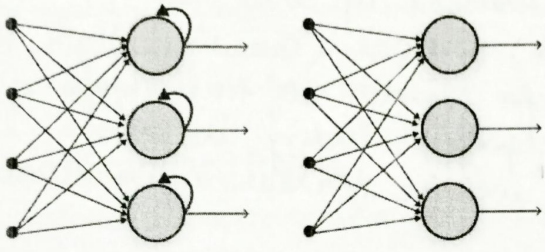
- **One To One:** There is only one pair here. A one-to-one architecture is used in traditional neural networks.
- **One To Many:** A single input in a one-to-many network might result in numerous outputs. One too many networks are used in the production of music, for example.
- **Many To One:** In this scenario, a single output is produced by combining many inputs from distinct time steps. Sentiment analysis and emotion identification use such networks, in which the class label is determined by a sequence of words.
- **Many To Many:** For many to many, there are numerous options. Two inputs yield three outputs. Machine translation systems, such as English to French or vice versa translation systems, use many to many networks.

Brief Overview of Recurrent Neural Networks (RNN)

Recurrent Neural Network Vs Feedforward Neural Network

A feed-forward neural network has only one route of information flow: from the input layer to the output layer, passing through the hidden layers. The data flows across the network in a straight route, never going through the same node twice.

The information flow between an RNN and a feed-forward neural network is depicted in the two figures below.



Recurrent Neural Network Feed-Forward Neural Network

Source: Uditvani.com

Feed-forward neural networks are poor predictions of what will happen next because they have no memory of the information they receive. Because it simply analyses the current input, a feed-forward network has no idea of temporal order. Apart from its training, it has no memory of what transpired in the past.

The information is in an RNN cycle via a loop. Before making a judgment, it evaluates the current input as well as what it has learned from past inputs. A recurrent neural network, on the other hand, may recall due to internal memory. It produces output, copies it, and then returns it to the network.

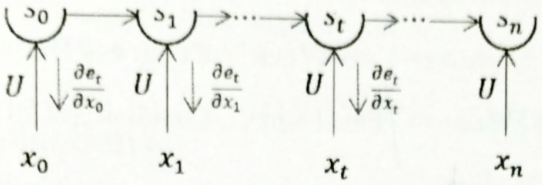
Backpropagation Through Time (BPTT)

When we apply a Backpropagation algorithm to a Recurrent Neural Network with time series data as its input, we call it backpropagation through time.

A single input is sent into the network at a time in a normal RNN, and a single output is obtained. Backpropagation, on the other hand, uses both the current and prior inputs as input. This is referred to as a timestep, and one timestep will consist of multiple time series data points entering the RNN at the same time.

ss Report
81/97

Brief Overview of Recurrent Neural Networks (RNN)

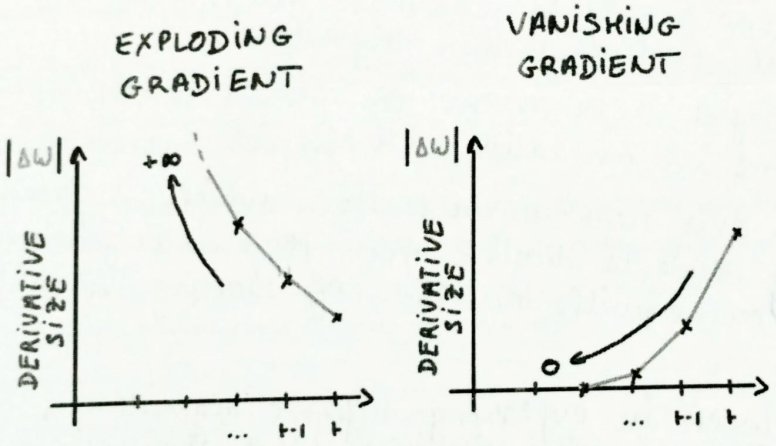


Source: Medium.com

The output of the neural network is used to calculate and collect the errors once it has trained on a time set and given you an output. The network is then rolled back up, and weights are recalculated and adjusted to account for the faults.

Two issues of Standard RNNs

There are two key challenges that RNNs have had to overcome, but in order to comprehend them, one must first grasp what a gradient is.



Source: GreatLearning.com

With regard to its inputs, a gradient is a partial derivative. If you're not sure what that implies, consider this: a gradient quantifies how much the output of a function varies when the inputs are changed slightly.

A function's slope is also known as its gradient. The steeper the slope, the faster a model can learn, the higher the gradient. The model, on the other hand, will stop learning if the slope is zero. A gradient is used to measure the change in all weights in relation to the change in error.

Brief Overview of Recurrent Neural Networks (RNN)

Fortunately, Sepp Hochreiter and Juergen Schmidhuber's LSTM concept solved the problem.

RNN Applications

Recurrent Neural Networks are used to tackle a variety of problems involving sequence data. There are many different types of sequence data, but the following are the most common: Audio, Text, Video, Biological sequences.

Using RNN models and sequence datasets, you may tackle a variety of problems, including :

- Speech recognition
- Generation of music
- Automated Translations
- Analysis of video action
- Sequence study of the genome and DNA

Basic Python Implementation (RNN with Keras)

Import the required libraries

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

Here's a simple Sequential model that processes integer sequences, embeds each integer into a 64-dimensional vector, and then uses an LSTM layer to handle the sequence of vectors.

```
model = keras.Sequential()
model.add(layers.Embedding(input_dim=1000, output_dim=64))
```

```
model.add(layers.LSTM(128))
```

```
model.add(layers.Dense(10))
```

```
model.summary()
```

Output:

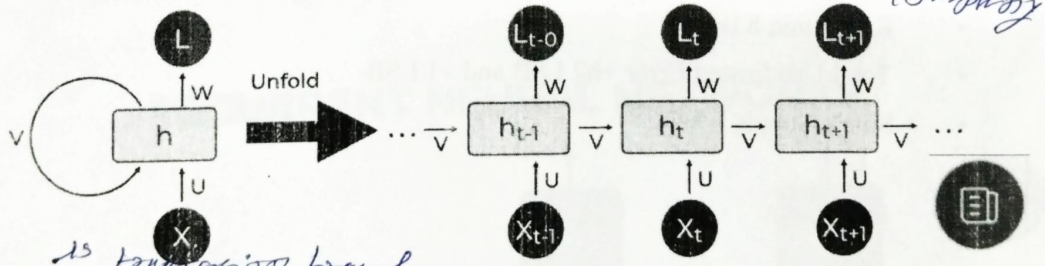
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, None, 64)          64000
_________________________________________________________________
lstm (LSTM)                  (None, 128)               98816
_________________________________________________________________
dense (Dense)                (None, 10)                1290
=================================================================
Total params: 164,106
```

Introduction to Recurrent Neural Network

In this article, we will introduce a new variation of neural network which is the **Recurrent Neural Network** also known as **(RNN)** that works better than a simple neural network when data is sequential like Time-Series data and text data.

What is Recurrent Neural Network (RNN)?

Recurrent Neural Network(RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is its **Hidden state**, which remembers some information about a sequence. The state is also referred to as *Memory State* since it remembers the previous input to the network. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.



REcurrent neural network

Architecture Of Recurrent Neural Network

RNNs have the same input and output architecture as any other deep neural architecture. However, differences arise in the way information flows from input to output. Unlike Deep neural networks where we have different weight matrices for each Dense network in RNN, the weight across the network remains the same. It calculates state hidden state H_i for every input X_i. **By using the following formulas:**

$$h = \sigma(UX + Wh_{-1} + B)$$

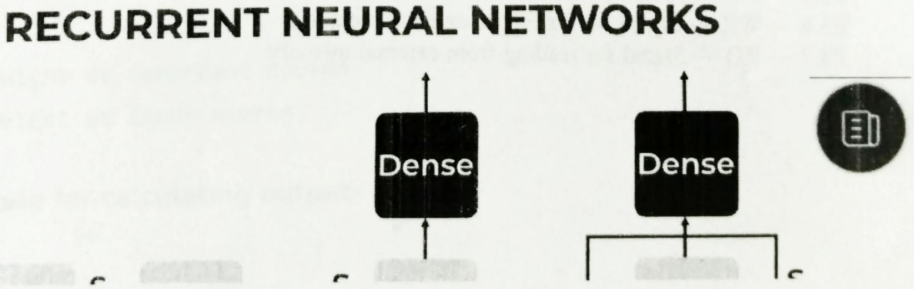
Check out our *Complete Machine Learning & Data Science*

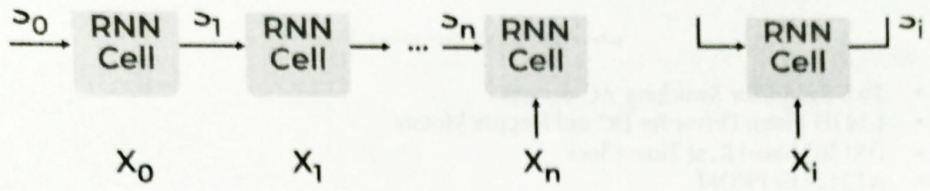
$$Y = O(Vh + C) \text{ Hence}$$

$$Y = f(X, h, W, U, V, B, C)$$

Here S is the State matrix which has element si as the state of the network at timestep i

The parameters in the network are W, U, V, c, b which are shared across timestep





What is Recurrent Neural Network

How RNN works

The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit. This hidden state signifies the past knowledge that the network currently holds at a given time step. This hidden state is updated at every time step to signify the change in the knowledge of the network about the past. The hidden state is updated using the following recurrence relation:-

The formula for calculating the current state:

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

where:

h_t -> current state
h_{t-1} -> previous state
x_t -> input state

Formula for applying Activation function(tanh):

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

where:

W_{hh} -> weight at recurrent neuron
W_{xh} -> weight at input neuron

The formula for calculating output:

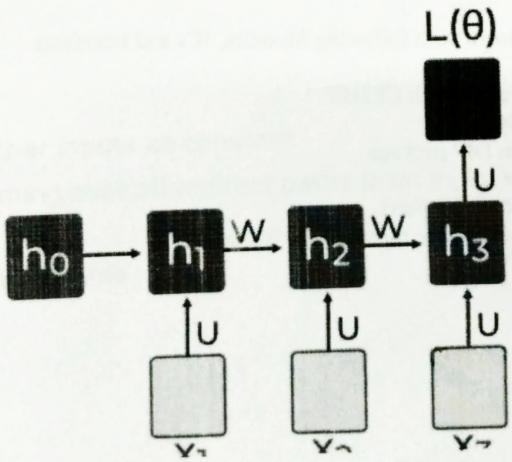
$$y_t = W_{hy}h_t$$

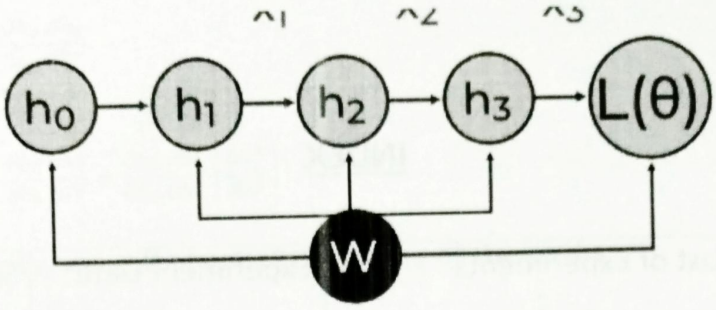
Y_t -> output
W_{hy} -> weight at output layer

These parameters are updated using Backpropagation. However, since RNN works on sequential data here we use an updated backpropagation which is known as Backpropagation through time.

Backpropagation Through Time (BPTT)

In RNN the neural network is in an ordered fashion and since in the ordered network each variable is computed one at a time in a specified order like first h1 then h2 then h3 so on. Hence we will apply backpropagation throughout all these hidden time states sequentially.





Backpropagation Through Time (BPTT) In RNN

```
L(θ)(loss function) depends on h3
h3 in turn depends on h2 and W
h2 in turn depends on h1 and W
h1 in turn depends on h0 and W
where h0 is a constant starting state.
```

$$\frac{\partial L(\theta)}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathbf{L}(\theta)}{\partial W}$$

For simplicity of this equation, we will apply backpropagation on only one row

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \frac{\partial h_3}{\partial W}$$

We already know how to compute this one as it is the same as any simple deep neural network backpropagation. $\frac{\partial L(\theta)}{\partial h_3}$ .However, we will see how to apply backpropagation to this term $\frac{\partial h_3}{\partial W}$

```
As we know h3 = σ(Wh2 + b)
```

And In such an ordered network, we can't compute $\frac{\partial h_3}{\partial W}$ by simply treating h3 as a constant because as it also depends on W. the total derivative $\frac{\partial h_3}{\partial W}$ has two parts

1. **Explicit:**
   $\frac{\partial h_3 +}{\partial W}$
   treating all other inputs as constant
2. **Implicit:** Summing over all indirect paths from h_3 to W

Let us see how to do this

$$\frac{\partial h_3}{\partial W} = \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial W}$$
$$= \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\left[\frac{\partial h_2^+}{\partial W} + \frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W}\right]$$
$$= \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\left[\frac{\partial h_1^+}{\partial W}\right]$$

For simplicity, we will short-circuit some of the paths

$$\frac{\partial h_3}{\partial W} = \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2^+}{\partial W} + \frac{\partial h_3}{\partial h_1}\frac{\partial h_1^+}{\partial W}$$

Finally, we have

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \cdot \frac{\partial h_3}{\partial W}$$

Where

$$\frac{\partial h_3}{\partial W} = \sum_{k=1}^{3} \frac{\partial h_3}{\partial h_k} \cdot \frac{\partial h_k}{\partial W}$$

Hence,

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \sum_{k=1}^{3} \frac{\partial h_3}{\partial h_k} \cdot \frac{\partial h_k}{\partial W}$$

This algorithm is called backpropagation through time (BPTT) as we backpropagate over all previous time steps

Training through RNN

1. A single-time step of the input is provided to the network.
2. Then calculate its current state using a set of current input and the previous state.
3. The current ht becomes ht-1 for the next time step.
4. One can go as many time steps according to the problem and join the information from all the previous states.
5. Once all the time steps are completed the final current state is used to calculate the output.
6. The output is then compared to the actual output i.e the target output and the error is generated.
7. The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained using Backpropagation through time.

Advantages of Recurrent Neural Network

1. An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.
2. Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.

Disadvantages of Recurrent Neural Network

1. Gradient vanishing and exploding problems.
2. Training an RNN is a very difficult task.
3. It cannot process very long sequences if using tanh or relu as an activation function.

Applications of Recurrent Neural Network

1. Language Modelling and Generating Text
2. Speech Recognition
3. Machine Translation
4. Image Recognition, Face detection
5. Time series Forecasting

Types Of RNN

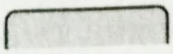
There are four types of RNNs based on the number of inputs and outputs in the network.

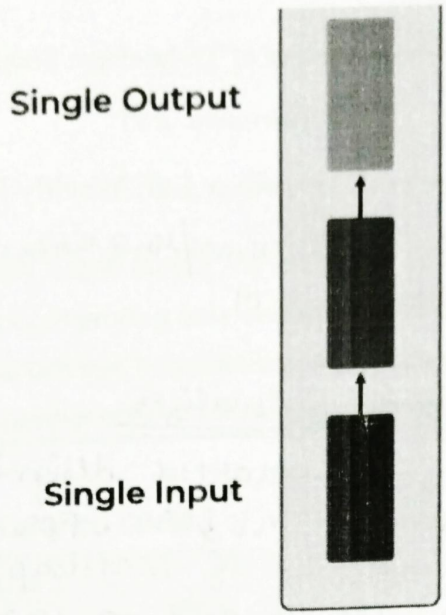
1. One to One
2. One to Many
3. Many to One
4. Many to Many

One to One

This type of RNN behaves the same as any simple Neural network it is also known as Vanilla Neural Network. In this Neural network, there is only one input and one output.

One to One

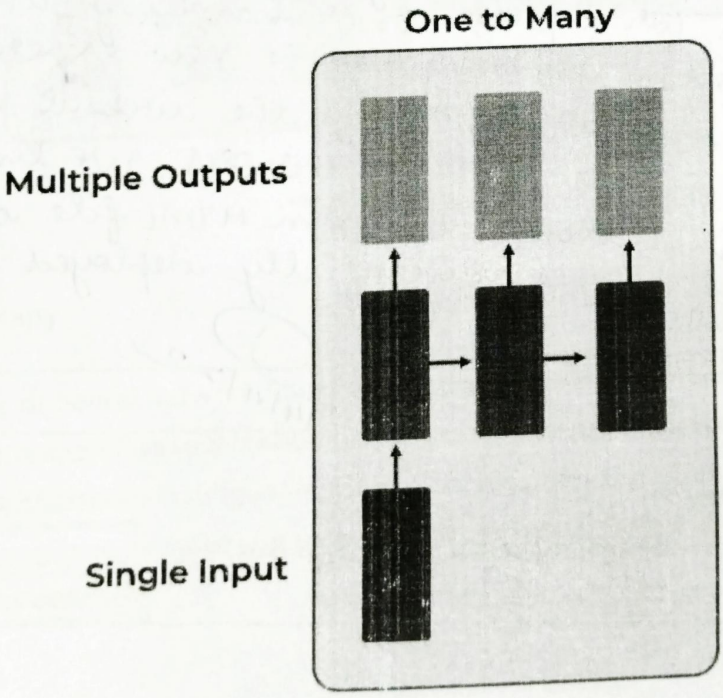




One to One RNN

One To Many

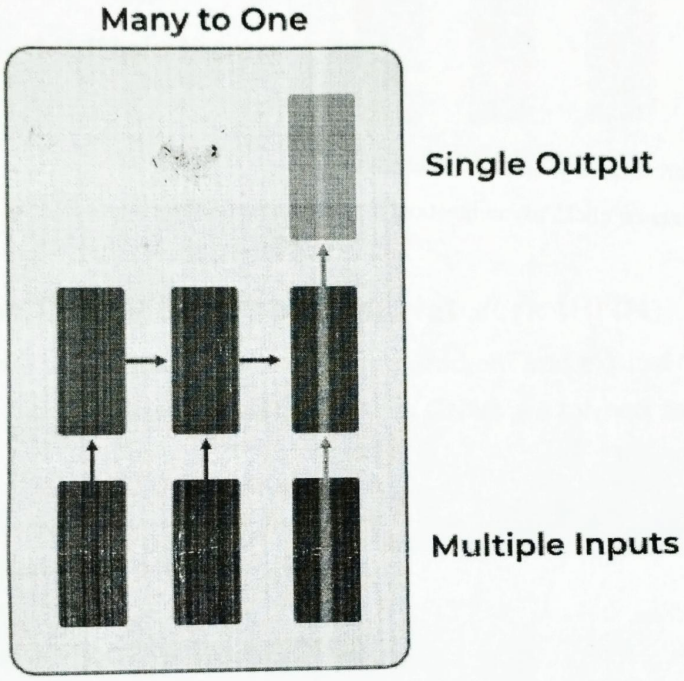
In this type of RNN, there is one input and many outputs associated with it. One of the most used examples of this network is Image captioning where given an image we predict a sentence having Multiple words.



One To Many RNN

Many to One

In this type of network, Many inputs are fed to the network at several states of the network generating only one output. This type of network is used in the problems like sentimental analysis. Where we give multiple words as input and predict only the sentiment of the sentence as output.



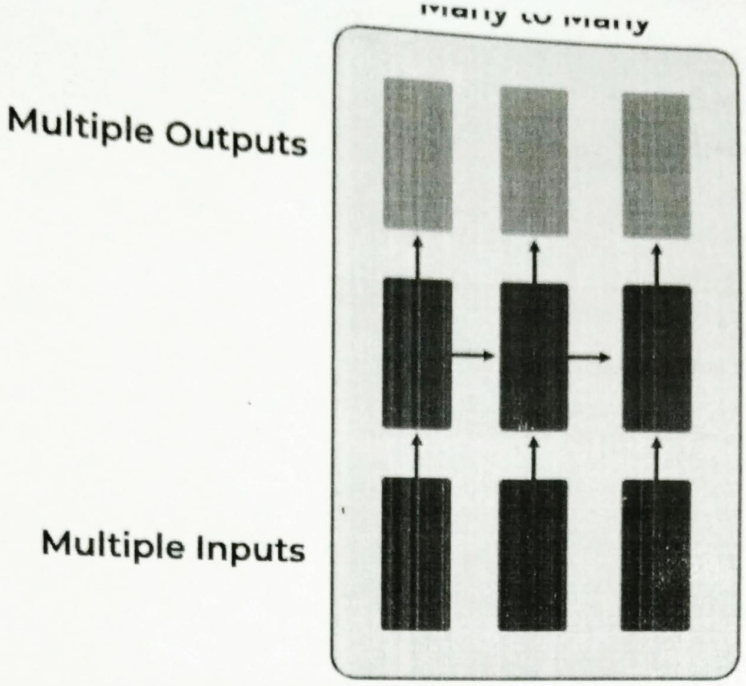
Many to One RNN

Many to Many

In this type of neural network, there are multiple inputs and multiple outputs corresponding to a problem. One Example of this Problem will be language translation. In language translation, we provide multiple words from one language as input and predict multiple words from the second language as output.

Google

Many to Many



Many to Many RNN

Variation Of Recurrent Neural Network (RNN)

To overcome the problems like vanishing gradient and exploding gradient descent several new advanced versions of RNNs are formed some of these are as ;

1. Bidirectional Neural Network (BiNN)
2. Long Short-Term Memory (LSTM)

Bidirectional Neural Network (BiNN)

A BiNN is a variation of a Recurrent Neural Network in which the input information flows in both direction and then the output of both direction are combined to produce the input. BiNN is useful in situations when the context of the input is more important such as Nlp tasks and Time-series analysis problems.

Long Short-Term Memory (LSTM)

Long Short-Term Memory works on the read-write-and-forget principle where given the input information network reads and writes the most useful information from the data and it forgets about the information which is not important in predicting the output. For doing this three new

gates are introduced in the RNN. In this way, only the selected information is passed through the network.

Difference between RNN and Simple Neural Network

RNN is considered to be the better version of deep neural when the data is sequential. There are significant differences between the RNN and deep neural networks they are listed as:

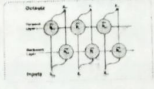
Recurrent Neural Network	Deep Neural Network
Weights are same across all the layers number of a Recurrent Neural Network	Weights are different for each layer of the network
Recurrent Neural Networks are used when the data is sequential and the number of inputs is not predefined.	A Simple Deep Neural network does not have any special method for sequential data also here the the number of inputs is fixed
The Numbers of parameter in the RNN are higher than in simple DNN	The Numbers of Parameter are lower than RNN
Exploding and vanishing gradients is the the major drawback of RNN	These problems also occur in DNN but these are not the major problem with DNN

Last Updated : 18 May, 2023

82

Similar Reads

Difference Between Feed-Forward Neural Networks and Recurrent Neural...



Bidirectional Recurrent Neural Network

Recurrent Neural Networks Explanation

Types of Recurrent Neural Networks (RNN) in Tensorflow

What is LSTM?

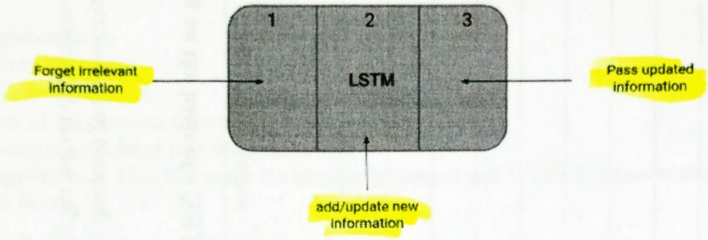
LSTM (Long Short-Term Memory) is a recurrent neural network (RNN) architecture widely used in Deep Learning. It excels at capturing long-term dependencies, making it ideal for sequence prediction tasks.

Unlike traditional neural networks, LSTM incorporates feedback connections, allowing it to process entire sequences of data, not just individual data points. This makes it highly effective in understanding and predicting patterns in sequential data like time series, text, and speech.

LSTM has become a powerful tool in artificial intelligence and deep learning, enabling breakthroughs in various fields by uncovering valuable insights from sequential data.

LSTM Architecture

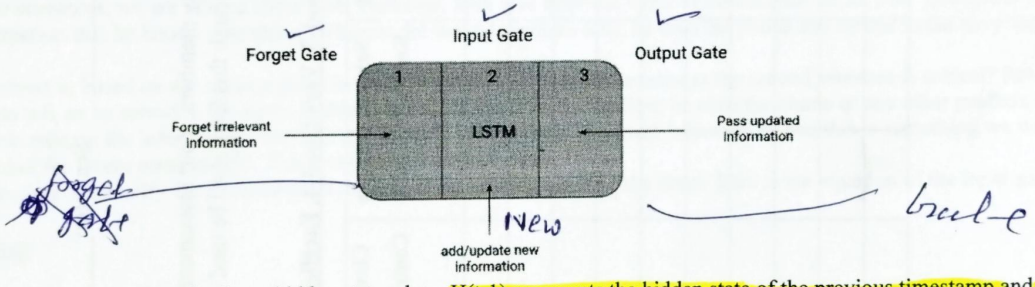
In the introduction to long short-term memory, we learned that it resolves the vanishing gradient problem faced by RNN, so now, in this section, we will see how it resolves this problem by learning the architecture of the LSTM. At a high level, LSTM works very much like an RNN cell. Here is the internal functioning of the LSTM network. The LSTM network architecture consists of three parts, as shown in the image below, and each part performs an individual function.



The Logic Behind LSTM

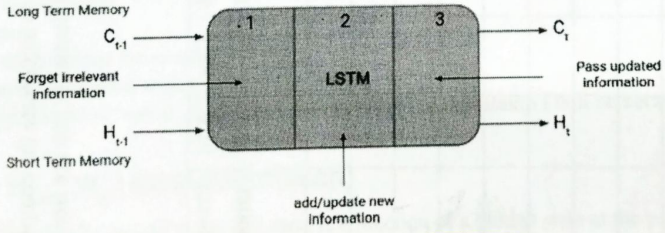
The first part chooses whether the information coming from the previous timestamp is to be remembered or is irrelevant and can be forgotten. In the second part, the cell tries to learn new information from the input to this cell. At last, in the third part, the cell passes the updated information from the current timestamp to the next timestamp. This one cycle of LSTM is considered a single-time step.

These three parts of an LSTM unit are known as gates. They control the flow of information in and out of the memory cell or lstm cell. The first gate is called **Forget gate, the** second gate is known as **the Input gate,** and the last one is **the Output gate**. An LSTM unit that consists of these three gates and a memory cell or lstm cell can be considered as a layer of neurons in traditional feedforward neural network, with each neuron having a hidden layer and a current state.

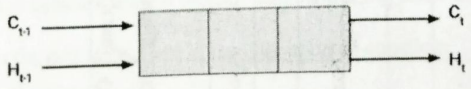


Just like a simple RNN, an LSTM also has a hidden state where H(t-1) represents the hidden state of the previous timestamp and Ht is the hidden state of the current timestamp. In addition to that, LSTM also has a cell state represented by C(t-1) and C(t) for the previous and current timestamps, respectively.

Here the hidden state is known as Short term memory, and the cell state is known as Long term memory. Refer to the following image.



It is interesting to note that the cell state carries the information along with all the timestamps.



LSTM

Example of LTSM Working

Let's take an example to understand how LSTM works. Here we have two sentences separated by a full stop. The first sentence is "Bob is a nice person," and the second sentence is "Dan, on the Other hand, is evil". It is very clear, in the first sentence, we are talking about Bob, and as soon as we encounter the full stop(.), we started talking about Dan.

As we move from the first sentence to the second sentence, our network should realize that we are no more talking about Bob. Now our subject is Dan. Here, the Forget gate of the network allows it to forget about it. Let's understand the roles played by these gates in LSTM architecture.

Forget Gate

In a cell of the LSTM neural network, the first step is to decide whether we should keep the information from the previous time step or forget it. Here is the equation for forget gate.

Forget Gate:

- $f_t = \sigma (x_t * U_f + H_{t-1} * W_f)$

Let's try to understand the equation, here

- Xt: input to the current timestamp.
- Uf: weight associated with the input
- Ht-1: The hidden state of the previous timestamp
- Wf: It is the weight matrix associated with the hidden state

Later, a sigmoid function is applied to it. That will make ft a number between 0 and 1. This ft is later multiplied with the cell state of the previous timestamp, as shown below.

$C_{t-1} * f_t = 0$ ...if $f_t = 0$ (forget everything)

$C_{t-1} * f_t = C_{t-1}$ ...if $f_t = 1$ (forget nothing)

Input Gate

Let's take another example.

"Bob knows swimming. He told me over the phone that he had served the navy for four long years."

So, in both these sentences, we are talking about Bob. However, both give different kinds of information about Bob. In the first sentence, we get the information that he knows swimming. Whereas the second sentence tells, he uses the phone and served in the navy for four years.

Now just think about it, based on the context given in the first sentence, which information in the second sentence is critical? First, he used the phone to tell, or he served in the navy. In this context, it doesn't matter whether he used the phone or any other medium of communication to pass on the information. The fact that he was in the navy is important information, and this is something we want our model to remember for future computation. This is the task of the Input gate.

The input gate is used to quantify the importance of the new information carried by the input. Here is the equation of the input gate

Input Gate:

- $i_t = \sigma (x_t * U_i + H_{t-1} * W_i)$

Here,

- Xt: Input at the current timestamp t
- Ui: weight matrix of input
- Ht-1: A hidden state at the previous timestamp
- Wi: Weight matrix of input associated with hidden state

Again we have applied the sigmoid function over it. As a result, the value of I at timestamp t will be between 0 and 1.

New Information

- $N_t = \tanh(x_t * U_c + H_{t-1} * W_c)$ (new information)

Now the new information that needed to be passed to the cell state is a function of a hidden state at the previous timestamp t-1 and input x at timestamp t. The activation function here is tanh. Due to the tanh function, the value of new information will be between -1 and 1. If the value of Nt is negative, the information is subtracted from the cell state, and if the value is positive, the information is added to the cell state at the current timestamp.

However, the Nt won't be added directly to the cell state. Here comes the updated equation:

$C_t = f_t * C_{t-1} + i_t * N_t$ (updating cell state)

Here, Ct-1 is the cell state at the current timestamp, and the others are the values we have calculated previously.

Output Gate

Now consider this sentence.

"Bob single-handedly fought the enemy and died for his country. For his contributions, brave_____."

During this task, we have to complete the second sentence. Now, the minute we see the word brave, we know that we are talking about a person. In the sentence, only Bob is brave, we can not say the enemy is brave, or the country is brave. So based on the current expectation, we have to give a relevant word to fill in the blank. That word is our output, and this is the function of our Output gate. Here is the equation of the Output gate, which is pretty similar to the two previous gates.

Output Gate:

- $o_t = \sigma(x_t * U_o + H_{t-1} * W_o)$

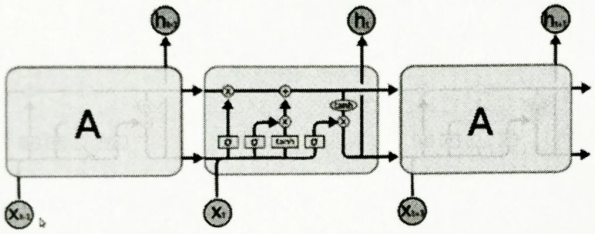
Its value will also lie between 0 and 1 because of this sigmoid function. Now to calculate the current hidden state, we will use Ot and tanh of the updated cell state. As shown below.

$$H_t = o_t * \tanh(C_t)$$

It turns out that the hidden state is a function of Long term memory (Ct) and the current output. If you need to take the output of the current timestamp, just apply the SoftMax activation on hidden state Ht.

$$\text{Output} = \text{Softmax}(H_t)$$

Here the token with the maximum score in the output is the prediction.
This is the More intuitive diagram of the LSTM network.



This diagram is taken from an interesting blog. I urge you all to go through it. Here is the link: *Understanding LSTM Networks*

LTSM vs RNN

Aspect	LSTM (Long Short-Term Memory)	RNN (Recurrent Neural Network)
Architecture	A type of RNN with additional memory cells	A basic type of RNN
Memory Retention	Handles long-term dependencies and prevents vanishing gradient problem	Struggles with long-term dependencies and vanishing gradient problem
Cell Structure	Complex cell structure with input, output, and forget gates	Simple cell structure with only hidden state
Handling Sequences	Suitable for processing sequential data	Also designed for sequential data, but limited memory
Training Efficiency	Slower training process due to increased complexity	Faster training process due to simpler architecture
Performance on Long Sequences	Performs better on long sequences	Struggles to retain information on long sequences
Usage	Best suited for tasks requiring long-term memory, such as language translation and sentiment analysis	Appropriate for simple sequential tasks, such as time series forecasting
Vanishing Gradient Problem	Addresses the vanishing gradient problem	Prone to the vanishing gradient problem

Types of Boltzmann Machines

Deep Learning models are broadly classified into supervised and unsupervised models.

Supervised DL models:

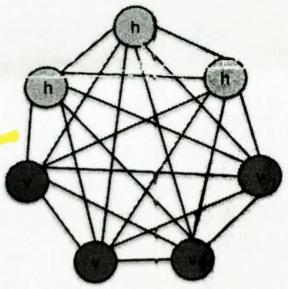
- Artificial Neural Networks (ANNs)
- Recurrent Neural Networks (RNNs)
- Convolutional Neural Networks (CNNs)

Unsupervised DL models:

- Self Organizing Maps (SOMs)
- Boltzmann Machines
- Autoencoders

Let us learn what exactly Boltzmann machines are, how they work and also implement a recommender system which recommends whether the user likes a movie or not based on the previous movies watched.

Boltzmann Machines is an unsupervised DL model in which every node is connected to every other node. That is, unlike the ANNs, CNNs, RNNs and SOMs, the Boltzmann Machines are **undirected** (or the connections are bidirectional). Boltzmann Machine is not a deterministic DL model but a **stochastic** or **generative** DL model. It is rather a representation of a certain system. There are two types of nodes in the Boltzmann Machine — **Visible nodes** — those nodes which we can and do measure, and the **Hidden nodes** – those nodes which we cannot or do not measure. Although the node types are different, the Boltzmann machine considers them as the same and everything works as one single system. The training data is fed into the Boltzmann Machine and the weights of the system are adjusted accordingly. Boltzmann machines help us understand abnormalities by learning about the working of the system in normal conditions. .



v - visible nodes, h - hidden nodes

Energy-Based Models:

Boltzmann Distribution is used in the sampling distribution of the Boltzmann Machine. The Boltzmann distribution is governed by the equation –

$P_i = e^{(-\epsilon_i/kT)} / \sum e^{(-\epsilon_j/kT)}$

P_i - probability of system being in state i
ϵ_i - Energy of system in state i
T - Temperature of the system
k - Boltzmann constant
$\sum e^{(-\epsilon_j/kT)}$ - Sum of values for all possible states of the system

Boltzmann Distribution describes different states of the system and thus Boltzmann machines create different states of the machine using this distribution. From the above equation, as the energy of system increases, the probability for the system to be in state 'i' decreases. Thus, the system is the most stable in its lowest energy state

gas is most stable when it spreads). Here, in Boltzmann machines, the energy of the system is defined in terms of the **weights of synapses**. Once the system is trained and the weights are set, the system always tries to find the lowest energy state for itself by adjusting the weights.

Types of Boltzmann Machines:

- Restricted Boltzmann Machines (RBMs)
- Deep Belief Networks (DBNs)
- Deep Boltzmann Machines (DBMs)

Restricted Boltzmann Machines (RBMs):

In a full Boltzmann machine, each node is connected to every other node and hence the connections grow **exponentially**. This is the reason we use RBMs. The restrictions in the node connections in RBMs are as follows

- Hidden nodes cannot be connected to one another.
- Visible nodes connected to one another.

Energy function example for Restricted Boltzmann Machine –

$E(v, h) = -\sum a_{ivi} - \sum b_{jhj} - \sum\sum v_{iwi,jhj}$

a, v - biases in the system - constants

$v_{i,hj}$ - visible node, hidden node

P(v, h) = Probability of being in a certain state

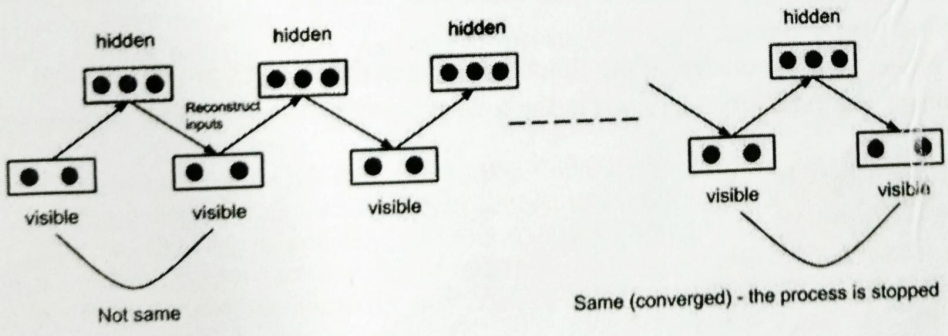
$P(v, h) = e^{(-E(v, h))/Z}$

Z - sum if values for all possible states

Suppose that we are using our RBM for building a recommender system that works on six (6) movies. RBM learns how to allocate the hidden nodes to certain features. By the process of **Contrastive Divergence**, we make the RBM close to our set of movies that is our case or scenario. RBM identifies which features are important by the training process. The training data is either 0 or 1 or missing data based on whether a user liked that movie (1), disliked that movie (0) or did not watch the movie (missing data). RBM automatically identifies important features.

Contrastive Divergence:

RBM adjusts its weights by this method. Using some randomly assigned initial weights, RBM calculates the hidden nodes, which in turn use the same weights to reconstruct the input nodes. Each hidden node is constructed from all the visible nodes and each visible node is reconstructed from all the hidden node and hence, the input is different from the reconstructed input, though the weights are the same. The process continues until the reconstructed input matches the previous input. The process is said to be converged at this stage. This entire procedure is known as **Gibbs Sampling**.



Gibb's Sampling

gas is most stable when it spreads). Here, in Boltzmann machines, the energy of the system is defined in terms of the **weights of synapses**. Once the system is trained and the weights are set, the system always tries to find the lowest energy state for itself by adjusting the weights.

Types of Boltzmann Machines:

- Restricted Boltzmann Machines (RBMs)
- Deep Belief Networks (DBNs)
- Deep Boltzmann Machines (DBMs)

Restricted Boltzmann Machines (RBMs):

In a full Boltzmann machine, each node is connected to every other node and hence the connections grow **exponentially**. This is the reason we use RBMs. The restrictions in the node connections in RBMs are as follows

- Hidden nodes cannot be connected to one another.
- Visible nodes connected to one another.

Energy function example for Restricted Boltzmann Machine –

$E(v, h) = -\sum a_{ivi} - \sum b_{jhj} - \sum\sum v_{iwi,jhj}$

a, v - biases in the system - constants

v_i, h_j - visible node, hidden node

P(v, h) = Probability of being in a certain state

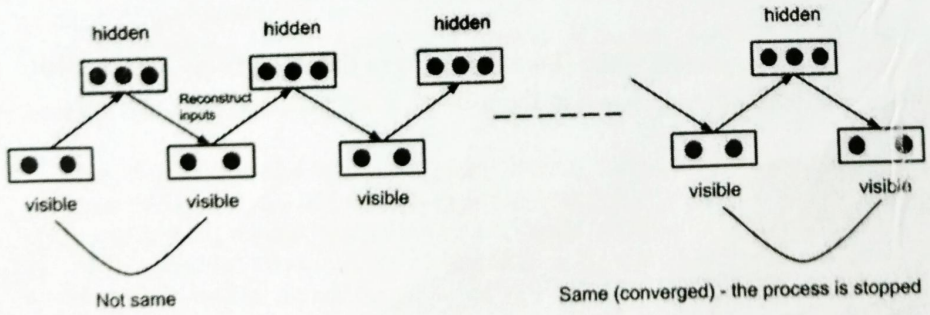
$P(v, h) = e^{(-E(v, h))/Z}$

Z - sum if values for all possible states

Suppose that we are using our RBM for building a recommender system that works on six (6) movies. RBM learns how to allocate the hidden nodes to certain features. By the process of **Contrastive Divergence**, we make the RBM close to our set of movies that is our case or scenario. RBM identifies which features are important by the training process. The training data is either 0 or 1 or missing data based on whether a user liked that movie (1), disliked that movie (0) or did not watch the movie (missing data). RBM automatically identifies important features.

Contrastive Divergence:

RBM adjusts its weights by this method. Using some randomly assigned initial weights, RBM calculates the hidden nodes, which in turn use the same weights to reconstruct the input nodes. Each hidden node is constructed from all the visible nodes and each visible node is reconstructed from all the hidden node and hence, the input is different from the reconstructed input, though the weights are the same. The process continues until the reconstructed input matches the previous input. The process is said to be converged at this stage. This entire procedure is known as **Gibbs Sampling**.



Gibb's Sampling

Gradient Formula gives the gradient of the log probability of the certain state of the system with respect to the weights of the system. It is given as follows –

$$d/dw_{ij}(\log(P(v^{0}))) = \langle vi0 * hj0 \rangle - \langle vi\infty * hj\infty \rangle$$

v - visible state, h- hidden state

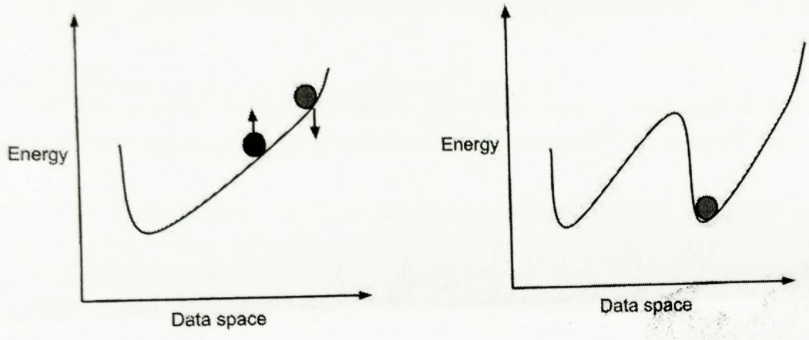
$\langle v_i^0 * h_j^0 \rangle$ - initial state of the system

$\langle v_i^\infty * h_j^\infty \rangle$ - final state of the system

$P(v^0)$ - probability that the system is in state v^0

w_{ij} - weights of the system

The above equations tell us – how the change in weights of the system will change the log probability of the system to be a particular state. The system tries to end up in the lowest possible energy state (most stable). Instead of continuing the adjusting of weights process until the current input matches the previous one, we can also consider the first few pauses only. It is sufficient to understand how to adjust our curve so as to get the lowest energy state. Therefore, we adjust the weights, redesign the system and energy curve such that we get the lowest energy for the current position. This is known as the **Hinton's shortcut**.



Hinton's Shortcut

Working of RBM – Illustrative Example –

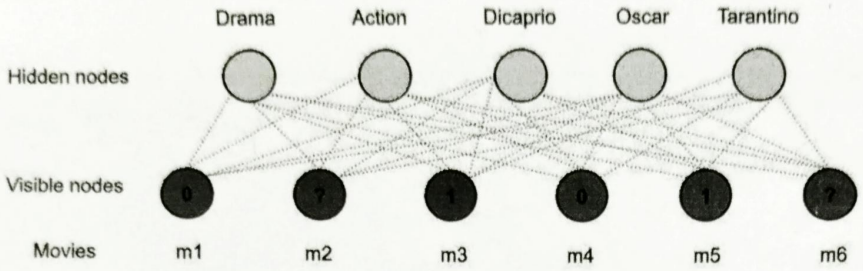
Consider – Mary watches four movies out of the six available movies and rates four of them. Say, she watched m_1, m_3, m_4 and m_5 and likes m_3, m_5 (rated 1) and dislikes the other two, that is m_1, m_4 (rated 0) whereas the other two movies – m2, m6 are unrated. Now, using our RBM, we will recommend one of these movies for her to watch next. Say –

- m_3, m_5 are of 'Drama' genre.
- m_1, m_4 are of 'Action' genre.
- 'Dicaprio' played a role in m_5.
- m_3, m_5 have won 'Oscar.'
- 'Tarantino' directed m_4.
- m_2 is of the 'Action' genre.
- m_6 is of both the genres 'Action' and 'Drama', 'Dicaprio' acted in it and it has won an 'Oscar'.

We have the following observations –

- Mary likes m_3, m_5 and they are of genre 'Drama,' she probably **likes 'Drama'** movies.
- Mary dislikes m_1, m_4 and they are of action genre, she probably **dislikes 'Action'** movies.
- Mary likes m_3, m_5 and they have won an 'Oscar', she probably **likes** an **'Oscar'** movie.
- Since 'Dicaprio' acted in m_5 and Mary likes it, she will probably **like** a movie in which **'Dicaprio'** acted.
- Mary does not like m_4 which is directed by Tarantino, she probably **dislikes** any movie directed by **'Tarantino'**.

Therefore, based on the observations and the details of m_2, m_6; our RBM **recommends m6** to Mary ('Drama', 'Dicaprio' and 'Oscar' matches both Mary's interests and m_6). This is how an RBM works and hence is used in recommender systems.



Working of RBM

Thus, RBMs are used to build Recommender Systems.

Deep Belief Networks (DBNs):

Suppose we stack several RBMs on top of each other so that the first RBM outputs are the input to the second RBM and so on. Such networks are known as Deep Belief Networks. The connections within each layer are undirected (since each layer is an RBM). Simultaneously, those in between the layers are directed (except the top two layers – the connection between the top two layers is undirected). There are two ways to train the DBNs-

1. **Greedy Layer-wise Training Algorithm –** The RBMs are trained layer by layer. Once the individual RBMs are trained (that is, the parameters – weights, biases are set), the direction is set up between the DBN layers.
2. **Wake-Sleep Algorithm –** The DBN is trained all the way up (connections going up – wake) and then down the network (connections going down — sleep).

Therefore, we stack the RBMs, train them, and once we have the parameters trained, we make sure that the connectic between the layers only work downwards (except for the top two layers).

Deep Boltzmann Machines (DBMs):

DBMs are similar to DBNs except that apart from the connections within layers, the connections between the layers are also **undirected** (unlike DBN in which the connections between layers are directed). DBMs can extract more complex or sophisticated features and hence can be used for more complex tasks.