

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

Autonomous Institution – UGC, Govt. of India



Department of Artificial Intelligence and Machine Learning

B. TECH (R-22 Regulation)

(IV YEAR – I SEM)

2025-26

DEEP LEARNING

(R22A6605)



LECTURE NOTES

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12(B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE-Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad-500100, Telangana State, India

Department of Computer Science and Engineering

(Artificial Intelligence and Machine Learning)

Vision

To be a premier center for academic excellence and research through innovative interdisciplinary collaborations and making significant contributions to the community, organizations, and society as a whole.

Mission

- ❖ To impart cutting-edge Artificial Intelligence technology in accordance with industry norms.
- ❖ To instil in students a desire to conduct research in order to tackle challenging technical problems for industry by sustaining the ethical values.
- ❖ To develop effective graduates who are responsible for their professional growth, leadership qualities and are committed to lifelong learning.

QUALITY POLICY

- ❖ To provide sophisticated technical infrastructure and to inspire students to reach their full potential.
- ❖ To provide students with a solid academic and research environment for a comprehensive learning experience.
- ❖ To provide research development, consulting, testing, and customized training to satisfy specific industrial demands, thereby encouraging self-employment and entrepreneurship among students.

For more information: www.mrcet.ac.in

SYLLABUS

**M R C E T CAMPUS | AUTONOMOUS INSTITUTION - UGC, GOVT.
OF INDIA**

IV Year B. Tech CSE (AIML)- I Sem

L/T/P/C 3/-/-/3

(R20A6610) DEEP LEARNING

COURSE OBJECTIVES:

1. To understand the basic concepts and techniques of Deep Learning and the need of Deep Learning techniques in real-world problems
2. To understand CNN algorithms and the way to evaluate performance of the CNN architectures.
3. To apply RNN and LSTM to learn, predict and classify the real-world problems in the paradigms of Deep Learning.
4. To understand, learn and design GANs for the selected problems.
5. To understand the concept of Auto-encoders and enhancing GANs using auto-encoders.

UNIT-I:

INTRODUCTION TO DEEP LEARNING: Historical Trends in Deep Learning, Why DL is Growing, Artificial Neural Network, Non-linear classification example using Neural Networks: XOR/XNOR, Single/Multiple Layer Perceptron, Feed Forward Network, Deep Feed- forward networks, Stochastic Gradient –Based learning, Hidden Units, Architecture Design, Back- Propagation.

UNIT-II:

CONVOLUTION NEURAL NETWORK (CNN): Introduction to CNNs and their applications in computer vision, CNN basic architecture, Activation functions- sigmoid, tanh, ReLU, Softmax layer, Types of pooling layers, Training of CNN in TensorFlow, various popular CNN architectures: VGG, Google Net, ResNet etc, Dropout, Normalization, Data augmentation

UNIT-III

RECURRENT NEURAL NETWORK (RNN): Introduction to RNNs and their applications in sequential data analysis, Back propagation through time (BPTT), Vanishing Gradient Problem, gradient clipping Long Short Term Memory (LSTM) Networks, Gated Recurrent Units, Bidirectional LSTMs, Bidirectional RNNs.

UNIT- IV

GENERATIVE ADVERSARIAL NETWORKS (GANs): Generative models, Concept and principles of GANs, Architecture of GANs (generator and discriminator networks), Comparison between discriminative and generative models, Generative Adversarial Networks (GANs), Applications of GANs

UNIT- V

AUTO-ENCODERS: Auto-encoders, Architecture and components of auto-encoders (encoder and decoder), Training an auto-encoder for data compression and reconstruction, Relationship between Autoencoders and GANs, Hybrid Models: Encoder-Decoder GANs.

TEXT BOOKS:

1. Deep Learning : An MIT Press Book by Ian Goodfellow and Yoshua Bengio Aaron Courville.
2. Michael Nielson, Neural Networks and Deep Learning, Determination Press, 2015.
3. Satish kumar, Neural networks: A classroom Approach, Tata McGraw-Hill Education, 2004

REFERENCES:

1. Deep Learning with Python, Francois Chollet, Manning publications 2018
2. Advanced Deep Learning with Keras, Rowel Atienza, PACKT Publications 2018

COURSE OUTCOMES:

CO1: Understand the basic concepts and techniques of Deep Learning and the need of Deep Learning techniques in real-world problems.

CO2: Understand CNN algorithms and the way to evaluate performance of the CNN architectures.

CO3: Apply RNN and LSTM to learn, predict and classify the real-world problems in the paradigms of Deep Learning.

CO4: Understand, learn and design GANs for the selected problems.

CO5: Understand the concept of Auto-encoders and enhancing GANs using auto-encoders.

UNIT-I:

INTRODUCTION TO DEEP LEARNING: Historical Trends in Deep Learning, Why DL is Growing, Artificial Neural Network, Non-linear classification example using Neural Networks: XOR/XNOR, Single/Multiple Layer Perceptron, Feed Forward Network, Deep Feed- forward networks, Stochastic Gradient –Based learning, Hidden Units, Architecture Design, Back- Propagation, Deep learning frameworks and libraries (e.g., TensorFlow/Keras, PyTorch).

INTRODUCTION TO DEEP LEARNING:

Deep learning is a branch of machine learning which is based on artificial neural networks. It is capable of learning complex patterns and relationships within data. In deep learning, we don't need to explicitly program everything. It has become increasingly popular in recent years due to the advances in processing power and the availability of large datasets. Because it is based on artificial neural networks (ANNs) also known as deep neural networks (DNNs). These neural networks are inspired by the structure and function of the human brain's biological neurons, and they are designed to learn from large amounts of data.

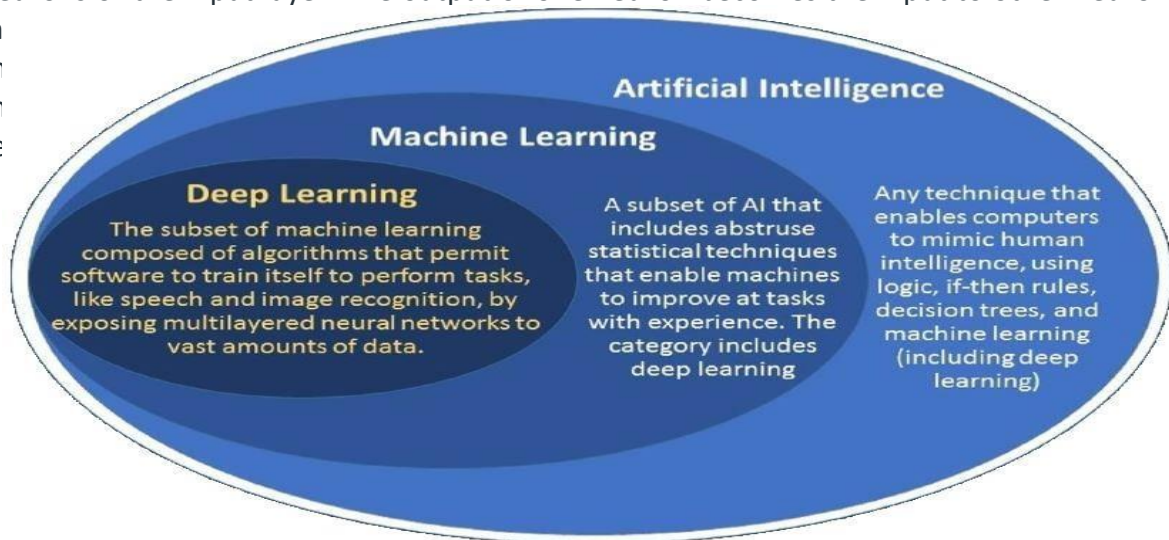
1. Deep Learning is a subfield of Machine Learning that involves the use of neural networks to model and solve complex problems. Neural networks are modeled after the structure and function of the human brain and consist of layers of interconnected nodes that process and transform data.
2. The key characteristic of Deep Learning is the use of deep neural networks, which have multiple layers of interconnected nodes. These networks can learn complex representations of data by discovering hierarchical patterns and features in the data. Deep Learning algorithms can automatically learn and improve from data without the need for manual feature engineering.
3. Deep Learning has achieved significant success in various fields, including image recognition, natural language processing, speech recognition, and recommendation systems. Some of the popular Deep Learning architectures include Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Deep Belief Networks (DBNs).
4. Training deep neural networks typically requires a large amount of data and computational resources. However, the availability of cloud computing and the development of specialized hardware, such as Graphics Processing Units (GPUs), has made it easier to train deep neural networks.

In summary, Deep Learning is a subfield of Machine Learning that involves the use of deep neural networks to model and solve complex problems. Deep Learning has achieved significant success in various fields, and its use is expected to continue to grow as more data becomes available, and more powerful computing resources become available.

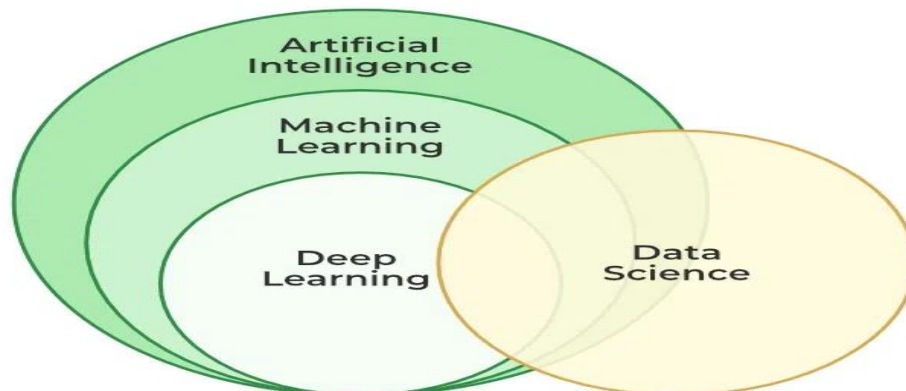
What is Deep Learning?

Deep learning is the branch of [machine learning](#) which is based on artificial neural network architecture. An artificial neural network or ANN uses layers of interconnected nodes called neurons that work together to process and learn from the input data.

In a fully connected Deep neural network, there is an input layer and one or more hidden layers connected one after the other. Each neuron receives input from the previous layer neurons or the input layer. The output of one neuron becomes the input to other neurons in the network.



Few Insights about AI, ML and DL



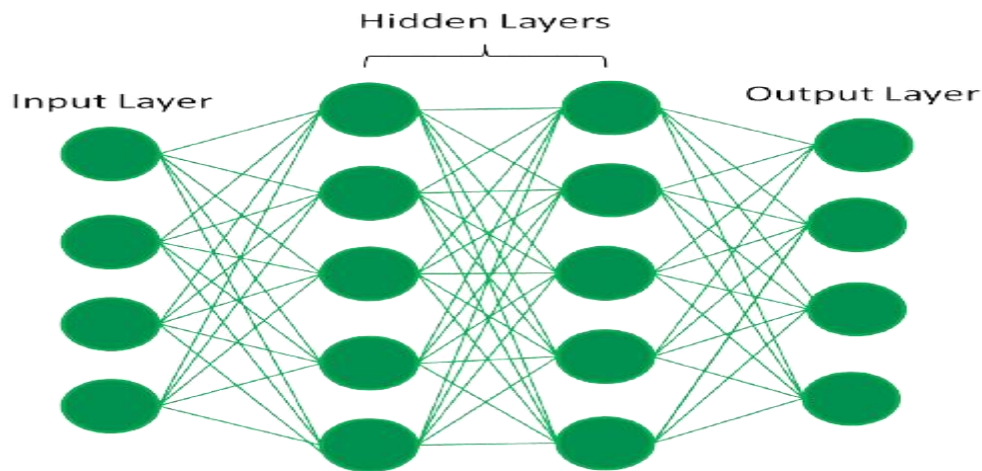
Today Deep learning has become one of the most popular and visible areas of machine learning, due to its success in a variety of applications, such as computer vision, natural language processing, and Reinforcement learning.

Deep learning can be used for supervised, unsupervised as well as reinforcement machine learning. it uses a variety of ways to process these.

- **Supervised Machine Learning:** [Supervised machine learning](#) is the [machine learning](#) technique in which the neural network learns to make predictions or classify data based on the labeled datasets. Here we input both input features along with the target variables. the neural network learns to make predictions based on the cost or error that comes from the difference between the predicted and the actual target, this process is known as backpropagation. Deep learning algorithms like Convolutional neural networks, Recurrent neural networks are used for many supervised tasks like image classifications and recognition, sentiment analysis, language translations, etc.
- **Unsupervised Machine Learning:** [Unsupervised machine learning](#) is the [machine learning](#) technique in which the neural network learns to discover the patterns or to cluster the dataset based on unlabeled datasets. Here there are no target variables. while the machine has to self-determined the hidden patterns or relationships within the datasets. Deep learning algorithms like autoencoders and generative models are used for unsupervised tasks like clustering, dimensionality reduction, and anomaly detection.
- **Reinforcement Machine Learning:** [Reinforcement Machine Learning](#) is the [machine learning](#) technique in which an agent learns to make decisions in an environment to maximize a reward signal. The agent interacts with the environment by taking action and observing the resulting rewards. Deep learning can be used to learn policies, or a set of actions, that maximizes the cumulative reward over time. Deep reinforcement learning algorithms like Deep Q networks and Deep Deterministic Policy Gradient (DDPG) are used to reinforce tasks like robotics and game playing etc.

Artificial neural networks:

[Artificial neural networks](#) are built on the principles of the structure and operation of human neurons. It is also known as neural networks or neural nets. An artificial neural network's input layer, which is the first layer, receives input from external sources and passes it on to the hidden layer, which is the second layer. Each neuron in the hidden layer gets information from the neurons in the previous layer, computes the weighted total, and then transfers it to the neurons in the next layer. These connections are weighted, which means that the impacts of the inputs from the preceding layer are more or less optimized by giving each input a distinct weight. These weights are then adjusted during the training process to enhance the performance of the model.



Fully Connected Artificial Neural Network

Artificial neurons, also known as units, are found in artificial neural networks. The whole Artificial Neural Network is composed of these artificial neurons, which are arranged in a series of layers. The complexities of neural networks will depend on the complexities of the underlying patterns in the dataset whether a layer has a dozen units or millions of units. Commonly, Artificial Neural Network has an input layer, an output layer as well as hidden layers. The input layer receives data from the outside world which the neural network needs to analyze or learn about.

In a fully connected artificial neural network, there is an input layer and one or more hidden layers connected one after the other. Each neuron receives input from the previous layer neurons or the input layer. The output of one neuron becomes the input to other neurons in the next layer of the network, and this process continues until the final layer produces the output of the network. Then, after passing through one or more hidden layers, this data is transformed into valuable data for the output layer. Finally, the output layer provides an output in the form of an artificial neural network's response to the data that comes in.

Units are linked to one another from one layer to another in the bulk of neural networks. Each of these links has weights that control how much one-unit influences another. The neural network learns more and more about the data as it moves from one unit to another, ultimately producing an output from the output layer.

Difference between Machine Learning and Deep Learning:

[Machine learning](#) and deep learning both are subsets of artificial intelligence but there are many similarities and differences between them.

Machine Learning	Deep Learning
------------------	---------------

Machine Learning	Deep Learning
Apply statistical algorithms to learn the hidden patterns and relationships in the dataset.	Uses artificial neural network architecture to learn the hidden patterns and relationships in the dataset.
Can work on the smaller amount of dataset	Requires the larger volume of dataset compared to machine learning
Better for the low-label task.	Better for complex task like image processing, natural language processing, etc.
Takes less time to train the model.	Takes more time to train the model.
A model is created by relevant features which are manually extracted from images to detect an object in the image.	Relevant features are automatically extracted from images. It is an end-to-end learning process.
Less complex and easy to interpret the result.	More complex, it works like the black box interpretations of the result are not easy.
It can work on the CPU or requires less computing power as compared to deep learning.	It requires a high-performance computer with GPU.

Types of neural networks:

Deep Learning models are able to automatically learn features from the data, which makes them well-suited for tasks such as image recognition, speech recognition, and natural language processing. The most widely used architectures in deep learning are

feedforward neural networks, convolutional neural networks (CNNs), and recurrent neural networks (RNNs).

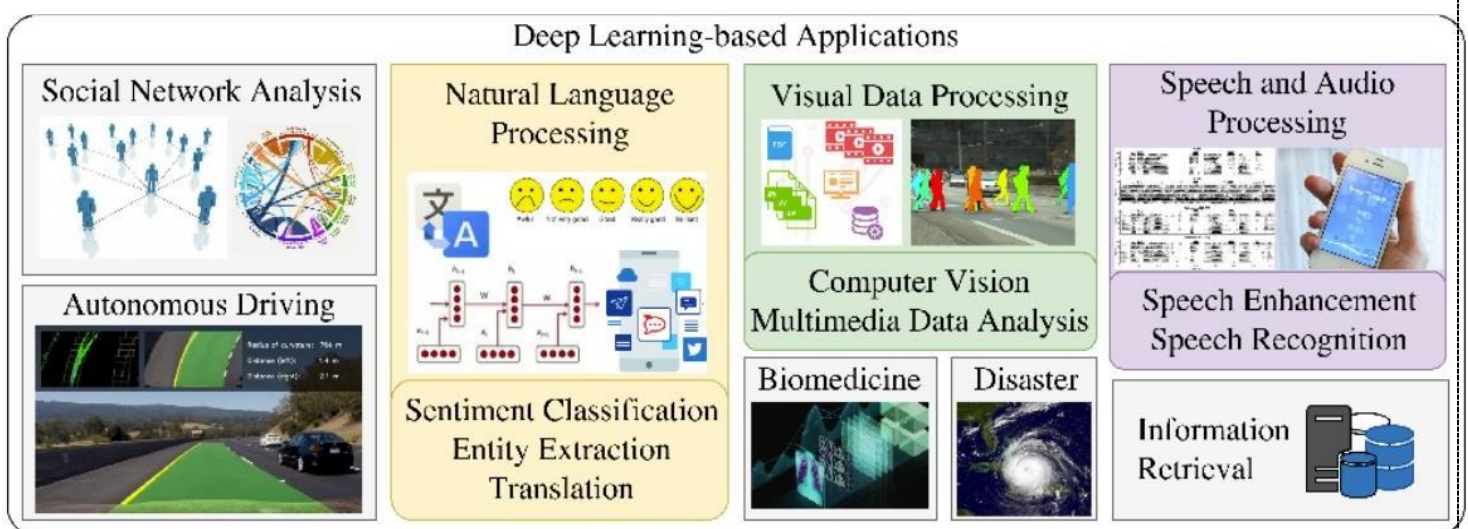
[Feedforward neural networks \(FNNs\)](#) are the simplest type of ANN, with a linear flow of information through the network. FNNs have been widely used for tasks such as image classification, speech recognition, and natural language processing.

[Convolutional Neural Networks \(CNNs\)](#) are specifically for image and video recognition tasks. CNNs are able to automatically learn features from the images, which makes them well-suited for tasks such as image classification, object detection, and image segmentation.

[Recurrent Neural Networks \(RNNs\)](#) are a type of neural network that is able to process sequential data, such as time series and natural language. RNNs are able to maintain an internal state that captures information about the previous inputs, which makes them well-suited for tasks such as speech recognition, natural language processing, and language translation.

Applications of Deep Learning :

The main applications of deep learning can be divided into computer vision, natural



language processing (NLP), and reinforcement learning.

Computer vision

In [computer vision](#), Deep learning models can enable machines to identify and understand visual data. Some of the main applications of deep learning in computer vision include:

- **Object detection and recognition:** Deep learning model can be used to identify and locate objects within images and videos, making it possible for machines to perform tasks such as self-driving cars, surveillance, and robotics.
- **Image classification:** Deep learning models can be used to classify images into categories such as animals, plants, and buildings. This is used in applications such as medical imaging, quality control, and image retrieval.
- **Image segmentation:** Deep learning models can be used for image segmentation into different regions, making it possible to identify specific features within images.

Natural language processing (NLP):

In NLP, the Deep learning model can enable machines to understand and generate human language. Some of the main applications of deep learning in NLP include:

- **Automatic Text Generation** – Deep learning model can learn the corpus of text and new text like summaries, essays can be automatically generated using these trained models.
- **Language translation:** Deep learning models can translate text from one language to another, making it possible to communicate with people from different linguistic backgrounds.
- **Sentiment analysis:** Deep learning models can analyze the sentiment of a piece of text, making it possible to determine whether the text is positive, negative, or neutral. This is used in applications such as customer service, social media monitoring, and political analysis.
- **Speech recognition:** Deep learning models can recognize and transcribe spoken words, making it possible to perform tasks such as speech-to-text conversion, voice search, and voice-controlled devices.






Reinforcement learning:

In reinforcement learning, deep learning works as training agents to take action in an environment to maximize a reward. Some of the main applications of deep learning in reinforcement learning include:

- **Game playing:** Deep reinforcement learning models have been able to beat human experts at games such as Go, Chess, and Atari.
- **Robotics:** Deep reinforcement learning models can be used to train robots to perform complex tasks such as grasping objects, navigation, and manipulation.
- **Control systems:** Deep reinforcement learning models can be used to control complex systems such as power grids, traffic management, and supply chain optimization.

Popular specific applications of DL:

5 Most Common Types of Deep Learning Vision Models

Classification	Segmentation	Object Detection
 <p>Identifies the concept of whole image and categorizes into different classes. (unit of interpretation: <u>image</u>)</p>	 <p>Recognizes an object, its shape and location within an image. (unit of interpretation: <u>pixel</u>)</p>	 <p>Distinguishes the class of each object and detect its location (unit of interpretation: <u>object</u>)</p>
 <p>Optical Character Recognition (OCR) Detects texts in the images and recognizes each character (unit of interpretation: <u>character</u>)</p>	 <p>Anomaly Detection Identifies outliers and captures rare items or observations which differs significantly from majority of the data</p>	

NEUROCLE

GPUS and TPUS.

3. Time-consuming: While working on sequential data depending on the computational resource it can take very large even in days or months.

4. Interpretability: Deep learning models are complex, it works like a black box. it is very difficult to interpret the result.
5. Overfitting: when the model is trained again and again, it becomes too specialized for the training data, leading to overfitting and poor performance on new data.

Advantages of Deep Learning:

1. High accuracy: Deep Learning algorithms can achieve state-of-the-art performance in various tasks, such as image recognition and natural language processing.
2. Automated feature engineering: Deep Learning algorithms can automatically discover and learn relevant features from data without the need for manual feature engineering.
3. Scalability: Deep Learning models can scale to handle large and complex datasets, and can learn from massive amounts of data.
4. Flexibility: Deep Learning models can be applied to a wide range of tasks and can handle various types of data, such as images, text, and speech.
5. Continual improvement: Deep Learning models can continually improve their performance as more data becomes available.

Disadvantages of Deep Learning:

1. High computational requirements: Deep Learning models require large amounts of data and computational resources to train and optimize.
2. Requires large amounts of labeled data: Deep Learning models often require a large amount of labeled data for training, which can be expensive and time- consuming to acquire.
3. Interpretability: Deep Learning models can be challenging to interpret, making it difficult to understand how they make decisions.
4. Black-box nature: Deep Learning models are often treated as black boxes, making it difficult to understand how they work and how they arrived at their predictions.

Overfitting: Deep Learning models can sometimes overfit to the training data, resulting in poor performance on new and unseen data.

In summary, while Deep Learning offers many advantages, including high accuracy and scalability, it also has some disadvantages, such as high computational requirements, the need for large amounts of labeled data, and interpretability challenges. These limitations need to be carefully considered when deciding whether to use Deep Learning for a specific task.

Historical Trends in Deep Learning:

Deep learning has experienced significant historical trends since its inception.

Here are some key milestones and trends that have shaped the field:

1. Early Developments: Deep learning traces its roots back to the 1960s with the development of artificial neural networks (ANNs).

- The idea of using interconnected nodes inspired by the human brain's structure laid the foundation for later deep learning advancements.

2. Winter of AI: In the 1970s and 1980s, deep learning faced a period of stagnation known as the "AI winter."

- Limited computational power, insufficient data, and theoretical challenges hindered progress in the field, leading to decreased interest and funding.

3. Backpropagation: In the 1980s, the backpropagation algorithm, which efficiently trains deep neural networks, was rediscovered and popularized.

- This breakthrough allowed for more efficient training of multi-layer neural networks, addressing some of the limitations faced during the AI winter.

4. Rise of Convolutional Neural Networks (CNNs): In the late 1990s and early 2000s, CNNs gained prominence in the field of computer vision.

- The LeNet-5 architecture developed by Yann LeCun revolutionized image recognition tasks and demonstrated the potential of deep learning in visual perception.

5. Big Data and GPUs: The early 2010s marked a turning point for deep learning with the advent of big data and the availability of powerful Graphics Processing Units (GPUs).

- The abundance of labeled data, combined with GPU acceleration, enabled the training of large-scale deep neural networks and significantly improved performance.

6. ImageNet and Deep Learning Renaissance: The ImageNet Large Scale Visual Recognition Challenge in 2012, won by a deep neural network known as AlexNet, brought deep learning into the spotlight.

- This event sparked a renaissance in the field, encouraging researchers to explore deep learning architectures and techniques across various domains.

7. Deep Learning in Natural Language Processing (NLP): Deep learning techniques, particularly recurrent neural networks (RNNs) and later transformer models, have made substantial advancements in NLP tasks.

- Models like LSTM (Long Short-Term Memory) and BERT (Bidirectional Encoder Representations from Transformers) have achieved state-of-the-art results in tasks like machine translation, sentiment analysis, and question answering.

8. Generative Models: The introduction of generative models like Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs) opened up possibilities for generating realistic images, videos, and audio.

- GANs, in particular, have demonstrated impressive capabilities in generating

synthetic data.

9. Transfer Learning and Pretraining: Transfer learning has become a prevalent technique in deep learning, enabling models to leverage knowledge from pretraining on large datasets and then fine-tune on specific tasks.

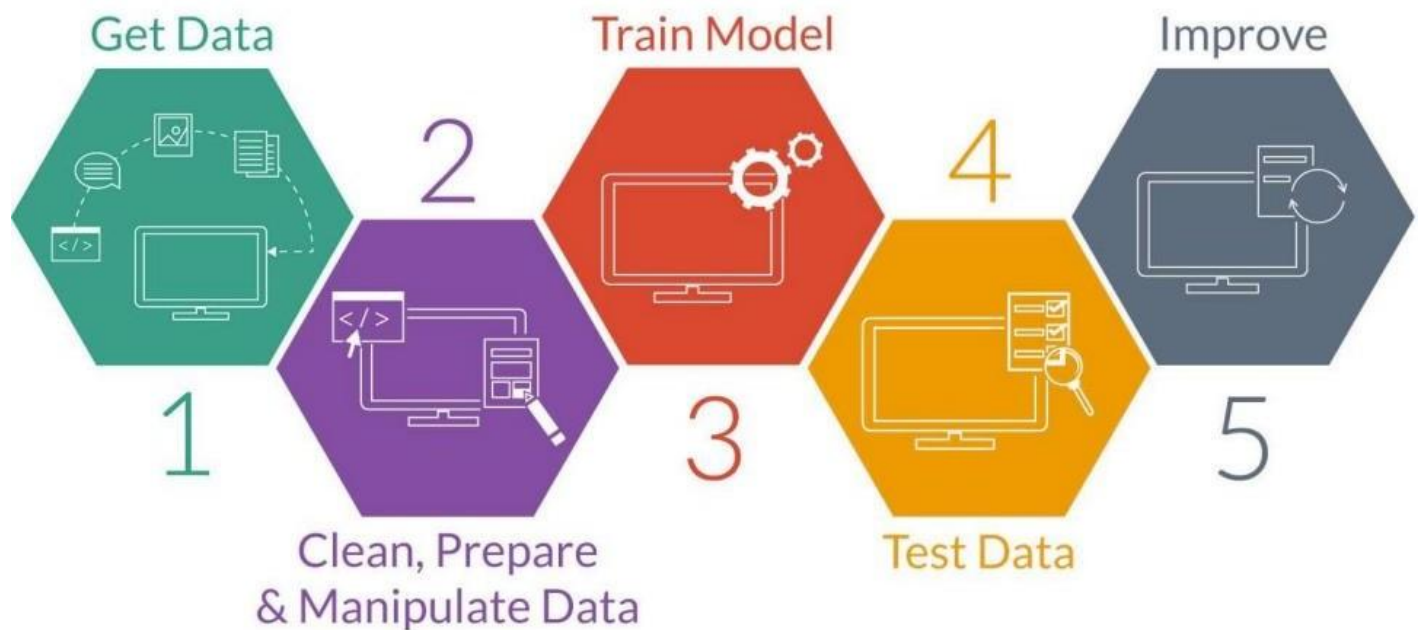
- This approach has led to significant performance improvements and reduced training time, especially in scenarios with limited labeled data.

10. Explainability and Interpretability: As deep learning models have become increasingly complex, researchers have focused on improving their explainability and interpretability.

- Techniques like attention mechanisms, saliency maps, and model-agnostic interpretability methods aim to shed light on the decision-making processes of deep learning models.

Why DL is Growing:

- Processing power needed for Deep learning is readily becoming available using GPUs, Distributed Computing and powerful CPUs.
- Moreover, as the data amount grows, Deep Learning models seem to outperform Machine Learning models.
- Focus on customization and real time decision.
- Uncover patterns that is hard to detect using traditional techniques. Find latent features (super variables) without significant manual feature engineering.



Process in ML/DL:

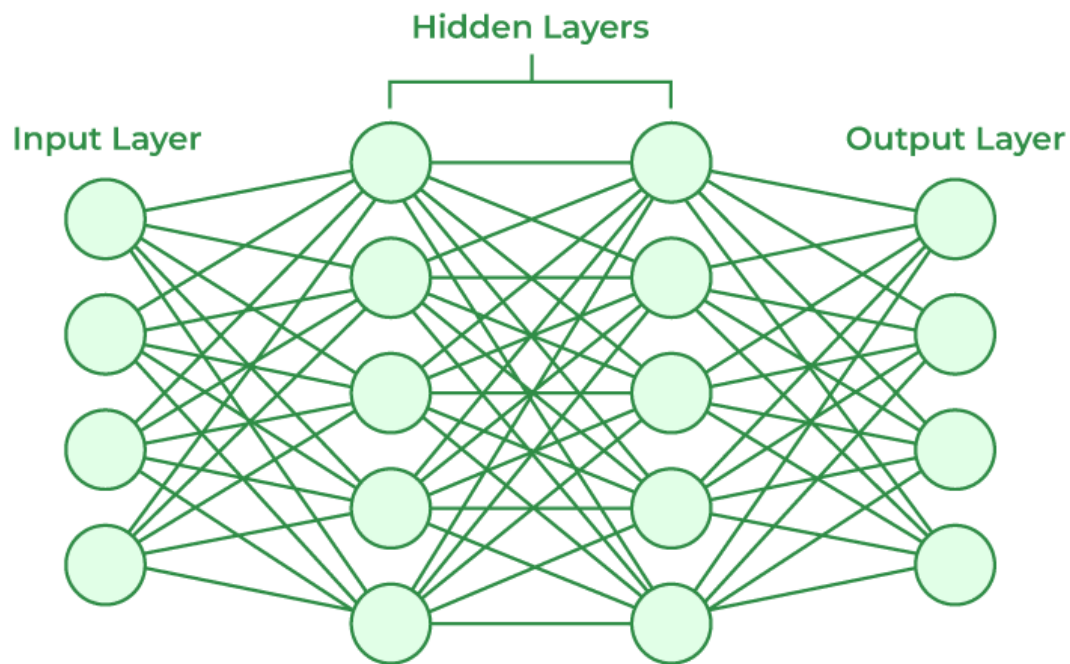
Artificial Neural Networks:

Artificial Neural Networks contain artificial neurons which are called **units**. These units are arranged in a series of layers that together constitute the whole Artificial Neural Network in a system.

A layer can have only a dozen units or millions of units as this depends on how the complex neural networks will be required to learn the hidden patterns in the dataset. Commonly, Artificial Neural Network has an input layer, an output layer as well as hidden layers.

The input layer receives data from the outside world which the neural network needs to analyze or learn about. Then this data passes through one or multiple hidden layers that transform the input into data that is valuable for the output layer. Finally, the output layer provides an output in the form of a response of the Artificial Neural Networks to input data provided.

In the majority of neural networks, units are interconnected from one layer to another. Each of these connections has weights that determine the influence of one unit on another unit. As the data transfers from one unit to another, the neural network learns more and more about the data which eventually results in an output from the output layer.



The structures and operations of human neurons serve as the basis for artificial neural networks. It is also known as neural networks or neural nets. The input layer of an artificial neural network is the first layer, and it receives input from external sources and releases it to the hidden layer, which is the second layer. In the hidden layer, each neuron receives input from the previous layer neurons, computes the weighted sum, and sends it to the neurons in the next layer. These connections are weighted means effects of the inputs from the previous layer are optimized more or less by assigning different-different weights to each input and it is adjusted during the training process by optimizing these weights for improved model performance.

Artificial neurons vs Biological neurons

The concept of artificial neural networks comes from biological neurons found in animal brains. So they share a lot of similarities in structure and function wise.

- **Structure:** The structure of artificial neural networks is inspired by biological neurons. A biological neuron has a cell body or soma to process the impulses, dendrites to receive them, and an axon that transfers them to other neurons. The input nodes of artificial neural networks receive input signals, the hidden layer nodes compute these input signals, and the output layer nodes compute the final output by processing the hidden layer's results using activation functions.

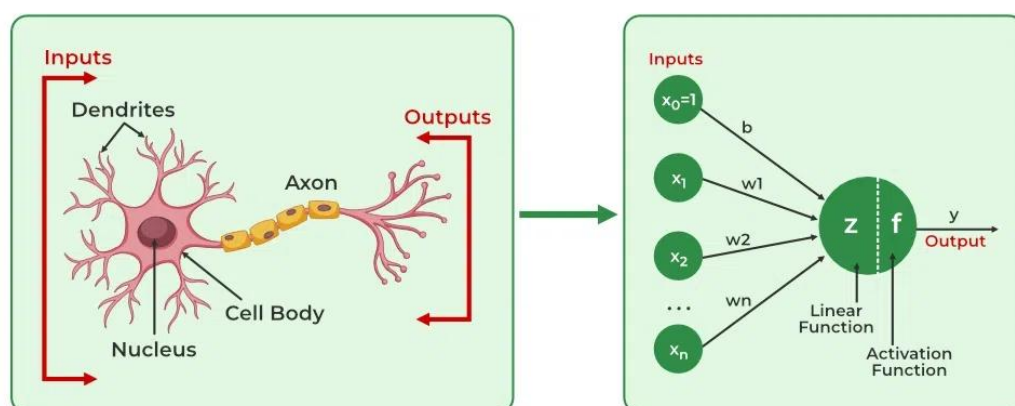
Biological Neuron	Artificial Neuron
Dendrite	Inputs
Cell nucleus or Soma	Nodes

Biological Neuron	Artificial Neuron
Synapses	Weights
Axon	Output

- **Synapses:** Synapses are the links between biological neurons that enable the transmission of impulses from dendrites to the cell body. Synapses are the weights that join the one-layer nodes to the next-layer nodes in artificial neurons. The strength of the links is determined by the weight value.
- **Learning:** In biological neurons, learning happens in the cell body nucleus or soma, which has a nucleus that helps to process the impulses. An action potential is produced and travels through the axons if the impulses are powerful enough to reach the threshold. This becomes possible by synaptic plasticity, which represents the ability of synapses to become stronger or weaker over time in reaction to changes in their activity. In artificial neural networks, backpropagation is a technique used for learning, which adjusts the weights between nodes according to the error or differences between predicted and actual outcomes.

Biological Neuron	Artificial Neuron
Synaptic plasticity	Backpropagations

- **Activation:** In biological neurons, activation is the firing rate of the neuron which happens when the impulses are strong enough to reach the threshold. In artificial neural networks, A mathematical function known as an activation function maps the input to the output, and executes activations.



How do Artificial Neural Networks learn?

Artificial neural networks are trained using a training set. For example, suppose you want to teach an ANN to recognize a cat. Then it is shown thousands of different images of cats so that the network can learn to identify a cat. Once the neural network has been trained enough using images of cats, then you need to check if it can identify cat images correctly. This is done by making the ANN classify the images it is provided by deciding whether they are cat images or not. The output obtained by the ANN is corroborated by a human-provided description of whether the image is a cat image or not. If the ANN identifies incorrectly then [back-propagation](#) is used to adjust whatever it has learned during training. [Backpropagation](#) is done by fine-tuning the weights of the connections in ANN units based on the error rate obtained. This process continues until the artificial neural network can correctly recognize a cat in an image with minimal possible error rates.

What are the types of Artificial Neural Networks?

- **[Feedforward Neural Network](#):** The feedforward neural network is one of the most basic artificial neural networks. In this ANN, the data or the input provided travels in a single direction. It enters into the ANN through the input layer and exits through the output layer while hidden layers may or may not exist. So, the feedforward neural network has a front-propagated wave only and usually does not have backpropagation.
- **[Convolutional Neural Network](#):** A Convolutional neural network has some similarities to the feed-forward neural network, where the connections between units have weights that determine the influence of one unit on another unit. But a CNN has one or more than one convolutional layer that uses a convolution operation on the input and then passes the result obtained in the form of output to the next layer. CNN has applications in speech and image processing which is particularly useful in computer vision.
- **Modular Neural Network:** A Modular Neural Network contains a collection of different neural networks that work independently towards obtaining the output with no interaction between them. Each of the different neural networks performs a different sub-task by obtaining unique inputs compared to other networks. The advantage of this modular neural network is that it breaks down a large and complex computational process into smaller components, thus decreasing its complexity while still obtaining the required output.
- **Radial basis function Neural Network:** Radial basis functions are those functions that consider the distance of a point concerning the center. RBF functions have two layers. In the first layer, the input is mapped into all the Radial basis functions in the hidden layer and then the output layer computes the output in the next step. Radial basis function nets are normally used to model the data that represents any underlying trend or function.
- **[Recurrent Neural Network](#):** The Recurrent Neural Network saves the output of a layer and feeds this output back to the input to better predict the outcome of the layer. The first layer in the RNN is quite similar to the feed-forward neural network and the recurrent neural network starts once the output of the first layer is computed. After this layer, each unit will remember some information from

the previous step so that it can act as a memory cell in performing computations.

Applications of Artificial Neural Networks

1. **Social Media:** Artificial Neural Networks are used heavily in Social Media. For example, let's take the '**People you may know**' feature on Facebook that suggests people that you might know in real life so that you can send them friend requests. Well, this magical effect is achieved by using Artificial Neural Networks that analyze your profile, your interests, your current friends, and also their friends and various other factors to calculate the people you might potentially know. Another common application of [Machine Learning](#) in social media is **facial recognition**. This is done by finding around 100 reference points on the person's face and then matching them with those already available in the database using convolutional neural networks.
2. **Marketing and Sales:** When you log onto E-commerce sites like Amazon and Flipkart, they will recommend your products to buy based on your previous browsing history. Similarly, suppose you love Pasta, then Zomato, Swiggy, etc. will show you restaurant recommendations based on your tastes and previous order history. This is true across all new-age marketing segments like Book sites, Movie services, Hospitality sites, etc. and it is done by implementing **personalized marketing**. This uses Artificial Neural Networks to identify the customer likes, dislikes, previous shopping history, etc., and then tailor the marketing campaigns accordingly.
3. **Healthcare:** Artificial Neural Networks are used in Oncology to train algorithms that can identify cancerous tissue at the microscopic level at the same accuracy as trained physicians. Various rare diseases may manifest in physical characteristics and can be identified in their premature stages by using **Facial Analysis** on the patient photos. So the full-scale implementation of Artificial Neural Networks in the healthcare environment can only enhance the diagnostic abilities of medical experts and ultimately lead to the overall improvement in the quality of medical care all over the world.
4. **Personal Assistants:** I am sure you all have heard of Siri, Alexa, Cortana, etc., and also heard them based on the phones you have!!! These are personal assistants and an example of speech recognition that uses **Natural Language Processing** to interact with the users and formulate a response accordingly. Natural Language Processing uses artificial neural networks that are made to handle many tasks of these personal assistants such as managing the language syntax, semantics, correct speech, the conversation that is going on, etc.

Neural Network, Non-linear classification example using Neural Networks: XOR/XNOR:

XOR problem with neural networks:

Among various logical gates, the XOR or also known as the "exclusive or" problem is one of the logical operations when performed on [binary](#) inputs that yield output for different

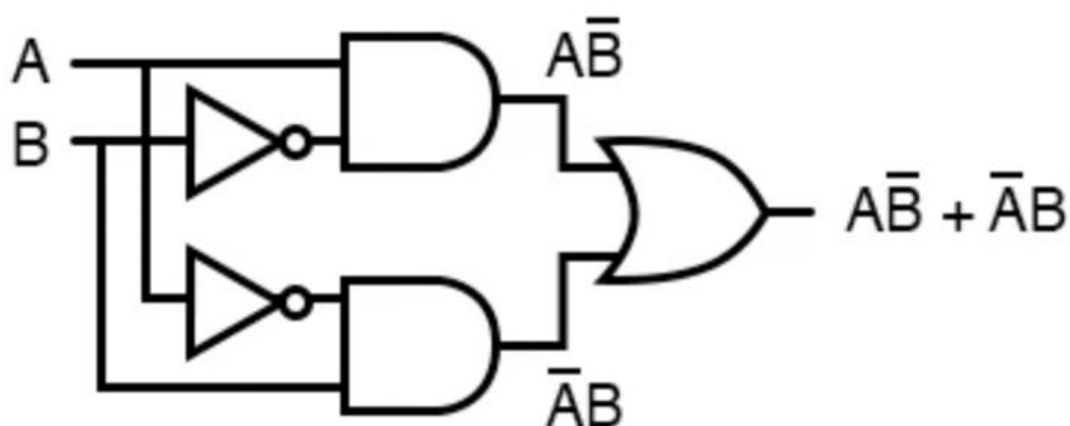
combinations of input, and for the same combination of input no output is produced. The outputs generated by the XOR logic are not [linearly](#) separable in the hyperplane. So, in this article let us see what is the XOR logic and how to integrate the XOR logic using [neural](#) networks.

From the below truth table, it can be inferred that XOR produces an output for different states of inputs and for the same inputs the XOR logic does not produce any output. The Output of XOR logic is yielded by the equation as shown below.

X	Y	Output
0	0	0
0	1	1
1	0	1
1	1	0

$$\text{Output} = X.Y' + X'.Y$$

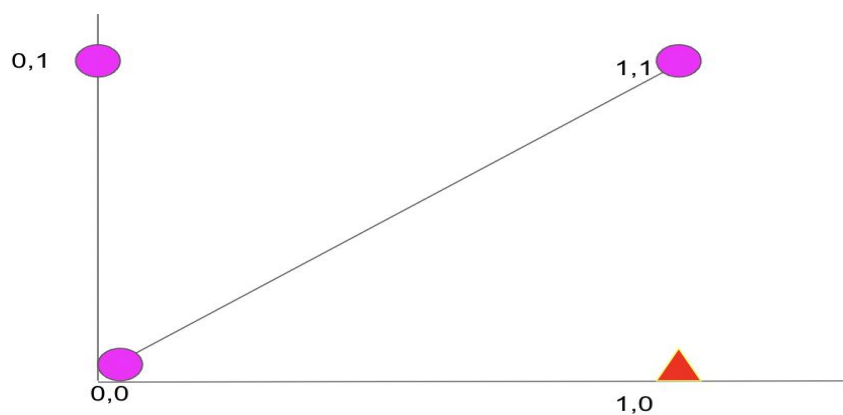
The XOR gate can be usually termed as a combination of NOT and AND gates and this type of logic finds its vast application in cryptography and fault tolerance. The logical diagram of an XOR gate is shown below.



The linear separability of points

Linear separability of points is the ability to classify the data points in the [hyperplane](#) by avoiding the overlapping of the classes in the planes. Each of the classes should fall above or below the separating line and then they are termed as linearly separable data points. With respect to logical gates operations like AND or OR the outputs generated by this logic are linearly separable in the hyperplane.

The linear separable data points appear to be as shown below.



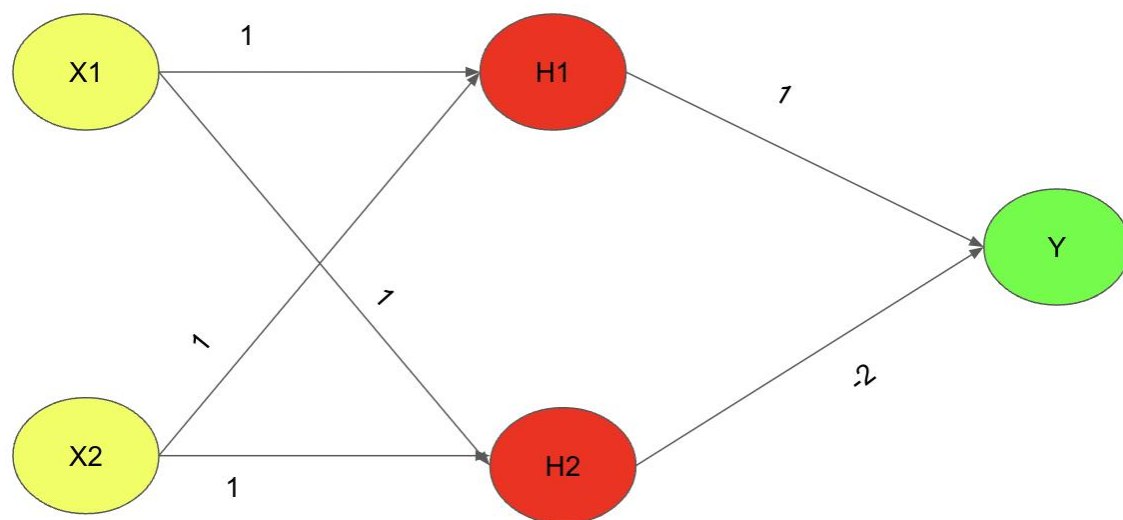
So here we can see that the pink dots and red triangle points in the plot do not overlap each other and the linear line is easily separating the two classes where the upper boundary of the plot can be considered as one classification and the below region can be considered as the other region of classification.

Need for linear separability in neural networks

Linear separability is required in neural networks as basic operations of neural networks would be in N-dimensional space and the data points of the neural networks have to be linearly separable to eradicate the issues with wrong weight updation and wrong classifications. Linear separability of data is also considered as one of the prerequisites which help in the easy interpretation of input spaces into points whether the network is positive and negative and linearly separate the data points in the hyperplane.

How to solve the XOR problem with neural networks:

The XOR problem with neural networks can be solved by using Multi-Layer Perceptrons or a neural network architecture with an input layer, hidden layer, and output layer. So during the forward **propagation** through the neural networks, the weights get updated to the corresponding layers and the XOR logic gets executed. The Neural network architecture to solve the XOR problem will be as shown below.



So with this overall architecture and certain weight parameters between each layer, the XOR logic output can be yielded through forward propagation. The overall neural network architecture uses the Relu activation function to ensure the weights updated in each of the processes to be 1 or 0 accordingly where for the positive set of weights the output at the particular neuron will be 1 and for a negative weight updation at the particular neuron will be 0 respectively. So let us understand one output for the first input state

Example: For $X1=0$ and $X2=0$ we should get an input of 0. Let us solve it.

Solution: Considering $X1=0$ and $X2=0$

$$H1 = \text{RELU}(0.1 + 0.1 + 0) = 0$$

$$H2 = \text{RELU}(0.1 + 0.1 + 0) = 0$$

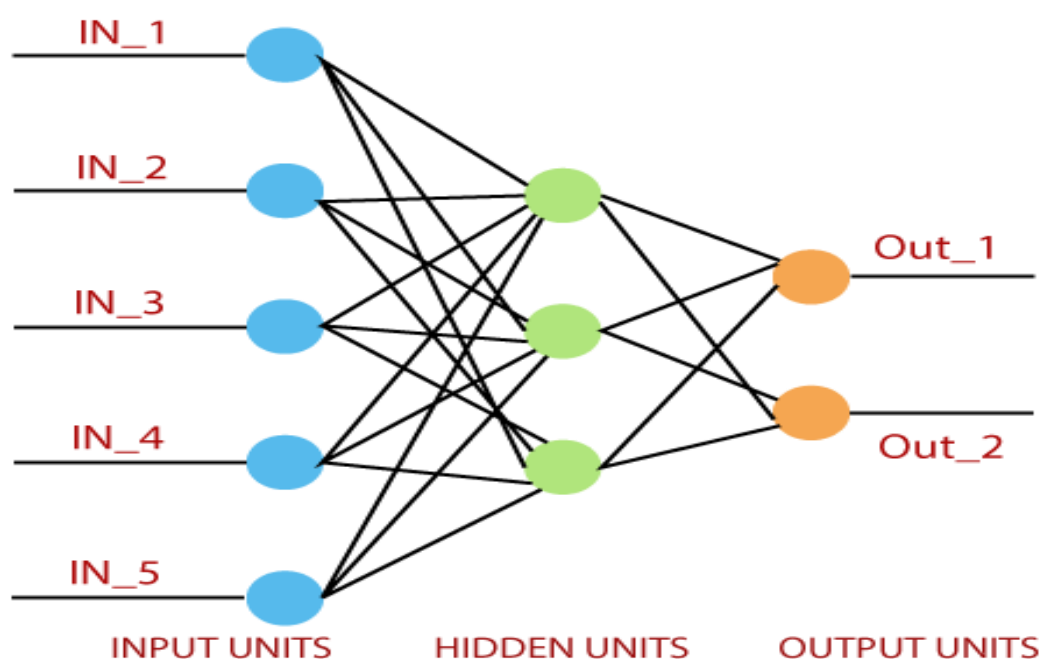
So now we have obtained the weights that were propagated from the input layer to the hidden layer. So now let us propagate from the hidden layer to the output layer

$$Y = \text{RELU}(0.1 + 0.(-2)) = 0$$

This is how multi-layer neural networks or also known as Multi-Layer perceptrons (MLP) are used to solve the XOR problem and for all other input sets the architecture provided above can be verified and the right outcome for XOR logic can be yielded.

So among the various logical operations, XOR logical operation is one such problem wherein linear separability of data points is not possible using single neurons or perceptrons. so for solving the XOR problem for neural networks it is necessary to use multiple neurons in the neural network architecture with certain weights and appropriate activation functions to solve the XOR problem with neural networks.

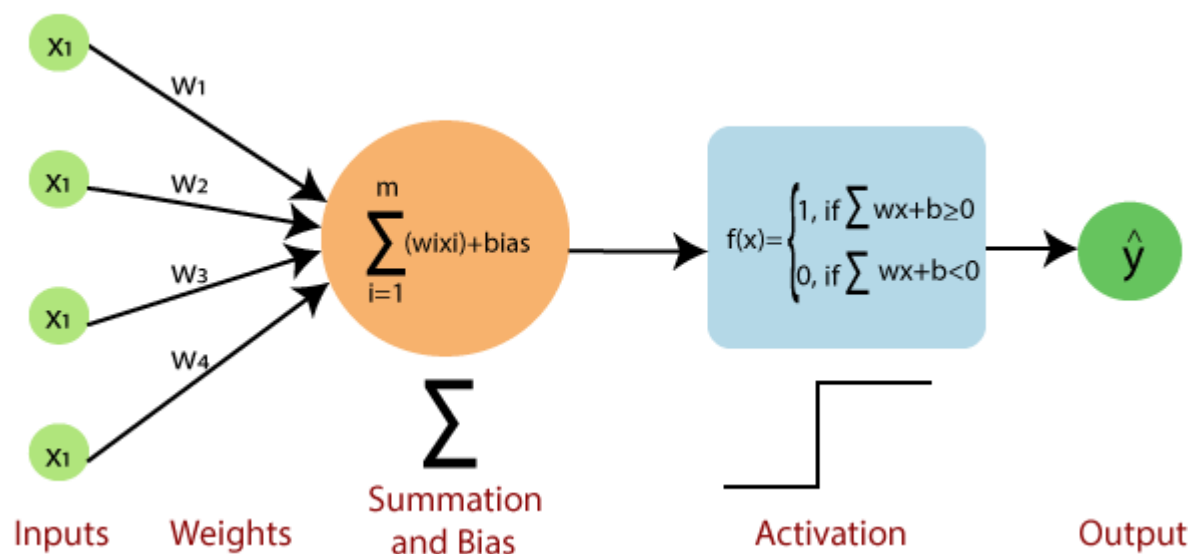
A perceptron is a neural network unit that does a precise computation to detect features in the input data. Perceptron is mainly used to classify the data into two parts. Therefore, it is also known as **Linear Binary Classifier**.



Perceptron uses the step function that returns +1 if the weighted sum of its input is 0 and -1.

The activation function is used to map the input between the required value like (0, 1) or (-1, 1).

A regular neural network looks like this:

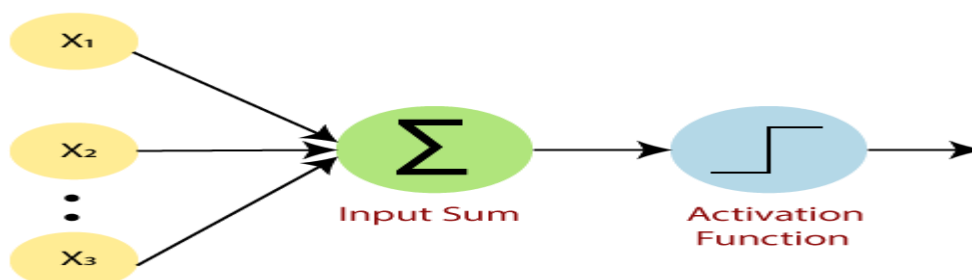


The perceptron consists of 4 parts.

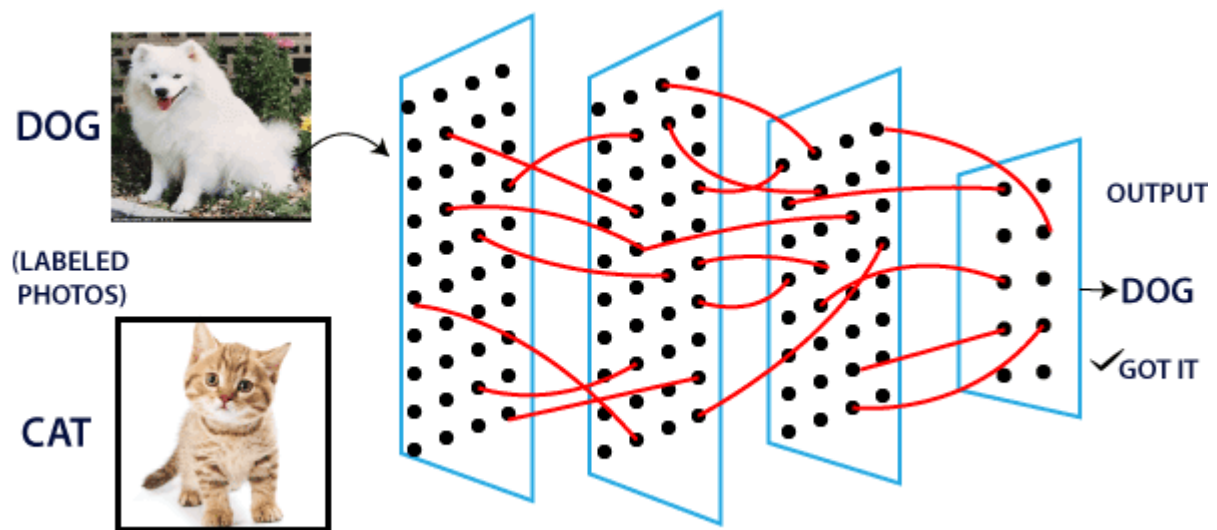
- **Input value or One input layer:** The input layer of the perceptron is made of artificial input neurons and takes the initial data into the system for further processing.
- **Weights and Bias:**

Weight: It represents the dimension or strength of the connection between units. If the weight to node 1 to node 2 has a higher quantity, then neuron 1 has a more considerable influence on the neuron.

Bias: It is the same as the intercept added in a linear equation. It is an additional parameter which task is to modify the output along with the weighted sum of the input to the other neuron.
- **Net sum:** It calculates the total sum.
- **Activation Function:** A neuron can be activated or not, is determined by an activation function. The activation function calculates a weighted sum and further adding bias with it to give the result.



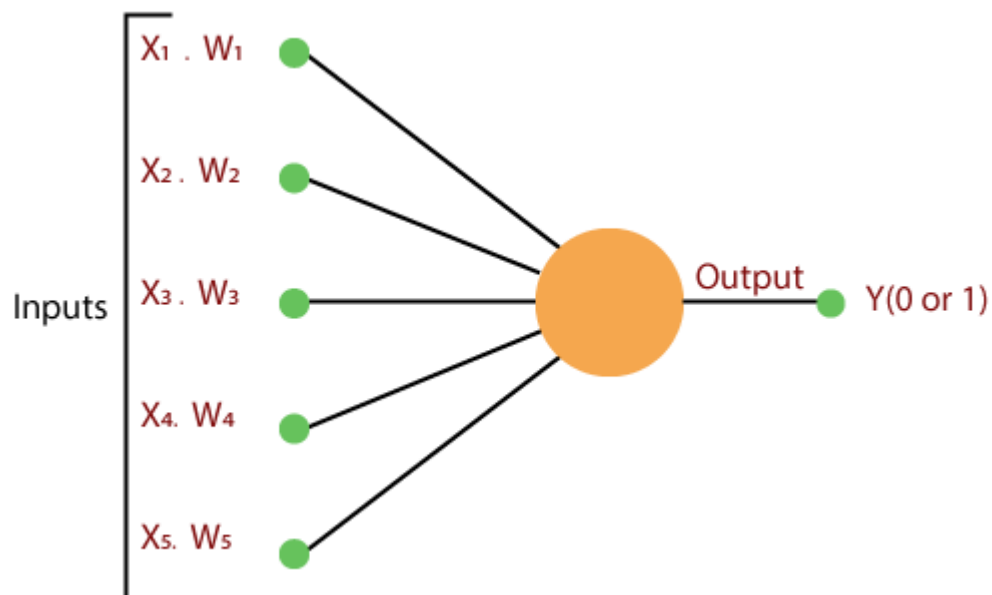
A standard neural network looks like the below diagram.



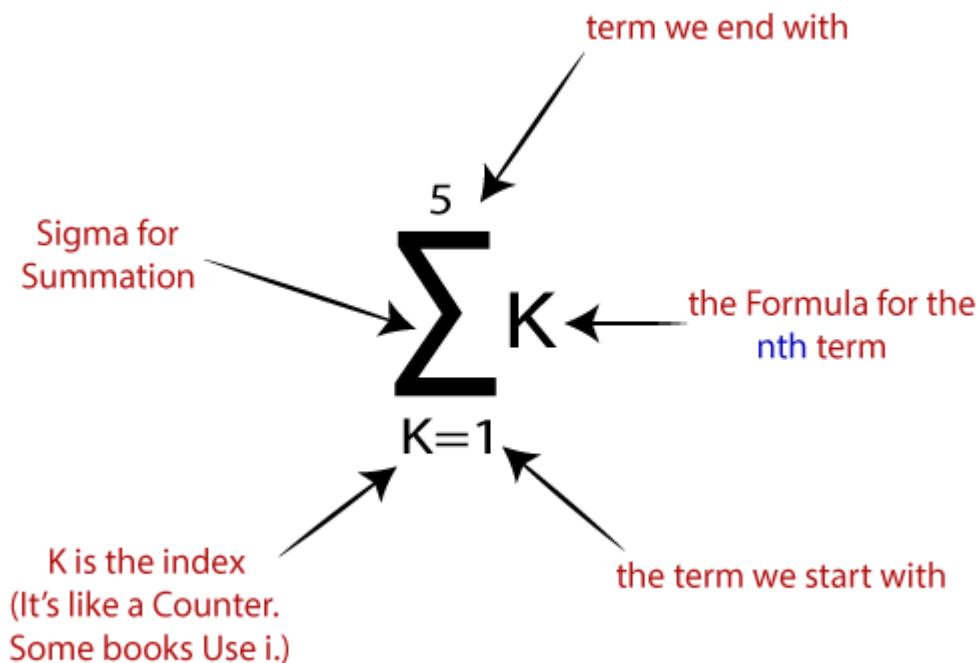
How does it work?

The perceptron works on these simple steps which are given below:

- In the first step, all the inputs x are multiplied with their weights w .



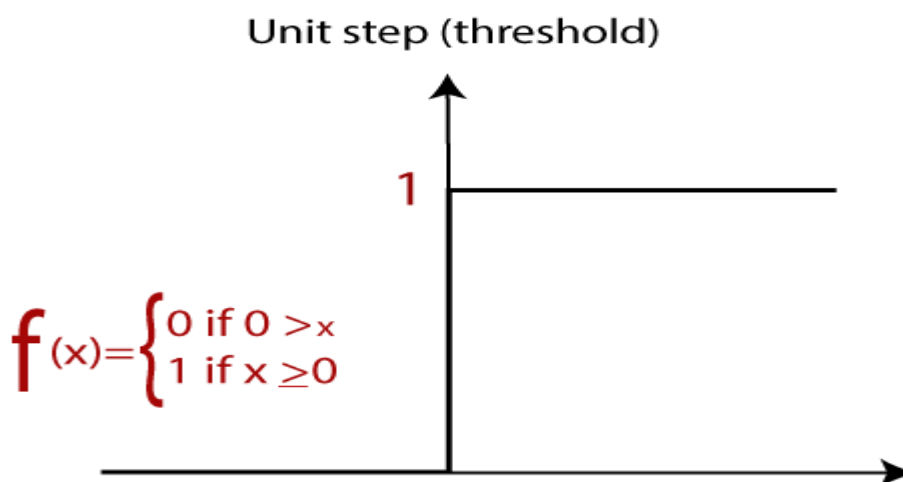
b. In this step, add all the increased values and call them the **Weighted sum**.



c. In our last step, apply the weighted sum to a correct **Activation Function**.

For Example:

A Unit Step Activation Function



There are two types of architecture. These types focus on the functionality of artificial neural networks as follows-

- Single Layer Perceptron

- Multi-Layer Perceptron

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 .

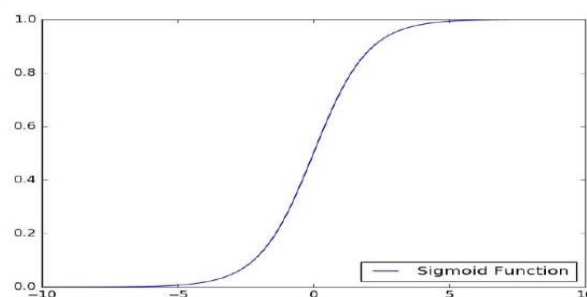
To understand the perceptron layer, it is necessary to comprehend artificial neural networks (ANNs).

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.

we will understand the concept of a multi-layer perceptron and its implementation in Python using the TensorFlow library.

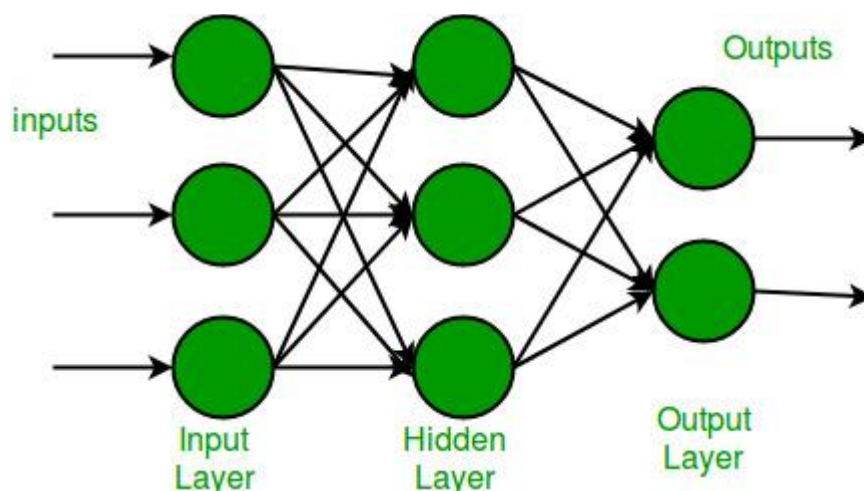
Multi-layer Perceptron :

Multi-layer perception is also known as MLP. It is fully connected dense layers, which transform any input dimension to the desired dimension. A multi-layer perception is a neural network that has multiple layers. To create a neural network we combine neurons together so that the outputs of some neurons are inputs of other neurons.

A gentle introduction to **neural networks and TensorFlow** can be found here:

- [Neural Networks](#)
- [Introduction to TensorFlow](#)

A multi-layer perceptron has one input layer and for each input, there is one neuron(or node), it has one output layer with a single node for each output and it can have any number of hidden layers and each hidden layer can have any number of nodes. A schematic diagram of a Multi-Layer Perceptron (MLP) is depicted below.



In the multi-layer perceptron diagram above, we can see that there are three inputs and thus three input nodes and the hidden layer has three nodes. The output layer gives two outputs, therefore there are two output nodes. The nodes in the input layer take input and forward it for further process, in the diagram above the nodes in the input layer forwards their output to each of the three nodes in the hidden layer, and in the same way, the hidden layer processes the information and passes it to the output layer.

Every node in the multi-layer perception uses a sigmoid activation function. The sigmoid activation function takes real values as input and converts them to numbers between 0 and 1 using the sigmoid formula.

Feed Forward Network:

Why are neural networks used:

Neuronal networks can theoretically estimate any function, regardless of its complexity. supervised learning is a method of determining the correct Y for a fresh X by learning a function that translates a given X into a specified Y. But what are the differences between neural networks and other methods of machine learning? The answer is based on the Inductive Bias phenomenon, a psychological phenomenon.

Machine learning models are built on assumptions such as the one where X and Y are related. An Inductive Bias of linear regression is the linear relationship between X and Y. In this way, a line or hyperplane gets fitted to the data.

When X and Y have a complex relationship, it can get difficult for a Linear Regression method to predict Y. For this situation, the curve must be multi-dimensional or approximate to the relationship.

A manual adjustment is needed sometimes based on the complexity of the function and the number of layers within the network. In most cases, trial and error methods combined with experience get used to accomplishing this. Hence, this is the reason these parameters are called hyperparameters.

What is a feed forward neural network:

Feed forward neural networks are [artificial neural networks](#) in which nodes do not form loops. This type of neural network is also known as a multi-layer neural network as all information is only passed forward.

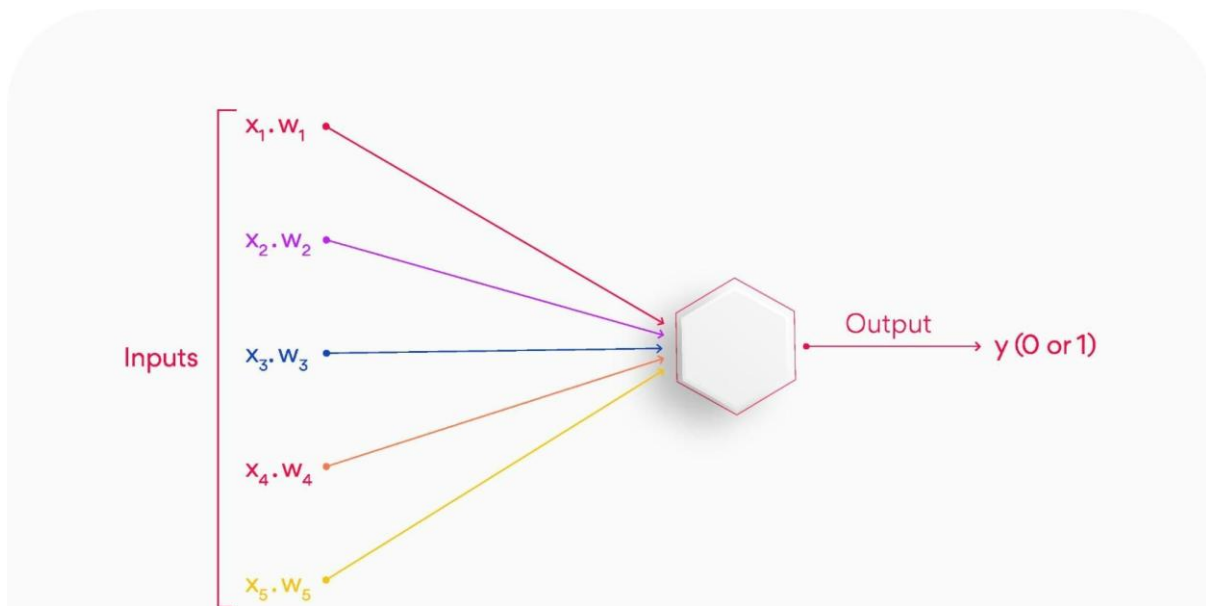
During data flow, input nodes receive data, which travel through hidden layers, and exit output nodes. No links exist in the network that could get used to by sending information back from the output node.

A feed forward neural network approximates functions in the following way:

- An algorithm calculates classifiers by using the formula $y = f^*(x)$.
- Input x is therefore assigned to category y.
- According to the feed forward model, $y = f(x; \theta)$. This value determines the closest approximation of the function.

Feed forward neural networks serve as the basis for object detection in photos, as shown in the Google Photos app.

What is the working principle of a feed forward neural network:



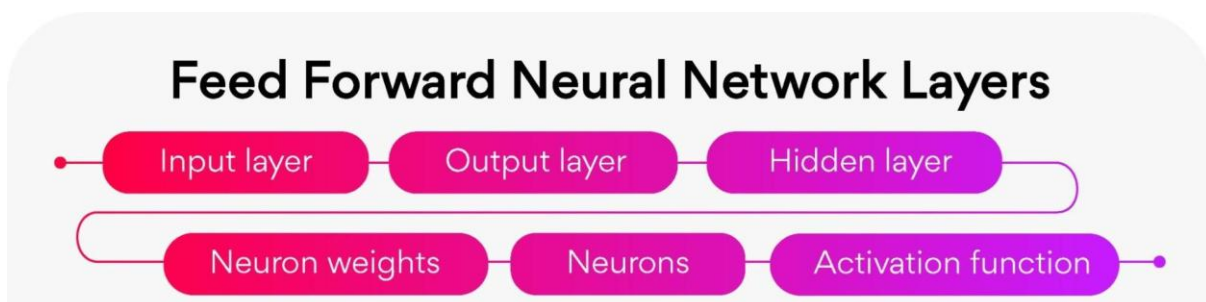
When the feed forward neural network gets simplified, it can appear as a single layer perceptron.

This model multiplies inputs with weights as they enter the layer. Afterward, the weighted input values get added together to get the sum. As long as the sum of the values rises above a certain threshold, set at zero, the output value is usually 1, while if it falls below the threshold, it is usually -1.

As a feed forward neural network model, the single-layer perceptron often gets used for classification. Machine learning can also get integrated into single-layer perceptrons. Through training, neural networks can adjust their weights based on a property called the delta rule, which helps them compare their outputs with the intended values.

As a result of training and learning, gradient descent occurs. Similarly, multi-layered perceptrons update their weights. But, this process gets known as back-propagation. If this is the case, the network's hidden layers will get adjusted according to the output values produced by the final layer.

Layers of feed forward neural network



- Input layer:

The neurons of this layer receive input and pass it on to the other layers of the network. Feature or attribute numbers in the dataset must match the number of neurons in the input layer.

- Output layer:

According to the type of model getting built, this layer represents the forecasted feature.

- Hidden layer:

Input and output layers get separated by hidden layers. Depending on the type of model, there may be several hidden layers.

There are several neurons in hidden layers that transform the input before actually transferring it to the next layer. This network gets constantly updated with weights in order to make it easier to predict.

- Neuron weights:

Neurons get connected by a weight, which measures their strength or magnitude. Similar to linear regression coefficients, input weights can also get compared.

Weight is normally between 0 and 1, with a value between 0 and 1.

- Neurons:

Artificial neurons get used in feed forward networks, which later get adapted from biological neurons. A neural network consists of artificial neurons.

Neurons function in two ways: first, they create weighted input sums, and second, they activate the sums to make them normal.

Activation functions can either be linear or nonlinear. Neurons have weights based on their inputs. During the learning phase, the network studies these weights.

- Activation Function:

Neurons are responsible for making decisions in this area.

According to the activation function, the neurons determine whether to make a linear or nonlinear decision. Since it passes through so many layers, it prevents the cascading effect from increasing neuron outputs.

An activation function can be classified into three major categories: sigmoid, Tanh, and Rectified Linear Unit (ReLU).

- Sigmoid:

Input values between 0 and 1 get mapped to the output values.

- Tanh:

A value between -1 and 1 gets mapped to the input values.

- Rectified linear Unit:

Only positive values are allowed to flow through this function. Negative values get mapped to 0.

Function in feed forward neural network

Feed Forward Neural Network Functions

1 Cost function

2 Loss function

3 Gradient learning algorithm

4 Output units

Cost function

In a feed forward neural network, the cost function plays an important role. The categorized data points are little affected by minor adjustments to weights and biases.

Thus, a smooth cost function can get used to determine a method of adjusting weights and biases to improve performance.

Following is a definition of the mean square error cost function:

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

Where,

w = the weights gathered in the network

b = biases

n = number of inputs for training

a = output vectors

x = input

$\|v\|$ = vector v 's normal length

Loss function

The loss function of a neural network gets used to determine if an adjustment needs to be made in the learning process.

Neurons in the output layer are equal to the number of classes. Showing the differences between predicted and actual probability distributions. Following is the cross-entropy loss for binary classification.

Cross Entropy Loss:

$$L(\Theta) = \begin{cases} -\log(\hat{y}) & \text{if } y = 1 \\ -\log(1 - \hat{y}) & \text{if } y = 0 \end{cases}$$

As a result of multiclass categorization, a cross-entropy loss occurs:

Cross Entropy Loss:

$$L(\Theta) = - \sum_{i=1}^k y_i \log(\hat{y}_i)$$

Gradient learning algorithm

In the gradient descent algorithm, the next point gets calculated by scaling the gradient at the current position by a learning rate. Then subtracted from the current position by the achieved value.

To decrease the function, it subtracts the value (to increase, it would add). As an example, here is how to write this procedure:

$$p_{n+1} = p_n - \eta \nabla f(p_n)$$

The gradient gets adjusted by the parameter η , which also determines the step size. Performance is significantly affected by the learning rate in machine learning.

Output units

In the output layer, output units are those units that provide the desired output or prediction, thereby fulfilling the task that the neural network needs to complete.

There is a close relationship between the choice of output units and the cost function. Any unit that can serve as a hidden unit can also serve as an output unit in a neural network.

Advantages of feed forward Neural Networks

- Machine learning can be boosted with feed forward neural networks' simplified architecture.
- Multi-network in the feed forward networks operate independently, with a moderated intermediary.
- Complex tasks need several neurons in the network.
- Neural networks can handle and process nonlinear data easily compared to perceptrons and sigmoid neurons, which are otherwise complex.
- A neural network deals with the complicated problem of decision boundaries.
- Depending on the data, the neural network architecture can vary. For example, convolutional neural networks (CNNs) perform exceptionally well in image processing, whereas [recurrent neural networks](#) (RNNs) perform well in text and voice processing.
- Neural networks need graphics processing units (GPUs) to handle large datasets for massive computational and hardware performance. Several GPUs get used widely in the market, including Kaggle Notebooks and Google Collab Notebooks.

Applications of feed forward neural networks:

Applications

- 1 Physiological feed forward system
- 2 Gene regulation and feed forward
- 3 Automation and machine management
- 4 Parallel feed forward compensation with derivative

There are many applications for these neural networks. The following are a few of them.

Physiological feed forward system

It is possible to identify feed forward management in this situation because the central involuntary regulates the heartbeat before exercise.

Gene regulation and feed forward

Detecting non-temporary changes to the atmosphere is a function of this motif as a feed forward system. You can find the majority of this pattern in the illustrious networks.

Automation and machine management

Automation control using feed forward is one of the disciplines in automation.

Parallel feed forward compensation with derivative

An open-loop transfer converts non-minimum part systems into minimum part systems using this technique.

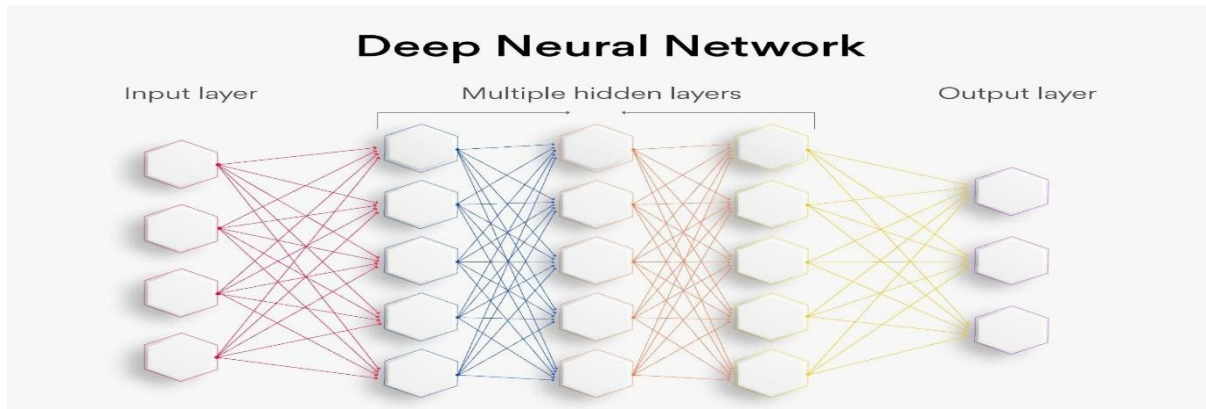
Understanding the math behind neural networks

Typical deep learning algorithms are neural networks (NNs). As a result of their unique structure, their popularity results from their 'deep' understanding of data.

Furthermore, NNs are flexible in terms of complexity and structure. Despite all the advanced stuff, they can't work without the basic elements: they may work better with the advanced stuff, but the underlying structure remains the same

Deep Feed- forward networks:

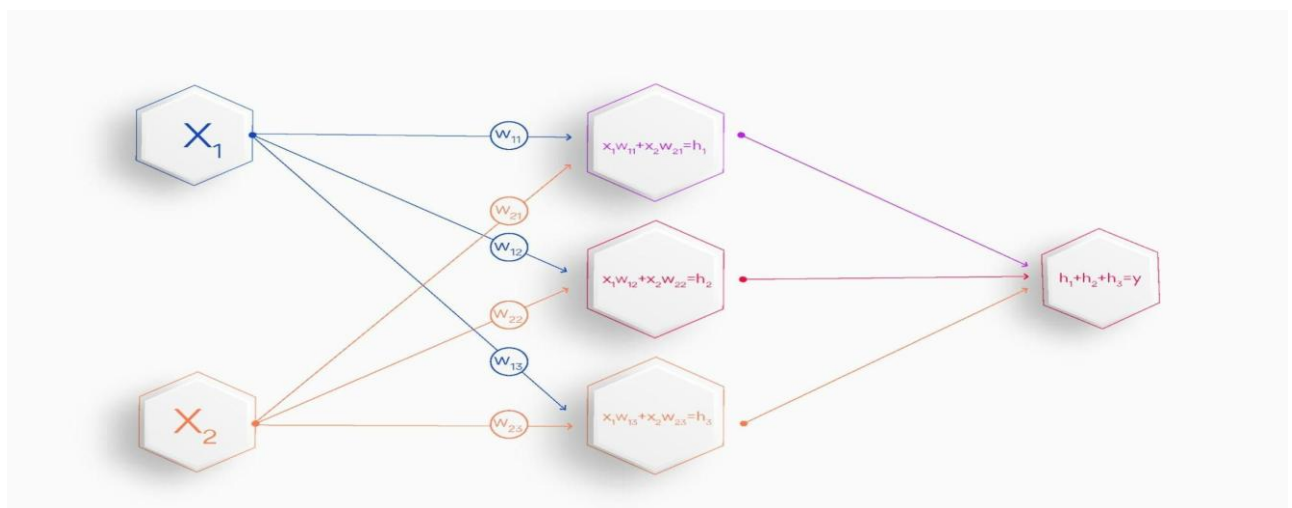
NNs get constructed similarly to our biological neurons, and they resemble the following:



Neurons are hexagons in this image. In neural networks, neurons get arranged into layers: input is the first layer, and output is the last with the hidden layer in the middle.

NN consists of two main elements that compute mathematical operations. Neurons calculate weighted sums using input data and synaptic weights since neural networks are just mathematical computations based on synaptic links.

The following is a simplified visualization:



In a matrix format, it looks as follows:

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} = \begin{bmatrix} x_1 w_{11} + x_2 w_{21} \\ x_1 w_{12} + x_2 w_{22} \\ x_1 w_{13} + x_2 w_{23} \end{bmatrix}' = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}'$$

In the third step, a vector of ones gets multiplied by the output of our hidden layer:

$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}' \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = h_1 + h_2 + h_3 = y$$

Using the output value, we can calculate the result. Understanding these fundamental concepts will make building NN much easier, and you will be amazed at how quickly you can do it. Every layer's output becomes the following layer's input.

The architecture of the network

In a network, the architecture refers to the number of hidden layers and units in each layer that make up the network.

A feed forward network based on the Universal Approximation Theorem must have a "squashing" activation function at least on one hidden layer.

The network can approximate any Borel measurable function within a finite-dimensional space with at least some amount of non-zero error when there are enough hidden units.

It simply states that we can always represent any function using the multi-layer perceptron (MLP), regardless of what function we try to learn.

Thus, we now know there will always be an MLP to solve our problem, but there is no specific method for finding it.

It is impossible to say whether it will be possible to solve the given problem if we use N layers with M hidden units.

Research is still ongoing, and for now, the only way to determine this configuration is by experimenting with it.

While it is challenging to find the appropriate architecture, we need to try many configurations before finding the one that can represent the target function.

There are two possible explanations for this. Firstly, the optimization algorithm may not find the correct parameters, and secondly, the training algorithms may use the wrong function because of overfitting.

What is backpropagation in feed forward neural network?

Backpropagation is a technique based on gradient descent. Each stage of a gradient descent process involves iteratively moving a function in the opposite direction of its gradient (the slope).

The goal is to reduce the cost function given the training data while learning a neural network. Network weights and biases of all neurons in each layer determine the cost function. Backpropagation gets used to calculate the gradient of the cost function iteratively. And then update weights and biases in the opposite direction to reduce the gradient.

We must define the error of the backpropagation formula to specify i-th neuron in the i-th layer of a network for the j-th training. Example as follows (in which $z_i^{[l](j)}$ represents the weighted input to the neuron, and L represents the loss.)

$$\delta_i^{[l](j)} = \frac{\partial \mathcal{L}(\hat{y}^{(j)}, y^{(j)})}{\partial z_i^{[l](j)}}$$

In backpropagation formulas, the error is defined as above:

Below is the full derivation of the formulas. For each formula below, L stands for the output layer, g for the activation function, ∇ the gradient, $W^{[l]T}$ layer l weights transposed.

A proportional activation of neuron i at layer l based on b_{li} bias from layer i to layer i, $w_{ik}^{[l]}$ weight from layer l to layer l-1, and $a_k^{[l-1](j)}$ activation of neuron k at layer l-1 for training example j.

$$\delta^{[L](j)} = \nabla_{\hat{y}^{(j)}} \mathcal{L} \odot (g^{[L]})'(z^{[L](j)}) = \hat{y}^{(j)} - y^{(j)}$$

$$\delta_i^{[l](j)} = W^{[l+1]T} \delta^{[l+1](j)} \odot (g^{[l]})'(z_i^{[l](j)})$$

$$\frac{\partial \mathcal{L}}{\partial b_i^{[l]}} = \delta_i^{[l](j)}$$

$$\frac{\partial \mathcal{L}}{\partial w_{ik}^{[l]}} = \delta_i^{[l](j)} a_k^{[l-1](j)}$$

The first equation shows how to calculate the error at the output layer for sample j. Following that, we can use the second equation to calculate the error in the layer just before the output layer.

Based on the error values for the next layer, the second equation can calculate the error in any layer. Because this algorithm calculates errors backward, it is known as backpropagation.

For sample j , we calculate the gradient of the loss function by taking the third and fourth equations and dividing them by the biases and weights.

We can update biases and weights by averaging gradients of the loss function relative to biases and weights for all samples using the average gradients.

The process is known as batch gradient descent. We will have to wait a long time if we have too many samples.

If each sample has a gradient, it is possible to update the biases/weights accordingly. The process is known as stochastic gradient descent.

Even though this algorithm is faster than batch gradient descent, it does not yield a good estimate of the gradient calculated using a single sample.

It is possible to update biases and weights based on the average gradients of batches. It gets referred to as mini-batch gradient descent and gets preferred over the other two.

Stochastic Gradient Descent (SGD):

Gradient Descent is an iterative optimization process that searches for an objective function's optimum value (Minimum/Maximum). It is one of the most used methods for changing a model's parameters in order to reduce a cost function in machine learning projects.

The primary goal of gradient descent is to identify the model parameters that provide the maximum accuracy on both training and test datasets. In gradient descent, the gradient is a vector pointing in the general direction of the function's steepest rise at a particular point. The algorithm might gradually drop towards lower values of the function by moving in the opposite direction of the gradient, until reaching the minimum of the function.

Types of Gradient Descent:

Typically, there are three types of Gradient Descent:

1. [Batch Gradient Descent](#)
2. Stochastic Gradient Descent
3. [Mini-batch Gradient Descent](#)

In this article, we will be discussing **Stochastic Gradient Descent (SGD)**.

Stochastic Gradient Descent (SGD):

Stochastic Gradient Descent (SGD) is a variant of the [Gradient Descent](#) algorithm that is used for optimizing machine learning models. It addresses the computational inefficiency of traditional Gradient Descent methods when dealing with large datasets in machine learning projects.

In SGD, instead of using the entire dataset for each iteration, only a single random training example (or a small batch) is selected to calculate the gradient and update the model

parameters. This random selection introduces randomness into the optimization process, hence the term “stochastic” in stochastic Gradient Descent

The advantage of using SGD is its computational efficiency, especially when dealing with large datasets. By using a single example or a small batch, the computational cost per iteration is significantly reduced compared to traditional Gradient Descent methods that require processing the entire dataset.

Stochastic Gradient Descent Algorithm :

- **Initialization:** Randomly initialize the parameters of the model.
- **Set Parameters:** Determine the number of iterations and the learning rate (alpha) for updating the parameters.
- **Stochastic Gradient Descent Loop:** Repeat the following steps until the model converges or reaches the maximum number of iterations:
 - a. Shuffle the training dataset to introduce randomness.
 - b. Iterate over each training example (or a small batch) in the shuffled order.
 - c. Compute the gradient of the cost function with respect to the model parameters using the current training example (or batch).
 - d. Update the model parameters by taking a step in the direction of the negative gradient, scaled by the learning rate.
 - e. Evaluate the convergence criteria, such as the difference in the cost function between iterations of the gradient.
- **Return Optimized Parameters:** Once the convergence criteria are met or the maximum number of iterations is reached, return the optimized model parameters.

In SGD, since only one sample from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier than your typical Gradient Descent algorithm. But that doesn't matter all that much because the path taken by the algorithm does not matter, as long as we reach the minimum and with a significantly shorter training time.

Hidden Units:

In [neural networks](#), a hidden layer is located between the input and output of the algorithm, in which the function applies weights to the inputs and directs them through an [activation function](#) as the output. In short, the hidden layers perform nonlinear transformations of the inputs entered into the network. Hidden layers vary depending on the function of the neural network, and similarly, the layers may vary depending on their associated weights.

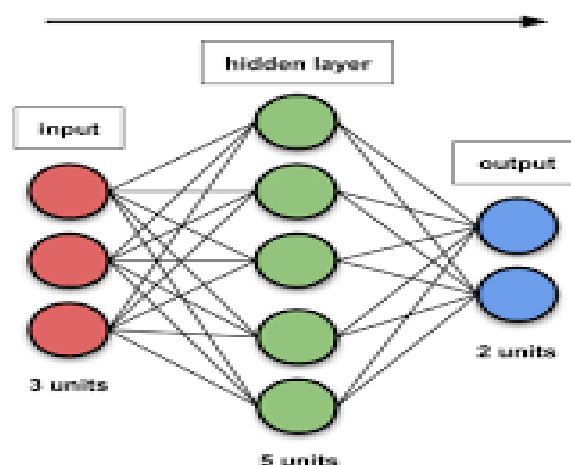
How does a Hidden Layer work?

Hidden layers, simply put, are layers of mathematical functions each designed to produce an output specific to an intended result. For example, some forms of hidden layers are known as squashing functions. These functions are particularly useful when the intended output of the algorithm is a [probability](#) because they take an input and produce an output value between 0 and 1, the range for defining probability.

Hidden layers allow for the function of a neural network to be broken down into specific transformations of the data. Each hidden layer function is specialized to produce a defined output. For example, a hidden layer functions that are used to identify human eyes and ears may be used in conjunction by subsequent layers to identify faces in images. While the functions to identify eyes alone are not enough to independently recognize objects, they can function jointly within a neural network.

Hidden Layers and Machine Learning:

Hidden layers are very common in neural networks, however their use and architecture often varies from case to case. As referenced above, hidden layers can be separated by their functional characteristics. For example, in a CNN used for object recognition, a hidden layer that is used to identify wheels cannot solely identify a car, however when placed in conjunction with additional layers used to identify windows, a large metallic body, and headlights, the neural network can then make predictions and identify possible cars within visual data.



Choosing Hidden Layers

1. Well if the data is linearly separable then you don't need any hidden layers at all.
2. If data is less complex and is having fewer dimensions or features then neural networks with 1 to 2 hidden layers would work.
3. If data is having large dimensions or features then to get an optimum solution, 3 to 5 hidden layers can be used.

It should be kept in mind that increasing hidden layers would also increase the complexity of the model and choosing hidden layers such as 8, 9, or in two digits may sometimes lead to overfitting.

Choosing Nodes in Hidden Layers

Once hidden layers have been decided the next task is to choose the number of nodes in each hidden layer.

1. The number of hidden neurons should be between the size of the input layer and the output layer.
2. The most appropriate number of hidden neurons is

$\text{Sqrt}(\text{input layer nodes} * \text{output layer nodes})$

1. The number of hidden neurons should keep on decreasing in subsequent layers to get more and more close to pattern and feature extraction and to identify the target class.

These above algorithms are only a general use case and they can be moulded according to use case. Sometimes the number of nodes in hidden layers can increase also in subsequent layers and the number of hidden layers can also be more than the ideal case.

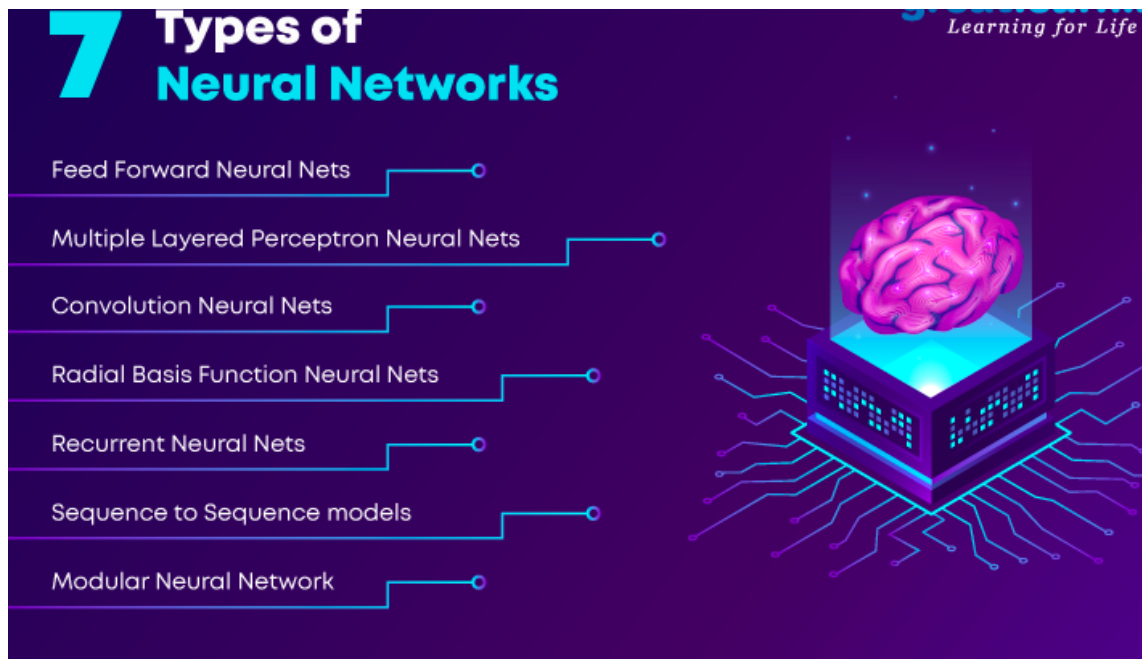
This whole depends upon the use case and problem statement that we are dealing with.

Architecture Design:

Types of neural networks models are listed below:

The nine types of neural networks are:

- Perceptron
- Feed Forward Neural Network
- [Multilayer Perceptron](#)
- Convolutional Neural Network
- Radial Basis Functional Neural Network
- Recurrent Neural Network
- LSTM – Long Short-Term Memory
- Sequence to Sequence Models
- Modular Neural Network



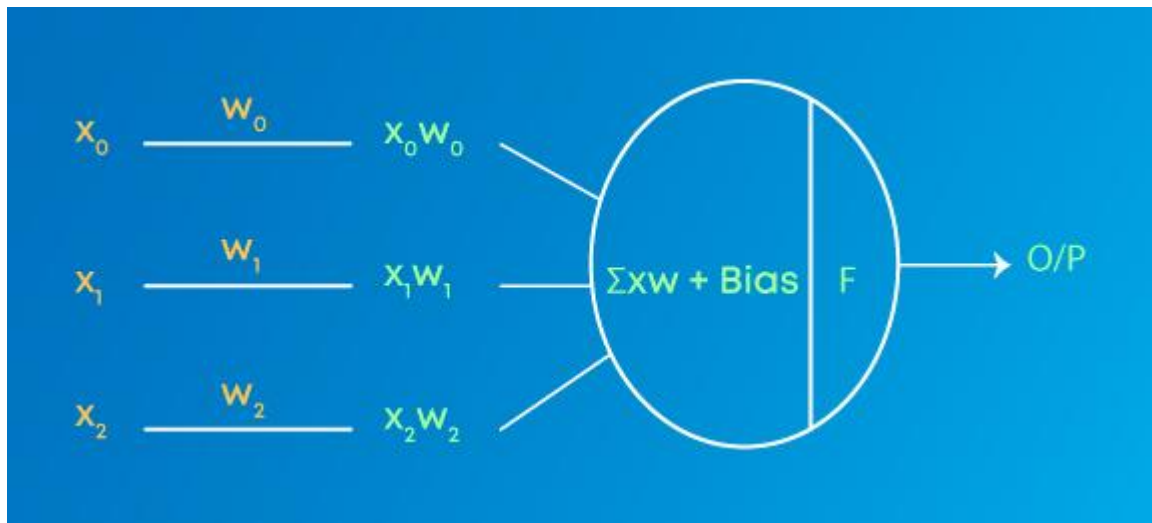
An Introduction to Artificial Neural Network

Neural networks represent [deep learning](#) using [artificial intelligence](#). Certain application scenarios are too heavy or out of scope for traditional machine learning algorithms to handle. As they are commonly known, Neural Network pitches in such scenarios and fills the gap. Also, enroll in the [neural networks and deep learning](#) course and enhance your skills today.

Artificial neural networks are inspired by the biological neurons within the human body which activate under certain circumstances resulting in a related action performed by the body in response. Artificial neural nets consist of various layers of interconnected artificial neurons powered by activation functions that help in switching them ON/OFF. Like traditional [machine algorithms](#), here too, there are certain values that neural nets learn in the training phase.

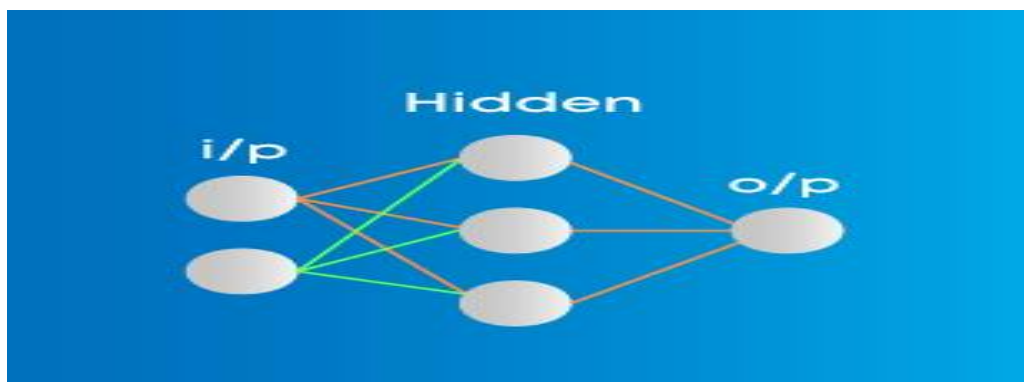
Briefly, each neuron receives a multiplied version of inputs and random weights, which is then added with a static bias value (unique to each neuron layer); this is then passed to an appropriate activation function which decides the final value to be given out of the neuron. There are various activation functions available as per the nature of input values. Once the output is generated from the final neural net layer, loss function (input vs output) is

calculated, and backpropagation is performed where the weights are adjusted to make the loss minimum. Finding optimal values of weights is what the overall operation focuses around. Please refer to the following for better understanding-



Weights are numeric values that are multiplied by inputs. In backpropagation, they are modified to reduce the loss. In simple words, weights are machine learned values from Neural Networks. They self-adjust depending on the difference between predicted outputs vs training inputs.

Activation Function is a mathematical formula that helps the neuron to switch ON/OFF.



- **Input layer** represents dimensions of the input vector.
- **Hidden layer** represents the intermediary nodes that divide the input space into regions with (soft) boundaries. It takes in a set of weighted input and produces output through an activation function.
- **Output layer** represents the output of the neural network.

Backpropagation:

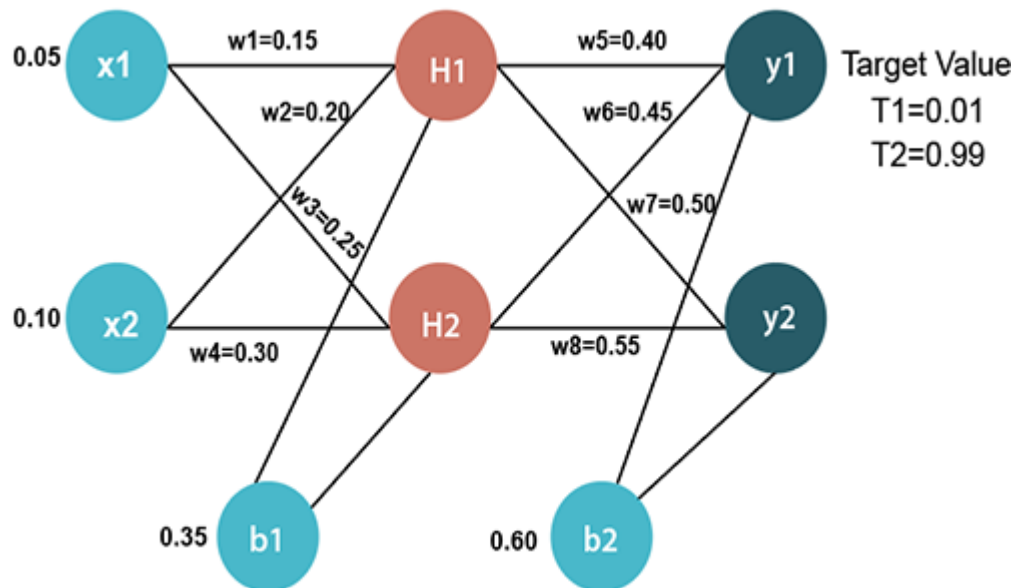
Backpropagation Process in Deep Neural Network

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

$X_1=0.05$

$X_2=0.10$

Initial weight

$$W1=0.15$$

$$W2=0.20$$

$$W3=0.25$$

$$W4=0.30 \quad w8=0.55$$

$$w5=0.40$$

$$w6=0.45$$

$$w7=0.50$$

Bias Values

$$b1=0.35 \quad b2=0.60$$

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

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

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

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

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

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

So, the total error is

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

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

$$= 0.274811084 + 0.0235600257$$

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

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 .

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

We perform backward process so first consider the last weight w_5 as

$$\text{Error}_{w_5} = \frac{\partial E_{\text{total}}}{\partial w_5} \dots \dots \dots (1)$$

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

From equation two, it is clear that we cannot partially differentiate it with respect to w_5 because there is no any w_5 . We split equation one into multiple terms so that we can easily differentiate it with respect to w_5 as

$$\frac{\partial E_{\text{total}}}{\partial w_5} = \frac{\partial E_{\text{total}}}{\partial y1_{\text{final}}} \times \frac{\partial y1_{\text{final}}}{\partial y1} \times \frac{\partial y1}{\partial w_5} \dots \dots \dots (3)$$

Now, we calculate each term one by one to differentiate E_{total} with respect to w_5 as

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

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

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

$$= -(0.01 - 0.75136507)$$

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

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

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

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

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

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

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

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

$$\begin{aligned}
 &= \frac{1 - y1_{\text{final}}}{y1_{\text{final}}} \times (y1_{\text{final}})^2 \\
 &= y1_{\text{final}} \times (1 - y1_{\text{final}}) \\
 &= 0.75136507 \times (1 - 0.75136507) \\
 \frac{\partial y1_{\text{final}}}{\partial y1} &= 0.186815602 \dots \dots \dots (8)
 \end{aligned}$$

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

$$\begin{aligned}
 \frac{\partial y1}{\partial w5} &= \frac{\partial (H1_{\text{final}} \times w5 + H2_{\text{final}} \times w6 + b2)}{\partial w5} \\
 &= H1_{\text{final}}
 \end{aligned}$$

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

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

$$\begin{aligned}
 \frac{\partial E_{\text{total}}}{\partial w5} &= \frac{\partial E_{\text{total}}}{\partial y1_{\text{final}}} \times \frac{\partial y1_{\text{final}}}{\partial y1} \times \frac{\partial y1}{\partial w5} \\
 &= 0.74136507 \times 0.186815602 \times 0.593269992 \\
 \text{Error}_{w5} &= \frac{\partial E_{\text{total}}}{\partial w5} = 0.0821670407 \dots \dots \dots (11)
 \end{aligned}$$

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

$$\begin{aligned}
 w5_{\text{new}} &= w5 - \eta \times \frac{\partial E_{\text{total}}}{\partial w5} \text{ Here, } \eta = \text{learning rate} = 0.5 \\
 &= 0.4 - 0.5 \times 0.0821670407 \\
 w5_{\text{new}} &= 0.35891648 \dots \dots \dots (12)
 \end{aligned}$$

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

$$\begin{aligned}
 w5_{\text{new}} &= 0.35891648 \\
 w6_{\text{new}} &= 408666186
 \end{aligned}$$

$$w7_{\text{new}} = 0.511301270$$

$$w8_{\text{new}} = 0.561370121$$

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

$$\text{Error}_{w1} = \frac{\partial E_{\text{total}}}{\partial w1}$$

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

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

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

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

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

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

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

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

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

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

We again Split both $\frac{\partial E_1}{\partial y_1}$ and $\frac{\partial E_2}{\partial y_2}$ because there is no any y_1 and y_2 term in E_1 and E_2 .
We split it as

$$\frac{\partial E_1}{\partial y_1} = \frac{\partial E_1}{\partial y_{1_{final}}} \times \frac{\partial y_{1_{final}}}{\partial y_1} \dots \dots (18)$$

$$\frac{\partial E_2}{\partial y_2} = \frac{\partial E_2}{\partial y_{2_{final}}} \times \frac{\partial y_{2_{final}}}{\partial y_2} \dots \dots (19)$$

Now, we find the value of $\frac{\partial E_1}{\partial y_1}$ and $\frac{\partial E_2}{\partial y_2}$ by putting values in equation (18) and (19) as

From equation (18)

$$\begin{aligned} \frac{\partial E_1}{\partial y_1} &= \frac{\partial E_1}{\partial y_{1_{final}}} \times \frac{\partial y_{1_{final}}}{\partial y_1} \\ &= \frac{\partial(\frac{1}{2}(T_1 - y_{1_{final}})^2)}{\partial y_{1_{final}}} \times \frac{\partial y_{1_{final}}}{\partial y_1} \\ &= 2 \times \frac{1}{2}(T_1 - y_{1_{final}}) \times (-1) \times \frac{\partial y_{1_{final}}}{\partial y_1} \end{aligned}$$

From equation (8)

$$\begin{aligned} &= 2 \times \frac{1}{2}(0.01 - 0.75136507) \times (-1) \times 0.186815602 \\ \frac{\partial E_1}{\partial y_1} &= \mathbf{0.138498562} \dots \dots (20) \end{aligned}$$

From equation (19)

$$\begin{aligned}
 \frac{\partial E_2}{\partial y_2} &= \frac{\partial E_2}{\partial y_{2_{final}}} \times \frac{\partial y_{2_{final}}}{\partial y_2} \\
 &= \frac{\partial \left(\frac{1}{2} (T_2 - y_{2_{final}})^2 \right)}{\partial y_{2_{final}}} \times \frac{\partial y_{2_{final}}}{\partial y_2} \\
 &= 2 \times \frac{1}{2} (T_2 - y_{2_{final}}) \times (-1) \times \frac{\partial y_{2_{final}}}{\partial y_2} \dots \dots \dots (21)
 \end{aligned}$$

$$y_{2_{final}} = \frac{1}{1 + e^{-y_2}} \dots \dots \dots (22)$$

$$\begin{aligned}
 \frac{\partial y_{2_{final}}}{\partial y_2} &= \frac{\partial \left(\frac{1}{1 + e^{-y_2}} \right)}{\partial y_2} \\
 &= \frac{e^{-y_2}}{(1 + e^{-y_2})^2} \\
 &= e^{-y_2} \times (y_{2_{final}})^2 \dots \dots \dots (23)
 \end{aligned}$$

$$y_{2_{final}} = \frac{1}{1 + e^{-y_2}}$$

$$e^{-y_2} = \frac{1 - y_{2_{final}}}{y_{2_{final}}} \dots \dots \dots (24)$$

Putting the value of e^{-y_2} in equation (23)

$$\begin{aligned}
 &= \frac{1 - y_{2_{final}}}{y_{2_{final}}} \times (y_{2_{final}})^2 \\
 &= y_{2_{final}} \times (1 - y_{2_{final}}) \\
 &= 0.772928465 \times (1 - 0.772928465) \\
 \frac{\partial y_{2_{final}}}{\partial y_2} &= \mathbf{0.175510053} \dots \dots \dots (25)
 \end{aligned}$$

From equation (21)

$$\begin{aligned}
 &= 2 \times \frac{1}{2} (0.99 - 0.772928465) \times (-1) \times 0.175510053 \\
 \frac{\partial E_1}{\partial y_1} &= \mathbf{-0.0380982366126414} \dots \dots \dots (26)
 \end{aligned}$$

Now from equation (16) and (17)

$$\begin{aligned}
 \frac{\partial E_1}{\partial H1_{final}} &= \frac{\partial E_1}{\partial y1} \times \frac{\partial y1}{\partial H1_{final}} \\
 &= 0.138498562 \times \frac{\partial(H1_{final} \times w_5 + H2_{final} \times w_6 + b2)}{\partial H1_{final}} \\
 &= 0.138498562 \times \frac{\partial(H1_{final} \times w_5 + H2_{final} \times w_6 + b2)}{\partial H1_{final}} \\
 &= 0.138498562 \times w_5 \\
 &= 0.138498562 \times 0.40
 \end{aligned}$$

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

$$\begin{aligned}
 \frac{\partial E_2}{\partial H1_{final}} &= \frac{\partial E_2}{\partial y2} \times \frac{\partial y2}{\partial H1_{final}} \\
 &= -0.0380982366126414 \times \frac{\partial(H1_{final} \times w_7 + H2_{final} \times w_8 + b2)}{\partial H1_{final}} \\
 &= -0.0380982366126414 \times w_7 \\
 &= -0.0380982366126414 \times 0.50
 \end{aligned}$$

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

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

$$\begin{aligned}
 \frac{\partial E_{total}}{\partial H1_{final}} &= \frac{\partial E_1}{\partial H1_{final}} + \frac{\partial E_2}{\partial H1_{final}} \\
 &= 0.0553994248 + (-0.0190491183063207)
 \end{aligned}$$

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

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

$$\begin{aligned}\frac{\partial H1_{final}}{\partial H1} &= \frac{\partial(\frac{1}{1+e^{-H1}})}{\partial H1} \\ &= \frac{e^{-H1}}{(1+e^{-H1})^2} \\ e^{-H1} \times (H1_{final})^2 \dots \dots \dots (30) \\ H1_{final} &= \frac{1}{1+e^{-H1}}\end{aligned}$$

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

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

$$\begin{aligned}&= \frac{1 - H1_{final}}{H1_{final}} \times (H1_{final})^2 \\ &= H1_{final} \times (1 - H1_{final}) \\ &= 0.593269992 \times (1 - 0.593269992) \\ \frac{\partial H1_{final}}{\partial H1} &= \mathbf{0.2413007085923199}\end{aligned}$$

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:

$$\begin{aligned}H1 &= H1_{final} \times w5 + H2_{final} \times w6 + b2 \dots \dots \dots (32) \\ \frac{\partial y1}{\partial w1} &= \frac{\partial(x1 \times w1 + x2 \times w3 + b1 \times 1)}{\partial w1} \\ &= x1\end{aligned}$$

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

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.

$$\begin{aligned}\frac{\partial E_{\text{total}}}{\partial w_1} &= \frac{\partial E_{\text{total}}}{\partial H_{1\text{final}}} \times \frac{\partial H_{1\text{final}}}{\partial H_1} \times \frac{\partial H_1}{\partial w_1} \\ &= 0.0364908241736793 \times 0.2413007085923199 \times 0.05 \\ \text{Error}_{w_1} &= \frac{\partial E_{\text{total}}}{\partial w_1} = 0.000438568 \dots \dots \dots (34)\end{aligned}$$

Now, we will calculate the updated weight $w_{1\text{new}}$ with the help of the following formula

$$\begin{aligned}w_{1\text{new}} &= w_1 - \eta \times \frac{\partial E_{\text{total}}}{\partial w_1} \quad \text{Here } \eta = \text{learning rate} = 0.5 \\ &= 0.15 - 0.5 \times 0.000438568 \\ w_{1\text{new}} &= 0.149780716 \dots \dots \dots (35)\end{aligned}$$

In the same way, we calculate $w_{2\text{new}}$, $w_{3\text{new}}$, and w_4 and this will give us the following values

$$\begin{aligned}w_{1\text{new}} &= 0.149780716 \\ w_{2\text{new}} &= 0.19956143 \\ w_{3\text{new}} &= 0.24975114 \\ w_{4\text{new}} &= 0.29950229\end{aligned}$$

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.

Deep learning frameworks and libraries (e.g., TensorFlow/Keras, PyTorch).

Deep Learning Frameworks

Keras, TensorFlow and PyTorch are among the top three frameworks that are preferred by Data Scientists as well as beginners in the field of Deep Learning. This comparison on Keras vs TensorFlow vs PyTorch will provide you with a crisp knowledge about the top Deep Learning Frameworks and help you find out which one is suitable for you. In this blog you will get a complete insight into the above three frameworks in the following sequence:

- [Introduction to Keras, TensorFlow & PyTorch](#)

- [Comparison Factors](#)
- [Final Verdict](#)

Introduction

Keras



Keras is an open source [neural network](#) library written in Python. It is capable of running on top of TensorFlow. It is designed to enable fast experimentation with **deep neural networks**.

TensorFlow



[TensorFlow](#) is an open-source software library for dataflow programming across a range of tasks. It is a symbolic math library that is used for **machine learning** applications like neural networks.

PyTorch



[PyTorch](#) is an open source **machine learning** library for Python, based on Torch. It is used for applications such as **natural language processing** and was developed by Facebook's AI research group.

Comparison Factors

All the three frameworks are related to each other and also have certain basic differences that distinguishes them from one another.

So lets have a look at the **parameters** that distinguish them:

- [Level of API](#)

- [Speed](#)
- [Architecture](#)
- [Debugging](#)
- [Dataset](#)
- [Popularity](#)

Level of API



Keras is a **high-level API** capable of running on top of TensorFlow, CNTK and Theano. It has gained favor for its ease of use and syntactic simplicity, facilitating fast development.

TensorFlow is a framework that provides both **high and low level** APIs. Pytorch, on the other hand, is a **lower-level API** focused on direct work with array expressions. It has gained immense interest in the last year, becoming a preferred solution for academic research, and applications of deep learning requiring optimizing custom expressions.

Speed



The performance is comparatively **slower** in **Keras** whereas **Tensorflow** and **PyTorch** provide a similar pace which is fast and suitable for **high performance**.

Architecture



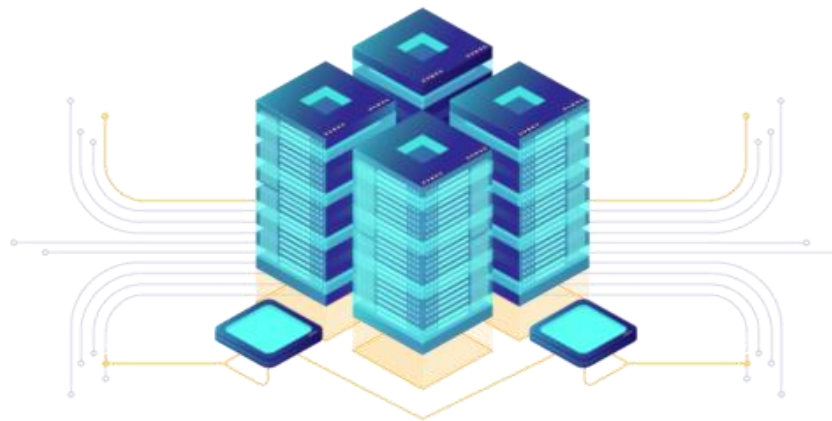
Keras has a **simple** architecture. It is more readable and concise . Tensorflow on the other hand is not very easy to use even though it provides Keras as a framework that makes work easier. PyTorch has a **complex** architecture and the readability is less when compared to Keras.

Debugging



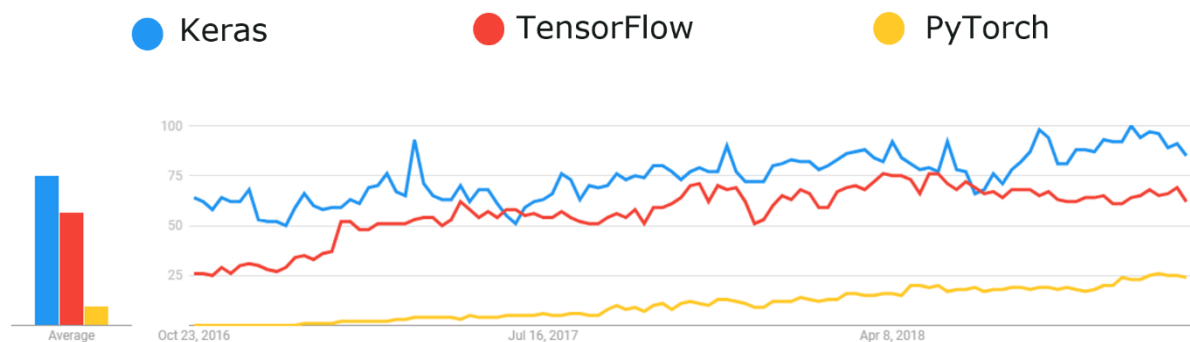
In keras, there is usually very **less frequent** need to debug simple networks. But in case of Tensorflow, it is quite **difficult** to perform debugging. **Pytorch** on the other hand has **better debugging capabilities** as compared to the other two.

Dataset



Keras is usually used for **small datasets** as it is comparatively slower. On the other hand, TensorFlow and PyTorch are used for **high performance** models and **large datasets** that require fast execution.

Popularity



With the increasing demand in the field of **Data Science**, there has been an enormous growth of **Deep learning technology** in the industry. With this, all the three frameworks have gained quite a lot of popularity. **Keras** tops the list followed by TensorFlow and PyTorch. It has gained immense popularity due to its **simplicity** when compared to the other two.

These were the parameters that distinguish all the three frameworks but there is no absolute answer to which one is better. The choice ultimately comes down to

- Technical background
- Requirements and
- Ease of Use

Final Verdict

Now coming to the final verdict of Keras vs TensorFlow vs PyTorch let's have a look at the situations that are most **preferable** for each one of these three deep learning frameworks



Keras is most suitable for:

- Rapid Prototyping
- Small Dataset
- Multiple back-end support



TensorFlow is most suitable for:

- Large Dataset
- High Performance
- Functionality
- [Object Detection](#)



PyTorch is most suitable for:

- Flexibility
- Short Training Duration
- Debugging capabilities

Now with this, we come to an end of this comparison on **Keras vs TensorFlow vs PyTorch**. I Hope you guys enjoyed this article and understood which Deep Learning Framework is most suitable for you.

*Now that you have understood the comparison between Keras, TensorFlow and PyTorch, check out the **AI and Deep Learning With Tensorflow** by Edureka, a trusted online learning company with a network of more than 250,000 satisfied learners spread across the globe. This Certification Training is curated by industry*

professionals as per the industry requirements & demands. You will master concepts such as SoftMax function, Autoencoder Neural Networks, Restricted Boltzmann Machine (RBM) and work with libraries like Keras & TFLearn.

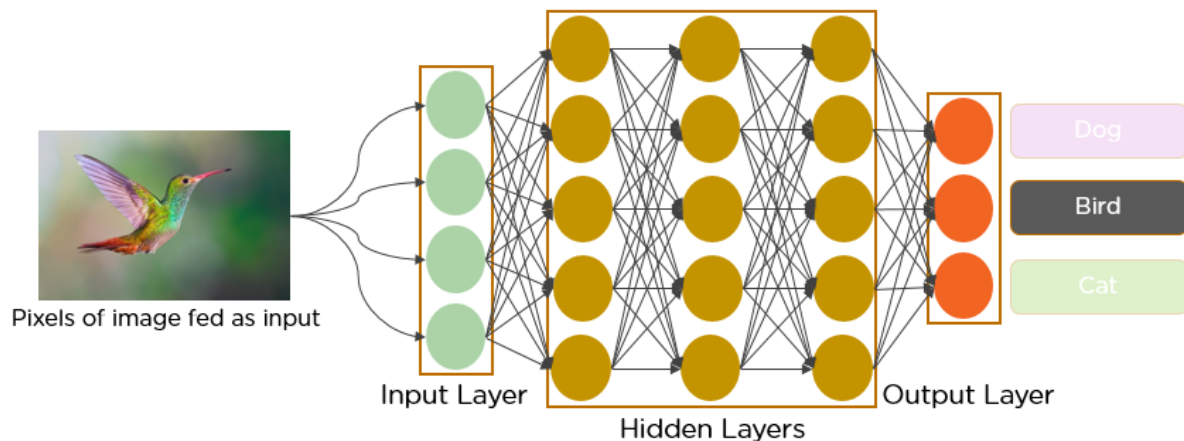
Also, discover your full abilities in becoming an AI and ML professional through our [Artificial Intelligence Course](#). Learn about various AI-related technologies like Machine Learning, Deep Learning, Computer Vision, Natural Language Processing, Speech Recognition, and Reinforcement learning.

UNIT-II:

CONVOLUTION NEURAL NETWORK (CNN): Introduction to CNNs and their applications in computer vision, CNN basic architecture, Activation functions-sigmoid, tanh, ReLU, Softmax layer, Types of pooling layers, Training of CNN in TensorFlow, various popular CNN architectures: VGG, Google Net, ResNet etc, Dropout, Normalization, Data augmentation

Introduction to CNNs and their applications in computer vision:

Deep Learning has proved to be a very powerful tool because of its ability to handle large amounts of data. The interest to use hidden layers has surpassed traditional techniques, especially in pattern recognition. One of the most popular deep neural networks is Convolutional Neural Networks (also known as CNN or ConvNet) in deep learning, especially when it comes to Computer Vision applications.



Since the 1950s, the early days of AI, researchers have struggled to make a system that can understand visual data. In the following years, this field came to be known as Computer Vision. In 2012, computer vision took a quantum leap when a group of researchers from the University of Toronto developed an AI model that surpassed the best image recognition algorithms, and that too by a large margin.

The AI system, which became known as AlexNet (named after its main creator, Alex Krizhevsky), won the 2012 ImageNet computer vision contest with an amazing 85 percent accuracy. The runner-up scored a modest 74 percent on the test.

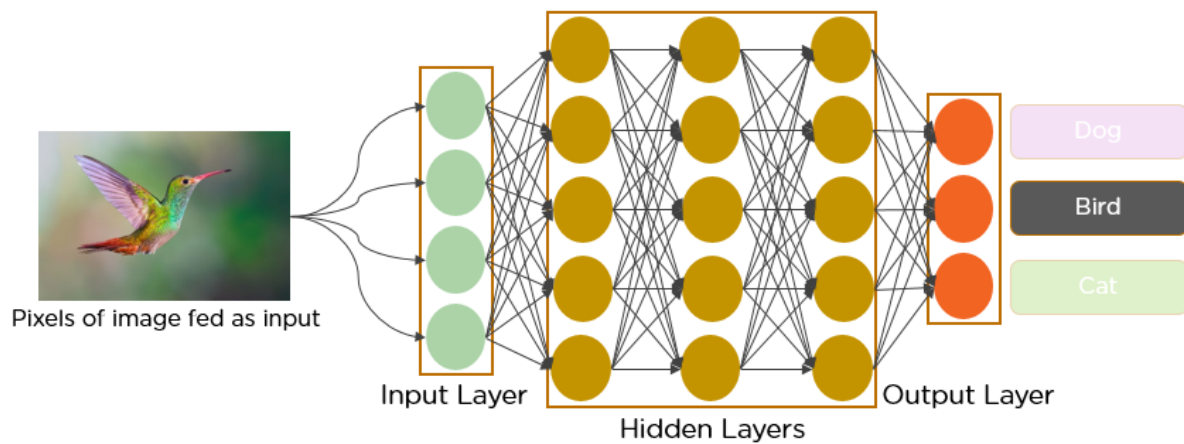
At the heart of AlexNet was Convolutional Neural Networks a special type of neural network that roughly imitates human vision.

Background of CNNs

CNN's were first developed and used around the 1980s. The most that a CNN could do at that time was recognize handwritten digits. It was mostly used in the postal sectors to read zip codes, pin codes, etc. The important thing to remember about any deep learning model is that it requires a large amount of data to train and also requires a lot of computing resources. This was a major drawback for CNNs at that period and hence CNNs were only

limited to the postal sectors and it failed to enter the world of machine learning.

In the past few decades, Deep Learning has proved to be a very powerful tool because of its ability to handle large amounts of data. The interest to use hidden layers has surpassed traditional techniques, especially in pattern recognition. One of the most popular deep neural networks is Convolutional Neural Networks (also known as CNN or ConvNet) in deep learning, especially when it comes to Computer Vision applications.



Since the 1950s, the early days of AI, researchers have struggled to make a system that can understand visual data. In the following years, this field came to be known as Computer Vision. In 2012, computer vision took a quantum leap when a group of researchers from the University of Toronto developed an AI model that surpassed the best image recognition algorithms, and that too by a large margin.

The AI system, which became known as AlexNet (named after its main creator, Alex Krizhevsky), won the 2012 ImageNet computer vision contest with an amazing 85 percent accuracy. The runner-up scored a modest 74 percent on the test.

At the heart of AlexNet was Convolutional Neural Networks a special type of neural network that roughly imitates human vision. Over the years CNNs have become a very important part of many Computer Vision applications and hence a

part of any computer vision course online. So let's take a look at the workings of CNNs or CNN algorithm in deep learning.

- [Background of CNNs](#)
- [What Is a CNN?](#)
- [How does it work?](#)
- [What Is a Pooling Layer?](#)
- [Limitations of CNNs](#)
- [Frequently Asked Questions](#)

Background of CNNs

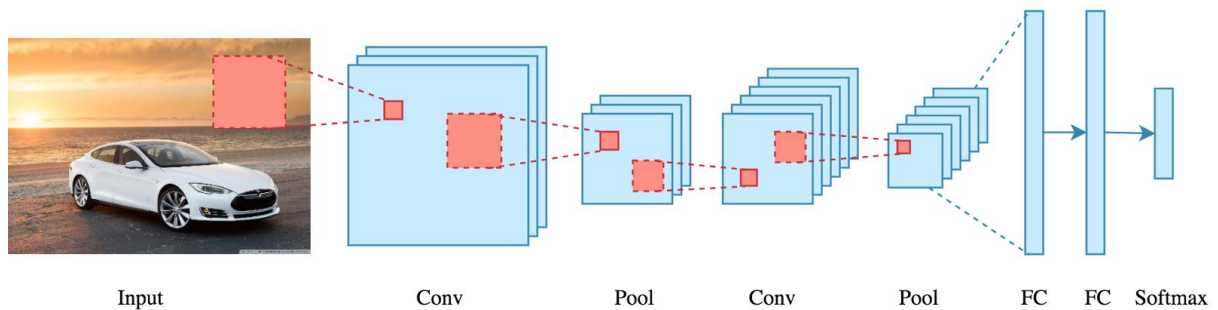
CNN's were first developed and used around the 1980s. The most that a CNN could do at that time was recognize handwritten digits. It was mostly used in the postal sectors to read zip codes, pin codes, etc. The important thing to remember about any deep learning model is that it requires a large amount of data to train and also requires a lot of computing resources. This was a major drawback for CNNs at that period and hence CNNs were only limited to the postal sectors and it failed to enter the world of machine learning.

In 2012 Alex Krizhevsky realized that it was time to bring back the branch of deep learning that uses multi-layered neural networks. The availability of large sets of data, to be more specific ImageNet datasets with millions of labeled images and an abundance of computing resources enabled researchers to revive CNNs.

What Is a CNN?

In deep learning, a **convolutional neural network (CNN/ConvNet)** is a class of deep neural networks, most commonly applied to analyze visual imagery. Now when we think of a neural network we think about matrix multiplications but that is not the case with ConvNet. It uses a special technique called Convolution. Now

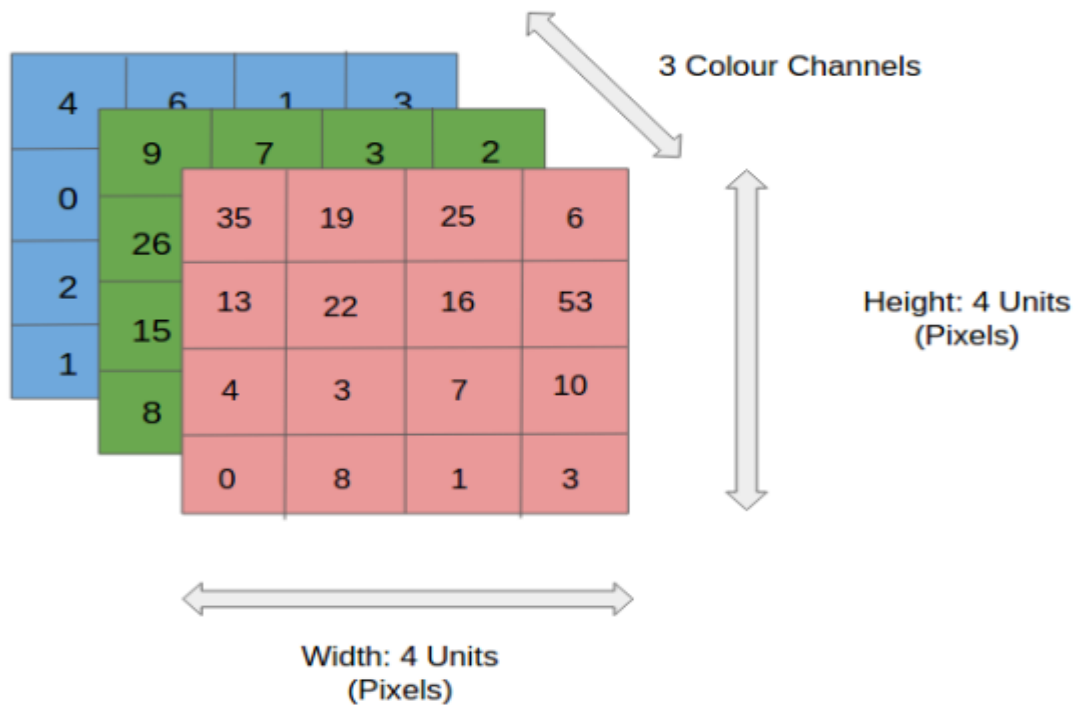
in mathematics **convolution** is a mathematical operation on two functions that produces a third function that expresses how the shape of one is modified by the other.



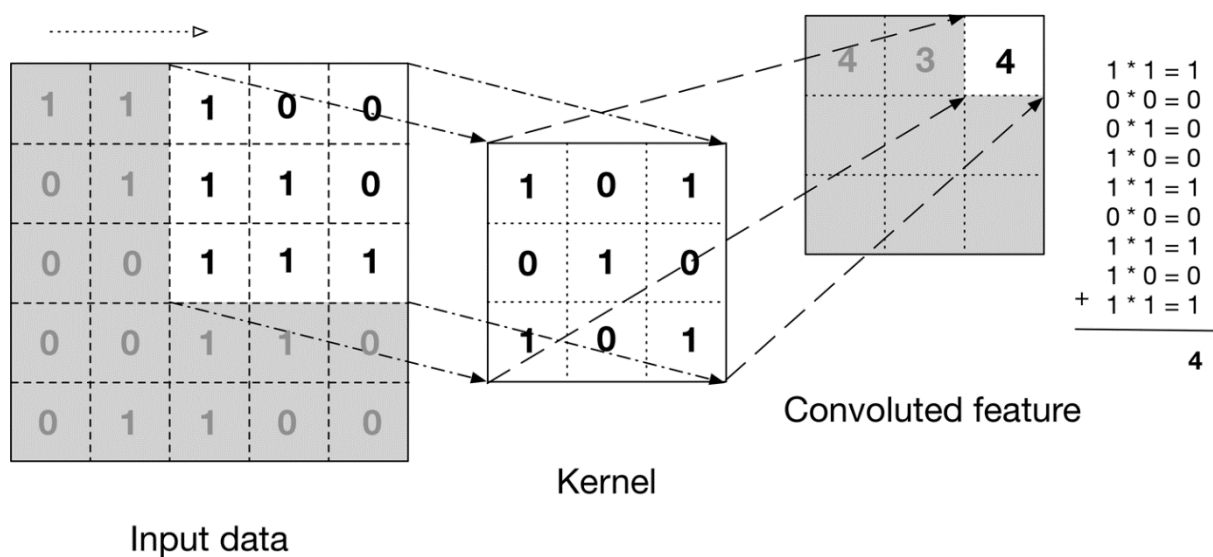
Bottom line is that the role of the ConvNet is to reduce the images into a form that is easier to process, without losing features that are critical for getting a good prediction.

How does it works?

Before we go to the working of CNN's let's cover the basics such as what is an image and how is it represented. An RGB image is nothing but a matrix of pixel values having three planes whereas a grayscale image is the same but it has a single plane. Take a look at this image to understand more.



For simplicity, let's stick with grayscale images as we try to understand how CNNs work.



The above image shows what a convolution is. We take a filter/kernel(3x3 matrix) and apply it to the input image to get the convolved feature. This convolved feature is passed on to the next layer.

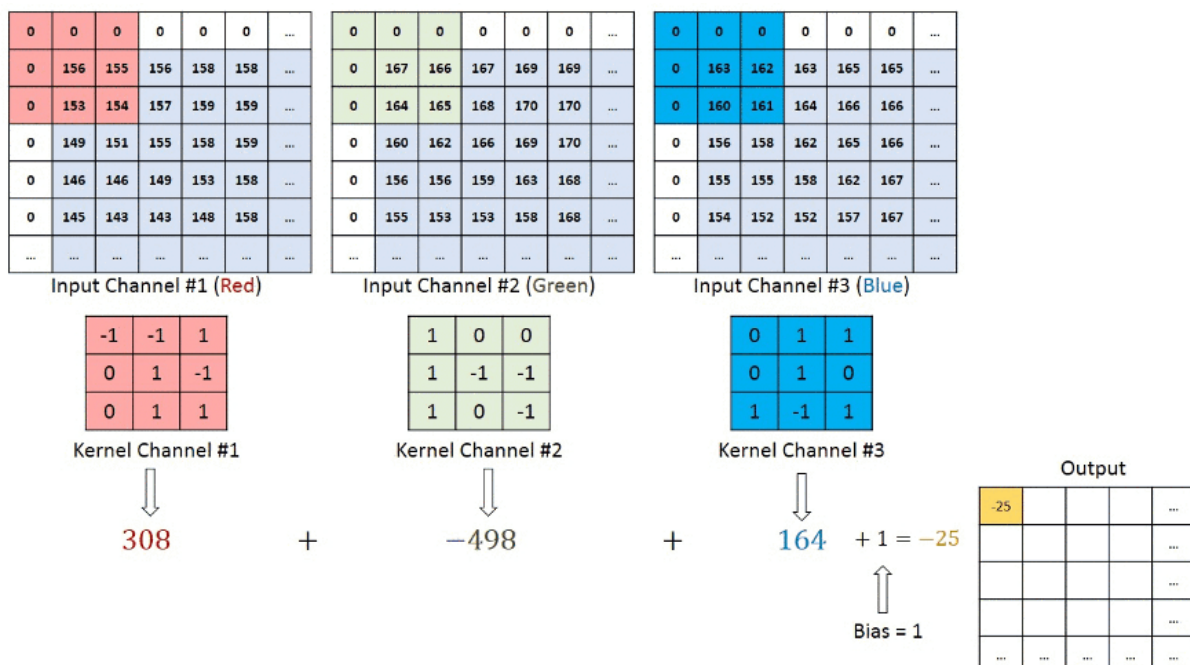
1 _{x1}	1 _{x0}	1 _{x1}	0	0
0 _{x0}	1 _{x1}	1 _{x0}	1	0
0 _{x1}	0 _{x0}	1 _{x1}	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved
Feature

In the case of RGB color, channel take a look at this animation to understand its working



Convolutional neural networks are composed of multiple layers of artificial neurons. Artificial neurons, a rough imitation of their biological counterparts,

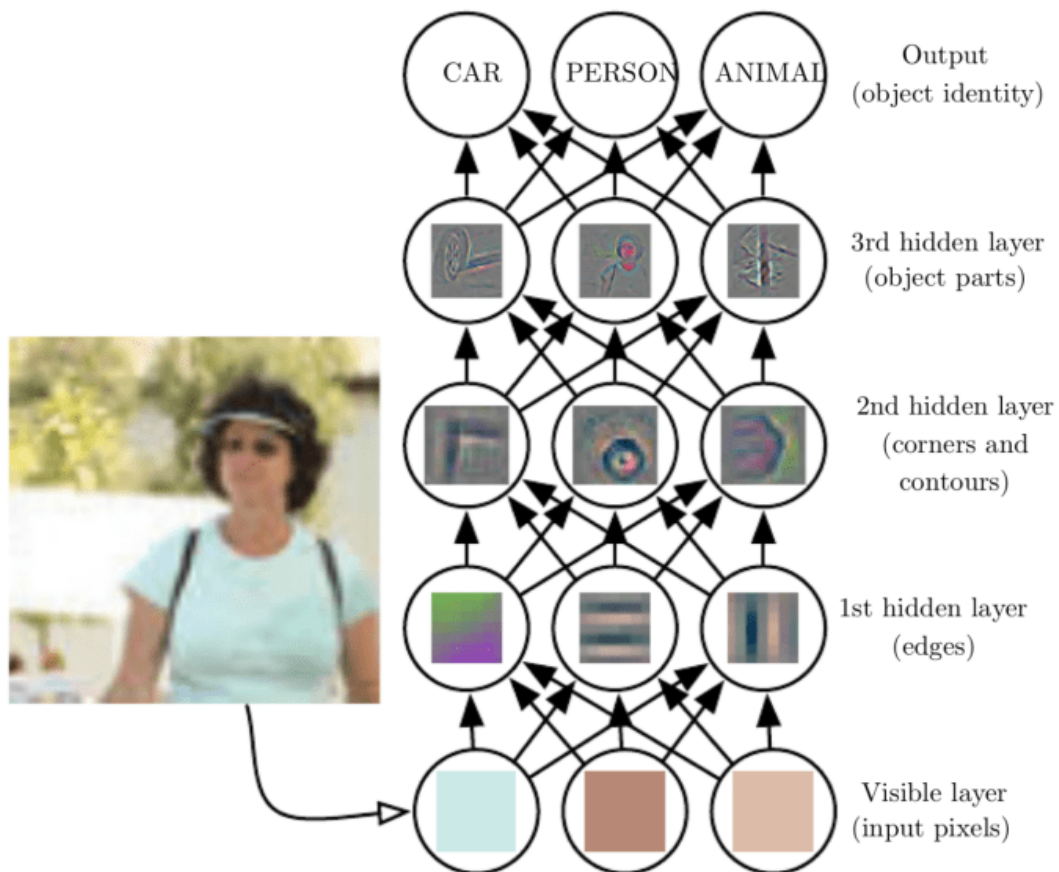
are mathematical functions that calculate the weighted sum of multiple inputs and outputs an activation value. When you input an image in a ConvNet, each layer generates several activation functions that are passed on to the next layer.

The first layer usually extracts basic features such as horizontal or diagonal edges. This output is passed on to the next layer which detects more complex features such as corners or combinational edges. As we move deeper into the network it can identify even more complex features such as objects, faces, etc.



Based on the activation map of the final convolution layer, the classification layer outputs a set of confidence scores (values between 0 and 1) that

specify how likely the image is to belong to a “class.” For instance, if you have a ConvNet that detects cats, dogs, and horses, the output of the final layer is the possibility that the input image contains any of those animals.



What Is a Pooling Layer?

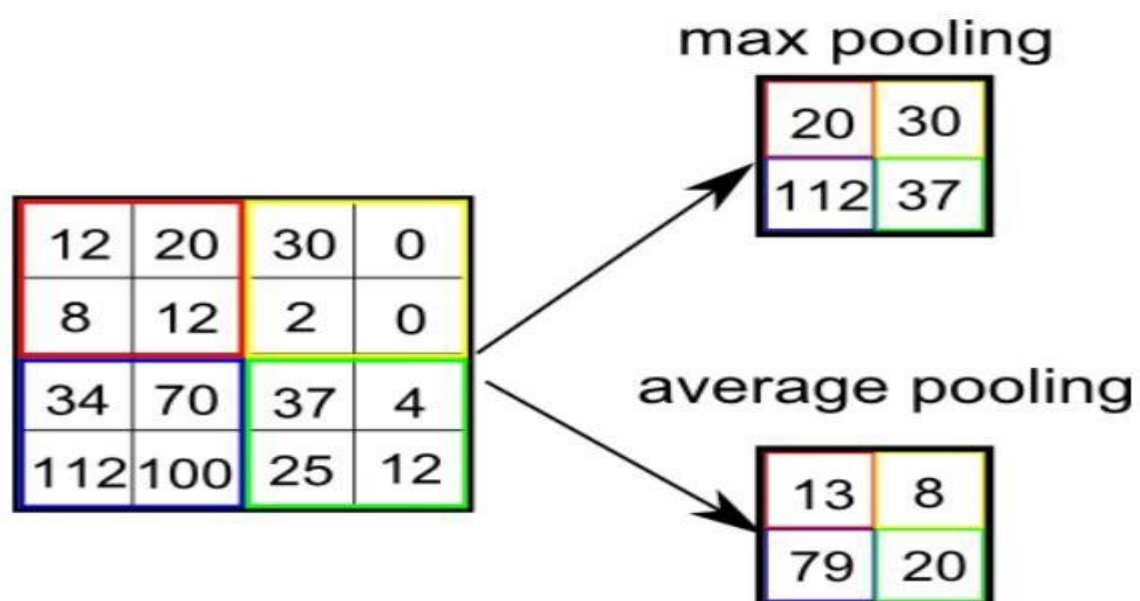
Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to **decrease the computational power required to process the data** by reducing the dimensions. There are two types of pooling average pooling and max pooling. I've only had experience with Max Pooling so far, I haven't faced any difficulties.

3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

So what we do in Max Pooling is we find the maximum value of a pixel from a portion of the image covered by the kernel. Max Pooling also performs as a **Noise Suppressant**. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction.

On the other hand, **Average Pooling** returns the **average of all the values** from the portion of the image covered by the Kernel. Average Pooling simply performs dimensionality reduction as a noise suppressing mechanism. Hence, we can say that **Max Pooling performs a lot better than Average Pooling**.



Benefits of Using CNNs for Machine and Deep Learning

Deep learning is a form of machine learning that requires a neural network with a minimum of three layers. Networks with multiple layers are more accurate than single-layer networks. Deep learning applications often use CNNs or RNNs (recurrent neural networks).

The CNN architecture is especially useful for image recognition and image classification, as well as other computer vision tasks because they can process large amounts of data and produce highly accurate predictions. CNNs can learn the features of an object through multiple iterations, eliminating the need for manual feature engineering tasks like feature extraction.

It is possible to retrain a CNN for a new recognition task or build a new model based on an existing network with trained weights. This is known as transfer learning. This enables ML model developers to apply CNNs to different use cases without starting from scratch.

What Are Convolutional Neural Networks (CNNs)?

A Convolutional Neural Network (CNN) is a type of deep learning algorithm specifically designed for image processing and recognition tasks. Compared to alternative classification models, CNNs require less preprocessing as they can automatically learn hierarchical feature representations from raw input images. They excel at assigning importance to various objects and features within the images through convolutional layers, which apply filters to detect local patterns.

The connectivity pattern in CNNs is inspired by the visual cortex in the human brain, where neurons respond to specific regions or receptive fields in the visual space. This architecture enables CNNs to effectively capture spatial relationships and patterns in images. By stacking multiple convolutional and pooling layers, CNNs can learn increasingly complex features, leading to high accuracy in tasks like image classification, object detection, and segmentation.

Convolutional Neural Network Architecture Model

Convolutional neural networks are known for their superiority over other artificial neural networks, given their ability to process visual, textual, and audio data. The CNN architecture comprises three main layers: convolutional layers, pooling layers, and a fully connected (FC) layer.

There can be multiple convolutional and pooling layers. The more layers in the network, the greater the complexity and (theoretically) the accuracy of the machine learning model. Each additional layer that processes the input data increases the model's ability to recognize objects and patterns in the data.

The Convolutional Layer

Convolutional layers are the key building block of the network, where most of the computations are carried out. It works by applying a filter to the input data to identify features. This filter, known as a feature detector, checks the image input's receptive fields for a given feature. This operation is referred to as convolution.

The filter is a two-dimensional array of weights that represents part of a 2-dimensional image. A filter is typically a 3×3 matrix, although there are other possible sizes. The filter is applied to a region within the input image and calculates a dot product between the pixels, which is fed to an output array. The filter then shifts and repeats the process until it has covered the whole image. The final output of all the filter processes is called the feature map.

The CNN typically applies the ReLU (Rectified Linear Unit) transformation to each feature map after every convolution to introduce nonlinearity to the ML model. A convolutional layer is typically followed by a pooling layer. Together, the convolutional and pooling layers make up a convolutional block.

Additional convolution blocks will follow the first block, creating a hierarchical structure with later layers learning from the earlier layers. For example, a CNN model might train to detect cars in images. Cars can be viewed as the sum of their parts, including the wheels, boot, and windscreen. Each feature of a car equates to a low-level pattern identified by the neural network, which then combines these parts to create a high-level pattern.

The Pooling Layers

A pooling or down sampling layer reduces the dimensionality of the input. Like a convolutional operation, pooling operations use a filter to sweep the whole input image, but it doesn't use weights. The filter instead uses an aggregation function to populate the output array based on the receptive field's values.

There are two key types of pooling:

- **Average pooling:** The filter calculates the receptive field's average value when it scans the input.
- **Max pooling:** The filter sends the pixel with the maximum value to populate the output array. This approach is more common than average pooling.

Pooling layers are important despite causing some information to be lost, because they help reduce the complexity and increase the efficiency of the CNN. It also reduces the risk of overfitting.

The Fully Connected Layer

The final layer of a CNN is a fully connected layer.

The FC layer performs classification tasks using the features that the previous layers and filters extracted.

Instead of ReLU functions, the FC layer typically uses a softmax function that classifies inputs more appropriately and produces a probability score between 0 and 1.

(OR)

Basic Architecture of CNN:

Basic Architecture

There are two main parts to a CNN architecture

- A convolution tool that separates and identifies the various features of the image for analysis in a process called as Feature Extraction.
- The network of feature extraction consists of many pairs of convolutional or pooling layers.
- A fully connected layer that utilizes the output from the convolution process and predicts the class of the image based on the features extracted in previous stages.
- This CNN model of feature extraction aims to reduce the number of features present in a dataset. It creates new features which summarises the existing features contained in an original set of features. There are many CNN layers as shown in the CNN architecture diagram.

Convolution Layers

There are three types of layers that make up the CNN which are the convolutional layers, pooling layers, and fully-connected (FC) layers. When these layers are stacked, a CNN architecture will be formed. In addition to these three layers, there are two more important parameters which are the dropout layer and the activation function which are defined below.

1. Convolutional Layer

This layer is the first layer that is used to extract the various features from the input images. In this layer, the mathematical operation of convolution is performed between the input image and a filter of a particular size $M \times M$. By sliding the filter over the input image, the dot product is taken between the filter and the parts of the input image with respect to the size of the filter ($M \times M$).

The output is termed as the Feature map which gives us information about the image such as the corners and edges. Later, this feature map is fed to other layers to learn several other features of the input image.

The convolution layer in CNN passes the result to the next layer once applying the convolution operation in the input. Convolutional layers in

CNN benefit a lot as they ensure the spatial relationship between the pixels is intact.

2. Pooling Layer

In most cases, a Convolutional Layer is followed by a Pooling Layer. The primary aim of this layer is to decrease the size of the convolved feature map to reduce the computational costs. This is performed by decreasing the connections between layers and independently operates on each feature map. Depending upon method used, there are several types of Pooling operations. It basically summarises the features generated by a convolution layer.

In Max Pooling, the largest element is taken from feature map. Average Pooling calculates the average of the elements in a predefined sized Image section. The total sum of the elements in the predefined section is computed in Sum Pooling. The Pooling Layer usually serves as a bridge between the Convolutional Layer and the FC Layer.

This CNN model generalises the features extracted by the convolution layer, and helps the networks to recognise the features independently. With the help of this, the computations are also reduced in a network.

3. Fully Connected Layer

The Fully Connected (FC) layer consists of the weights and biases along with the neurons and is used to connect the neurons between two different layers. These layers are usually placed before the output layer and form the last few layers of a CNN Architecture.

In this, the input image from the previous layers are flattened and fed to the FC layer. The flattened vector then undergoes few more FC layers where the mathematical functions operations usually take place. In this stage, the classification process begins to take place. The reason two layers are connected is that two fully connected layers will perform better than a single connected layer. These layers in CNN reduce the human supervision

4. Dropout

Usually, when all the features are connected to the FC layer, it can cause overfitting in the training dataset. Overfitting occurs when a particular model works so well on the training data causing a negative impact in the model's performance when used on a new data.

To overcome this problem, a dropout layer is utilised wherein a few neurons are dropped from the neural network during training process resulting in reduced size of the model. On passing a dropout of 0.3, 30% of the nodes are dropped out randomly from the neural network.

Dropout results in improving the performance of a machine learning model as it prevents overfitting by making the network simpler. It drops neurons from the neural networks during training.

5. Activation Functions

Finally, one of the most important parameters of the CNN model is the activation function. They are used to learn and approximate any kind of continuous and complex relationship between variables of the network. In simple words, it decides which information of the model should fire in the forward direction and which ones should not at the end of the network.

It adds non-linearity to the network. There are several commonly used activation functions such as the ReLU, Softmax, tanH and the Sigmoid functions. Each of these functions have a specific usage. For a binary classification CNN model, sigmoid and softmax functions are preferred a for a multi-class classification, generally softmax us used. In simple terms, activation functions in a CNN model determine whether a neuron should be activated or not. It decides whether the input to the work is important or not to predict using mathematical operations.

Activate functions:

3 Types of Neural Networks Activation Functions

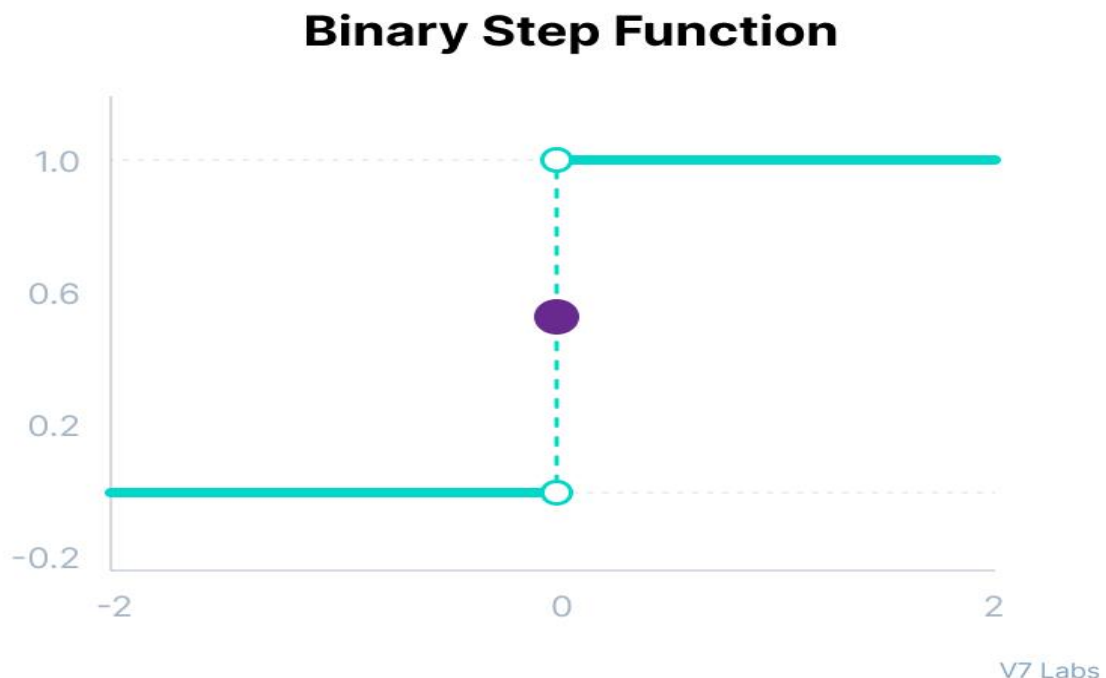
Now, as we've covered the essential concepts, let's go over the most popular neural networks activation functions.

Binary Step Function

Binary step function depends on a threshold value that decides whether a neuron should be activated or not.

The input fed to the activation function is compared to a certain threshold; if the input is greater than it, then the neuron is activated, else it is deactivated, meaning that its output is not passed on to the next hidden layer.

Binary Step Function



Mathematically it can be represented as:

Binary step

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

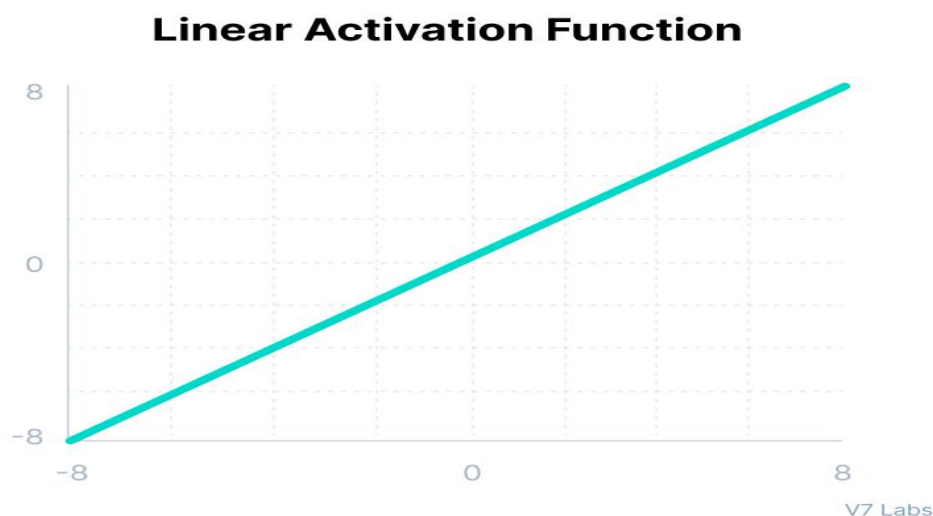
Here are some of the limitations of binary step function:

- It cannot provide multi-value outputs—for example, it cannot be used for multi-class classification problems.
- The gradient of the step function is zero, which causes a hindrance in the backpropagation process.

Linear Activation Function

The linear activation function, also known as "no activation," or "identity function" (multiplied x1.0), is where the activation is proportional to the input.

The function doesn't do anything to the weighted sum of the input, it simply spits out the value it was given.



Linear Activation Function

Mathematically it can be represented as:

$$\textbf{Linear}$$
$$f(x) = x$$

However, a linear activation function has two major problems :

- It's not possible to use backpropagation as the derivative of the function is a constant and has no relation to the input x .
- All layers of the neural network will collapse into one if a linear activation function is used. No matter the number of layers in the neural network, the last layer will still be a linear function of the first layer. So, essentially, a linear activation function turns the neural network into just one layer.

Non-Linear Activation Functions

The linear activation function shown above is simply a linear regression model.

Because of its limited power, this does not allow the model to create complex mappings between the network's inputs and outputs.

Non-linear activation functions solve the following limitations of linear activation functions:

- They allow backpropagation because now the derivative function would be related to the input, and it's possible to go back and understand which weights in the input neurons can provide a better prediction.
- They allow the stacking of multiple layers of neurons as the output would now be a non-linear combination of input passed through multiple layers. Any output can be represented as a functional computation in a neural network.

Now, let's have a look at ten different non-linear neural networks activation functions and their characteristics.

Non-Linear Neural Networks Activation Functions

Sigmoid / Logistic Activation Function

This function takes any real value as input and outputs values in the range of 0 to 1.

The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0, as shown below.



Sigmoid/Logistic Activation Function

Mathematically it can be represented as:

Sigmoid / Logistic

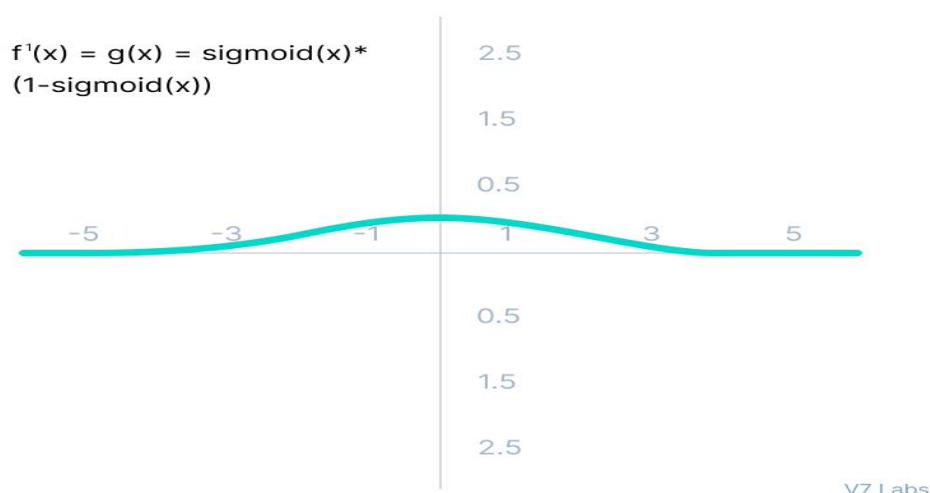
$$f(x) = \frac{1}{1 + e^{-x}}$$

Here's why sigmoid/logistic activation function is one of the most widely used functions:

- It is commonly used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice because of its range.
- The function is differentiable and provides a smooth gradient, i.e., preventing jumps in output values. This is represented by an S-shape of the sigmoid activation function.

The limitations of sigmoid function are discussed below:

- The derivative of the function is $f'(x) = \text{sigmoid}(x) * (1 - \text{sigmoid}(x))$.



The derivative of the Sigmoid Activation Function

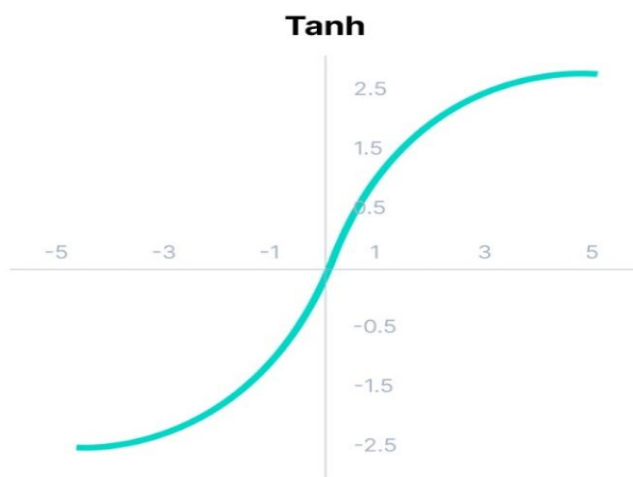
As we can see from the above Figure, the gradient values are only significant for range -3 to 3, and the graph gets much flatter in other regions.

It implies that for values greater than 3 or less than -3, the function will have very small gradients. As the gradient value approaches zero, the network ceases to learn and suffers from the *Vanishing gradient* problem.

- The output of the logistic function is not symmetric around zero. So the output of all the neurons will be of the same sign. This makes the [training of the neural network](#) more difficult and unstable.

Tanh Function (Hyperbolic Tangent)

Tanh function is very similar to the sigmoid/logistic activation function, and even has the same S-shape with the difference in output range of -1 to 1. In Tanh, the larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.



Tanh Function (Hyperbolic Tangent)

Mathematically it can be represented as:

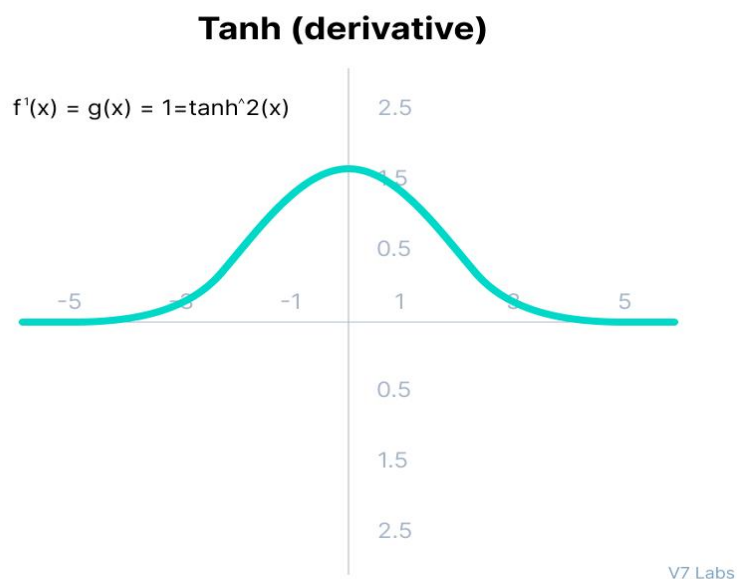
$$\text{Tanh}$$
$$f(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$$

Advantages of using this activation function are:

- The output of the tanh activation function is Zero centered; hence we can easily map the output values as strongly negative, neutral, or strongly positive.

- Usually used in hidden layers of a neural network as its values lie between -1 to 1; therefore, the mean for the hidden layer comes out to be 0 or very close to it. It helps in centering the data and makes learning for the next layer much easier.

Have a look at the gradient of the tanh activation function to understand its limitations.



Gradient of the Tanh Activation Function

As you can see— it also faces the problem of vanishing gradients similar to the sigmoid activation function. Plus the gradient of the tanh function is much steeper as compared to the sigmoid function.

💡 Note: **Although both sigmoid and tanh face vanishing gradient issue, tanh is zero centered, and the gradients are not restricted to move in a certain direction. Therefore, in practice, tanh nonlinearity is always preferred to sigmoid nonlinearity.**

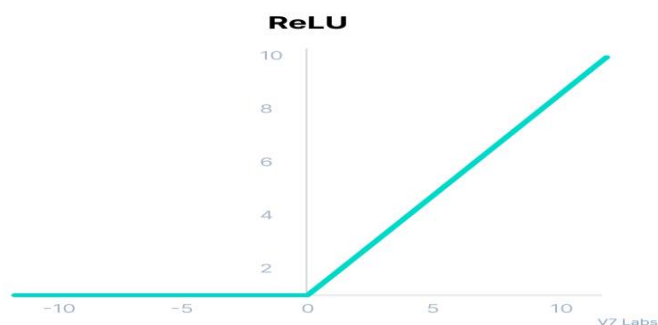
ReLU Function

ReLU stands for Rectified Linear Unit.

Although it gives an impression of a linear function, ReLU has a derivative function and allows for backpropagation while simultaneously making it computationally efficient.

The main catch here is that the ReLU function does not activate all the neurons at the same time.

The neurons will only be deactivated if the output of the linear transformation is less than 0.



ReLU Activation Function

Mathematically it can be represented as:

ReLU

$$f(x) = \max(0, x)$$

The advantages of using ReLU as an activation function are as follows:

- Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh functions.
- ReLU accelerates the convergence of gradient descent towards the global minimum of the [loss function](#) due to its linear, non-saturating property.

The limitations faced by this function are:

- The Dying ReLU problem, which I explained below.

The Dying ReLU problem



The Dying ReLU problem

The negative side of the graph makes the gradient value zero. Due to this reason, during the backpropagation process, the weights and biases for some

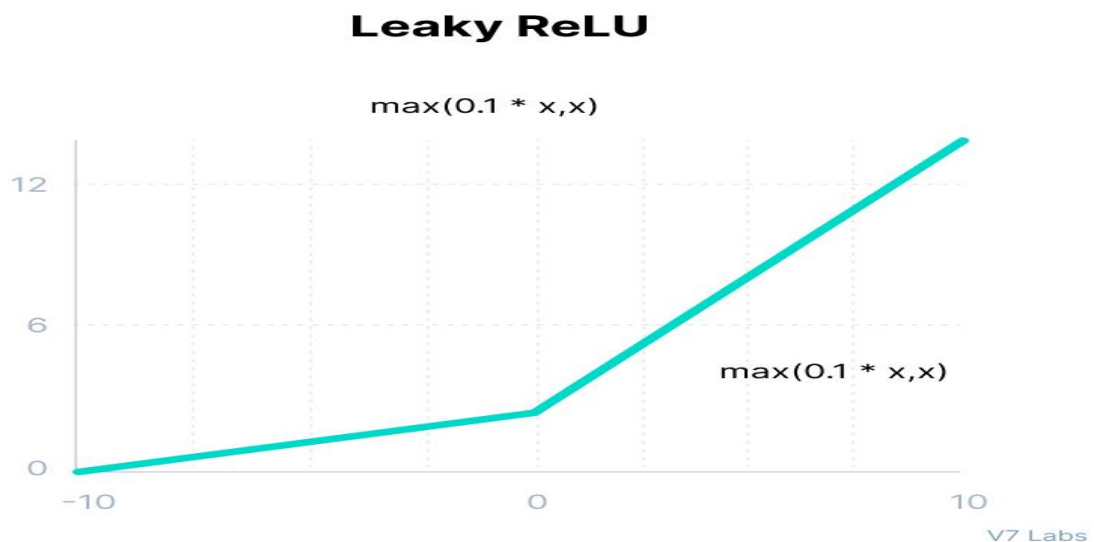
neurons are not updated. This can create dead neurons which never get activated.

- All the negative input values become zero immediately, which decreases the model's ability to fit or train from the data properly.

Note: For building the most reliable ML models, [split your data into train, validation, and test sets](#).

Leaky ReLU Function

Leaky ReLU is an improved version of ReLU function to solve the Dying ReLU problem as it has a small positive slope in the negative area.



Leaky ReLU

Mathematically it can be represented as:

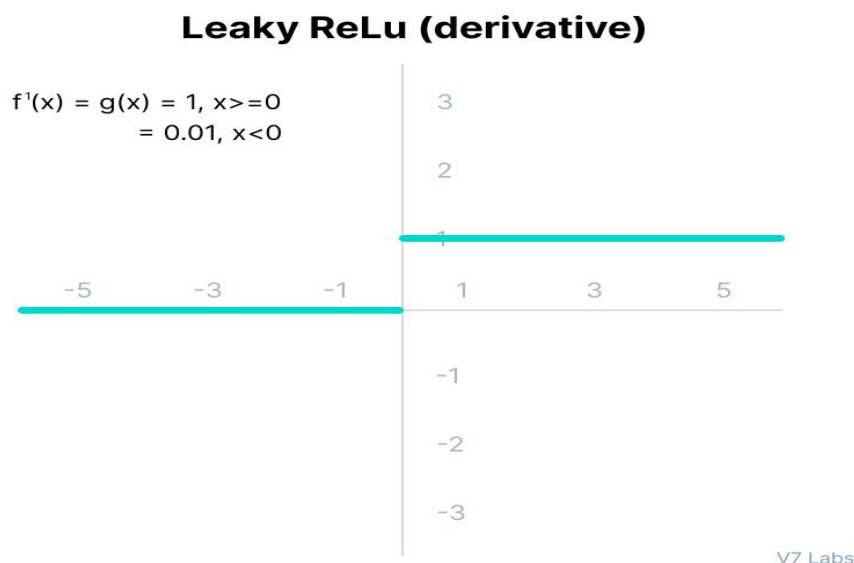
Leaky ReLU

$$f(x) = \max(0.1x, x)$$

The advantages of Leaky ReLU are same as that of ReLU, in addition to the fact that it does enable backpropagation, even for negative input values.

By making this minor modification for negative input values, the gradient of the left side of the graph comes out to be a non-zero value. Therefore, we would no longer encounter dead neurons in that region.

Here is the derivative of the Leaky ReLU function.



The derivative of the Leaky ReLU function

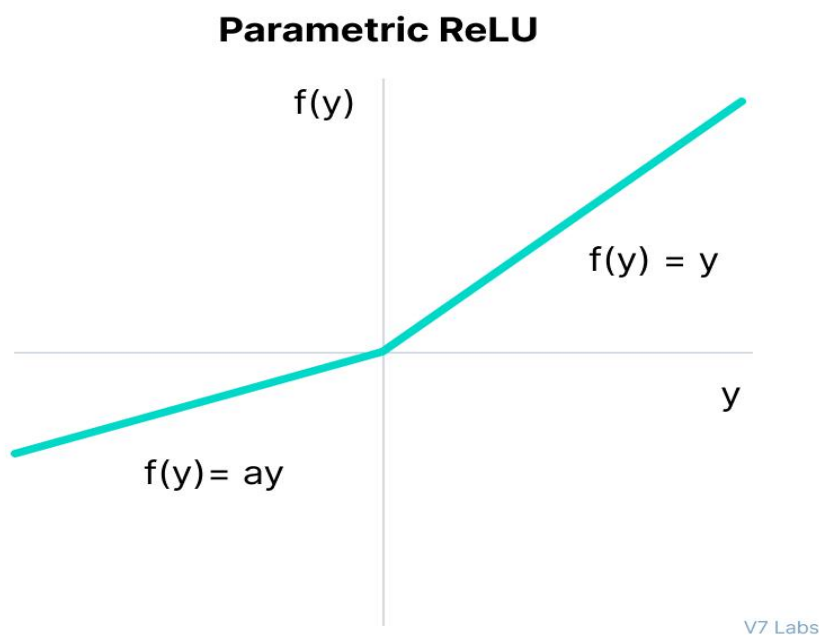
The limitations that this function faces include:

- The predictions may not be consistent for negative input values.
- The gradient for negative values is a small value that makes the learning of model parameters time-consuming.

Parametric ReLU Function

Parametric ReLU is another variant of ReLU that aims to solve the problem of gradient's becoming zero for the left half of the axis.

This function provides the slope of the negative part of the function as an argument a . By performing backpropagation, the most appropriate value of a is learnt.



Parametric ReLU

Mathematically it can be represented as:

Parametric ReLU

$$f(x) = \max(ax, x)$$

Where "a" is the slope parameter for negative values.

The parameterized ReLU function is used when the leaky ReLU function still fails at solving the problem of dead neurons, and the relevant information is not successfully passed to the next layer.

This function's limitation is that it may perform differently for different problems depending upon the value of slope parameter **a**.

Types of pooling Layers:

A Convolutional neural network(CNN) is a special type of Artificial Neural Network that is usually used for image recognition and processing due to its ability to recognize patterns in images. It eliminates the need to extract features from visual data manually. It learns images by sliding a filter of some size on them and learning not just the features from the data but also keeps Translation invariance.

The typical structure of a CNN consists of three basic layers

1. **Convolutional layer:** These layers **generate a feature map** by sliding a filter over the input image and recognizing patterns in images.
2. **Pooling layers:** These layers **downsample the feature map** to introduce Translation invariance, which reduces the overfitting of the CNN model.
3. **Fully Connected Dense Layer:** This layer contains the **same number of units as the number of classes** and the output activation function such as “softmax” or “sigmoid”

What are Pooling layers?

Pooling layers are one of the building blocks of Convolutional Neural Networks. Where Convolutional layers **extract features** from images, Pooling layers **consolidate the features** learned by CNNs. Its purpose is to gradually shrink the representation's spatial dimension to minimize the number of parameters and computations in the network.

Why are Pooling layers needed?

The feature map produced by the filters of Convolutional layers is location-dependent. For example, If an object in an image has shifted a bit it might not be recognizable by the Convolutional layer. So, it means that the feature map records the precise positions of features in the input. What pooling layers provide is “Translational Invariance” which makes the CNN invariant to translations, i.e., even if the input of the CNN is translated, the CNN will still be able to recognize the features in the input.

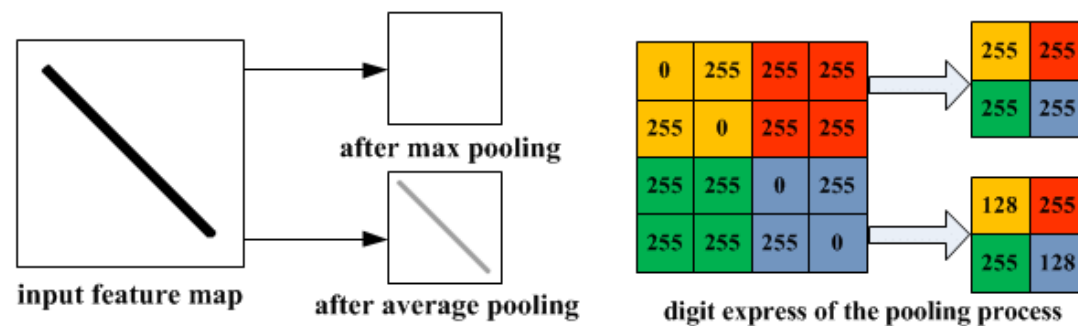
In all cases, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change—Page 342, Deep Learning by Ian Goodfellow, 2016.

How do Pooling layers achieve that? A Pooling layer is added after the Convolutional layer(s), as seen in the structure of a CNN above. It downsamples the output of the Convolutional layers by sliding the filter of some size with some stride size and calculating the maximum or average of the input.

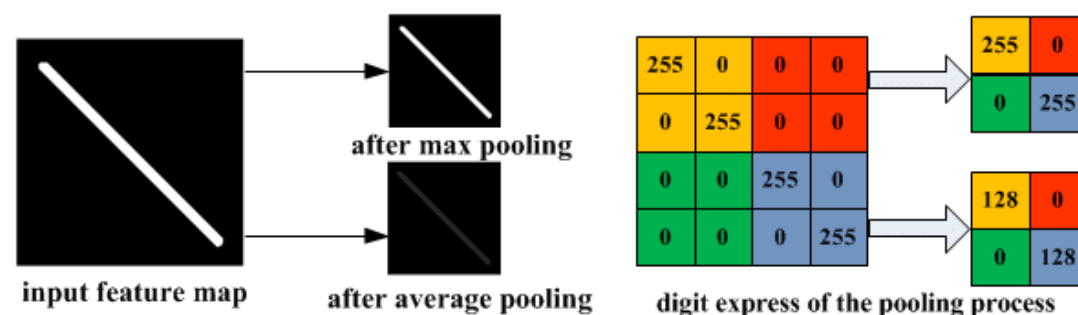
There are **two types of poolings** that are used:

1. **Max pooling:** This works by selecting the maximum value from every pool. Max Pooling retains the **most prominent** features of the feature map, and the returned image is sharper than the original image.

2. **Average pooling:** This pooling layer works by getting the average of the pool. Average pooling retains the **average values** of features of the feature map. It smoothes the image while keeping the essence of the feature in an image.



(a) Illustration of max pooling drawback



(b) Illustration of average pooling drawback

Image source

Let's explore the working of Pooling Layers using TensorFlow. Create a NumPy array and reshape it.

Max Pooling

Create a MaxPool2D layer with pool_size=2 and strides=2. Apply the MaxPool2D layer to the matrix, and you will get the MaxPooled output in the tensor form. By applying it to the matrix, the Max pooling layer will go through the matrix by computing the max of each 2×2 pool with a jump of 2. Print the shape of the tensor. Use tf.squeeze to remove dimensions of size 1 from the shape of a tensor.

Average Pooling

Create an AveragePooling2D layer with the same 2 pool_size and strides. Apply the AveragePooling2D layer to the matrix. By applying it to the matrix, the average pooling layer will go through the matrix by computing the average of 2×2 for each pool with a jump of 2. Print the shape of the matrix and Use tf.squeeze to convert the output into a readable form by removing all 1 size dimensions.

The GIF here explains how these pooling layers go through the input matrix and computes the maximum or average for max pooling and average pooling, respectively.

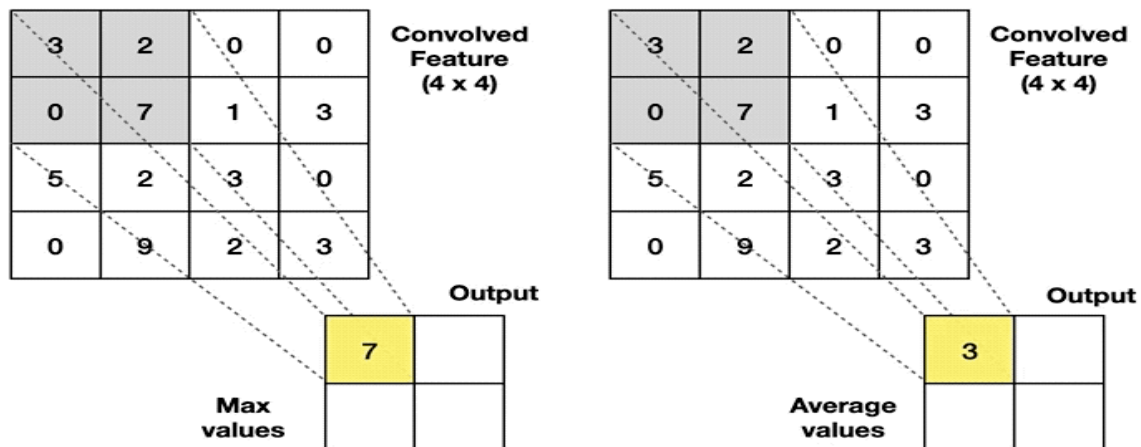
Max Pooling

Take the **highest** value from the area covered by the kernel

Average Pooling

Calculate the **average** value from the area covered by the kernel

Example: Kernel of size 2 x 2; stride=(2,2)



Max Pooling and Average Pooling being performed — [Source](#)

Global Pooling Layers

Global Pooling Layers often replace the classifier's fully connected or Flatten layer. The model instead ends with a convolutional layer that produces as many feature maps as there are target classes and performs global average pooling on each of the feature maps to combine each feature map into a single value.

Create the same NumPy array but with a different shape. By keeping the same shape as above, the Global Pooling layers will reduce them to one value.

Global Average Pooling

Considering a tensor of shape $h \times w \times n$, the output of the Global Average Pooling layer is a single value across $h \times w$ that summarizes the presence of the feature. Instead of downsizing the patches of the input feature map, the Global Average Pooling layer downsizes the whole $h \times w$ into 1 value by taking the average.

Global Max Pooling

With the tensor of shape $h \times w \times n$, the output of the Global Max Pooling layer is a single value across $h \times w$ that summarizes the presence of a feature. Instead of downsizing the patches of the input feature map, the Global Max Pooling layer downsizes the whole $h \times w$ into 1 value by taking the maximum.

Training of CNN in TensorFlow

Training of CNN in TensorFlow

The MNIST database (**Modified National Institute of Standard Technology database**) is an extensive database of handwritten digits, which is used for training various image processing systems. It was created by "**reintegrating**" samples from the original dataset of the **MNIST**.

If we are familiar with the building blocks of Connects, we are ready to build one with TensorFlow. We use the MNIST dataset for image classification.

Preparing the data is the same as in the previous tutorial. We can run code and jump directly into the architecture of CNN.

Here, we are executing our code in **Google Colab** (an online editor of machine learning).

We can go to TensorFlow editor through the below link: <https://colab.research.google.com>

These are the steps used to training the CNN (Convolutional Neural Network).

Steps:

Step 1: Upload Dataset

Step 2: The Input layer

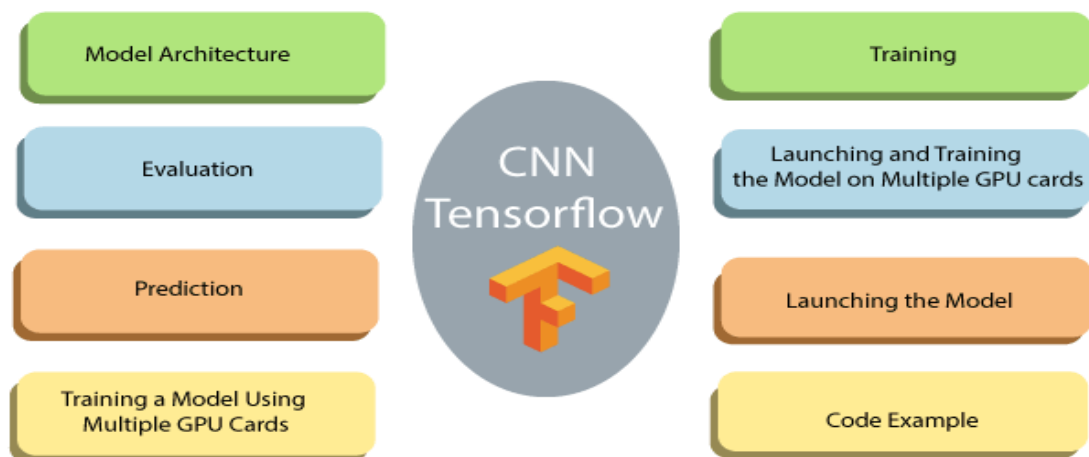
Step 3: Convolutional layer

Step 4: Pooling layer

Step 5: Convolutional layer and Pooling Layer

Step 6: Dense layer

Step 7: Logit Layer



Step 1: Upload Dataset

The MNIST dataset is available with scikit for learning in this URL (Unified Resource Locator). We can download it and store it in our downloads. We can upload it with `fetch_mldata` ('MNIST Original').

Create a test/train set

We need to split the dataset into **train_test_split**.

Scale the features

Finally, we scale the function with the help of **MinMax Scaler**.

1. **import** numpy as np
2. **import** tensorflow as tf
- 3.
4. from sklearn.datasets **import** fetch_mldata
5. #Change USERNAME by the username of the machine
6. ##Windows USER
7. mnist = fetch_mldata('C:\\Users\\USERNAME\\Downloads\\MNIST original')
8. ## Mac User
9. mnist = fetch_mldata('/Users/USERNAME/Downloads/MNIST original')
10. print(mnist.data.shape)
11. print(mnist.target.shape)
12. from sklearn.model_selection **import** train_test_split

```

13. A_train, A_test, B_train, B_test = train_test_split(mnist.data,mnist.target, test_size=0.2, random_state=45)
14. B_train = B_train.astype(int)
15. B_test = B_test.astype(int)
16. batch_size =len(X_train)
17. print(A_train.shape, B_train.shape,B_test.shape )
18. ## rescale
19. from sklearn.preprocessing import MinMaxScaler
20. scaler = MinMaxScaler()
21. # Train the Dataset
22. X_train_scaled = scaler.fit_transform(A_train.astype(np.float65))

1. #test the dataset
2. X_test_scaled = scaler.fit_transform(A_test.astype(np.float65))
3. feature_columns = [tf.feature_column.numeric_column('x',shape=A_train_scaled.shape[1:])]
4. X_train_scaled.shape[1:]

```

Defining the CNN (Convolutional Neural Network)

CNN uses filters on the pixels of any image to learn detailed patterns compared to global patterns with a traditional neural network. To create CNN, we have to define:

1. **A convolutional Layer:** Apply the number of filters to the feature map. After convolution, we need to use a relay activation function to add non-linearity to the network.
2. **Pooling Layer:** The next step after the Convention is to downsampling the maximum facility. The objective is to reduce the mobility of the feature map to prevent overfitting and improve the computation speed. Max pooling is a traditional technique, which splits feature maps into subfields and only holds maximum values.
3. **Fully connected Layers:** All neurons from the past layers are associated with the other next layers. The CNN has classified the label according to the features from convolutional layers and reduced with any pooling layer.

CNN Architecture

- **Convolutional Layer:** It applies 14 5x5 filters (extracting 5x5-pixel sub-regions),
- **Pooling Layer:** This will perform max pooling with a 2x2 filter and stride of 2 (which specifies that pooled regions do not overlap).

- **Convolutional Layer:** It applies 36 5x5 filters, with ReLU activation function
- **Pooling Layer:** Again, performs max Pooling with a 2x2 filter and stride of 2.
- **1,764 neurons**, with the dropout regularization rate of 0.4 (where the probability of 0.4 that any given element will be dropped in training)
- **Dense Layer (Logits Layer):** There are ten neurons, one for each digit target class (0-9).

Important modules to use in creating a CNN:

1. `Conv2d ()`. Construct a two-dimensional convolutional layer with the number of filters, filter kernel size, padding, and activation function like arguments.
2. `max_pooling2d ()`. Construct a two-dimensional pooling layer using the max-pooling algorithm.
3. `Dense ()`. Construct a dense layer with the hidden layers and units

We can define a function to build CNN.

Let's see in detail how to construct every building block before wrapping everything in the function.

Step 2: Input layer

1. `#Input layer`
2. `def cnn_model_fn(mode, features, labels):`
3. `input_layer = tf.reshape(tensor= features["x"],shape=[-1, 26, 26, 1])`

We need to define a tensor with the shape of the data. For that, we can use the **module `tf.reshape`**. In this module, we need to declare the tensor to reshape and to shape the tensor. The first argument is the feature of the data, that is defined in the argument of a function.

A picture has a width, a height, and a channel. The **MNIST** dataset is a monochromatic picture with the **28x28** size. We set the batch size into -1 in the shape argument so that it takes the shape of the features ["x"]. The advantage is to tune the batch size to hyperparameters. If the batch size is 7, the tensor feeds **5,488** values (**28 * 28 * 7**).

Step 3: Convolutional Layer

1. `# first CNN Layer`
2. `conv1 = tf.layers.conv2d(`
3. `inputs= input_layer,`
4. `filters= 18,`

5. `kernel_size= [7, 7],`
6. `padding="same",`
7. `activation=tf.nn.relu)`

The first convolutional layer has 18 filters with the kernel size of 7x7 with equal padding. The same padding has both the output tensor and input tensor have the same width and height. TensorFlow will add zeros in the rows and columns to ensure the same size.

We use the Relu activation function. The output size will be [28, 28, and 14].

Step 4: Pooling layer

The next step after the convolutional is pooling computation. The pooling computation will reduce the extension of the data. We can use the module `max_pooling2d` with a size of 3x3 and stride of 2. We use the previous layer as input. The output size can be [batch_size, 14, 14, and 15].

1. `##first Pooling Layer`
2. `pool1 = tf.layers.max_pooling2d (inputs=conv1,`
3. `pool_size=[3, 3], strides=2)`

Step 5: Pooling Layer and Second Convolutional Layer

The second CNN has exactly 32 filters, with the output size of [batch_size, 14, 14, 32]. The size of the pooling layer has the same as ahead, and output shape is [batch_size, 14, 14, and 18].

1. `conv2 = tf.layers.conv2d(`
2. `inputs=pool1,`
3. `filters=36,`
4. `kernel_size=[5, 5],`
5. `padding="same",`
6. `activation=tf.nn.relu)`
7. `pool2 = tf.layers.max_pooling2d (inputs=conv2, pool_size=[2, 2],strides=2).`

Step6: Fully connected (Dense) Layer

We have to define the fully-connected layer. The feature map has to be compressed before to be combined with the dense layer. We can use the module `reshape` with a size of **7*7*36**.

The dense layer will connect **1764** neurons. We add a Relu activation function and can add a Relu activation function. We add a dropout regularization term with a rate of 0.3, meaning 30 percent of the weights will be 0. The dropout takes place only along the training phase. The **cnn_model_fn()** has an argument mode to declare if the model needs to be trained or to be evaluated.

1. pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 36])
2. dense = tf.layers.dense(inputs=pool2_flat, units=7 * 7 * 36, activation=tf.nn.relu)
3. dropout = tf.layers.dropout(
4. inputs=dense, rate=0.3, training=mode == tf.estimator.ModeKeys.TRAIN)

Step 7: Logits Layer

Finally, we define the last layer with the prediction of model. The output shape is equal to the batch size 12, equal to the total number of images in the layer.

1. #Logit Layer
2. logits = tf.layers.dense(inputs=dropout, units=12)

We can create a dictionary that contains classes and the possibility of each class. The module returns the highest value with tf.argmax () if the logit layers. The softmax function returns the probability of every class.

various popular CNN architectures: VGG, Google Net, ResNet etc:

Types of Convolutional Neural Network Algorithms

LeNet LeNet is a pioneering CNN designed for recognizing handwritten characters. It was proposed by Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner in the late 1990s. LeNet consists of a series of convolutional and pooling layers, as well as a fully connected layer and softmax classifier. It was among the first successful applications of deep learning for computer vision. It has been used by banks to identify numbers written on cheques in grayscale input images.

VGG

VGG (Visual Geometry Group) is a research group within the Department of Engineering Science at the University of Oxford. The VGG group is well-known for its work in computer vision, particularly in the area of convolutional neural networks (CNNs).

One of the most famous contributions from the VGG group is the VGG model, also known as VGGNet. The VGG model is a deep neural network that achieved state-of-the-art performance on the ImageNet

Large Scale Visual Recognition Challenge in 2014, and has been widely used as a benchmark for image classification and object detection tasks.

The VGG model is characterized by its use of small convolutional filters (3×3) and deep architecture (up to 19 layers), which enables it to learn increasingly complex features from input images. The VGG model also uses max pooling layers to reduce the spatial resolution of the feature maps and increase the receptive field, which can improve its ability to recognize objects of varying scales and orientations.

The VGG model has inspired many subsequent research efforts in deep learning, including the development of even deeper neural networks and the use of residual connections to improve gradient flow and training stability.

ResNet

ResNet (short for “Residual Neural Network”) is a family of deep convolutional neural networks designed to overcome the problem of vanishing gradients that are common in very deep networks. The idea behind ResNet is to use “residual blocks” that allow for the direct propagation of gradients through the network, enabling the training of very deep networks.

A residual block consists of two or more convolutional layers followed by an activation function, combined with a shortcut connection that bypasses the convolutional layers and adds the original input directly to the output of the convolutional layers after the activation function.

This allows the network to learn residual functions that represent the difference between the convolutional layers’ input and output, rather than trying to learn the entire mapping directly. The use of residual blocks enables the training of very deep networks, with hundreds or thousands of layers, significantly alleviating the issue of vanishing gradients.

GoogLeNet

GoogLeNet is a deep convolutional neural network developed by researchers at Google. It was introduced in 2014 and won the ILSVRC (ImageNet Large-Scale Visual Recognition Challenge) that year, with a top-five error rate of 6.67%.

GoogLeNet is notable for its use of the Inception module, which consists of multiple parallel convolutional layers with different filter sizes, followed by a pooling layer, and concatenation of the outputs. This design allows the network to learn features at multiple scales and resolutions, while keeping the computational cost manageable. The network also includes auxiliary classifiers at intermediate layers, which encourage the network to learn more discriminative features and prevent overfitting.

GoogLeNet builds upon the ideas of previous convolutional neural networks, including LeNet, which was one of the first successful applications of deep learning in computer vision. However, GoogLeNet is much deeper and more complex than LeNet.

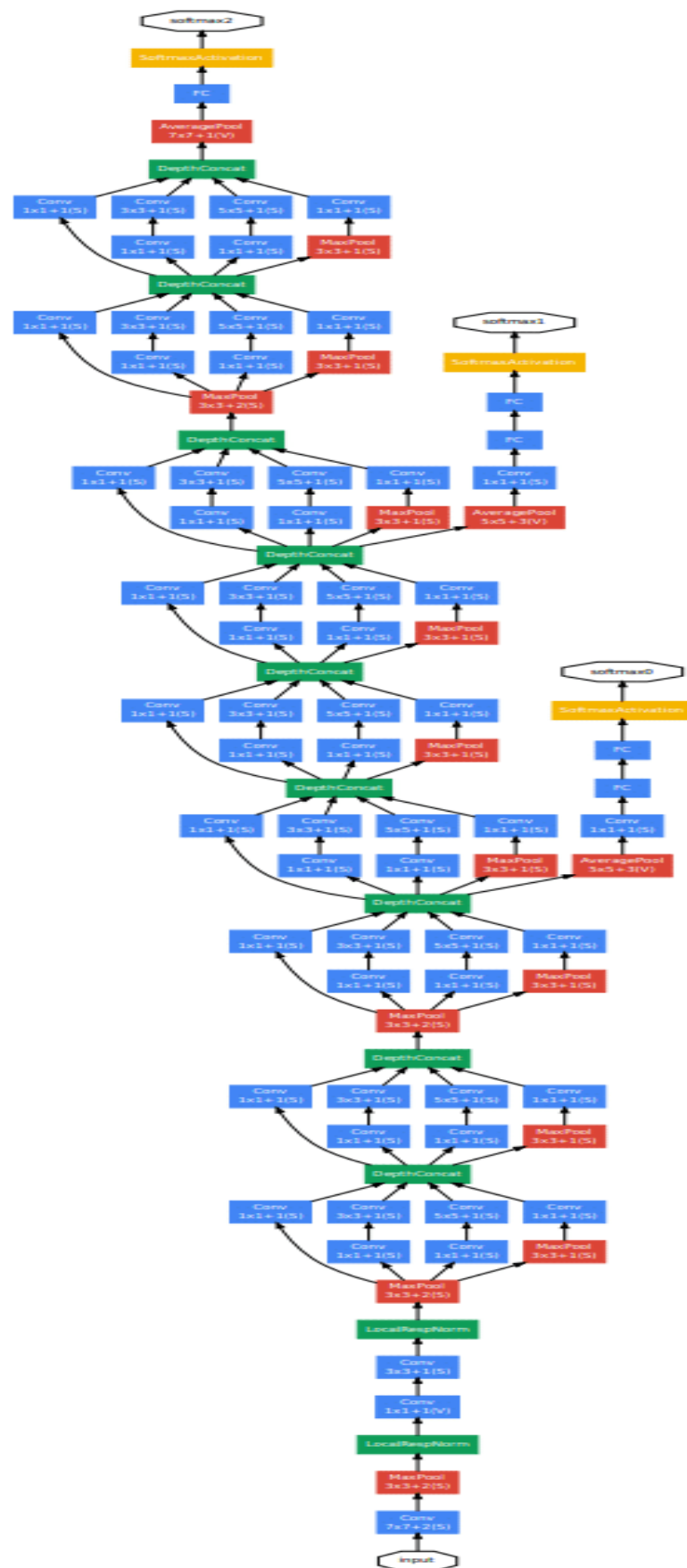


Figure 3: GoogLeNet network with all the bells and whistles

Dropout:

What is a Dropout?

The term “dropout” refers to dropping out the nodes (input and hidden layer) in a neural network (as seen in Figure 1). All the forward and backwards connections with a dropped node are temporarily removed, thus creating a new network architecture out of the parent network. The nodes are dropped by a dropout probability of p .

Let's try to understand with a given input x : {1, 2, 3, 4, 5} to the fully connected layer. We have a dropout layer with probability $p = 0.2$ (or keep probability = 0.8). During the forward propagation (training) from the input x , 20% of the nodes would be dropped, i.e. the x could become {1, 0, 3, 4, 5} or {1, 2, 0, 4, 5} and so on. Similarly, it applied to the hidden layers.

For instance, if the hidden layers have 1000 neurons (nodes) and a dropout is applied with drop probability = 0.5, then 500 neurons would be randomly dropped in every iteration (batch).

Generally, for the input layers, the keep probability, i.e. 1- drop probability, is closer to 1, 0.8 being the best as suggested by the authors. For the hidden layers, the greater the drop probability more sparse the model, where 0.5 is the most optimised keep probability, that states dropping 50% of the nodes.

So how does dropout solves the problem of overfitting?

How does it solve the Overfitting problem?

In the overfitting problem, the model learns the statistical noise. To be precise, the main motive of training is to decrease the loss function, given all the units (neurons). So in overfitting, a unit may change in a way that fixes up the mistakes of the other units. This leads to complex co-adaptations, which in turn leads to the overfitting problem because this complex co-adaptation fails to generalise on the unseen dataset.

Now, if we use dropout, it prevents these units to fix up the mistake of other units, thus preventing co-adaptation, as in every iteration the presence of a unit is highly unreliable. So by randomly dropping a few units (nodes), it forces the layers to take more or less responsibility for the input by taking a probabilistic approach.

This ensures that the model is getting generalised and hence reducing the overfitting problem.

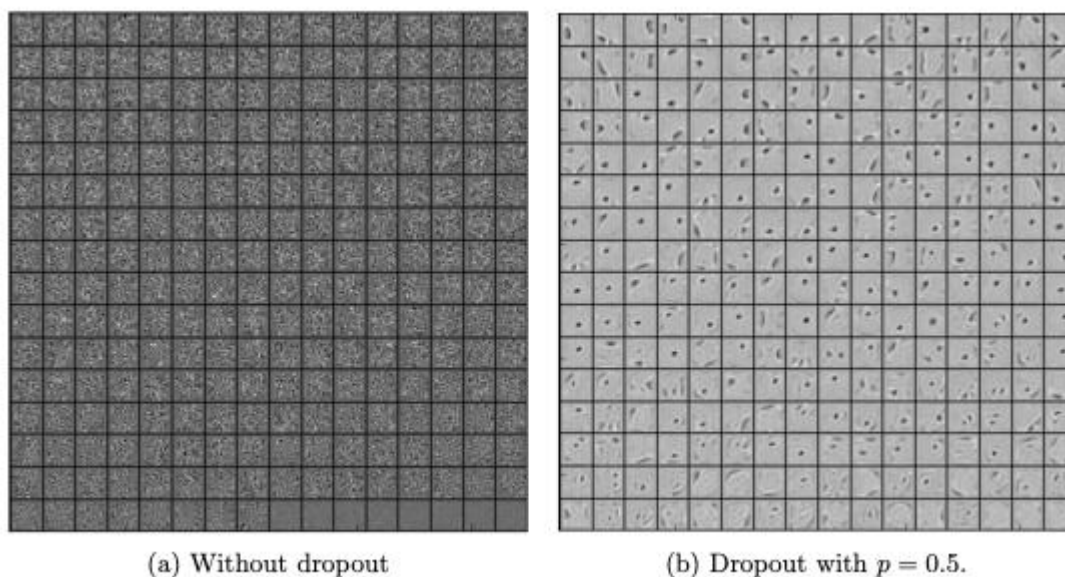


Figure 2: (a) Hidden layer features without dropout; (b) Hidden layer features with dropout

From figure 2, we can easily make out that the hidden layer with dropout is learning more of the generalised features than the co-adaptations in the layer without dropout. It is quite apparent, that dropout breaks such inter-unit relations and focuses more on generalisation.

Dropout Implementation

Enough of the talking! Let's head to the mathematical explanation of the dropout.

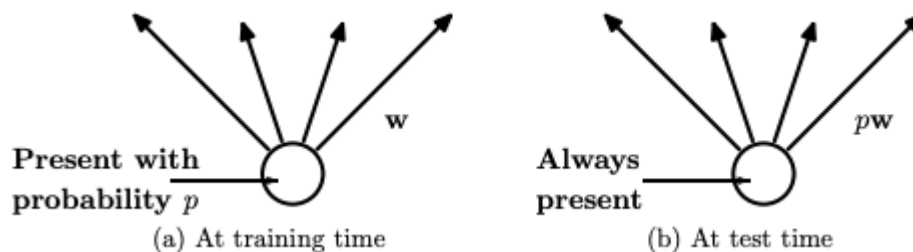


Figure 3: (a) A unit (neuron) during training is present with a probability p and is connected to the next layer with weights ' \mathbf{w} '; (b) A unit during inference/prediction is always present and is connected to the next layer with weights, ' \mathbf{pw} '

In the original implementation of the dropout layer, during training, a unit (node/neuron) in a layer is selected with a keep probability (1-drop probability). This creates a thinner architecture in the given training batch, and every time this architecture is different.

In the standard neural network, during the forward propagation we have the following equations:

$$z_i^{(l+1)} = \mathbf{w}_i^{(l+1)} \mathbf{y}^l + b_i^{(l+1)},$$

$$y_i^{(l+1)} = f(z_i^{(l+1)}),$$

Figure 4: Forward propagation of a standard neural network

where:

z : denote the vector of output from layer $(l + 1)$ before activation

y : denote the vector of outputs from layer l

w : weight of the layer l

b : bias of the layer l

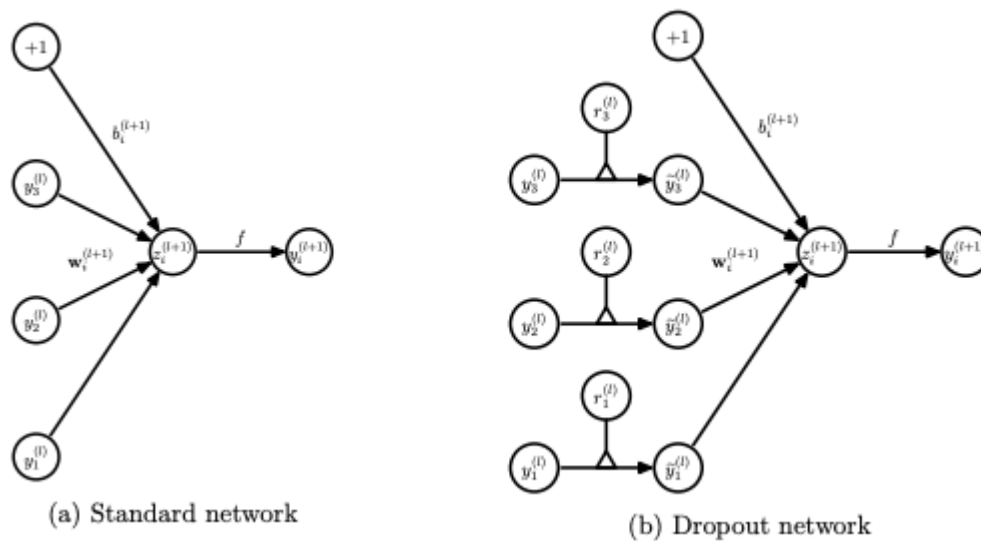
Further, with the activation function, z is transformed into the output for layer $(l+1)$.

Now, if we have a dropout, the forward propagation equations change in the following way:

$$\begin{aligned} r_j^{(l)} &\sim \text{Bernoulli}(p), \\ \tilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)}, \\ z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \tilde{\mathbf{y}}^{(l)} + b_i^{(l+1)}, \\ y_i^{(l+1)} &= f(z_i^{(l+1)}). \end{aligned}$$

Figure 5: Forward propagation of a layer with dropout

So before we calculate \mathbf{z} , the input to the layer is sampled and multiplied element-wise with the independent Bernoulli variables. \mathbf{r} denotes the Bernoulli random variables each of which has a probability p of being 1. Basically, \mathbf{r} acts as a mask to the input variable, which ensures only a few units are kept according to the keep probability of a dropout. This ensures that we have thinned outputs “ $\mathbf{y}(\text{bar})$ ”, which is given as an input to the layer during feed-forward propagation.



Training Deep Neural Networks is a difficult task that involves several problems to tackle. Despite their huge potential, they can be slow and be prone to overfitting. Thus, studies on methods to solve these problems are constant in Deep Learning research.

Batch Normalization – commonly abbreviated as Batch Norm – is one of these methods. Currently, it is a widely used technique in the field of Deep Learning. It improves the learning speed of Neural Networks and provides regularization, avoiding overfitting.

But why is it so important? How does it work? Furthermore, how can it be applied to non-regular networks such as Convolutional Neural Networks?

Normalization:

Normalization is a pre-processing technique used to standardize data. In other words, having different sources of data inside the same range. Not normalizing the data before training can cause problems in our network, making it drastically harder to train and decrease its learning speed.

For example, imagine we have a car rental service. Firstly, we want to predict a fair price for each car based on competitors' data. We have two features per car: the age in years and the total amount of kilometers it has been driven for. These can have very different ranges, ranging from 0 to 30 years, while distance could go from 0 up to hundreds of thousands of kilometers. We don't want features to have these differences in ranges, as the value with the higher range might bias our models into giving them inflated importance.

There are two main methods to normalize our data. The most straightforward method is to scale it to a range from 0 to 1:

The data point to normalize, the mean of the data set, the highest value, and the lowest value. This technique is generally used in the inputs of the data. The non-normalized data points with wide ranges can cause instability in Neural Networks. The relatively large inputs can cascade down to the layers, causing problems such as exploding gradients.

The other technique used to normalize data is forcing the data points to have a mean of 0 and a standard deviation of 1, using the following formula:

$$x_{normalized} = \frac{x - m}{s}$$

being the data point to normalize, the mean of the data set, and the standard deviation of the data set. Now, each data point mimics a standard normal distribution. Having all the features on this scale, none of them will have a bias, and therefore, our models will learn better.

In Batch Norm, we use this last technique to normalize batches of data inside the network itself.

Batch Normalization

Batch Norm is a normalization technique done between the layers of a Neural Network instead of in the raw data. It is done along mini-batches instead of the full data set. It serves to speed up training and use higher learning rates, making learning easier.

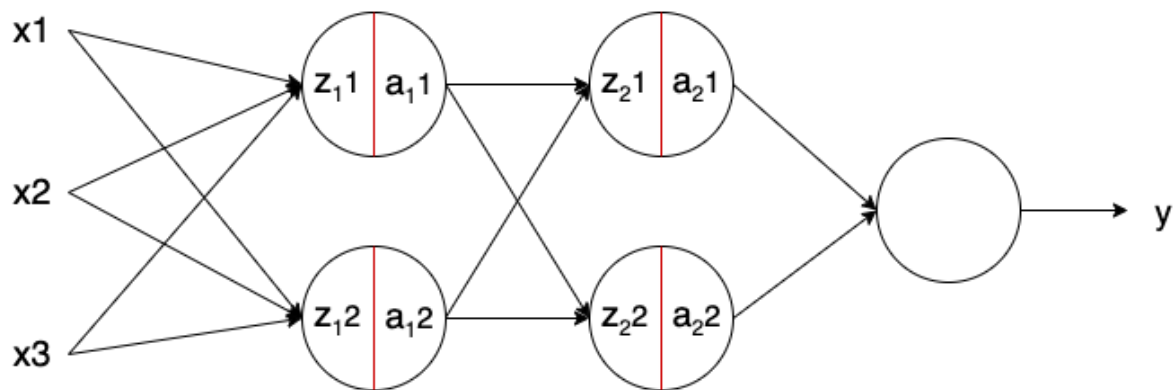
Following the technique explained in the previous section, we can define the normalization formula of Batch Norm as:

$$z^N = \left(\frac{z - m_z}{s_z} \right)$$

being m_z the mean of the neurons' output and s_z the standard deviation of the neurons' output.

How Is It Applied?

In the following image, we can see a regular feed-forward Neural Network: are the inputs, the output of the neurons, the output of the activation functions, and the output of the network:



Batch Norm – in the image represented with a red line – is applied to the neurons' output just before applying the activation function. Usually, a neuron without Batch Norm would be computed as follows:

$$z = g(w, x) + b; \quad a = f(z)$$

being the linear transformation of the neuron, the weights of the neuron, the bias of the neurons, and the activation function. The model learns the parameters and . Adding Batch Norm, it looks as:

$$z = g(w, x); \quad z^N = \left(\frac{z - m_z}{s_z} \right) \cdot \gamma + \beta; \quad a = f(z^N)$$

being the output of Batch Norm, the mean of the neurons' output, the standard deviation of the output of the neurons, and and learning parameters of Batch Norm. Note that the bias of the neurons () is removed. This is because as we subtract the mean m_z , any constant over the values of z – such as b – can be ignored as it will be subtracted by itself.

The parameters β and γ shift the mean and standard deviation, respectively. Thus, the outputs of Batch Norm over a layer result in a distribution with a mean and a standard deviation of γ . These values are learned over epochs and the other learning parameters, such as the weights of the neurons, aiming to decrease the loss of the model.

Data Augmentation:

What is Data Augmentation in a CNN:

Algorithms can use machine learning to identify different objects and classify them for image recognition. This evolving technology includes using Data Augmentation to produce better-performing models. Machine learning models need to identify an object in any condition, even if it is rotated, zoomed in, or a grainy image. Researchers needed an artificial way of adding training data with realistic modifications.

Data augmentation is the addition of new data artificially derived from existing training data. Techniques include resizing, flipping, rotating, cropping, padding, etc. It helps to address issues like overfitting and data scarcity, and it makes the model robust with better performance.

Data Augmentation provides many possibilities to alter the original image and can be useful to add enough data for larger models. It is important to learn the techniques of Data Augmentation and its advantages and disadvantages. In this post, I'll cover all the details you need and show you a Python example using PyTorch.

Data Augmentation in a CNN:

Convolutional Neural Networks (CNNs) can do amazing things if there is sufficient data. However, selecting the correct amount of training data for all of the features that need to be trained is a difficult question. If the user does not have enough, the network can overfit on the training data. Realistic images contain a variety of sizes, poses, zoom, lighting, noise, etc.

To make the network robust to these commonly encountered factors, the method of Data Augmentation is used. By rotating input images to different angles, flipping images along different axes, or translating/cropping the images the network will encounter these phenomena during training.

As more parameters are added to a CNN, it requires more examples to show to the machine learning model. Deeper networks can have higher performance, but the tradeoff is increased training data needs and increased training time.

Data Augmentation Techniques	Data Augmentation Factor
------------------------------	--------------------------

Data Augmentation Techniques	Data Augmentation Factor
Flipping	2-4x (in each direction)
Rotation	Arbitrary
Translation	Arbitrary
Scaling	Arbitrary
Salt and Pepper Noise Addition	At least 2x (depends on the implementation)

A table outlining the factor by which different methods multiply the existing training data.

Data Augmentation Techniques:

Some libraries use Data Augmentation by actually copying the training images and saving these copies as part of the total. This produces new training examples to feed to the machine learning model. Other libraries simply define a set of transforms to perform on the input training data. These transforms are applied randomly. As a result, the space the optimizer is searching is increased. This has the advantage that it does not require extra disk space to augment the training.

Image Data Augmentation is now a famous and common method used with CNNs and involves techniques such as:

- Flips
- Rotation (at 90 degrees and finer angles)
- Translation
- Scaling
- Salt and Pepper noise addition

Data Augmentation has even been used in applications like sound recognition. In the next sections, I'll cover these Data Augmentation methods in detail.

Flips:

By Flipping images, the optimizer will not become biased that particular features of an image are only on one side. To do this augmentation, the

original training image is flipped vertically or horizontally over one axis of the image. As a result, the features continually change directions.



Stella the Puppy sitting on a car seat



Stella the Puppy Flipped over the vertical axis.

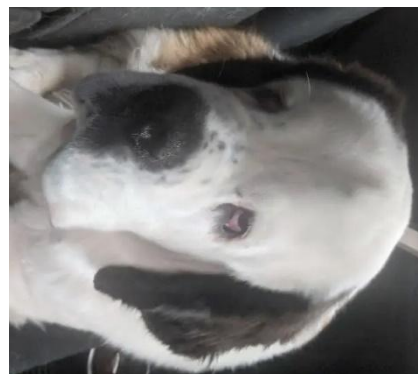
Flipping is a similar augmentation as rotation, however, it produces mirror images. A particular feature such as the head of a person either stays on top, on the left, on the right, or at the bottom of the image.

Rotation:

Rotation is an augmentation that is commonly performed at 90-degree angles but can even happen at smaller or minute angles if the need for more data is great. For rotation, the background color is commonly fixed so that it can blend when the image is rotated. Otherwise, the model can assume the background change is a distinct feature. This works best when the background is the same in all rotated images.



Stella the Puppy sitting on a car seat



Stella the Puppy rotated 90 degrees.

Specific features move in rotations. For example, the head of a person will be rotated 10, 22.7, or -8 degrees. However, rotation does not change the orientation of the feature and will not produce mirror images like flips. This helps models not consider the angle to be a distinct feature of the human.

Translation:

Translation of an image means shifting the main object in the image in various directions. For example, consider a person in the center with all their parts visible in the frame and take it as a base image. Next, shift the person to one corner with the legs cut from the bottom as one translated image.



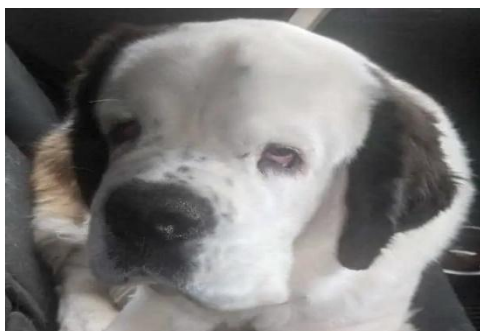
Stella the Puppy sitting on a car seat



Stella the Puppy translated and cropped so she's only partly visible.

Scaling:

Scaling provides more diversity in the training data of a machine learning model. Scaling the image will ensure that the object is recognized by the network regardless of how zoomed in or out the image is. Sometimes the object is tiny in the center. Sometimes, the object is zoomed in the image and even cropped at some parts.



Stella the Puppy sitting on a car seat Stella the Puppy scaled up to be even larger than she is in **real life.**

Salt and Pepper Noise Addition

Salt and pepper noise addition is the addition of black and white dots (looking like salt and pepper) to the image. This simulates dust and imperfections in real photos. Even if the camera of the photographer is blurry or has spots on it, the image would be better recognized by the model. The training data set is augmented to train the model with more realistic images.



Stella the Puppy sitting on a car seat Stella the Puppy with Salt and Pepper noise added to the **image**

Benefits of Data Augmentation in a CNN

There are many benefits of using Data Augmentation:

- Prediction improvement in a model becomes more accurate because Data Augmentation helps in recognizing samples the model has never seen before.
- There is enough data for the model to understand and train all the available parameters. This can be essential in applications where data collection is difficult.
- Helps prevent the model from overfitting due to Data Augmentation creating more variety in the data.
- Can save time in areas where collecting more data is time-consuming.
- Can reduce the cost required for collecting a variety of data if data collection is costly.

Drawbacks of Data Augmentation:

Data Augmentation is not useful when the variety required by the application cannot be artificially generated. For example, if one were training a bird recognition model and the training data contained only red birds. The training data could be augmented by generating pictures with the color of the bird varied.

However, the artificial augmentation method may not capture the realistic color details of birds when there is not enough variety of data to start with. For example, if

the augmentation method simply varied red for blue or green, etc. Realistic non-red birds may have more complex color variations and the model may fail to recognize the color. Having sufficient data is still important if one wants Data Augmentation to work properly.

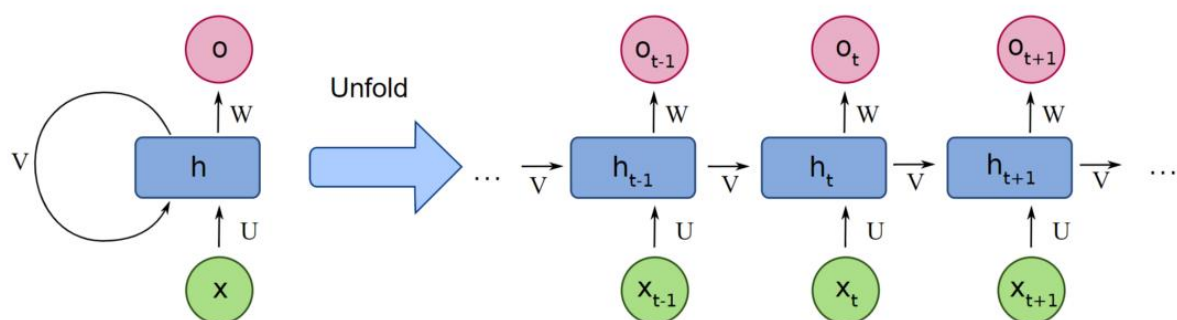
UNIT-III

RECURRENT NEURAL NETWORK (RNN): Introduction to RNNs and their applications in sequential data analysis, Back propagation through time (BPTT), Vanishing Gradient Problem, gradient clipping Long Short-Term Memory (LSTM) Networks, Gated Recurrent Units, Bidirectional LSTMs, Bidirectional RNNs.

Introduction to RNNs and their applications in sequential data analysis:

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*.

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.



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

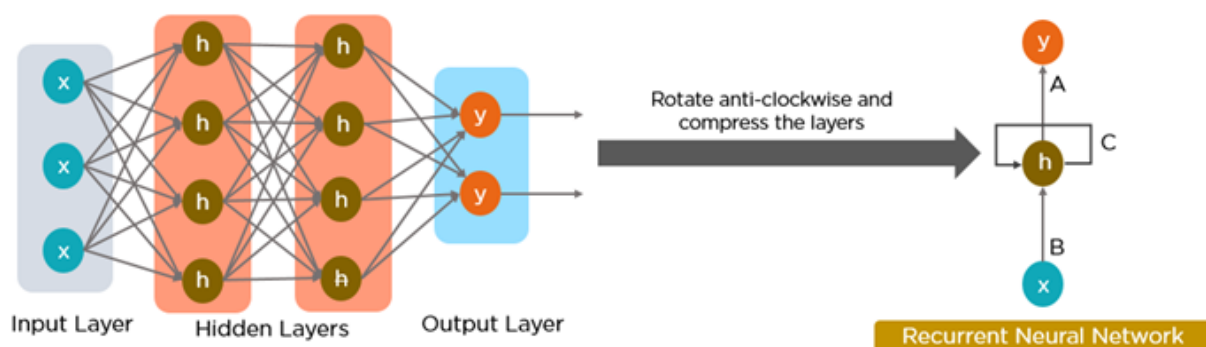
data other than sequential data, such as geographical or graphical data, because of its design.

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 with, has really pushed RNNs to the forefront.

What is a Recurrent Neural Network (RNN)?

Neural networks imitate the function of the human brain in the fields of AI, machine learning, and deep learning, allowing computer programs to recognize patterns and solve common issues.

RNNs are a type of neural network that can be used to model sequence data. RNNs, which are formed from feedforward networks, are similar to human brains in their behaviour. Simply said, recurrent neural networks can anticipate sequential data in a way that other algorithms can't.



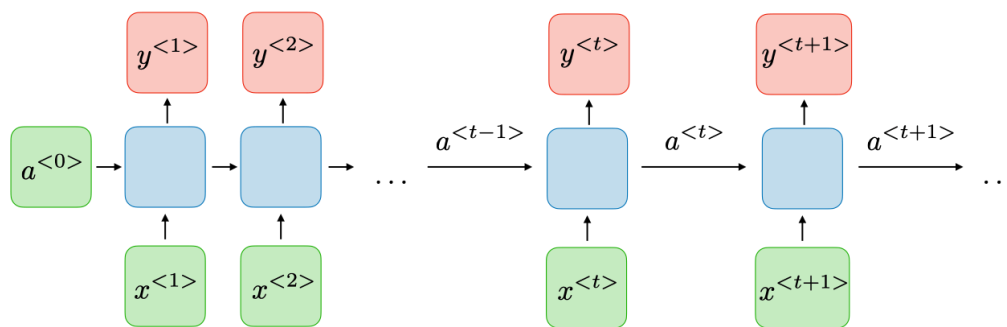
All of the inputs and outputs in standard neural networks are independent of one another, however in some circumstances, such as when predicting the next word of a phrase, the prior words are necessary, and so the previous words must be remembered. As a result, RNN was created, which

used a Hidden Layer to overcome the problem. The most important component of RNN is the Hidden state, which remembers specific information about a sequence.

RNNs have a Memory that stores all information about the calculations. It employs the same settings for each input since it produces the same outcome by performing the same task on all inputs or hidden layers.

The Architecture of a Traditional RNN

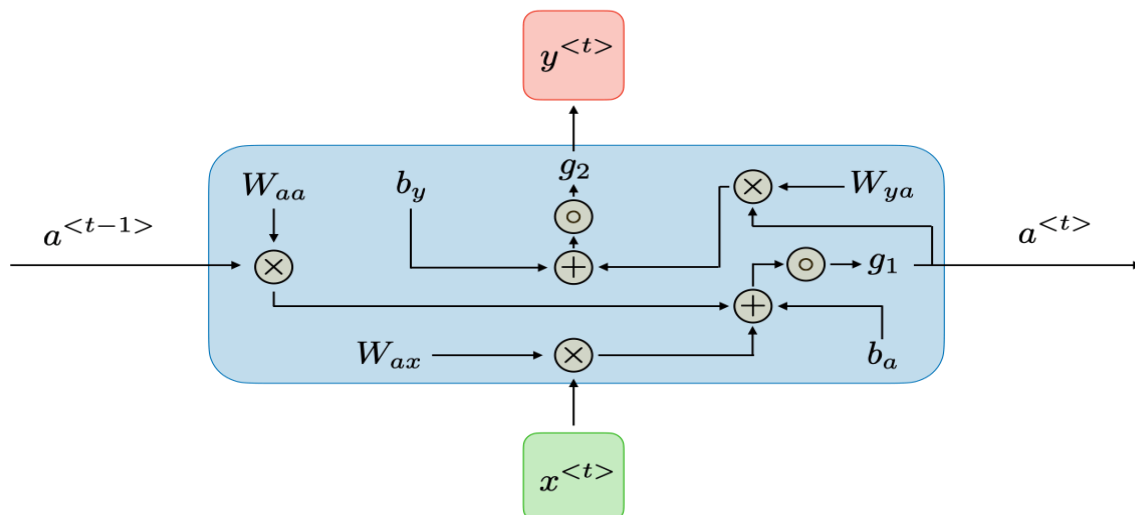
RNNs are a type of neural network that has hidden states and allows past outputs to be used as inputs. They usually go like this:



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

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

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



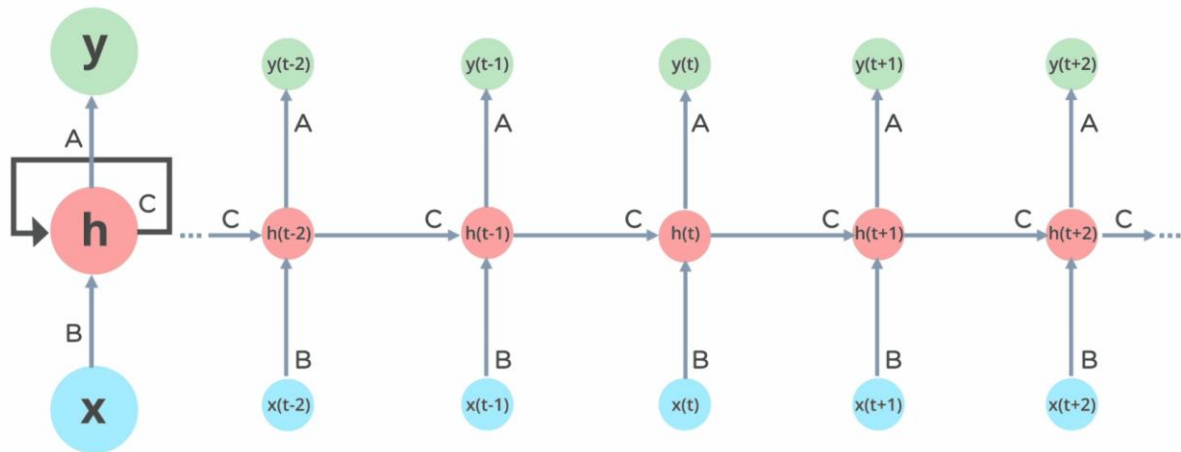
RNN architecture can vary depending on the problem you're trying to solve. From those with a single input and output to those with many (with variations between).

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.

How does Recurrent Neural Networks work?

The information in recurrent neural networks cycles through a loop to the middle-hidden layer.



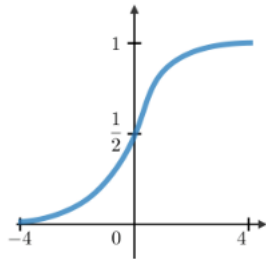
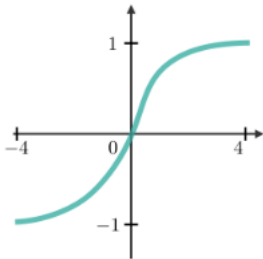
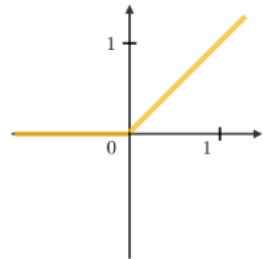
The input layer \mathbf{x} receives and processes the neural network's input before passing it on to the middle layer.

Multiple hidden layers can be found in the middle layer \mathbf{h} , each with its own activation functions, weights, and biases. You can utilize a recurrent neural network if the various parameters of different hidden layers are not impacted by the preceding layer, i.e. There is no memory in the neural network.

The different activation functions, weights, and biases will be standardized by the Recurrent Neural Network, ensuring that each hidden layer has the same characteristics. Rather than constructing numerous hidden layers, it will create only one and loop over it as many times as necessary.

Common Activation Functions:

A neuron's activation function dictates whether it should be turned on or off. Nonlinear functions usually transform a neuron's output to a number between 0 and 1 or -1 and 1.

Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1 + e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$
		

Source: MLtutorial.com

The following are some of the most commonly utilized functions:

- **Sigmoid:** The formula $g(z) = 1/(1 + e^{-z})$ is used to express this.
- **Tanh:** The formula $g(z) = (e^z - e^{-z})/(e^z + e^{-z})$ is used to express this.
- **Relu:** The formula $g(z) = \max(0, z)$ is used to express this.

Applications of RNN Networks

1. Machine Translation:

RNN can be used to build a deep learning model that can translate text from one language to another without the need for human intervention. You can, for example, translate a text from your native language to English.

2. Text Creation:

RNNs can also be used to build a deep learning model for text generation. Based on the previous sequence of words/characters used in the text, a trained model learns the likelihood of occurrence of a word/character. A model can be trained at the character, n-gram, sentence, or paragraph level.

3. Captioning of images:

The process of creating text that describes the content of an image is known as image captioning. The image's content can depict the object as well as the action of the object on the image. In the image below, for example, the trained deep learning model using RNN can describe the image as "A lady in a green coat is reading a book under a tree."

4. Recognition of Speech:

This is also known as Automatic Speech Recognition (ASR), and it is capable of converting human speech into written or text format. Don't mix up speech recognition and voice recognition; speech recognition primarily focuses on converting voice data into text, whereas voice recognition identifies the user's voice.

Speech recognition technologies that are used on a daily basis by various users include Alexa, Cortana, Google Assistant, and Siri.

5. Forecasting of Time Series:

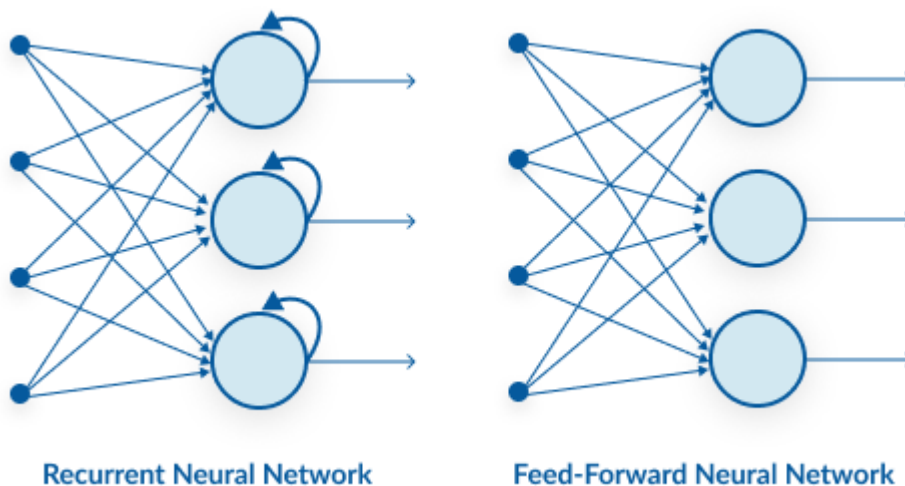
After being trained on historical time-stamped data, an RNN can be used to create a time series prediction model that predicts the future outcome. The stock market is a good example.

You can use stock market data to build a machine learning model that can forecast future stock prices based on what the model learns from historical data. This can assist investors in making data-driven investment decisions.

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.



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-RNN:

Backpropagation is a training algorithm that we use for training neural networks. When preparing a neural network, we are tuning the network's weights to minimize the error concerning the available actual values with the help of the Backpropagation algorithm. Backpropagation is a supervised learning algorithm as we find errors concerning already given values.

The backpropagation training algorithm aims to modify the weights of a neural network to minimize the error of the network results compared to some expected output in response to corresponding inputs.

The general algorithm of Backpropagation is as follows:

1. We first train input data and propagate it through the network to get an output.

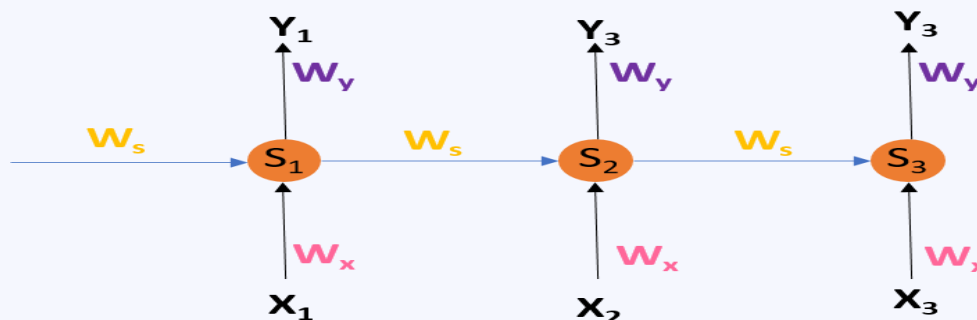
2. Compare the predicted outcomes to the expected results and calculate the error.
3. Then, we calculate the derivatives of the error concerning the network weights.
4. We use these calculated derivatives to adjust the weights to minimize the error.
5. Repeat the process until the error is minimized.

In simple words, Backpropagation is an algorithm where the information of cost function is passed on through the neural network in the backward direction. The Backpropagation training algorithm is ideal for training feed-forward neural networks on fixed-sized input-output pairs.

Unrolling The Recurrent Neural Network

We will briefly discuss RNN to understand how the backpropagation algorithm is applied to recurrent neural networks or RNN. Recurrent Neural Network deals with sequential data. RNN predicts outputs using not only the current inputs but also by considering those that occurred before it. In other words, the current outcome depends on the current production and a memory element (which evaluates the past inputs).

The below figure depicts the architecture of RNN :



We use Backpropagation for training such networks with a slight change. We don't independently train the network at a specific time "t." We train it at a particular time "t" as well as all that has happened before time "t" like $t-1$, $t-2$, $t-3$. S_1 , S_2 , S_3 are the hidden states at time t_1 , t_2 , t_3 , respectively, and W_s is the associated weight matrix.

x_1 , x_2 , x_3 are the inputs at time t_1 , t_2 , t_3 , respectively, and W_x is the associated weight matrix.

Y_1 , Y_2 , Y_3 are the outcomes at time t_1 , t_2 , t_3 , respectively, and W_y is the associated weight matrix.

At time t_0 , we feed input x_0 to the network and output y_0 . At time t_1 , we provide input x_1 to the network and receive an output y_1 . From the figure, we can see that to calculate the outcome. The network uses input x and the cell state from the previous timestamp. To calculate specific Hidden state and output at each step, here is the formula:

$$S_t = g_1(W_x x_t + W_s S_{t-1})$$

$$Y_t = g_2(W_Y S_t)$$

To calculate the error, we take the output and calculate its error concerning the actual result, but we have multiple outputs at each timestamp. Thus, the regular Backpropagation won't work here. Therefore, we modify this algorithm and call the new algorithm as Backpropagation through time.

Backpropagation Through Time

It is important to note that W_s , W_x , and W_y do not change across the timestamps, which means that for all inputs in a sequence, the values of these weights are the same.

The error function is defined as:

$$E_t = (d_t - Y_t)^2$$

Now the question arises: What is the total loss for this network? How do we update the weights W_s , W_x , and W_y ?

The total loss we have to calculate is the sum in overall timestamps, i.e., $E_0 + E_1 + E_2 + E_3 + \dots$

Now to calculate the error gradient concerning W_s , W_x , and W_y . It is relatively easy to calculate the loss derivative concerning W_y as the derivative only depends on the current timestamp values.

Formula:

$$\frac{\delta E_N}{\delta W_Y} = \frac{\delta E_N}{\delta Y_N} \cdot \frac{\delta Y_N}{\delta W_Y}$$

But when calculating the derivative of loss concerning W_s and W_x , it becomes tricky.

Formula:

$$\frac{\delta E}{\delta W_s} = \frac{\delta E}{\delta s_3} \cdot \frac{\delta s_3}{\delta W_s}$$

The value of s_3 depends on s_2 , which is a function of W_s . Therefore we cannot calculate the derivative of s_3 , taking s_2 as constant. In RNN networks, the derivative has two parts, implicit and explicit. We assume all other inputs as constant in the explicit part, whereas we sum over all the indirect paths in the implicit part.

Therefore we calculate the derivative as :

$$\frac{\delta E_3}{\delta W_s} = \frac{\delta E_3}{\delta Y_3} \cdot \frac{\delta Y_3}{\delta S_3} \cdot \frac{\delta S_3}{\delta W_s} + \frac{\delta E_3}{\delta Y_3} \cdot \frac{\delta Y_3}{\delta S_3} \cdot \frac{\delta S_3}{\delta S_2} \cdot \frac{\delta S_2}{\delta W_s} + \frac{\delta E_3}{\delta Y_3} \cdot \frac{\delta Y_3}{\delta S_3} \cdot \frac{\delta S_3}{\delta S_2} \cdot \frac{\delta S_2}{\delta S_1} \cdot \frac{\delta S_1}{\delta W_s}$$

The general expression can be written as:

$$\frac{\delta E_N}{\delta W_s} = \sum_{i=1}^N \frac{\delta E_N}{\delta Y_N} \cdot \frac{\delta Y_N}{\delta S_i} \cdot \frac{\delta S_i}{\delta W_s}$$

Similarly, for W_x , it can be written as:

$$\frac{\delta E_N}{\delta W_x} = \sum_{i=1}^N \frac{\delta E_N}{\delta Y_N} \cdot \frac{\delta Y_N}{\delta S_i} \cdot \frac{\delta S_i}{\delta W_x}$$

Now that we have calculated all three derivatives, we can easily update the weights. This algorithm is known as Backpropagation through time, as we used values across all the timestamps to calculate the gradients.

The algorithm at a glance:

- We feed a sequence of timestamps of input and output pairs to the network.
- Then, we unroll the network then calculate and accumulate errors across each timestamp.
- Finally, we roll up the network and update weights.
- Repeat the process.

Limitations of BPTT:

BPTT has difficulty with local optima. Local optima are a more significant issue with recurrent neural networks than feed-forward neural networks. The recurrent feedback in such networks creates chaotic responses in the error surface, which causes local optima to occur frequently and in the wrong locations on the error surface.

When using BPTT in RNN, we face problems such as exploding gradient and vanishing gradient. To avoid issues such as exploding gradient, we use a gradient clipping method to check if the gradient value is greater than the threshold or not at each timestamp. If it is, we normalize it. This helps to tackle exploding gradient.

We can use BPTT up to a limited number of steps like 8 or 10. If we backpropagate further, the gradient becomes too negligible and is a Vanishing gradient problem. To avoid the vanishing gradient problem, some of the possible solutions are:

- Using ReLU activation function in place of tanh or sigmoid activation function.
- Proper initializing the weight matrix can reduce the effect of vanishing gradients. For example, using an identity matrix helps us tackle this problem.
- Using gated cells such as LSTM or GRUs.

Vanishing Gradient Problem:

The [gradient descent](#) algorithm finds the global minimum of the cost function that is going to be an optimal setup for the network.

As you might also recall, information travels through the neural network from input neurons to the output neurons, while the error is calculated and propagated back through the network to update the weights.

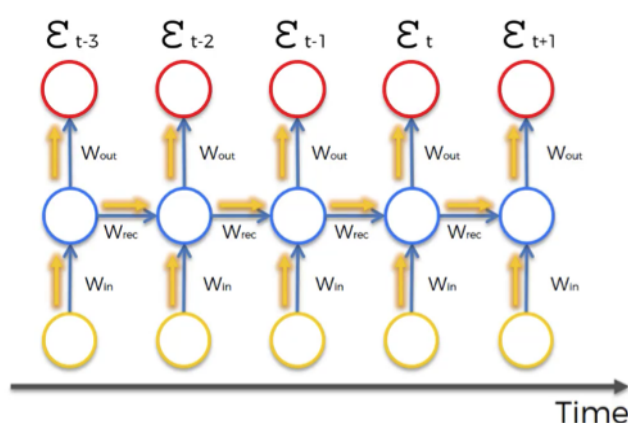
It works quite similarly for RNNs, but here we've got a little bit more going on.

- Firstly, information travels through time in RNNs, which means that information from previous time points is used as input for the next time points.
- Secondly, you can calculate the cost function, or your error, at each time point.

Basically, during the training, your cost function compares your outcomes (red circles on the image below) to your desired output.

As a result, you have these values throughout the time series, for every single one of these red circles.

The Vanishing Gradient Problem



Formula Source: Razvan Pascanu et al. (2013)

Let's focus on one error term ϵ_t .

You've calculated the cost function ϵ_t , and now you want to propagate your cost function back through the network because you need to update the weights.

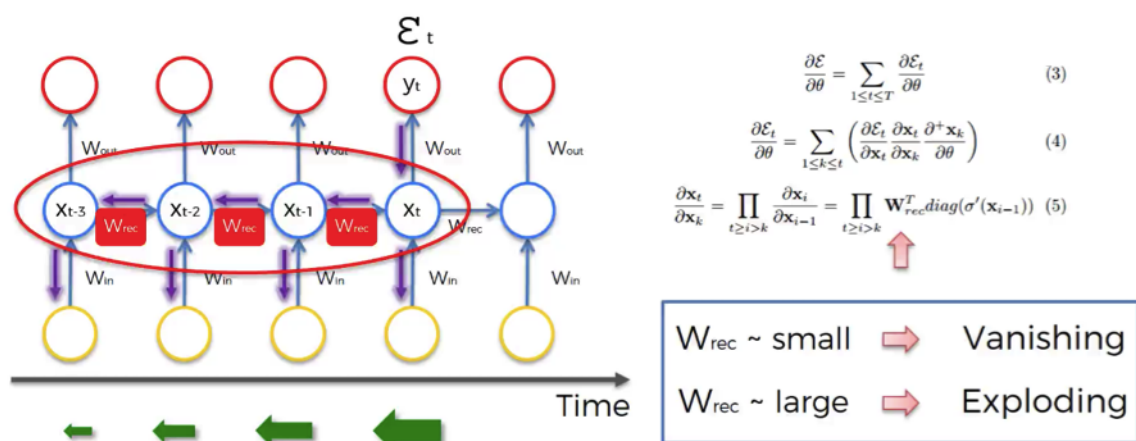
Essentially, every single neuron that participated in the calculation of the output, associated with this cost function, should have its weight updated in order to minimize that error. And the thing with RNNs is that it's not just the neurons directly below this output layer that contributed but all of the neurons far back in time. So, you have to propagate all the way back through time to these neurons.

The problem relates to updating w_{rec} (weight recurring) – the weight that is used to connect the hidden layers to themselves in the unrolled temporal loop.

For instance, to get from x_{t-3} to x_{t-2} we multiply x_{t-3} by w_{rec} . Then, to get from x_{t-2} to x_{t-1} we again multiply x_{t-2} by w_{rec} . So, we multiply with the same exact weight multiple times, and this is where the problem arises: when you multiply something by a small number, your value decreases very quickly.

As we know, weights are assigned at the start of the neural network with the random values, which are close to zero, and from there the network trains them up. But, when you start with w_{rec} close to zero and multiply x_t , x_{t-1} , x_{t-2} , x_{t-3} , ... by this value, your gradient becomes less and less with each multiplication.

The Vanishing Gradient Problem



Formula Source: Razvan Pascanu et al. (2013)

What does this mean for the network?

The lower the gradient is, the harder it is for the network to update the weights and the longer it takes to get to the final result.

For instance, 1000 epochs might be enough to get the final weight for the time point t , but insufficient for training the weights for the time point $t-3$ due to a very low gradient at this point. However, the problem is not only that half of the network is not trained properly.

The output of the earlier layers is used as the input for the further layers. Thus, the training for the time point t is happening all along based on inputs that are coming from untrained layers. So, because of the vanishing gradient, the whole network is not being trained properly.

To sum up, if w_{rec} is small, you have vanishing gradient problem, and if w_{rec} is large, you have exploding gradient problem

For the vanishing gradient problem, the further you go through the network, the lower your gradient is and the harder it is to train the weights, which has a domino effect on all of the further weights throughout the network.

That was the main roadblock to using Recurrent Neural Networks. But let's now check what are the possible solutions to this problem.

Solutions to the vanishing gradient problem

In case of exploding gradient, you can:

- stop backpropagating after a certain point, which is usually not optimal because not all of the weights get updated;
- penalize or artificially reduce gradient;
- put a maximum limit on a gradient.

In case of vanishing gradient, you can:

- initialize weights so that the potential for vanishing gradient is minimized;
- have Echo State Networks that are designed to solve the vanishing gradient problem;
- have Long Short-Term Memory Networks (LSTMs).

Gradient clipping Long Short-Term Memory (LSTM) Networks:

Training a neural network can become unstable given the choice of error function, learning rate, or even the scale of the target variable.

Large updates to weights during training can cause a numerical overflow or underflow often referred to as “exploding gradients.”

The problem of exploding gradients is more common with recurrent neural networks, such as LSTMs given the accumulation of gradients unrolled over hundreds of input time steps.

A common and relatively easy solution to the exploding gradients problem is to change the derivative of the error before propagating it backward through the network and using it to update the weights. Two approaches include rescaling the gradients given a chosen vector norm and clipping gradient values that exceed a preferred range. Together, these methods are referred to as “gradient clipping.”

- Training neural networks can become unstable, leading to a numerical overflow or underflow referred to as exploding gradients.
- The training process can be made stable by changing the error gradients either by scaling the vector norm or clipping gradient values to a range.
- How to update an MLP model for a regression predictive modeling problem with exploding gradients to have a stable training process using gradient clipping methods.

Exploding Gradients and Clipping

Neural networks are trained using the stochastic gradient descent optimization algorithm.

This requires first the estimation of the loss on one or more training examples, then the calculation of the derivative of the loss, which is propagated backward through the network in order to update the weights. Weights are updated using a fraction of the back propagated error controlled by the [learning rate](#).

It is possible for the updates to the weights to be so large that the weights either overflow or underflow their numerical precision. In practice, the weights can take on the value of an “NaN” or “Inf” when they overflow or underflow and for practical purposes the network will be useless from that point forward, forever predicting NaN values as signals flow through the invalid weights.

“ The difficulty that arises is that when the parameter gradient is very large, a gradient descent parameter update could throw the parameters very far, into a region where the objective function is larger, undoing much of the work that had been done to reach the current solution.”

The underflow or overflow of weights is generally refers to as an instability of the network training process and is known by the name “*exploding gradients*” as the unstable training process causes the network to fail to train in such a way that the model is essentially useless.

In a given neural network, such as a Convolutional Neural Network or Multilayer Perceptron, this can happen due to a poor choice of configuration. Some examples include:

- Poor choice of learning rate that results in large weight updates.
- Poor choice of data preparation, allowing large differences in the target variable.
- Poor choice of loss function, allowing the calculation of large error values.

Exploding gradients is also a problem in recurrent neural networks such as the Long Short-Term Memory network given the accumulation of error gradients in the [unrolled recurrent structure](#).

Exploding gradients can be avoided in general by careful configuration of the network model, such as choice of small learning rate, scaled target variables, and a [standard loss function](#). Nevertheless, exploding gradients may still be an issue with recurrent networks with a large number of input time steps.

“One difficulty when training LSTM with the full gradient is that the derivatives sometimes become excessively large, leading to numerical problems. To prevent this, [we] clipped the derivative of the loss with respect to the network inputs to the LSTM layers (before the sigmoid and tanh functions are applied) to lie within a predefined range”

A common solution to exploding gradients is to change the error derivative before propagating it backward through the network and using it to update the weights. By rescaling the error derivative, the updates to the weights will also be rescaled, dramatically decreasing the likelihood of an overflow or underflow.

There are two main methods for updating the error derivative; they are:

- Gradient Scaling.
- Gradient Clipping.

Gradient scaling involves normalizing the error gradient vector such that vector norm (magnitude) equals a defined value, such as 1.0.

... one simple mechanism to deal with a sudden increase in the norm of the gradients is to rescale them whenever they go over a threshold

Gradient clipping involves forcing the gradient values (element-wise) to a specific minimum or maximum value if the gradient exceeded an expected range. Together, these methods are often simply referred to as “*gradient clipping*.”

“When the traditional gradient descent algorithm proposes to make a very large step, the gradient clipping heuristic intervenes to reduce the step size to be small enough

that it is less likely to go outside the region where the gradient indicates the direction of approximately steepest descent.”

It is a method that only addresses the numerical stability of training deep neural network models and does not offer any general improvement in performance.

The value for the gradient vector norm or preferred range can be configured by trial and error, by using common values used in the literature or by first observing common vector norms or ranges via experimentation and then choosing a sensible value.

In our experiments we have noticed that for a given task and model size, training is not very sensitive to this [gradient norm] hyperparameter and the algorithm behaves well even for rather small thresholds.

It is common to use the same gradient clipping configuration for all layers in the network. Nevertheless, there are examples where a larger range of error gradients are permitted in the output layer compared to hidden layers.

The output derivatives [...] were clipped in the range $[-100, 100]$, and the LSTM derivatives were clipped in the range $[-10, 10]$. Clipping the output gradients proved vital for numerical stability; even so, the networks sometimes had numerical problems late on in training, after they had started overfitting on the training data.

Gated Recurrent Unit (GRU):

A Gated Recurrent Unit (GRU) is a Recurrent Neural Network (RNN) architecture type. Like other RNNs, a GRU can process sequential data such as time series, natural language, and speech. The main difference between a GRU and other RNN architectures, such as the Long Short-Term Memory (LSTM) network, is how the network handles information flow through time.

Take a look at the following sentence:

"My mom gave me a bicycle on my birthday because she knew that I wanted to go biking with my friends."

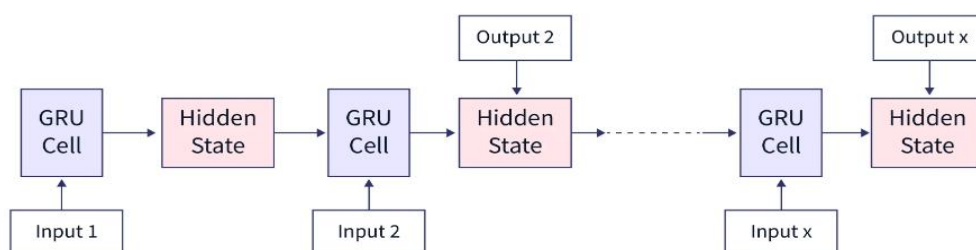
As we can see from the above sentence, words that affect each other can be further apart. For example, "bicycle" and "go biking" are closely related but are placed further apart in the sentence.

An RNN network finds tracking the state with such a long context difficult. It needs to find out what information is important. However, a GRU cell greatly alleviates this problem.

GRU network was invented in 2014. It solves problems involving long sequences with contexts placed further apart, like the above biking example. This is possible because of how the GRU cell in the GRU architecture is built. Let us now delve deeper into the understanding and working of the GRU network.

Understanding the GRU Cell:

The Gated Recurrent Unit (GRU) cell is the basic building block of a GRU network. It comprises three main components: an update gate, a reset gate, and a candidate hidden state.



One of the key advantages of the GRU cell is its simplicity. Since it has fewer parameters than a long short-term memory (LSTM) cell, it is faster to train and run and less prone to overfitting.

Additionally, one thing to remember is that the GRU cell architecture is simple, the cell itself is a black box, and the final decision on how much we should consider the past state and how much should be forgotten is taken by this GRU cell. We need to look inside and understand what the cell is thinking.

Compare GRU vs LSTM

Here is a comparison of Gated Recurrent Unit (GRU) and Long Short-Term Memory (LSTM) networks

	GRU	LSTM
Structure	Simpler structure with two gates (update and reset gate)	More complex structure with three gates (input, forget, and output gate)
Parameters	Fewer parameters (3 weight matrices)	More parameters (4 weight matrices)
Training	Faster to train	Slow to train
Space Complexity	In most cases, GRU tend to use fewer memory resources due to its simpler structure and fewer parameters, thus better suited for	LSTM has a more complex structure and a larger number of parameters, thus might require more memory resources and could be less effective for large datasets

GRU

large datasets or sequences.

Generally performed similarly to LSTM on many tasks, but in some cases, GRU has been shown to outperform LSTM and vice versa.

Performance It's better to try both and see which works better for your dataset and task.

LSTM

or sequences.

LSTM generally performs well on many tasks but is more computationally expensive and requires more memory resources. LSTM has advantages over GRU in natural language understanding and machine translation tasks.

The Architecture of GRU

A GRU cell keeps track of the important information maintained throughout the network. A GRU network achieves this with the following two gates:

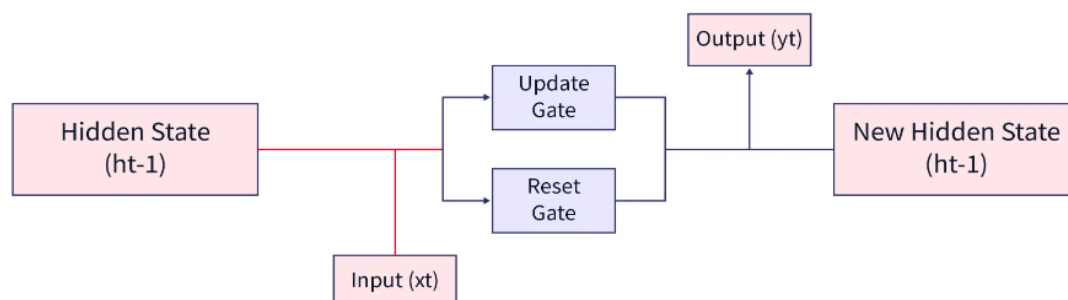
- Reset Gate
- Update Gate.

Given below is the simplest architectural form of a GRU cell.

As shown below, a GRU cell takes two inputs:

1. The previous hidden state
2. The input in the current timestamp.

The cell combines these and passes them through the update and reset gates. To get the output in the current timestep, we must pass this hidden state through a dense layer with softmax activation to predict the output. Doing so, a new hidden state is obtained and then passed on to the next time step.



Update gate

An update gate determines what current GRU cell will pass information to the next GRU cell. It helps in keeping **track of the most important information**.

Let us see how the output of the Update Gate is obtained in a GRU cell. The input to the update gate is the hidden layer at the previous timestep $h(t-1)$ and the current input (x_t) .

Both have their weights associated with them which are learned during the training process. Let us say that the weights associated with $h(t-1)$ is $U(z)$, and that of x_t is Wz . The output of the update gate Z_t is given by,

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

Reset gate

A reset gate **identifies the unnecessary information** and decides what information to be laid off from the GRU network. Simply put, it decides what information to delete at the specific timestamp.

Let us see how the output of the Reset Gate is obtained in a GRU cell. The input to the reset gate is the hidden layer at the previous timestep $h(t-1)$ and the current input x_t . Both have their weights associated with them which are learned during the training process. Let us say that the weights associated with $h(t-1)$ is U_r , and that of x_t is W_r . The output of the update gate r_t is given by,

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

PS: It is important to note that the weights associated with the hidden layer at the previous timestep and the current input are different for both gates. The values for these weights are learned during the training process.

How Does GRU Work?

Gated Recurrent Unit (GRU) networks process sequential data, such as time series or natural language, bypassing the hidden state from one time step to the next. The hidden state is a vector that captures the information from the past time steps relevant to the current time step. The main idea behind a GRU is to allow the network to decide what information from the last time step is relevant to the current time step and what information can be discarded.

Candidate Hidden State

A candidate's hidden state is calculated from the reset gate. This is used to determine the information stored from the past. This is generally called the memory component in a GRU cell. It is calculated by,

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

Here, W - weight associated with the current input r_t - Output of the reset gate U - Weight associated with the hidden layer of the previous timestep ' h_t' ' - Candidate hidden state

Hidden state

The following formula gives the new hidden state and depends on the update gate and candidate hidden state.

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

Here, z_t - Output of update gate
 Candidate hidden state h_{t-1} - Hidden state at the previous timestep

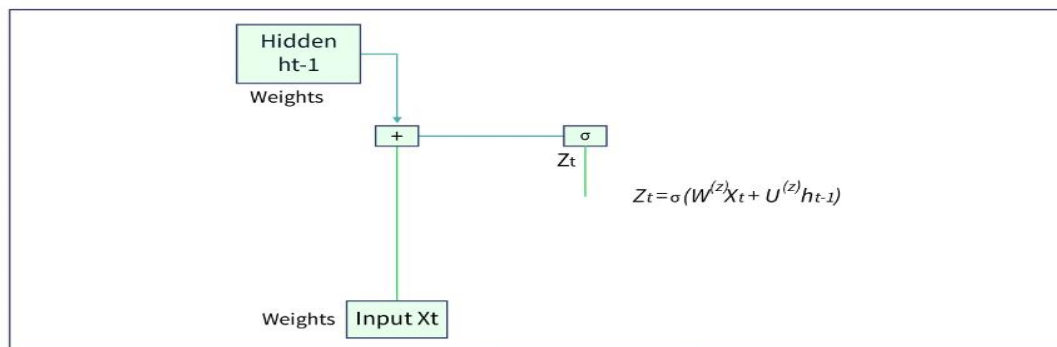
As we can see in the above formula, whenever z_t is 0, the information at the previously hidden layer gets forgotten. It is updated with the value of the new candidate hidden layer (as $1-z_t$ will be 1). If z_t is 1, then the information from the previously hidden layer is maintained. This is how the most relevant information is passed from one state to the next.

Now, we have all the basics to understand a GRU network's forward propagation (i.e., working). Without any further ado, let us get started.

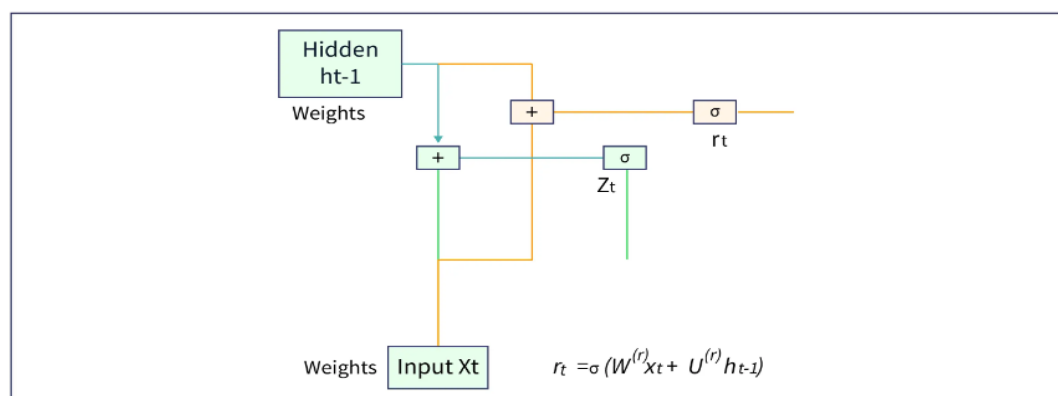
Forward Propagation in a GRU Cell

In a Gated Recurrent Unit (GRU) cell, the forward propagation process includes several steps:

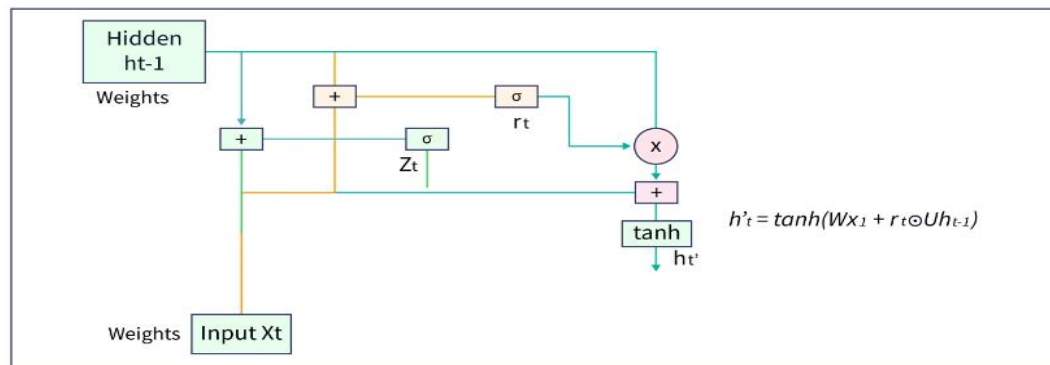
- Calculate the output of the update gate (z_t) using the update gate formula:



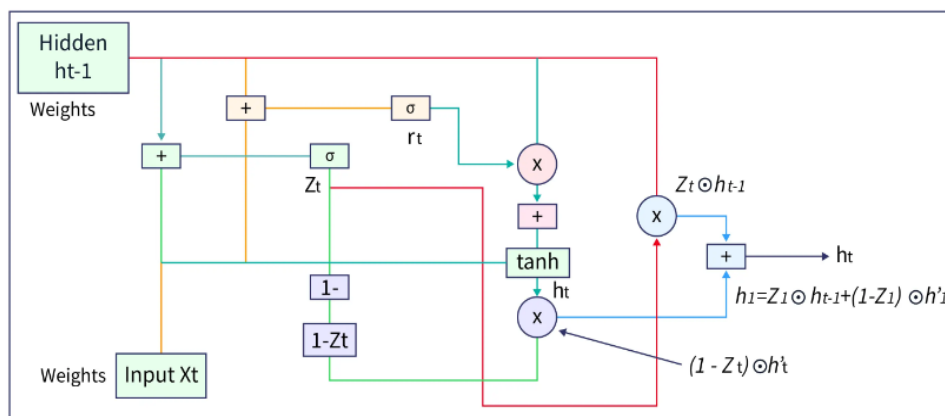
- Calculate the output of the reset gate (r_t) using the reset gate formula



- Calculate the candidate's hidden state



- Calculate the new hidden state

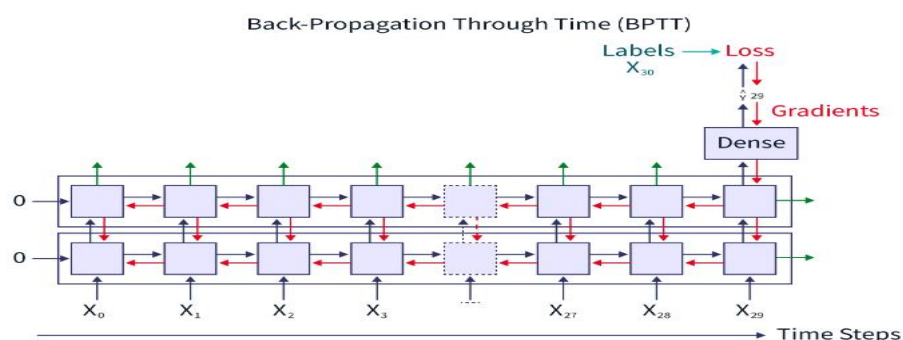


This is how forward propagation happens in a GRU cell at a GRU network.

We have a question about how the weights are learnt in a GRU network to make the right prediction. Let's understand that in the next section.

Backpropagation in a GRU Cell

Take a look at the image below. Let each hidden layer (orange colour) represent a GRU cell.



In the above image, we can see that whenever the network predicts wrongly, the network compares it with the original label, and the loss is then propagated throughout the network. This happens until all the weights' values are identified so that the value of the loss function used to compute the loss is minimum. During this time, the weights and biases associated with the hidden layers and the input are fine-tuned.

What are the differences and similarities between LSTM and GRU in terms of architecture and performance?

LSTM and GRU are two types of recurrent neural networks (RNNs) that can handle sequential data, such as text, speech, or video. They are designed to overcome the problem of vanishing or exploding gradients that affect the training of standard RNNs. However, they have different architectures and performance characteristics that make them suitable for different applications. In this article, you will learn about the differences and similarities between LSTM and GRU in terms of architecture and performance.

LSTM Architecture

LSTM stands for long short-term memory, and it consists of a series of memory cells that can store and update information over long time steps. Each memory cell has three gates: an input gate, an output gate, and a forget gate. The input gate decides what information to add to the cell state, the output gate decides what information to output from the cell state, and the forget gate decides what information to discard from the cell state. The gates are learned by the network based on the input and the previous hidden state.

GRU Architecture

GRU stands for gated recurrent unit, and it is a simplified version of LSTM. It has only two gates: a reset gate and an update gate. The reset gate decides how much of the previous hidden state to keep, and the update gate decides how much of the new input to incorporate into the hidden state. The hidden state also acts as the cell state and the output, so there is no separate output gate. The GRU is easier to implement and requires fewer parameters than the LSTM.

Performance Comparison

The performance of LSTM and GRU depends on the task, the data, and the hyperparameters. Generally, LSTM is more powerful and flexible than GRU, but it is also more complex and prone to overfitting. GRU is faster and more efficient than LSTM, but it may not capture long-term dependencies as well as LSTM. Some empirical studies have shown that LSTM and GRU perform similarly on many natural language processing tasks, such as sentiment analysis, machine translation, and text generation. However, some tasks may benefit from the specific features of LSTM or GRU, such as image captioning, speech recognition, or video analysis.

Similarities Between LSTM and GRU

Despite their differences, LSTM and GRU share some common characteristics that make them both effective RNN variants. They both use gates to control the information flow and to avoid the vanishing or exploding gradient problem. They both can learn long-term dependencies and capture sequential patterns in the data. They both can be stacked into multiple layers to increase the depth and complexity of the network. They both can be combined with other neural network architectures, such as convolutional neural networks (CNNs) or attention mechanisms, to enhance their performance.

Differences Between LSTM and GRU

The main differences between LSTM and GRU lie in their architectures and their trade-offs. LSTM has more gates and more parameters than GRU, which gives it more

flexibility and expressiveness, but also more computational cost and risk of overfitting. GRU has fewer gates and fewer parameters than LSTM, which makes it simpler and faster, but also less powerful and adaptable. LSTM has a separate cell state and output, which allows it to store and output different information, while GRU has a single hidden state that serves both purposes, which may limit its capacity. LSTM and GRU may also have different sensitivities to the hyperparameters, such as the learning rate, the dropout rate, or the sequence length.

Bidirectional LSTM

Introduction:

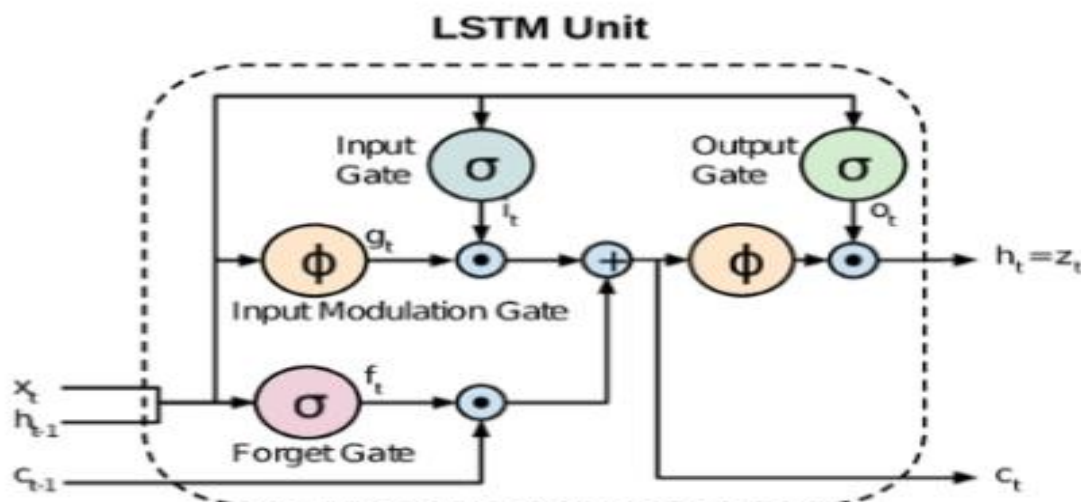
To understand the working of Bi-LSTM first, we need to understand the unit cell of LSTM and LSTM network. LSTM stands for long short-term memory. In 1977 Hochreiter and Schmidhuber introduced LSTM networks. These are the most commonly used recurrent neural networks.

Need of LSTM

As we know that sequential data is better handled by recurrent neural networks, but sometimes it is also necessary to store the result of the previous data. For example, “I will play cricket” and “I can play cricket” are two different sentences with different meanings. As we can see, the meaning of the sentence depends on a single word so, it is necessary to store the data of previous words. But no such memory is available in simple RNN. To solve this problem, we need to study a term called LSTM.

The Architecture of the LSTM Unit

The image below is the architecture of the LSTM unit.



The LSTM unit has three gates:

Input gate

First, the current state $x(t)$ and previous hidden state $h(t-1)$ are passed into the input gate, i.e., the second sigmoid function. The $x(t)$ and $h(t-1)$ values are transformed between 0 and 1, where 0 is important, and 1 is not important. Furthermore, the current and hidden state information will be passed through the tanh function. The output from the tanh function will range from -1 to 1, and it will help to regulate the network. The

output values generated from the activation functions are ready for point-by-point multiplication.

Forget gate

The forget gate decides which information needs to be kept for further processing and which can be ignored. The hidden state $h(t-1)$ and current input $X(t)$ information are passed through the sigmoid function. After passing the values through the sigmoid function, it generates values between 0 and 1 that conclude whether the part of the previous output is necessary (by giving the output closer to 1).

Output gate

The output gate helps in deciding the value of the next hidden state. This state contains information on previous inputs. First, the current and previously hidden state values are passed into the third sigmoid function. Then the new cell state generated from the cell state is passed through the tanh function. Both these outputs are multiplied point-by-point. Based upon the final value, the network decides which information the hidden state should carry. This hidden state is used for prediction.

Finally, the new cell state and the new hidden state are carried over to the next step.

To conclude, the forget gate determines which relevant information from the prior steps is needed. The input gate decides what relevant information can be added from the current step, and the output gates finalize the next hidden state.

How do LSTM works?

The Lengthy Short Term Memory architecture was inspired by an examination of error flow in current RNNs, which revealed that long time delays were inaccessible to existing designs due to backpropagated error, which either blows up or decays exponentially.

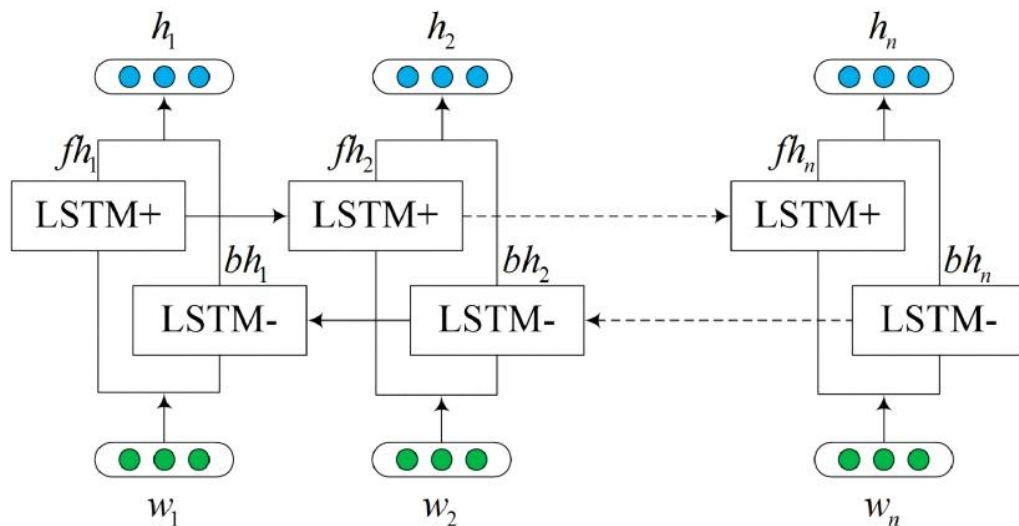
An LSTM layer is made up of memory blocks that are recurrently linked. These blocks can be thought of as a differentiable version of a digital computer's memory chips. Each one has recurrently connected memory cells as well as three multiplicative units – the input, output, and forget gates – that offer continuous analogs of the cells' write, read, and reset operations.

What is Bi-LSTM?

Bidirectional LSTM networks function by presenting each training sequence forward and backward to two independent LSTM networks, both of which are coupled to the same output layer. This means that the Bi-LSTM contains comprehensive, sequential information about all points before and after each point in a particular sequence.

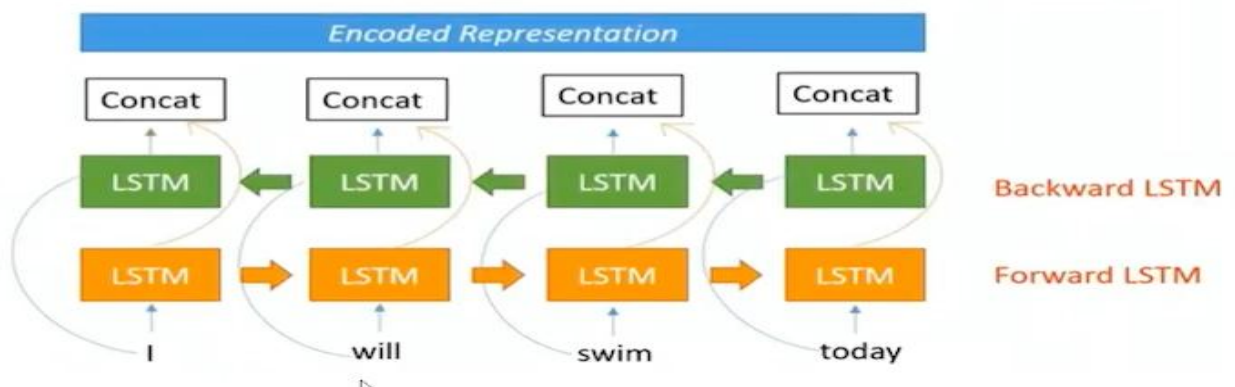
In other words, rather than encoding the sequence in the forward direction only, we encode it in the backward direction as well and concatenate the results from both forward and backward LSTM at each time step. The encoded representation of each word now understands the words before and after the specific word.

Below is the basic architecture of Bi-LSTM.



Working of Bi-LSTM

Let us understand the working of Bi-LSTM using an example. Consider the sentence “I will swim today”. The below image represents the encoded representation of the sentence in the Bi-LSTM network.



So when forward LSTM occurs, “I” will be passed into the LSTM network at time $t = 0$, “will” at $t = 1$, “swim” at $t = 2$, and “today” at $t = 3$. In backward LSTM “today” will be passed into the network at time $t = 0$, “swim” at $t = 1$, “will” at $t = 2$, and “I” at $t = 3$. In this way, results both forward and backward LSTM at each time step is calculated.

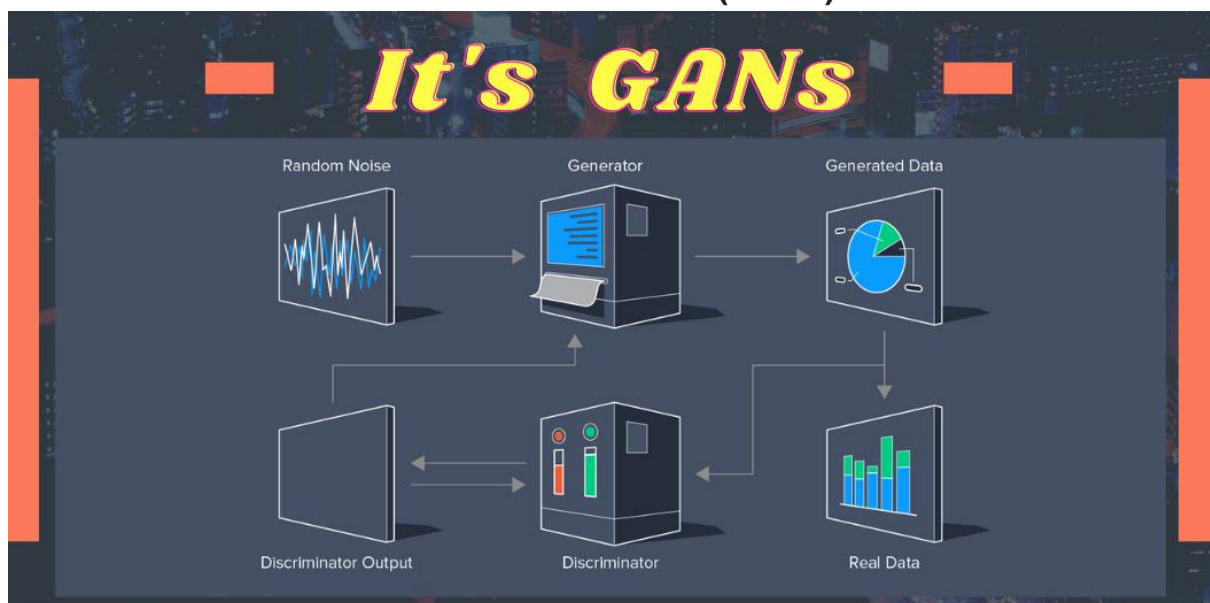
UNIT- IV

GENERATIVE ADVERSARIAL NETWORKS (GANS): Generative models, Concept and principles of GANs, Architecture of GANs (generator and discriminator networks), Comparison between discriminative and generative models, Generative Adversarial Networks (GANs), Applications of GANs

Generative Adversarial Networks and its models:

Introduction:

What are Generative Adversarial Networks (GANs):



Generative Adversarial Networks (GANs) were developed in 2014 by Ian Goodfellow and his teammates. GAN is basically an approach to generative modeling that generates a new set of data based on training data that look like training data. GANs have two main blocks (two neural networks) which compete with each other and are able to capture, copy, and analyze the variations in a dataset. The two models are usually called Generator and Discriminator which we will cover in Components on GANs. To understand the term GAN let's break it into separate three parts

- **Generative** – To learn a generative model, which describes how data is generated in terms of a probabilistic model. In simple words, it explains how data is generated visually.
- **Adversarial** – The training of the model is done in an adversarial setting.
- **Networks** – use deep neural networks for training purposes.

The generator network takes random input (typically noise) and generates samples, such as images, text, or audio, that resemble the training data it was trained on. The goal of the generator is to produce samples that are indistinguishable from real data.

The discriminator network, on the other hand, tries to distinguish between real and generated samples. It is trained with real samples from the training data and generated samples from the generator. The discriminator's objective is to correctly classify real data as real and generated data as fake.

The training process involves an adversarial game between the generator and the discriminator. The generator aims to produce samples that fool the discriminator, while the discriminator tries to improve its ability to distinguish between real and generated data. This adversarial training pushes both networks to improve over time.

As training progresses, the generator becomes more adept at producing realistic samples, while the discriminator becomes more skilled at differentiating between real and generated data. Ideally, this process converges to a point where the generator is capable of generating high-quality samples that are difficult for the discriminator to distinguish from real data.

GANs have demonstrated impressive results in various domains, such as image synthesis, text generation, and even video generation. They have been used for tasks like generating realistic images, creating deepfakes, enhancing low-resolution images, and more. GANs have greatly advanced the field of generative modeling

and have opened up new possibilities for creative applications in artificial intelligence.

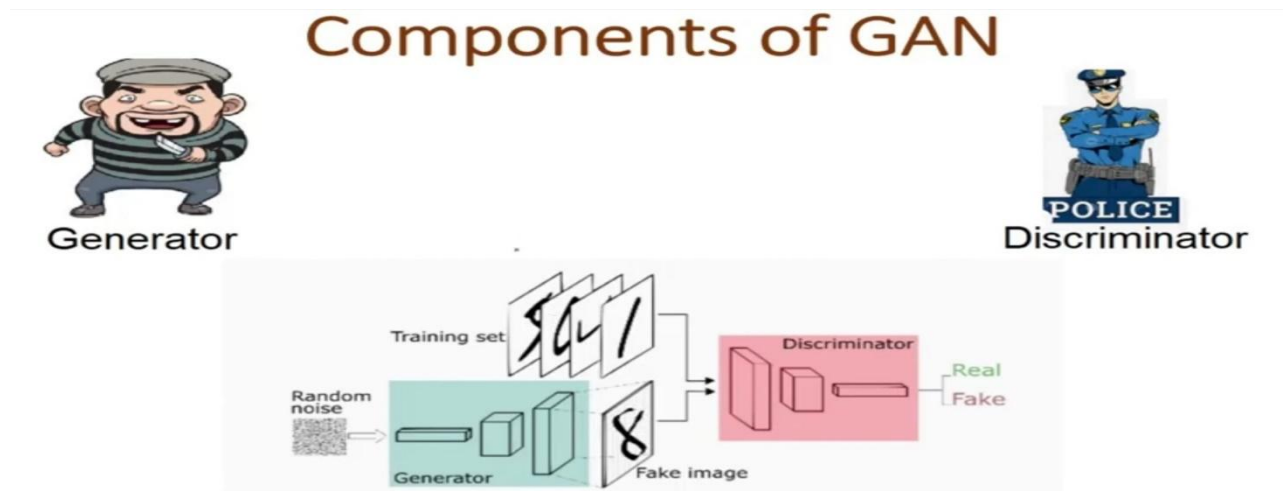
Why GANs was Developed?

Machine learning algorithms and neural networks can easily be fooled to misclassify things by adding some amount of noise to data. After adding some amount of noise, the chances of misclassifying the images increase. Hence the small rise that, is it possible to implement something that neural networks can start visualizing new patterns like sample train data. Thus, GANs were built that generate new fake results similar to the original.

Components of Generative Adversarial Networks (GANs)

What is Geometric Intuition behind the working of GANs?

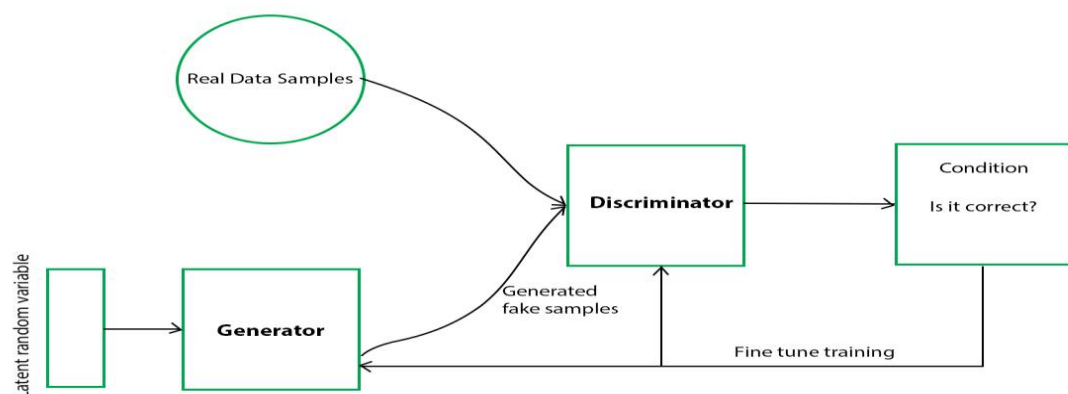
Two major components of GANs are Generator and Discriminator. The role of the generator is like a thief to generate the fake samples based on the original sample and make the discriminator fool to understand Fake as real. On the other hand, a Discriminator is like a Police whose role is to identify the abnormalities in the samples created by Generator and classify them as Fake or real. This competition between both the component goes on until the level of perfection is achieved where Generator wins making a Discriminator fool on fake data.



Now let us understand, what is this two-component to understand the training process of GAN intuitively.

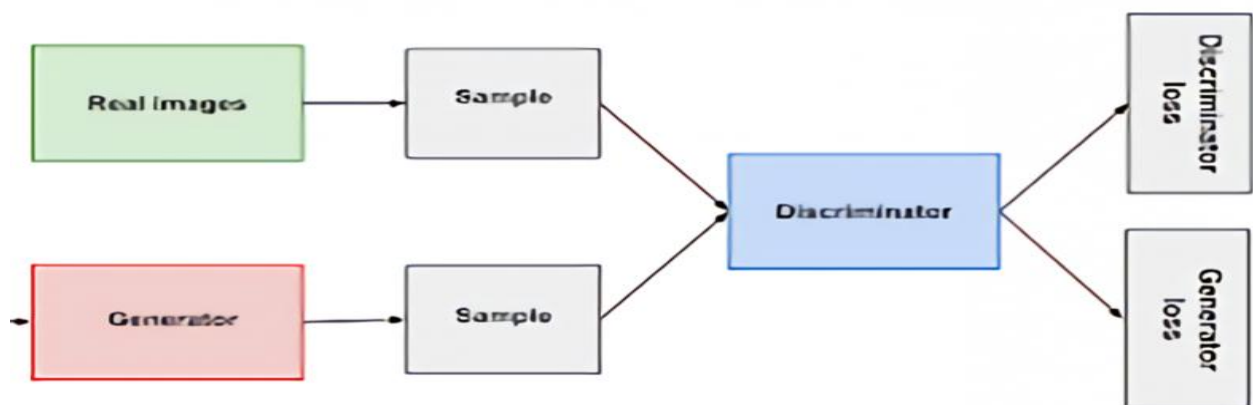
1) Discriminator – It is a supervised approach means It is a simple classifier that predicts data is fake or real. It is trained on real data and provides feedback to a generator.

2) Generator – It is an unsupervised learning approach. It will generate data that is fake data based on original(real) data. It is also a neural network that has hidden layers, activation, loss function. Its aim is to generate the fake image based on feedback and make the discriminator fool that it cannot predict a fake image. And when the discriminator is made a fool by the generator, the training stops and we can say that a generalized GAN model is created.



Here the generative model captures the distribution of data and is trained in such a manner to generate the new sample that tries to maximize the probability of the discriminator to make a mistake(maximize discriminator loss). The discriminator on other hand is based on a model that estimates the probability that the sample it receives is from training data not from the generator and tries to classify it accurately and minimize the GAN accuracy. Hence the GAN network is formulated as a minimax game where the Discriminator is trying to minimize its reward $V(\mathbf{D}, \mathbf{G})$ and the generator is trying to maximize the Discriminator loss.

Now you might be wondering how is an actual architecture of GAN, and how two neural networks are build and training and prediction is done? To simplify it have a look at the below general architecture of GAN.



We know that both components are neural networks. we can see that generator output is directly connected to the input of the discriminator. And discriminator predicts it and through backpropagation, the generator receives a feedback signal to update weights and improve performance. The discriminator is a feed-forward neural network.

Training & Prediction of Generative Adversarial Networks (GANs):

We know the geometric intuition of GAN, Now let us understand the training of Gan. In this section training of Generator and Discriminator will separately be clear to you.

Step-1) Define a Problem

The problem statement is key to the success of the project so the first step is to define your problem. GANs work with a different set of problems you are aiming so you need to define What you are creating like audio, poem, text, Image is a type of problem.

Step-2) Select Architecture of GAN

There are many different types of GAN, that we will study further. we have to define which type of GAN architecture we are using.

Step-3) Train Discriminator on Real Dataset

Now Discriminator is trained on a real dataset. It is only having a forward path, no backpropagation is there in the training of the Discriminator in n epochs. And the Data you are providing is without Noise and only contains real images, and for fake images, Discriminator uses instances created by the generator as negative output. Now, what happens at the time of discriminator training.

- It classifies both real and fake data.
- The discriminator loss helps improve its performance and penalize it when it misclassifies real as fake or vice-versa.
- weights of the discriminator are updated through discriminator loss.

Step-4) Train Generator

Provide some Fake inputs for the generator(Noise) and It will use some random noise and generate some fake outputs. when Generator is trained, Discriminator is Idle and when Discriminator is trained, Generator is Idle. During generator training through any random noise as input, it tries to transform it into meaningful data. to get meaningful output from the generator takes time and runs under many epochs. steps to train a generator are listed below.

- get random noise and produce a generator output on noise sample
- predict generator output from discriminator as original or fake.
- we calculate discriminator loss.
- perform backpropagation through discriminator, and generator both to calculate gradients.
- Use gradients to update generator weights.

Step-5) Train Discriminator on Fake Data

The samples which are generated by Generator will pass to Discriminator and It will predict the data passed to it is Fake or real and provide feedback to Generator again.

Step-6) Train Generator with the output of Discriminator

Again Generator will be trained on the feedback given by Discriminator and try to improve performance.

This is an iterative process and continues running until the Generator is not successful in making the discriminator fool.



Generative Adversarial Networks (GANs) Loss Function

I hope that the working of the GAN network is completely understandable and now let us understand the loss function it uses and minimize and maximize in this iterative process. The generator tries to minimize the following loss function while the discriminator tries to maximize it. It is the same as a minimax game if you have ever played.

$$\min_G \max_D V(D, G)$$

$$V(D, G) = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

- $D(x)$ is the discriminator's estimate of the probability that real data instance x is real.
- \mathbb{E}_x is the expected value over all real data instances.
- $G(z)$ is the generator's output when given noise z .
- $D(G(z))$ is the discriminator's estimate of the probability that a fake instance is real.
- \mathbb{E}_z is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances $G(z)$).
- The formula is derived from cross-entropy between

Challenges Faced by Generative Adversarial Networks (GANs)

1. The problem of stability between generator and discriminator. We do not want that discriminator should be too strict, we want to be lenient.
2. Problem to determine the positioning of objects. suppose in a picture we have 3 horse and generator have created 6 eyes and 1 horse.
3. The problem in understanding the global objects – GANs do not understand the global structure or holistic structure which is similar to the problem of perspective. It means sometimes GAN generates an image that is unrealistic and cannot be possible.

A problem in understanding the perspective – It cannot understand the 3-d images and if we train it on such types of images then it will fail to create 3-d images because today GANs are capable to work on 1-d images.

Different Types of Generative Adversarial Networks (GANs):

1) DC GAN – It is a Deep convolutional GAN. It is one of the most used, powerful, and successful types of GAN architecture. It is implemented with help of ConvNets in place of a Multi-layered perceptron. The ConvNets use a convolutional stride and are built without max pooling and layers in this network are not completely connected.

2) Conditional GAN and Unconditional GAN (CGAN) – Conditional GAN is deep learning neural network in which some additional parameters are used. Labels are also put in inputs of Discriminator in order to help the discriminator to classify the input correctly and not easily fool by the generator.

3) Least Square GAN(LSGAN) – It is a type of GAN that adopts the least-square loss function for the discriminator. Minimizing the objective function of LSGAN results in minimizing the Pearson divergence.

4) Auxiliary Classifier GAN(ACGAN) – It is the same as CGAN and an advanced version of it. It says that the Discriminator should not only classify the image as real or fake but should also provide the source or class label of the input image.

5) Dual Video Discriminator GAN – DVD-GAN is a generative adversarial network for video generation built upon the BigGAN architecture. DVD-GAN uses two discriminators: **a Spatial Discriminator and a Temporal Discriminator.**

6) SRGAN – Its main function is to transform low resolution to high resolution known as Domain Transformation.

7) Cycle GAN

It is released in 2017 which performs the task of Image Translation. Suppose we have trained it on a horse image dataset and we can translate it into zebra images.

8) Info GAN – Advance version of GAN which is capable to learn to disentangle representation in an unsupervised learning approach.

Top Generative Adversarial Networks Applications:

Generate Examples for Image Datasets:

GANs can be used to generate new examples for image datasets in various domains, such as medical imaging, satellite imagery, and natural language processing. By generating synthetic data, researchers can augment existing datasets and improve the performance of machine learning models.

Generate Photographs of Human Faces:

GANs can generate realistic photographs of human faces, including images of people who do not exist in the real world. You can use these rendered images for various purposes, such as creating avatars for online games or social media profiles.

Generate Realistic Photographs:

GANs can generate realistic photographs of various objects and scenes, including landscapes, animals, and architecture. These rendered images can be used to augment existing image datasets or to create entirely new datasets.

Generate Cartoon Characters:

GANs can be used to generate cartoon characters that are similar to those found in popular movies or television shows. These developed characters can create new content or customize existing characters in games and other applications.

Image-to-Image Translation:

GANs can translate images from one domain to another, such as converting a photograph of a real-world scene into a line drawing or a painting. You can create new content or transform existing images in various ways.

Text-to-Image Translation:

GANs can be used to generate images based on a given text description. You can use it to create visual representations of concepts or generate images for machine learning tasks.

Semantic-Image-to-Photo Translation:

GANs can translate images from a semantic representation (such as a label map or a segmentation map) into a realistic photograph. You can use it to generate synthetic data for training machine learning models or to visualize concepts more practically.

Face Frontal View Generation:

GANs can generate frontal views of faces from images that show the face at an angle. You can use it to improve face recognition algorithms' performance or synthesize pictures for use in other applications.

Generate New Human Poses:

GANs can generate images of people in new poses, such as difficult or impossible for humans to achieve. It can be used to create new content or to augment existing image datasets.

Photos to Emojis:

GANs can be used to convert photographs of people into emojis, creating a more personalized and expressive form of communication.

Photograph Editing:

GANs can be used to edit photographs in various ways, such as changing the background, adding or removing objects, or altering the appearance of people or animals in the image.

Face Aging:

GANs can be used to generate images of people at different ages, allowing users to visualize how they might look in the future or to see what they might have looked like in the past.

Difference Between Discriminative and Generative Models

Some of the differences between the Discriminative and Generative Models.

Core Idea

Discriminative models draw boundaries in the data space, while generative models try to model how data is placed throughout the space. A generative model explains how the data was generated, while a discriminative model focuses on predicting the labels of the data.

Mathematical Intuition

In mathematical terms, discriminative machine learning trains a model, which is done by learning parameters that maximize the conditional probability $P(Y|X)$. On the other hand, a generative model learns parameters by maximizing the joint probability of $P(X, Y)$.

Applications

Discriminative models recognize existing data, i.e., discriminative modeling identifies tags and sorts data and can be used to classify data, while Generative modeling produces something.

Since these models use different approaches to machine learning, both are suited for specific tasks i.e., Generative models are useful for unsupervised learning tasks. In contrast, discriminative models are useful for supervised learning tasks. GANs(Generative adversarial networks) can be thought of as a competition between the generator, which is a component of the generative model, and the discriminator, so basically, it is generative vs. discriminative model.

Outliers

Generative models have more impact on outliers than discriminative models.

Computational Cost

Discriminative models are computationally cheap as compared to generative models.

Comparison Between Discriminative and Generative Models:

Some of the comparisons based on the following criteria between Discriminative and Generative Models:

Based on Performance

Generative models need fewer data to train compared with discriminative models since generative models are more biased as they make stronger assumptions, i.e., **assumption of conditional independence**.

Based on Missing Data

In general, if we have missing data in our dataset, then Generative models can work with these missing data, while discriminative models can't. This is because, in generative models, we can still estimate the posterior by marginalizing the unseen variables. However, discriminative models usually require all the features X to be observed.

Based on the Accuracy Score

If the assumption of conditional independence violates, then at that time, generative models are less accurate than discriminative models.

Based on Applications

Discriminative models are called “**discriminative**” since they are useful for discriminating Y 's label, i.e., target outcome, so they can only solve classification problems. In contrast, Generative models have more applications besides classification, such as samplings, Bayes learning, MAP inference, etc.

Generative Models vs Discriminative Models:

Machine learning (ML) and deep learning (DL) are two of the most exciting and constantly changing fields of study of the 21st century. Using these technologies, machines are given the ability to learn from past data and predict or make decisions from future, unseen data.

The inspiration comes from the human mind, how we use past experiences to help us make informed decisions in the present and the future. And while there are already many applications of ML and DL, the future possibilities are endless.

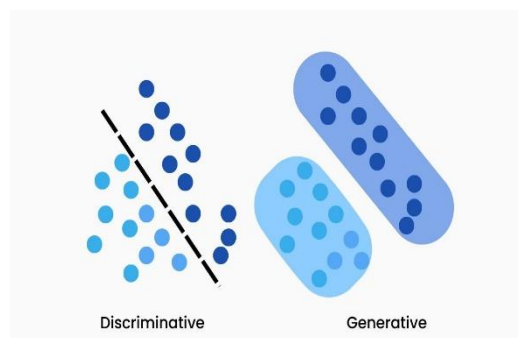
Computers utilize mathematics, algorithms, and data pipelines to draw meaningful inferences from raw data since they cannot perceive data and information like humans - not yet, at least. There are two ways we can improve a machine's efficiency: either get more data or come up with newer or more robust algorithms.

Quintillions of data are generated all over the world almost daily, so getting fresh data is easy. But in order to work with this gigantic amount of data, we need new algorithms or we need to scale up existing ones.

Mathematics, especially branches like calculus, probability, statistics, etc., is the backbone of these algorithms or models. They can be widely divided into two groups:

1. Discriminative models
2. Generative models

Mathematically, generative classifiers assume a functional form for $P(Y)$ and $P(X|Y)$, then generate estimated parameters from the data and use the Bayes' theorem to calculate $P(Y|X)$ (posterior probability). Meanwhile, discriminative classifiers assume a functional form of $P(Y|X)$ and estimate the parameters directly from the provided data.



Discriminative model

The majority of discriminative models, aka conditional models, are used for supervised machine learning. They do what they 'literally' say, separating the data points into different classes and learning the boundaries using probability estimates and maximum likelihood.

Outliers have little to no effect on these models. They are a better choice than generative models, but this leads to misclassification problems which can be a major drawback.

Here are some examples and a brief description of the widely used discriminative models:

1. Logistic regression: Logistic regression can be considered the linear regression of classification models. The main idea behind both the algorithms is similar, but while linear regression is used for predicting a continuous dependent variable, logistic regression is used to differentiate between two or more classes.

2. Support vector machines: This is a powerful learning algorithm with applications in both regression and classification scenarios. An n -dimensional space containing the data points is divided into classes by decision boundaries using support vectors. The best boundary is called a hyperplane.

3. Decision trees: A graphical tree-like model is used to map decisions and their probable outcomes. It could be thought of as a robust version of If-else statements.

A few other examples are commonly-used neural nets, k-nearest neighbor (KNN), conditional random field (CRF), random forest, etc.

Generative model

As the name suggests, generative models can be used to generate new data points. These models are usually used in unsupervised machine learning problems.

Generative models go in-depth to model the actual data distribution and learn the different data points, rather than model just the decision boundary between classes.

These models are prone to outliers, which is their only drawback when compared to discriminative models. The mathematics behind generative models is quite intuitive too. The method is not direct like in the case of discriminative models. To calculate $P(Y|X)$, they first estimate the prior probability $P(Y)$ and the likelihood probability $P(X|Y)$ from the data provided.

Putting the values into Bayes' theorem's equation, we get an accurate value for $P(Y|X)$.

$$posterior = \frac{prior \times likelihood}{evidence} \Rightarrow P(Y|X) = \frac{P(Y) \cdot P(X|Y)}{P(X)}$$

Here are some examples as well as a description of generative models:

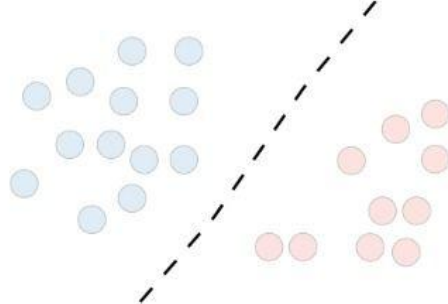
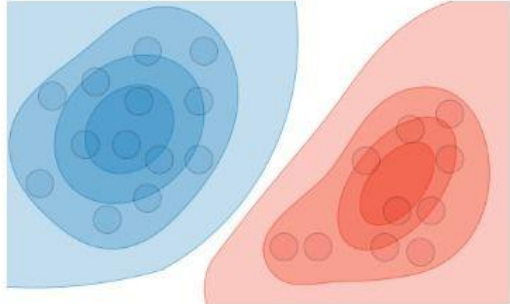
1. Bayesian network: Also known as Bayes' network, this model uses a directed acyclic graph (DAG) to draw Bayesian inferences over a set of random variables to calculate probabilities. It has many applications like prediction, anomaly detection, time series prediction, etc.

2. Autoregressive model: Mainly used for time series modeling, it finds a correlation between past behaviors to predict future behaviors.

3. Generative adversarial network (GAN): It's based on deep learning technology and uses two sub models. The generator model trains and generates new data points and the discriminative model classifies these 'generated' data points into real or fake.

Some other examples include Naive Bayes, Markov random field, hidden Markov model (HMM), latent Dirichlet allocation (LDA), etc.

Discriminative vs generative: Which is the best fit for Deep Learning?

	Discriminative model	Generative model
Goal	Directly estimate $P(y x)$	Estimate $P(x y)$ to then deduce $P(y x)$
What's learned	Decision boundary	Probability distributions of the data
Illustration		
Examples	Regressions, SVMs	GDA, Naive Bayes

Discriminative models divide the data space into classes by learning the boundaries, whereas generative models understand how the data is embedded into the space. Both the approaches are widely different, which makes them suited for specific tasks.

Deep learning has mostly been using supervised machine learning algorithms like artificial neural networks (ANNs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). ANN is the earliest in the trio and leverages artificial neurons, backpropagation, weights, and biases to identify patterns based on the inputs. CNN is mostly used for image recognition and computer vision tasks. It works by pooling important features from an input image. RNN, which is the latest of the three, is used in advanced fields like natural language processing, handwriting recognition, time series analysis, etc.

These are the fields where discriminative models are effective and better used for deep learning as they work well for supervised tasks.

Apart from these, deep learning and neural nets can be used to cluster images based on similarities. Algorithms like autoencoder, Boltzmann machine, and self-organizing maps are popular unsupervised deep learning algorithms. They make use of generative models for tasks like exploratory data analysis (EDA) of high dimensional datasets, image denoising, image compression, anomaly detection and even generating new images.

[This Person Does Not Exist - Random Face Generator](#) is an interesting website that uses a type of generative model called StyleGAN to create realistic human faces, even though the people in these images don't exist!



UNIT- V AUTO-ENCODERS: Auto-encoders, Architecture and components of auto-encoders (encoder and decoder), Training an auto-encoder for data compression and reconstruction, Relationship between Autoencoders and GANs, Hybrid Models: Encoder-Decoder GANs.

Auto-encoders:

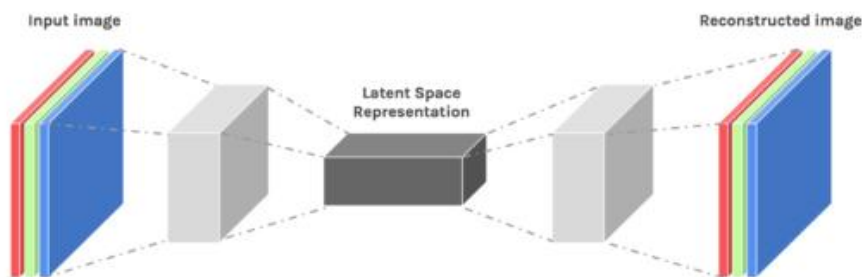
Autoencoders are a type of deep learning algorithm that are designed to receive an input and transform it into a different representation. They play an important part in image construction.

Artificial Intelligence encircles a wide range of technologies and techniques that enable computer systems to solve problems like Data Compression which is used in computer vision, computer networks, computer architecture, and many other fields.

Autoencoders are *unsupervised neural networks* that use machine learning to do this compression for us.

What Are Autoencoders?

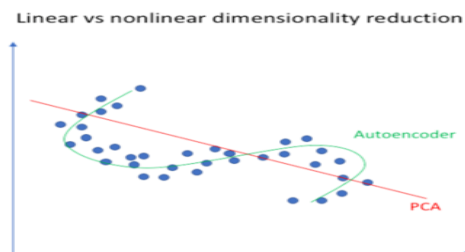
An autoencoder **neural network** is an **Unsupervised Machine learning** algorithm that applies backpropagation, setting the target values to be equal to the inputs. Autoencoders are used to reduce the size of our inputs into a smaller representation. If anyone needs the original data, they can reconstruct it from the compressed data.



We have a similar machine learning algorithm ie. PCA (principal component analysis) which does the same task.

Autoencoders: Its Emergence

Autoencoders are preferred over PCA because:



- An autoencoder can learn **non-linear transformations** with a **non-linear activation function** and multiple layers.
- It doesn't have to learn dense layers. It can use **convolutional layers** to learn which is better for video, image and series data.
- It is more efficient to learn several layers with an autoencoder rather than learn one huge transformation with PCA.
- An autoencoder provides a representation of each layer as the output.
- It can make use of **pre-trained layers** from another model to apply transfer learning to enhance the encoder/decoder.

We will look at a few Industrial Applications of Autoencoders.

Applications of Autoencoders

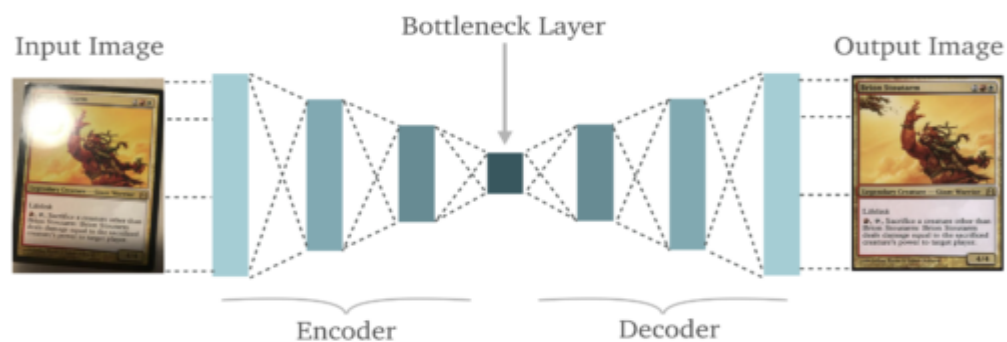
Image Coloring



Autoencoders are used for converting any black and white picture into a colored image. Depending on what is in the picture, it is possible to tell what the color should be.

Feature variation

It extracts only the required features of an image and generates the output by removing any noise or unnecessary interruption.



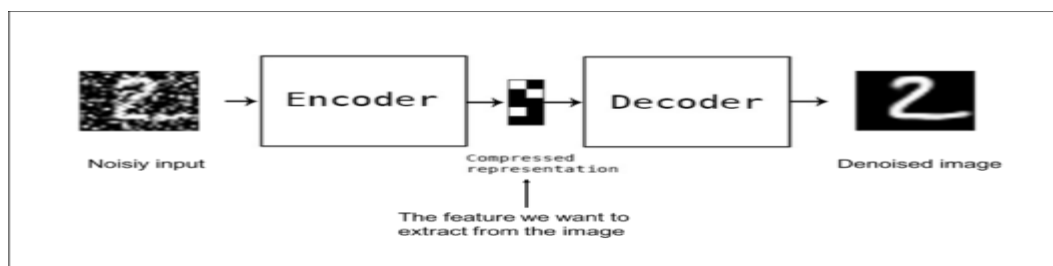
Dimensionality Reduction

The reconstructed image is the same as our input but with reduced dimensions. It helps in providing the similar image with a reduced pixel value.



Denoising Image

The input seen by the autoencoder is not the raw input but a stochastically corrupted version. A denoising autoencoder is thus trained to reconstruct the original input from the noisy version.



Watermark Removal

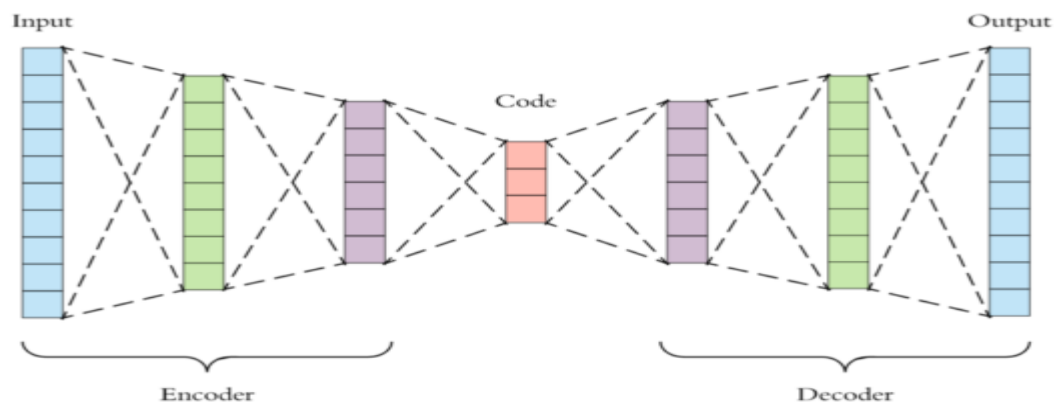
It is also used for removing watermarks from images or to remove any object while filming a video or a movie.



Architecture of Autoencoders

An Autoencoder consist of three layers:

1. **Encoder**
2. **Code**
3. **Decoder**



- **Encoder:** This part of the network compresses the input into a **latent space representation**. The encoder layer **encodes** the input image as a compressed representation in a reduced dimension. The compressed image is the distorted version of the original image.
- **Code:** This part of the network represents the compressed input which is fed to the decoder.
- **Decoder:** This layer **decodes** the encoded image back to the original dimension. The decoded image is a lossy reconstruction of the original image and it is reconstructed from the latent space representation.



The layer between the encoder and decoder, ie. the code is also known as **Bottleneck**. This is a well-designed approach to decide which aspects of observed data are relevant information and what aspects can be discarded. It does this by balancing two criteria:

- Compactness of representation, measured as the compressibility.
- It retains some behaviourally relevant variables from the input.

Training an auto-encoder for data compression and reconstruction:

An autoencoder consists of two parts: an encoder network and a decoder network. The encoder network compresses the input data, while the decoder network reconstructs the compressed data back into its original form. The compressed data, also known as the bottleneck layer, is typically much smaller than the input data.

The encoder network takes the input data and maps it to a lower-dimensional representation. This lower-dimensional representation is the compressed data. The decoder network takes this compressed data and maps it back to the original input data. The decoder network is essentially the inverse of the encoder network.

The bottleneck layer is the layer in the middle of the autoencoder that contains the compressed data. This layer is much smaller than the input data, which is what allows for compression. The size of the bottleneck layer determines the amount of compression that can be achieved.

Autoencoders differ from other deep learning architectures, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs), in that they do not require labeled data. Autoencoders can learn the underlying structure of the data without any explicit labels.

Image Compression with Autoencoders

There are two types of image compression: lossless and lossy. Lossless compression methods preserve all of the data in the original image, while lossy compression methods discard some of the data to achieve higher compression rates.

Autoencoders can be used for both lossless and lossy compression. Lossless compression can be achieved by using a bottleneck layer that is the same size as the input data. In this case, the autoencoder essentially learns to encode and decode the input data without any loss of information.

Lossy compression can be achieved by using a bottleneck layer that is smaller than the input data. In this case, the autoencoder learns to discard some of the data to achieve higher compression rates. The amount of data that is discarded depends on the size of the bottleneck layer.

Here are some examples of image compression using autoencoders:

- A 512×512 color image can be compressed to a 64×64 grayscale image using an autoencoder with a bottleneck layer of size 64.
- A 256×256 grayscale image can be compressed to a 128×128 grayscale image using an autoencoder with a bottleneck layer of size 128.

The effectiveness of autoencoder-based compression techniques can be evaluated by comparing the compressed and reconstructed images to the original images. The most common evaluation metric is the peak signal-to-noise ratio (PSNR), which measures the amount of noise introduced by the compression algorithm. Higher PSNR values indicate better compression quality.

Image Reconstruction with Autoencoders

Autoencoders are a type of neural network that can be used for image compression and reconstruction. The process involves compressing an image into a smaller representation and then reconstructing it back to its original form. Image reconstruction is the process of creating an image from compressed data.

Explanation of image reconstruction from compressed data:

The compressed data can be thought of as a compressed version of the original image. To reconstruct the image, the compressed data is fed through a decoder network, which expands the data back to its original size. The reconstructed image will not be identical to the original, but it will be a close approximation.

How autoencoders can be used for image reconstruction:

Autoencoders use a loss function to determine how well the reconstructed image matches the original. The loss function calculates the difference between the reconstructed image and the original image. The goal of the autoencoder is to minimize the loss function so that the reconstructed image is as close to the original as possible.

Examples of image reconstruction using autoencoders:

An example of image reconstruction using autoencoders is the MNIST dataset, which consists of handwritten digits. The autoencoder is trained on the dataset to compress and reconstruct the images. Another example is the CIFAR-10 dataset, which consists of 32x32 color images of objects. The autoencoder can be trained on this dataset to compress and reconstruct the images.

Evaluation of the effectiveness of autoencoder-based reconstruction techniques:

The effectiveness of autoencoder-based reconstruction techniques can be evaluated using metrics such as peak signal-to-noise ratio (PSNR) and structural similarity index (SSIM). PSNR measures the quality of the reconstructed image by comparing it to the original image, while SSIM measures the structural similarity between the reconstructed and original images.

Variations of Autoencoders for Image Compression and Reconstruction

Autoencoders can be modified and improved for better image compression and reconstruction. Some of the variations of autoencoders are:

Denoising autoencoders:

Denoising autoencoders are used to remove noise from images. The autoencoder is trained on noisy images and is trained to reconstruct the original image from the noisy input.

Variational autoencoders:

Variational autoencoders (VAEs) are a type of autoencoder that learn the probability distribution of the input data. VAEs are trained to generate new samples from the learned distribution. This makes VAEs suitable for image generation tasks.

Convolutional autoencoders:

Convolutional autoencoders (CAEs) use convolutional neural networks (CNNs) for image compression and reconstruction. CNNs are specialized neural networks that can learn features from images.

Comparison of the effectiveness of different types of autoencoders for image compression and reconstruction:

The effectiveness of different types of autoencoders for image compression and reconstruction can be compared using metrics such as PSNR and SSIM. CAEs are generally more effective for image compression and reconstruction than other types of autoencoders. VAEs are better suited for image generation tasks.

Real-Time Examples:

A real-time example of an autoencoder for image compression and reconstruction is Google's Guetzli algorithm. Guetzli uses a combination of a perceptual metric and a psycho-visual model to compress images while maintaining their quality. Another example is the Deep Image Prior algorithm, which uses a convolutional neural network to reconstruct images from compressed data.

Applications of Autoencoders for Image Compression and Reconstruction

Autoencoders have become increasingly popular for image compression and reconstruction tasks due to their ability to learn efficient representations of the input data. In this, we will explore some of the common applications of autoencoders for image compression and reconstruction.

Medical Imaging:

Autoencoders have shown great promise in medical imaging applications such as Magnetic Resonance Imaging (MRI), Computed Tomography (CT), and X-Ray imaging. The ability of autoencoders to learn feature representations from high-dimensional data has made them useful for compressing medical images while preserving diagnostic information. For example, researchers have developed a deep learning-based autoencoder approach for compressing 3D MRI images, which achieved higher compression ratios than traditional compression methods while preserving diagnostic quality. This can have significant implications for improving the storage and transmission of medical images, especially in resource-limited settings.

Video Compression:

Autoencoders have also been used for video compression, where the goal is to compress a sequence of images into a compact representation that can be transmitted or stored efficiently. One example of this is the video codec AV1, which uses a combination of autoencoders and traditional compression methods to achieve higher compression rates while maintaining video quality. The autoencoder component of the codec is used to learn spatial and temporal features of the video frames, which are then used to reduce redundancy in the video data.

Autonomous Vehicles:

Autoencoders are also useful for autonomous vehicle applications, where the goal is to compress high-resolution camera images captured by the vehicle's sensors while preserving critical information for navigation and obstacle detection. For example, researchers have developed an autoencoder-based approach for compressing images captured by a self-driving car, which achieved high compression ratios while preserving the accuracy of object detection algorithms. This can have significant implications for improving the performance and reliability of autonomous vehicles, especially in scenarios where high-bandwidth communication is not available.

Social Media and Web Applications:

Autoencoders have also been used in social media and web applications, where the goal is to reduce the size of image files to improve website loading times and reduce bandwidth usage. For example, Facebook uses an autoencoder-based approach for compressing images uploaded to their platform, which achieves high compression ratios while preserving image quality. This has led to faster loading times for images on the platform and reduced data usage for users.

Comparison of the effectiveness of autoencoder-based compression and reconstruction techniques for different applications:

The effectiveness of autoencoder-based compression and reconstruction techniques can vary depending on the application and the specific requirements of the task. For example, in medical imaging applications, the preservation of diagnostic information is critical, while in social media applications, image quality and loading times may be more important. Researchers have compared the effectiveness of autoencoder-based compression and reconstruction techniques with traditional compression methods and have found that autoencoder-based methods often outperform traditional methods in terms of compression ratio and image quality.

Relationship between Autoencoders and GANs:

Autoencoders and GANs are both powerful techniques for learning from data in an unsupervised way, but they have some differences and trade-offs. Autoencoders are easier to train and more stable, but they tend to produce blurry or distorted reconstructions or generations. GANs are harder to train and more prone to mode collapse, where they produce only a few modes of the data distribution, but they tend to produce sharper and more diverse generations. Depending on your goal and your data, you might prefer one or the other, or even combine them in a hybrid model.

Autoencoders are unsupervised models, which means that they are not trained on labeled data. Instead, they are trained on unlabeled data and learn to reconstruct the input data. GANs, on the other hand, are supervised models, which means that they are trained on labeled data. The generator in a GAN is trained to generate data that looks like the labeled data, and the discriminator is trained to distinguish between real and fake data. Autoencoders are typically used for tasks such as image denoising and compression. GANs are typically used for tasks such as image generation and translation.

Hybrid Models: Encoder-Decoder GANs:

How can you combine GANs and autoencoders to create hybrid models for various tasks?

Generative adversarial networks (GANs) and autoencoders are two powerful types of artificial neural networks that can learn from data and generate new samples. But what if you could combine them to create hybrid models that can perform various tasks, **such as image synthesis, anomaly detection**, or domain adaptation? In this article, you will learn how GANs and autoencoders work, and how you can combine them to create hybrid models for various tasks.

GANs and autoencoders

GANs are composed of two networks: a generator and a discriminator. The generator tries to create realistic samples from random noise, while the discriminator tries to distinguish between real and fake samples. The two networks compete with each other, improving their skills over time. Autoencoders are composed of two networks: an encoder and a decoder. The encoder compresses the input data into a lower-dimensional representation, while the decoder reconstructs the input data from the representation. The goal is to minimize the reconstruction error, while learning useful features from the data.

Hybrid models

Hybrid models are models that combine GANs and autoencoders in different ways, depending on the task and the objective. For example, you can use an autoencoder as the generator of a GAN, and train it to fool the discriminator, while also minimizing the reconstruction error. This way, you can generate realistic samples that are similar to the input data, but also have some variations. Alternatively, you can use a GAN as the encoder of an autoencoder, and train it to encode the input data into a latent space that is compatible with the discriminator. This way, you can learn a meaningful representation of the data that can be used for downstream tasks, such as classification or clustering.

Image synthesis

One of the most common tasks for hybrid models is image synthesis, which is the process of creating new images from existing ones, or from scratch. For example, you can use a hybrid model to synthesize images of faces, animals, or landscapes, by using an autoencoder as the generator of a GAN, and feeding it with real images or random noise. This way, you can create diverse and realistic images that preserve the attributes of the input data, but also have some variations. You can also use a hybrid model to synthesize images of different domains, such as converting photos to paintings, or day to night, by using a GAN as the encoder of an autoencoder, and feeding it with images from both domains. This way, you can learn a common latent space that can be used to transfer the style or the attributes of one domain to another.

Anomaly detection

Another task for hybrid models is anomaly detection, which is the process of identifying abnormal or unusual patterns in the data, such as outliers, frauds, or defects. For example, you can use a hybrid model to detect anomalies in images, such as damaged products, or medical conditions, by using an autoencoder as the

generator of a GAN, and feeding it with normal images. This way, you can train the autoencoder to reconstruct normal images well, but fail to reconstruct abnormal images. Then, you can use the reconstruction error or the discriminator score as a measure of anomaly. You can also use a hybrid model to detect anomalies in time series, such as sensor readings, or financial transactions, by using a GAN as the encoder of an autoencoder, and feeding it with normal time series. This way, you can train the GAN to encode normal time series well, but fail to encode abnormal time series. Then, you can use the latent space or the discriminator score as a measure of anomaly.

Domain adaptation

A third task for hybrid models is domain adaptation, which is the process of adapting a model trained on one domain to work on another domain, without requiring labeled data from the target domain. For example, you can use a hybrid model to adapt a model trained on images of handwritten digits to work on images of handwritten letters, by using a GAN as the encoder of an autoencoder, and feeding it with images from both domains. This way, you can train the GAN to encode both domains into a shared latent space that is invariant to the domain differences. Then, you can use the latent space as the input for a classifier or a decoder that can work on both domains.

