1. Discuss about Historical trends in Deep Learning?

A Short History Of Deep Learning —

Deep learning is a topic that is making big waves at the moment. It is basically a branch of machine learning (another hot topic) that uses algorithms to e.g. recognize objects and understand human speech. Scientists have used deep learning algorithms with multiple processing layers (hence "deep") to make better models from large quantities of unlabelled data (such as photos with no description, voice recordings or videos on YouTube).

It's one kind of supervised machine learning, in which a computer is provided a training set of examples to learn a function, where each example is a pair of an input and an output from the function.

Very simply: if we give the computer a picture of a cat and a picture of a ball, and show it which one is the cat, we can then ask it to decide if subsequent pictures are cats. The computer compares the image to its training set and makes an answer. Today's algorithms can also do this unsupervised; that is, they don't need every decision to be pre-programmed.

Of course, the more complex the task, the bigger the training set has to be. Google's voice recognition algorithms operate with a massive training set — yet it's not nearly big enough to predict every possible word or phrase or question you could put to it.

But it's getting there. Deep learning is responsible for recent advances in computer vision, speech recognition, natural language processing, and audio recognition.

Deep learning is based on the concept of artificial neural networks, or computational systems that mimic the way the human brain functions. And so, our brief history of deep learning must start with those neural networks.

1943: Warren McCulloch and Walter Pitts create a computational model for neural networks based on mathematics and algorithms called threshold logic.

1958: Frank Rosenblatt creates the perceptron, an algorithm for pattern recognition based on a two-layer computer neural network using simple addition and subtraction. He also proposed additional layers with mathematical notations, but these wouldn't be realized until 1975.

1980: Kunihiko Fukushima proposes the Neoconitron, a hierarchical, multilayered artificial neural network that has been used for handwriting recognition and other pattern recognition problems.

1989: Scientists were able to create algorithms that used deep neural networks, but training times for the systems were measured in days, making them impractical for real-world use.

1992: Juyang Weng publishes Cresceptron, a method for performing 3-D object recognition automatically from cluttered scenes.

Mid-2000s: The term "deep learning" begins to gain popularity after a paper by Geoffrey Hinton and Ruslan Salakhutdinov showed how a many-layered neural network could be pre-trained one layer at a time.

2009: NIPS Workshop on Deep Learning for Speech Recognition discovers that with a large enough data set, the neural networks don't need pre-training, and the error rates drop significantly.

2012: Artificial pattern-recognition algorithms achieve human-level performance on certain tasks. And Google's deep learning algorithm discovers cats.

2014: Google buys UK artificial intelligence startup Deepmind for £400m

2015: Facebook puts deep learning technology – called DeepFace – into operations to automatically tag and identify Facebook users in photographs. Algorithms perform superior face recognition tasks using deep networks that take into account 120 million parameters.

2016: Google DeepMind's algorithm AlphaGo masters the art of the complex board game Go and beats the professional go player Lee Sedol at a highly publicised tournament in Seoul.

The promise of deep learning is not that computers will start to think like humans. That's a bit like asking an apple to become an orange. Rather, it demonstrates that given a large enough data set, fast enough

processors, and a sophisticated enough algorithm, computers can begin to accomplish tasks that used to be completely left in the realm of human perception — like recognizing cat videos on the web (and other, perhaps more useful purposes).

2. Explain about Machine Learning Basics?

What is Machine Learning?

In 1959, **Arthur Samuel**, a computer scientist who pioneered the study of artificial intelligence, described machine learning as "the study that gives computers the ability to learn without being explicitly programmed."

Alan Turing's seminal paper (**Turing,** 1950) introduced a benchmark standard for demonstrating machine intelligence, such that a machine has to be intelligent and responsive in a manner that cannot be differentiated from that of a human being.

Machine Learning is an application of artificial intelligence where a computer/machine learns from the past experiences (input data) and makes future predictions. The performance of such a system should be at least human level.

A more technical definition given by **Tom M. Mitchell**'s (1997) : "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." Example:

A handwriting recognition learning problem: Task T: recognizing and classifying handwritten words within images

Performance measure P: percent of words correctly classified, accuracy

Training experience E: a data-set of handwritten words with given classifications

In order to perform the task T, the system learns from the data-set provided. A data-set is a collection of many examples. An example is a collection of features.

Machine Learning Categories

Machine Learning is generally categorized into three types:

- **Supervised Learning,**
- **Unsupervised Learning,**
- **Reinforcement learning**

Supervised Learning: In supervised learning the machine experiences the examples along with the labels or targets for each example. The labels in the data help the algorithm to correlate the features.

Two of the most common supervised machine learning tasks is **classification** and **regression**.

In **classification** problems the machine must learn to predict discrete values. That is, the machine must predict the most probable category, class, or label for new examples. Applications of classification include predicting whether a stock's price will rise or fall, or deciding if a news article belongs to the politics or leisure section.

In **regression** problems the machine must predict the value of a continuous response variable. Examples of regression problems include predicting the sales for a new product, or the salary for a job based on its description.

Unsupervised Learning: When we have unclassified and unlabeled data, the system attempts to uncover patterns from the data. There is no label or target given for the examples. One common task is to group similar examples together called clustering.

Reinforcement Learning: Reinforcement learning refers to goal-oriented algorithms, which learn how to attain a complex objective (goal) or maximize along a particular dimension over many steps. This method allows machines and software agents to automatically determine the ideal behavior within a specific context in order to maximize its performance. Simple reward feedback is required for the agent to learn which action is best; this is known as the reinforcement signal. For example, maximize the points won in a game over many moves.

3. Explain about Deep Learning Algorithms:

Deep Learning Algorithms: Deep learning algorithms are dynamically made to run through several **layers** of neural networks, which are nothing but a set of decision-making networks that are pre-trained to serve a task. Later, each of these is passed through simple layered representations and move on to the next layer. However, most machine learning is trained to work fairly well on datasets that have to deal with hundreds of features or columns. For a data set to be structured or unstructured, machine learning tends to fail mostly because they fail to recognize a simple image having a dimension of **800x1000** in RGB. It becomes quite unfeasible for a traditional machine learning algorithm to handle such depths. This is where deep learning. The following are the Deep Learning Algorithms:

1. Convolutional Neural Networks (CNNs)

CNN's popularly known as **ConvNets** majorly consists of several layers and are specifically used for image processing and detection of objects. It was developed in **1998** by **Yann LeCun** and was first called **LeNet.** Back then, it was developed to recognize digits and zip code characters. CNNs have wide usage in identifying the image of the satellites, medical image processing, series forecasting, and anomaly detection.

2. Long Short Term Memory Networks (LSTMs)

LSTMs can be defined as **Recurrent Neural Networks** (RNN) that are programmed to learn and adapt for dependencies for the long term. It can memorize and recall past data for a greater period and by default, it is its sole behavior. LSTMs are designed to retain over time and henceforth they are majorly used in time series predictions because they can restrain memory or previous inputs.

3. Recurrent Neural Networks (RNNs)

Recurrent Neural Networks or RNNs consist of some directed connections that form a cycle that allow the input provided from the LSTMs to be used as input in the current phase of RNNs. These inputs are deeply embedded as inputs and enforce the memorization ability of LSTMs lets these inputs get absorbed for a period in the internal memory. RNNs are therefore dependent on the inputs that are preserved by LSTMs and work under the synchronization phenomenon of LSTMs. RNNs are mostly used in captioning the image, time series analysis, recognizing handwritten data, and translating data to machines.

4. Generative Adversarial Networks (GANs)

GANs are defined as deep learning algorithms that are used to generate new instances of data that match the training data. GAN usually consists of two components namely a **generator** that learns to generate false data and a **discriminator** that adapts itself by learning from this false data. Over some time, GANs have gained immense usage since they are frequently being used to clarify **astronomical images** and simulate **lensing** the gravitational dark matter.

5. Radial Basis Function Networks (RBFNs)

RBFNs are specific types of neural networks that follow a feed-forward approach and make use of radial functions as activation functions. They consist of **three** layers namely the **input layer, hidden layer,** and **output layer** which are mostly used for **time-series prediction, regression testing,** and **classification.**

6. Multilayer Perceptrons (MLPs)

MLPs are the base of deep learning technology. It belongs to a class of feed-forward neural networks having various layers of **perceptrons.** These perceptrons have various activation functions in them. MLPs also have connected input and output layers and their number is the same. Also, there's a layer that remains hidden amidst these two layers. MLPs are mostly used to build **image and speech recognition** systems or some other types of the **translation software.**

7. Self Organizing Maps (SOMs)

SOMs were invented by **Teuvo Kohenen** for achieving data visualization to understand the dimensions of data through artificial and self-organizing neural networks. The attempts to achieve data visualization to solve problems are mainly done by what humans cannot visualize. These data are generally high-dimensional so there are lesser chances of human involvement and of course less error.

8. Deep Belief Networks (DBNs)

DBNs are called generative models because they have various layers of latent as well as stochastic variables. The latent variable is called a **hidden unit** because they have binary values. DBNs are also called **Boltzmann Machines** because the **RGM** layers are stacked over each other to establish communication with previous and consecutive layers. DBNs are used in applications like video and image recognition as well as capturing motional objects.

9. Restricted Boltzmann Machines (RBMs)

RBMs were developed by **Geoffrey Hinton** and resemble stochastic neural networks that learn from the probability distribution in the given input set. This algorithm is mainly used in the field of dimension **reduction, regression** and **classification, topic modeling** and are considered the building blocks of DBNs. RBIs consist of two layers namely the **visible layer** and the **hidden layer**.

4. <u>Explain about Linear algebra for Machine Learning?</u>

Linear Algebra for Machine learning : Machine learning has a strong connection with mathematics. Each machine learning algorithm is based on the concepts of mathematics & also with the help of mathematics, one can choose the correct algorithm by considering training time, complexity, number of features, etc. ***Linear Algebra is an essential field of mathematics, which defines the study of vectors, matrices, planes, mapping, and lines required for linear transformation.***

The term Linear Algebra was initially introduced in the early 18th century to find out the unknowns in Linear equations and solve the equation easily; hence it is an important branch of mathematics that helps study data. Also, no one can deny that Linear Algebra is undoubtedly the important and primary thing to process the applications of Machine Learning. It is also a prerequisite to start learning Machine Learning and data science.

Linear algebra plays a vital role and key foundation in machine learning***, and it enables ML algorithms to run on a huge number of datasets***.

The concepts of linear algebra are widely used in developing algorithms in machine learning. Although it is used almost in each concept of Machine learning, specifically, it can perform the following task:

- **Optimization of data.**
- **Applicable in loss functions, regularization, covariance matrices, Singular Value Decomposition (SVD), Matrix Operations, and support vector machine classification.**
- **Implementation of Linear Regression in Machine Learning.**

Besides the above uses, linear algebra is also used in neural networks and the data science field.

Basic mathematics principles and concepts like Linear algebra are the foundation of Machine Learning and Deep Learning systems. To learn and understand Machine Learning or Data Science, one needs to be familiar with linear algebra and optimization theory

Why learn Linear Algebra before learning Machine Learning?

Linear Algebra is just similar to the flour of bakery in Machine Learning. As the cake is based on flour similarly, every Machine Learning Model is also based on Linear Algebra. Further, the cake also needs more ingredients like egg, sugar, cream, soda. Similarly, Machine Learning also requires more concepts as vector calculus, probability, and optimization theory. So, we can say that Machine Learning creates a useful model with the help of the above-mentioned mathematical concepts.

Below are some popular examples of linear algebra in Machine learning:

- **Datasets and Data Files**
- **Linear Regression**
- **Recommender Systems**
- **One-hot encoding**
- **Regularization**
- **Principal Component Analysis**
- **Images and Photographs**
- **Singular-Value Decomposition**
- **Deep Learning**
- **Latent Semantic Analysis**

Below are some benefits of learning Linear Algebra before Machine learning:

- **Better Graphic experience**
- **Improved Statistics**
- **Creating better Machine Learning algorithms**
- **Estimating the forecast of Machine Learning**
- **Easy to Learn**

Notation: Notation in linear algebra enables you to read algorithm descriptions in papers, books, and websites to understand the algorithm's working. Even if you use for-loops rather than matrix operations, you will be able to piece things together.

Operations: Working with an advanced level of abstractions in vectors and matrices can make concepts clearer, and it can also help in the description, coding, and even thinking capability. In linear algebra, it is required to learn the basic operations such as addition, multiplication, inversion, transposing of matrices, vectors, etc.

Matrix Factorization: One of the most recommended areas of linear algebra is matrix factorization, specifically matrix deposition methods such as SVD and QR.

5. Explain about Machine Learning Testing?

Machine Learning testing: First of all, what are we trying to achieve when performing ML testing, as well as any software testing whatsoever?

- Quality assurance is required to make sure that the software system works according to the requirements. Were all the features implemented as agreed? Does the program behave as expected? All the parameters that you test the program against should be stated in the technical specification document.

- Moreover, software testing has the power to point out all the defects and flaws during development. You don’t want your clients to encounter bugs after the software is released and come to you waving their fists. Different kinds of testing allow us to catch bugs that are visible only during runtime.

However, in machine learning, a programmer usually inputs the data and the desired behavior, and the logic is elaborated by the machine. This is especially true for deep learning. Therefore, the purpose of machine learning testing is, first of all, to ensure that this learned logic will remain consistent, no matter how many times we call the program.

Model evaluation in machine learning testing

Usually, software testing includes:

Unit tests. The program is broken down into blocks, and each element (unit) is tested separately.

Regression tests. They cover already tested software to see if it doesn't suddenly break.

Integration tests. This type of testing observes how multiple components of the program work together.

Moreover, there are certain rules that people follow: don't merge the code before it passes all the tests, always test newly introduced blocks of code, when fixing bugs, write a test that captures the bug.

Machine learning adds up more actions to your to-do list. You still need to follow ML's best practices. Moreover, every ML model needs not only to be tested but **evaluated**. Your model should generalize well. This is not what we usually understand by testing, but evaluation is needed to make sure that the performance is satisfactory.

First of all, you split the database into three non-overlapping sets. You use a training set to train the model. Then, to evaluate the performance of the model, you use two sets of data:

- **Validation set.** Having only a training set and a testing set is not enough if you do many rounds of hyperparameter-tuning (which is always). And that can result in overfitting. To avoid that, you can select a small validation data set to evaluate a model. Only after you get maximum accuracy on the validation set, you make the testing set come into the game.

- **Test set (or holdout set).** Your model might fit the training dataset perfectly well. But where are the guarantees that it will do equally well in real-life? In order to assure that, you select samples for a testing set from your training set — examples that the machine hasn't seen before. It is important to remain unbiased during selection and draw samples at random. Also, you should not use the same set many times to avoid training on your test data. Your test set should be large enough to provide statistically meaningful results and be representative of the data set as a whole.

- But just as test sets, validation sets "wear out" when used repeatedly. The more times you use the same data to make decisions about hyperparameter settings or other model improvements, the less confident you are that the model will generalize well on new, unseen data. So it is a good idea to collect more data to 'freshen up' the test set and validation set.

Cross-validation: Cross-validation is a model evaluation technique that can be performed even on a limited dataset. The training set is divided into small subsets, and the model is trained and validated on each of these samples.

Evaluate models using metrics: Evaluating the performance of the model using different metrics is integral to every data science project. Here is what you have to keep an eye on:

Accuracy: Accuracy is a metric for how much of the predictions the model makes are true. The higher the accuracy is, the better. However, it is not the only important metric when you estimate the performance.

6. Elaborate Cross Validation?

Cross-Validation in Machine Learning: Cross-validation is a technique for validating the model efficiency by training it on the subset of input data and testing on previously unseen subset of the input data. ***We can also say that it is a technique to check how a statistical model generalizes to an independent dataset***.

In machine learning, there is always the need to test the stability of the model. It means based only on the training dataset; we can't fit our model on the training dataset. For this purpose, we reserve a particular sample of the dataset, which was not part of the training dataset. After that, we test our model on that sample before deployment, and this complete process comes under cross-validation. This is something different from the general train-test split.

Hence the basic steps of cross-validations are:

- Reserve a subset of the dataset as a validation set.
- Provide the training to the model using the training dataset.

- Now, evaluate model performance using the validation set. If the model performs well with the validation set, perform the further step, else check for the issues.

Methods used for Cross-Validation

There are some common methods that are used for cross-validation. These methods are given below:

1. **Validation Set Approach**
2. **Leave-P-out cross-validation**
3. **Leave one out cross-validation**
4. **K-fold cross-validation**
5. **Stratified k-fold cross-validation**

Validation Set Approach: We divide our input dataset into a training set and test or validation set in the validation set approach. Both the subsets are given 50% of the dataset.

But it has one of the big disadvantages that we are just using a 50% dataset to train our model, so the model may miss out to capture important information of the dataset. It also tends to give the underfitted model.

Leave-P-out cross-validation: In this approach, the p datasets are left out of the training data. It means, if there are total n datapoints in the original input dataset, then n-p data points will be used as the training dataset and the p data points as the validation set. This complete process is repeated for all the samples, and the average error is calculated to know the effectiveness of the model.

There is a disadvantage of this technique; that is, it can be computationally difficult for the large p.

Leave one out cross-validation: This method is similar to the leave-p-out cross-validation, but instead of p, we need to take 1 dataset out of training. It means, in this approach, for each learning set, only one datapoint is reserved, and the remaining dataset is used to train the model. This process repeats for each datapoint. Hence for n samples, we get n different training set and n test set. It has the following features:

- In this approach, the bias is minimum as all the data points are used.
- The process is executed for n times; hence execution time is high.
- This approach leads to high variation in testing the effectiveness of the model as we iteratively check against one data point.

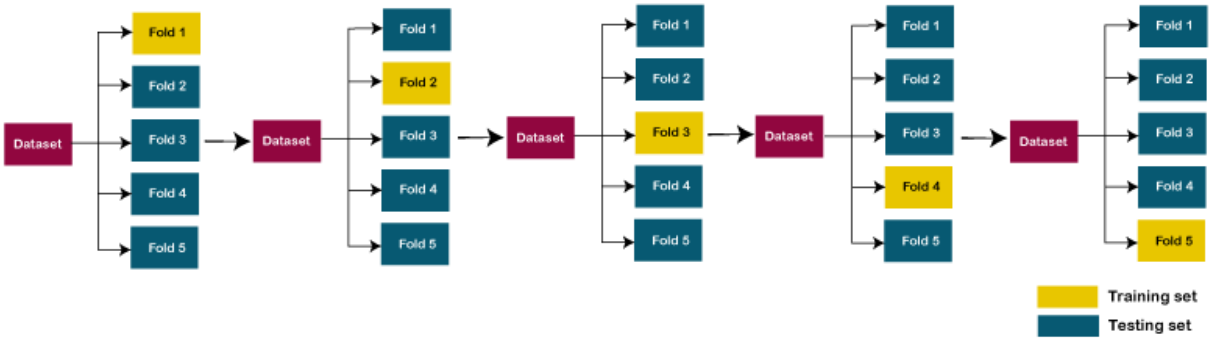
K-Fold Cross-Validation: K-fold cross-validation approach divides the input dataset into K groups of samples of equal sizes. These samples are called **folds**. For each learning set, the prediction function uses k-1 folds, and the rest of the folds are used for the test set. This approach is a very popular CV approach because it is easy to understand, and the output is less biased than other methods.

The steps for k-fold cross-validation are:

- Split the input dataset into K groups
- For each group:
  - Take one group as the reserve or test data set.
  - Use remaining groups as the training dataset
  - Fit the model on the training set and evaluate the performance of the model using the test set.

Let's take an example of 5-folds cross-validation. So, the dataset is grouped into 5 folds. On 1^{st} iteration, the first fold is reserved for test the model, and rest are used to train the model. On 2^{nd} iteration, the second fold is used to test the model, and rest are used to train the model. This process will continue until each fold is not used for the test fold.

Consider the below diagram:



Stratified k-fold cross-validation: This technique is similar to k-fold cross-validation with some little changes. This approach works on stratification concept, it is a process of rearranging the data to ensure that each fold or group is a good representative of the complete dataset. To deal with the bias and variance, it is one of the best approaches.

It can be understood with an example of housing prices, such that the price of some houses can be much high than other houses. To tackle such situations, a stratified k-fold cross-validation technique is useful.

Holdout Method

This method is the simplest cross-validation technique among all. In this method, we need to remove a subset of the training data and use it to get prediction results by training it on the rest part of the dataset.

The error that occurs in this process tells how well our model will perform with the unknown dataset. Although this approach is simple to perform, it still faces the issue of high variance, and it also produces misleading results sometimes.

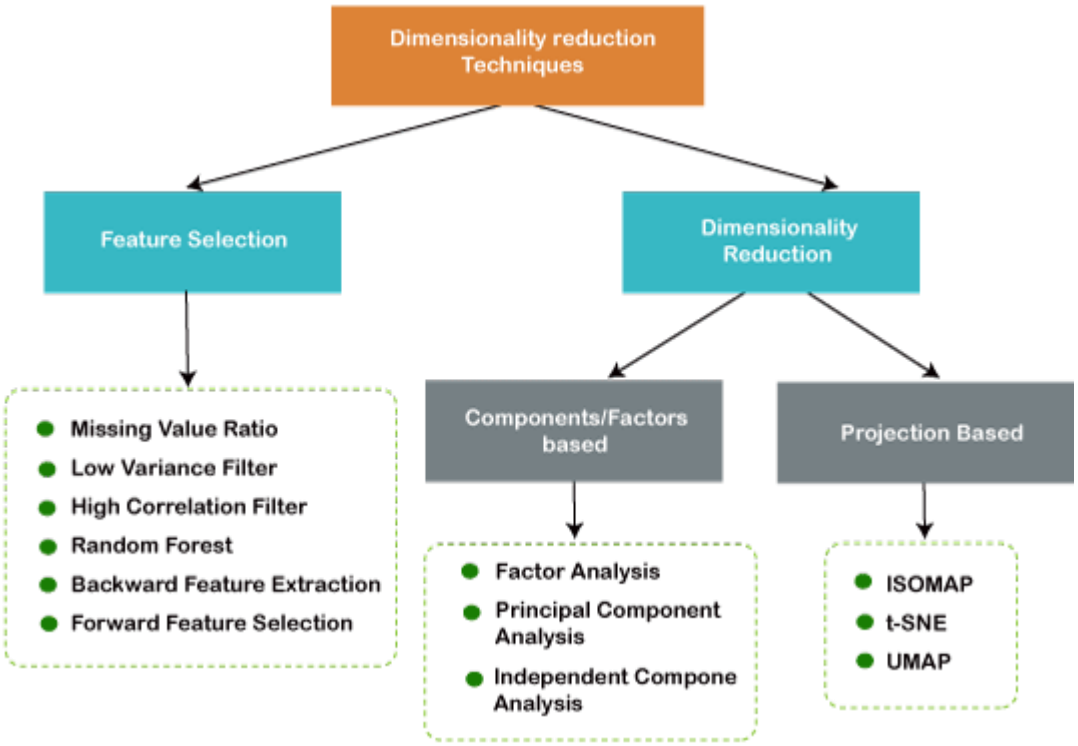
7. Illustrate the Dimensionality Reduction?

Dimensionality Reduction: The number of input features, variables, or columns present in a given dataset is known as dimensionality, and the process to reduce these features is called dimensionality reduction.

A dataset contains a huge number of input features in various cases, which makes the predictive modeling task more complicated. Because it is very difficult to visualize or make predictions for the training dataset with a high number of features, for such cases, dimensionality reduction techniques are required to use.

Dimensionality reduction technique can be defined as, ***"It is a way of converting the higher dimensions dataset into lesser dimensions dataset ensuring that it provides similar information."*** These techniques are widely used in machine learning for obtaining a better fit predictive model while solving the classification and regression problems.

It is commonly used in the fields that deal with high-dimensional data, such as **speech recognition, signal processing, bioinformatics, etc. It can also be used for data visualization, noise reduction, cluster analysis**, etc



The Curse of Dimensionality: Handling the high-dimensional data is very difficult in practice, commonly known as the *curse of dimensionality*. If the dimensionality of the input dataset increases, any machine learning algorithm and model becomes more complex. As the number of features increases, the number of samples also gets increased proportionally, and the chance of overfitting also increases. If the machine learning model is trained on high-dimensional data, it becomes overfitted and results in poor performance.

Hence, it is often required to reduce the number of features, which can be done with dimensionality reduction.

Benefits of applying Dimensionality Reduction

Some benefits of applying dimensionality reduction technique to the given dataset are given below:

- By reducing the dimensions of the features, the space required to store the dataset also gets reduced.
- Less Computation training time is required for reduced dimensions of features.
- Reduced dimensions of features of the dataset help in visualizing the data quickly.
- **I**t removes the redundant features (if present) by taking care of multi-collinearity.

Disadvantages of dimensionality Reduction: There are also some disadvantages of applying the dimensionality reduction, which are given below:

- Some data may be lost due to dimensionality reduction.
- In the PCA dimensionality reduction technique, sometimes the principal components required to consider are unknown.

8. Explain about Approaches of Dimension reduction?

Approaches of Dimension Reduction: There are two ways to apply the dimension reduction technique, which are given below:

Feature Selection: Feature selection is the process of selecting the subset of the relevant features and leaving out the irrelevant features present in a dataset to build a model of high accuracy. In other words, it is a way of selecting the optimal features from the input dataset.

Three methods are used for the feature selection:

1. Filters Methods: In this method, the dataset is filtered, and a subset that contains only the relevant features is taken. Some common techniques of filters method are:

- **Correlation**
- **Chi-Square Test**
- **ANOVA**
- **Information Gain, etc.**

2 . Wrappers Methods: The wrapper method has the same goal as the filter method, but it takes a machine learning model for its evaluation. In this method, some features are fed to the ML model, and evaluate the performance. The performance decides whether to add those features or remove to increase the accuracy of the model. This method is more accurate than the filtering method but complex to work. Some common techniques of wrapper methods are:

- Forward Selection
- Backward Selection
- Bi-directional Elimination

3. Embedded Methods: Embedded methods check the different training iterations of the machine learning model and evaluate the importance of each feature. Some common techniques of Embedded methods are:

- **LASSO**
- **Elastic Net**
- **Ridge Regression, etc.**

Feature Extraction: Feature extraction is the process of transforming the space containing many dimensions into space with fewer dimensions. This approach is useful when we want to keep the whole information but use fewer resources while processing the information.

Some common feature extraction techniques are:

a. Principal Component Analysis

b. Linear Discriminant Analysis

c. Kernel PCA

d. Quadratic Discriminant Analysis

Common techniques of Dimensionality Reduction

a. Principal Component Analysis
a. Backward Elimination
b. Forward Selection
c. Score comparison
d. Missing Value Ratio
e. Low Variance Filter
f. High Correlation Filter
g. Random Forest
h. Factor Analysis
i. Auto-Encoder

Principal Component Analysis (PCA)

Principal Component Analysis is a statistical process that converts the observations of correlated features into a set of linearly uncorrelated features with the help of orthogonal transformation. These new transformed features are called the **Principal Components**. It is one of the popular tools that is used for exploratory data analysis and predictive modeling.

PCA works by considering the variance of each attribute because the high attribute shows the good split between the classes, and hence it reduces the dimensionality. Some real-world applications of PCA are ***image processing, movie recommendation system, optimizing the power allocation in various communication channels.***

Backward Feature Elimination: The backward feature elimination technique is mainly used while developing Linear Regression or Logistic Regression model. Below steps are performed in this technique to reduce the dimensionality or in feature selection:

- In this technique, firstly, all the n variables of the given dataset are taken to train the model.
- The performance of the model is checked.
- Now we will remove one feature each time and train the model on n-1 features for n times, and will compute the performance of the model.
- We will check the variable that has made the smallest or no change in the performance of the model, and then we will drop that variable or features; after that, we will be left with n-1 features.
- Repeat the complete process until no feature can be dropped.

In this technique, by selecting the optimum performance of the model and maximum tolerable error rate, we can define the optimal number of features require for the machine learning algorithms.

Forward Feature Selection: Forward feature selection follows the inverse process of the backward elimination process. It means, in this technique, we don't eliminate the feature; instead, we will find the best features that can produce the highest increase in the performance of the model. Below steps are performed in this technique:

- We start with a single feature only, and progressively we will add each feature at a time.
- Here we will train the model on each feature separately.
- The feature with the best performance is selected.
- The process will be repeated until we get a significant increase in the performance of the model.

Missing Value Ratio: If a dataset has too many missing values, then we drop those variables as they do not carry much useful information. To perform this, we can set a threshold level, and if a variable has missing values more than that threshold, we will drop that variable. The higher the threshold value, the more efficient the reduction.

Low Variance Filter: As same as missing value ratio technique, data columns with some changes in the data have less information. Therefore, we need to calculate the variance of each variable, and all data columns with variance lower than a given threshold are dropped because low variance features will not affect the target variable.

High Correlation Filter: High Correlation refers to the case when two variables carry approximately similar information. Due to this factor, the performance of the model can be degraded. This correlation between the independent numerical variable gives the calculated value of the correlation coefficient. If this value is higher than the threshold value, we can remove one of the variables from the dataset. We can consider those variables or features that show a high correlation with the target variable.

Random Forest: Random Forest is a popular and very useful feature selection algorithm in machine learning. This algorithm contains an in-built feature importance package, so we do not need to program it separately. In this technique, we need to generate a large set of trees against the target variable, and with the help of usage statistics of each attribute, we need to find the subset of features.

Random forest algorithm takes only numerical variables, so we need to convert the input data into numeric data using **hot encoding**.

Factor Analysis: Factor analysis is a technique in which each variable is kept within a group according to the correlation with other variables, it means variables within a group can have a high correlation between themselves, but they have a low correlation with variables of other groups.

We can understand it by an example, such as if we have two variables Income and spend. These two variables have a high correlation, which means people with high income spends more, and vice versa. So, such variables are put into a group, and that group is known as the **factor**. The number of these factors will be reduced as compared to the original dimension of the dataset.

Auto-encoders: One of the popular methods of dimensionality reduction is auto-encoder, which is a type of ANN or artificial neural network, and its main aim is to copy the inputs to their outputs. In this, the input is compressed into latent-space representation, and output is occurred using this representation. It has mainly two parts:

- **Encoder:** The function of the encoder is to compress the input to form the latent-space representation.
- **Decoder:** The function of the decoder is to recreate the output from the latent-space representation.

9.Explain about Hyperparameters?

Hyperparameters in Machine Learning: *Hyperparameters in Machine learning are those parameters that are explicitly defined by the user to control the learning process.* **These hyperparameters are used to improve the learning of the model, and their values are set before starting the learning process of the model.**

What are hyperparameters?

In Machine Learning/Deep Learning, a model is represented by its parameters. In contrast, a training process involves selecting the best/optimal hyperparameters that are used by learning algorithms to provide the best result. So, what are these hyperparameters? The answer is, "***Hyperparameters are defined as the parameters that are explicitly defined by the user to control the learning process.***"

Here the prefix "hyper" suggests that the parameters are top-level parameters that are used in controlling the learning process. The value of the Hyperparameter is selected and set by the machine learning engineer before the learning algorithm begins training the model. **Hence, these are external to the model, and their values cannot be changed during the training process**.

Some examples of Hyperparameters in Machine Learning

- The k in kNN or K-Nearest Neighbour algorithm
- Learning rate for training a neural network
- Train-test split ratio
- Batch Size
- Number of Epochs
- Branches in Decision Tree
- Number of clusters in Clustering Algorithm

Difference between Parameter and Hyperparameter?

There is always a big confusion between Parameters and hyperparameters or model hyperparameters. So, in order to clear this confusion, let's understand the difference between both of them and how they are related to each other.

Model Parameters: Model parameters are configuration variables that are internal to the model, and a model learns them on its own. For example**, W Weights or Coefficients of independent variables in the** Linear regression model**. or** Weights or Coefficients of independent variables in SVM, weight, and biases of a neural network, cluster centroid in clustering. **Some key points for model parameters are as follows:**

- They are used by the model for making predictions.
- They are learned by the model from the data itself
- These are usually not set manually.
- These are the part of the model and key to a machine learning Algorithm.

Model Hyperparameters: Hyperparameters are those parameters that are explicitly defined by the user to control the learning process. Some key points for model parameters are as follows:

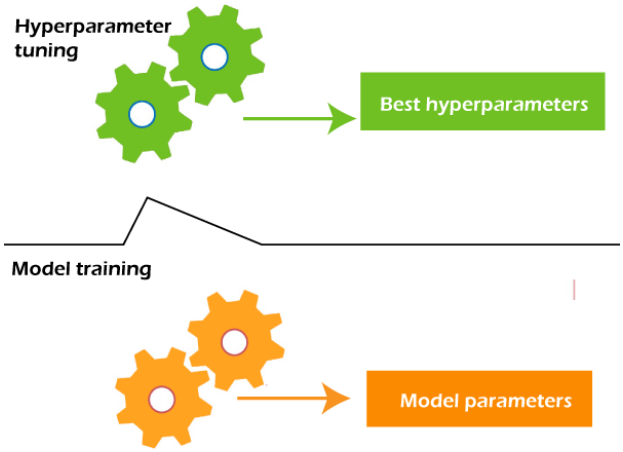
- These are usually defined manually by the machine learning engineer.

- One cannot know the exact best value for hyperparameters for the given problem. The best value can be determined either by the rule of thumb or by trial and error.
- Some examples of Hyperparameters are **the learning rate for training a neural network, K in the KNN algorithm,**

Categories of Hyperparameters: Broadly hyperparameters can be divided into two categories, which are given below:

1. **Hyperparameter for Optimization**
2. **Hyperparameter for Specific Models**

Hyperparameter for Optimization: The process of selecting the best hyperparameters to use is known as hyperparameter tuning, and the tuning process is also known as hyperparameter optimization. Optimization parameters are used for optimizing the model.



Some of the popular optimization parameters are as follows:

- **Learning Rate:** The learning rate is the hyperparameter in optimization algorithms that controls how much the model needs to change in response to the estimated error for each time when the model's weights are updated. It is one of the crucial parameters while building a neural network, and also it determines the frequency of cross-checking with model parameters. Selecting the optimized learning rate is a challenging task because if the learning rate is very less, then it may slow down the training process. On the other hand, if the learning rate is too large, then it may not optimize the model properly.
- **Batch Size:** To enhance the speed of the learning process, the training set is divided into different subsets, which are known as a batch. **Number of Epochs:** An epoch can be defined as the complete cycle for training the machine learning model. Epoch represents an iterative learning process. The number of epochs varies from model to model, and various models are created with more than one epoch. To determine the right number of epochs, a validation error is taken into account. The number of epochs is increased until there is a reduction in a validation error. If there is no improvement in reduction error for the consecutive epochs, then it indicates to stop increasing the number of epochs.

Hyperparameter for Specific Models: Hyperparameters that are involved in the structure of the model are known as hyperparameters for specific models. These are given below:

- **A number of Hidden Units:** Hidden units are part of neural networks, which refer to the components comprising the layers of processors between input and output units in a neural network.

It is important to specify the number of hidden units hyperparameter for the neural network. It should be between the size of the input layer and the size of the output layer. More specifically, the number of hidden units should be 2/3 of the size of the input layer, plus the size of the output layer.

for complex functions, it is necessary to specify the number of hidden units, but it should not overfit the model.

- **Number of Layers:** A neural network is made up of vertically arranged components, which are called layers. There are mainly **input layers, hidden layers, and output layers**. A 3-layered neural network gives a better performance than a 2-layered network. For a Convolutional Neural network, a greater number of layers make a better model.

10. Explain about "estimators"?

Estimators

Estimation is a statistical term for finding some estimate of unknown parameter, given some data. Point Estimation is the attempt to provide the single best prediction of some quantity of interest.

Quantity of interest can be:

- A single parameter
- A vector of parameters — e.g., weights in linear regression
- A whole function

Point estimator

To distinguish estimates of parameters from their true value, a point estimate of a parameter **θ**is represented by **θˆ.** Let **{x(1) , x(2) ,..x(m)}** be **m** independent and identically distributed data points. Then a point estimator is any function of the data:

$$\hat{\boldsymbol{\theta}}_m = g(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)})$$

This definition of a point estimator is very general and allows the designer of an estimator great flexibility. While almost any function thus qualifies as an estimator, a good estimator is a function whose output is close to the true underlying θ that generated the training data.

Point estimation can also refer to estimation of relationship between input and target variables referred to as function estimation.

Function Estimation

Here we are trying to predict a variable y given an input vector x. We assume that there is a function **f(x)** that describes the approximate relationship between y and x. For example,

we may assume that **y = f(x) + ε**, where **ε** stands for the part of **y** that is not predictable from **x**. In function estimation, we are interested in approximating **f** with a model or estimate **fˆ**. Function estimation is really just the same as estimating a parameter **θ**; the function estimator **fˆ**is simply a point estimator in function space. Ex: in polynomial regression we are either estimating a parameter ***w*** or estimating a function mapping from ***x*** to **y**.

11. Explain about Bias and Variance?

Bias and Variance: Bias and variance measure two different sources of error in an estimator. Bias measures the expected deviation from the true value of the function or parameter. Variance on the other hand, provides a measure of the deviation from the expected estimator value that any particular sampling of the data is likely to cause.

Bias: The bias of an estimator is defined as:

$$\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta}$$

where the expectation is over the data (seen as samples from a random variable)and ***θ***is the true underlying value of ***θ***used to define the data generating distribution.

An estimator **θˆm** is said to be unbiased if **bias(θˆm) = 0**, which implies that **E(θˆm) = θ**.

Variance and Standard Error: The variance of an estimator **Var(θˆ)** where the random variable is the training set. Alternately, the square root of the variance is called the standard error, denoted standard error **SE(ˆθ)**. The variance or the standard error of an estimator provides a measure of how we would expect the estimate we compute from data to vary as we independently re-sample the dataset from the underlying data generating process.

Just as we might like an estimator to exhibit low bias we would also like it to have relatively low variance.

Having discussed the definition of an estimator, let us now discuss some commonly used estimators.

Maximum Likelihood Estimator (MLE): Maximum Likelihood Estimation can be defined as a method for estimating parameters (such as the mean or variance) from sample data such that the probability (likelihood) of obtaining the observed data is maximized.

Consider a set of **m** examples **X = {x(1), . . . , x(m)}** drawn independently from the true but unknown data generating distribution **Pdata(x)**. Let **Pmodel(x; θ)** be a parametric family of probability distributions over the same space indexed by **θ**. In other words, **Pmodel(x; θ)** maps any configuration **x**to a real number estimating the true probability **Pdata(x)**.
The maximum likelihood estimator for **θ**is then defined as:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} p_{\mathrm{model}}(\mathbb{X}; \boldsymbol{\theta})$$

Since we assumed the examples to be i.i.d, the above equation can be written in the product form as:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\mathrm{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})$$

This product over many probabilities can be inconvenient for a variety of reasons. For example, it is prone to numerical underflow. Also, to find the maxima/minima of this function, we can take the derivative of this function w.r.t **θ**and equate it to 0. Since we have terms in product here, we need to apply the chain rule which is quite cumbersome with products. To obtain a more convenient but equivalent optimization problem, we observe that taking the logarithm of the likelihood does not change its arg max but does conveniently transform a product into a sum and since log is a strictly increasing function (natural log function is a monotone transformation), it would not impact the resulting value of $\boldsymbol{\theta}$.

So we have:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\mathrm{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}).$$

Two important properties: Consistency & Efficiency

Consistency: As the number of training examples approaches infinity, the maximum likelihood estimate of a parameter converges to the true value of the parameter.

Efficiency: A way to measure how close we are to the true parameter is by the expected mean squared error, computing the squared difference between the estimated and true parameter values, where the expectation is over **m** training samples from the data generating distribution. That parametric mean squared error decreases as **m** increases, and for **m** large, the Cramér-Rao lower bound shows that no consistent estimator has a lower mean squared error than the maximum likelihood estimator.

For the reasons of consistency and efficiency, maximum likelihood is often considered the preferred estimator to use for machine learning.

When the number of examples is small enough to yield over-fitting behavior, regularization strategies such as weight decay may be used to obtain a biased version of maximum likelihood that has less variance when training data is limited.

Maximum A Posteriori (MAP) Estimation: Following Bayesian approach by allowing the prior to influence the choice of the point estimate. The MAP can be used to obtain a point estimate of an unobserved

quantity on the basis of empirical data. The MAP estimate chooses the point of maximal posterior probability (or maximal probability density in the more common case of continuous θ):

$$\boldsymbol{\theta}_{\mathrm{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \boldsymbol{x}) = \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta})$$

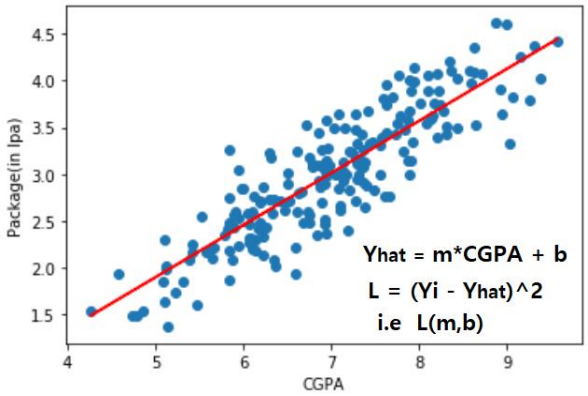
Where on the right hand side, **log p(x|θ)** is the standard log likelihood term, and **log p(θ)**, corresponding to the prior distribution.

As with full Bayesian inference, MAP Bayesian inference has the advantage of leveraging information that is brought by the prior and cannot be found in the training data. This additional information helps to reduce the variance in the MAP point estimate (in comparison to the ML estimate). However, it does so at the price of increased bias.

12. **Elaborate Loss function?**

Loss Function in Deep Learning: In mathematical optimization and decision theory, a loss or cost function (sometimes also called an error function) is a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event.

In simple terms, the Loss function is a method of evaluating how well your algorithm is modeling your dataset. It is a mathematical function of the parameters of the machine learning algorithm.
In simple linear regression, prediction is calculated using slope(m) and intercept(b). the loss function for this is the (Yi – Yihat)^2 i.e loss function is the function of slope and intercept.



Why Loss Function in Deep Learning is Important?
Famous author Peter Druker says You can't improve what you can't measure. That's why the loss function comes into the picture to evaluate how well your algorithm is modeling your dataset.

The loss function is very important in machine learning or deep learning. let's say you are working on any problem and you have trained a machine learning model on the dataset and are ready to put it in front of your client. But how can you be sure that this model will give the optimum result? Is there a metric or a technique that will help you quickly evaluate your model on the dataset? Yes, here loss functions come into play in machine learning or deep learning,

Loss Function in Deep Learning

1. Regression
    1. MSE(Mean Squared Error)
    2. MAE(Mean Absolute Error)
    3. Hubber loss
2. Classification
    1. Binary cross-entropy
    2. Categorical cross-entropy
3. Auto Encoder

KL Divergence

4. GAN
    1. Discriminator loss
    2. Minmax GAN loss
5. Object detection
    Focal loss
6. Word embeddings
    Triplet loss

Regression Loss:

1. Mean Squared Error/Squared loss/ L2 loss: The Mean Squared Error (MSE) is the simplest and most common loss function. To calculate the MSE, you take the difference between the actual value and model prediction, square it, and average it across the whole dataset.

$$MSE = \frac{1}{N}\sum_{i}^{N}(Yi - \hat{Y}i)^2$$

2. Mean Absolute Error/ L1 loss: The Mean Absolute Error (MAE) is also the simplest loss function. To calculate the MAE, you take the difference between the actual value and model prediction and average it across the whole dataset.

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|Yi - \hat{Y}i|$$

3. Huber Loss: In statistics, the Huber loss is a loss function used in robust regression, that is less sensitive to outliers in data than the squared error loss.

$$Huber = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}(y_i - \hat{y}_i)^2 \qquad |y_i - \hat{y}_i| \le \delta$$

$$Huber = \frac{1}{n}\sum_{i=1}^{n}\delta\left(|y_i - \hat{y}_i| - \frac{1}{2}\delta\right) \qquad |y_i - \hat{y}_i| > \delta$$

- n – the number of data points.
- y – the actual value of the data point. Also known as true value.
- ŷ – the predicted value of the data point. This value is returned by the model.
- δ – defines the point where the Huber loss function transitions from a quadratic to linear.

B. Classification Loss

1**. Binary Cross Entropy/log loss**: It is used in binary classification problems like two classes. example a person has covid or not or my article gets popular or not.

Binary cross entropy compares each of the predicted probabilities to the actual class output which can be either 0 or 1. It then calculates the score that penalizes the probabilities based on the distance from the expected value. That means how close or far from the actual value.

$$\text{Log Loss} = -\frac{1}{N}\sum_{i=1}^{N} y_i \log\hat{y}_i + (1-y_i)\log(1-\hat{y}_i)$$

- yi – actual values
- yihat – Neural Network prediction

2. **Categorical Cross Entropy**: Categorical Cross entropy is used for Multiclass classification and softmax regression.

loss function = -sum up to k(yjlagyjhat) where k is classes

$$\text{Loss} = -\sum_{j=1}^{K} y_j \log(\hat{y}_j)$$

where k is number of classes in the data

cost function = -1/n(sum upto n(sum j to k (yijloghijhat))

$$\text{Cost} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k}[y_{ij}\log(\hat{y}_{ij})]$$

where

- k is classes,
- y = actual value
- yhat – Neural Network prediction

13. **Elaborate Regularization?**

Regularization:

Regularization is one of the most important concepts of machine learning. It is a technique to prevent the model from overfitting by adding extra information to it.

Sometimes the machine learning model performs well with the training data but does not perform well with the test data. It means the model is not able to predict the output when deals with unseen data by introducing noise in the output, and hence the model is called overfitted. This problem can be deal with the help of a regularization technique.

This technique can be used in such a way that it will allow to maintain all variables or features in the model by reducing the magnitude of the variables. Hence, it maintains accuracy as well as a generalization of the model.

It mainly regularizes or reduces the coefficient of features toward zero. In simple words, "*In regularization technique, we reduce the magnitude of the features by keeping the same number of features.*"

How does Regularization Work?

Regularization works by adding a penalty or complexity term to the complex model. Let's consider the simple linear regression equation:

$$y= \beta0+\beta1x1+\beta2x2+\beta3x3+\cdots+\beta nxn +b$$

In the above equation, Y represents the value to be predicted

X1, X2, …Xn are the features for Y.

β0,β1,…..βn are the weights or magnitude attached to the features, respectively. Here represents the bias of the model, and b represents the intercept.

Linear regression models try to optimize the β0 and b to minimize the cost function. The equation for the cost function for the linear model is given below:

$$\sum_{i=1}^{M}(y_i - y'_i)^2 - \sum_{i=1}^{M}(y_i - \sum_{j=0}^{n}\beta_j * Xij)^2$$

Now, we will add a loss function and optimize parameter to make the model that can predict the accurate value of Y. The loss function for the linear regression is called as **RSS or Residual sum of squares.**

Techniques of Regularization

There are mainly two types of regularization techniques, which are given below:

- **Ridge Regression**
- **Lasso Regression**

Ridge Regression

- Ridge regression is one of the types of linear regression in which a small amount of bias is introduced so that we can get better long-term predictions.
- Ridge regression is a regularization technique, which is used to reduce the complexity of the model. It is also called as **L2 regularization**.
- In this technique, the cost function is altered by adding the penalty term to it. The amount of bias added to the model is called **Ridge Regression penalty**. We can calculate it by multiplying with the lambda to the squared weight of each individual feature.
- The equation for the cost function in ridge regression will be:

$$\sum_{i=1}^{M}(y_i - y'_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{n}\beta_j * x_{ij}\right)^2 + \lambda\sum_{j=0}^{n}\beta_j{}^2$$

- In the above equation, the penalty term regularizes the coefficients of the model, and hence ridge regression reduces the amplitudes of the coefficients that decreases the complexity of the model.
- As we can see from the above equation, if the values of **λ tend to zero, the equation becomes the cost function of the linear regression model.** Hence, for the minimum value of λ, the model will resemble the linear regression model.
- A general linear or polynomial regression will fail if there is high collinearity between the independent variables, so to solve such problems, Ridge regression can be used.
- It helps to solve the problems if we have more parameters than samples.

Lasso Regression:

- Lasso regression is another regularization technique to reduce the complexity of the model. It stands for **Least Absolute and Selection Operator.**
- It is similar to the Ridge Regression except that the penalty term contains only the absolute weights instead of a square of weights.
- Since it takes absolute values, hence, it can shrink the slope to 0, whereas Ridge Regression can only shrink it near to 0.
- It is also called as **L1 regularization.** The equation for the cost function of Lasso regression will be:

$$\sum_{i=1}^{M}(y_i - y'_i)^2 = \sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{n}\beta_j * x_{ij}\right)^2 + \lambda\sum_{j=0}^{n}|\beta_j|$$

- Some of the features in this technique are completely neglected for model evaluation.
- Hence, the Lasso regression can help us to reduce the overfitting in the model as well as the feature selection.

Key Difference between Ridge Regression and Lasso Regression:

- **Ridge regression** is mostly used to reduce the overfitting in the model, and it includes all the features present in the model. It reduces the complexity of the model by shrinking the coefficients.
- **Lasso regression** helps to reduce the overfitting in the model as well as feature selection.

1. Discuss about Biological Neuron?

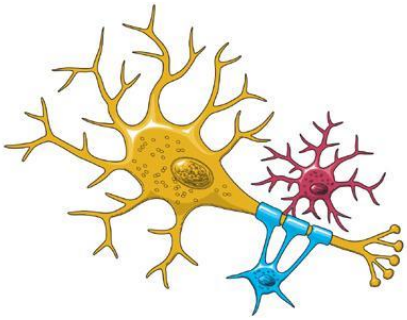
In the context of a neural network, a neuron is the most fundamental unit of processing. It's also called a perceptron. A neural network is based on the way a human brain works. So, we can say that it simulates the way the biological neurons signal to one another.

Biological Neuron: We can define neurons as the information carriers that use electrical impulses and chemical signals to transmit information. The neurons transmit the information in the following two areas:

1. different parts of the brain
2. the brain and the nervous system

Thus, whatever we think, feel, and later do is all due to the working of the neurons.

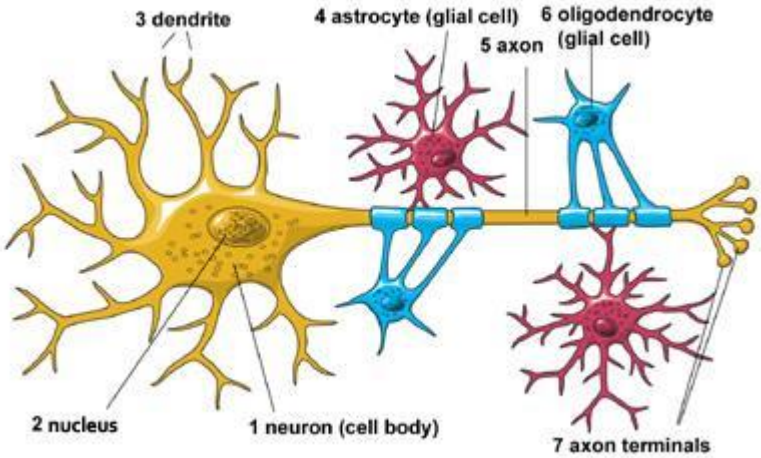
The following figure shows a typical biological neuron:



A neuron has the following three basic parts:

1. **Cell body**
2. **Cell extension Axon**
3. **Cell extension Dendrite**

The following figure shows the architecture of a biological neuron:



The nucleus in the cell body controls the cell's functioning. The axon extension (having a long tail) transmits messages from the cell. Dendrites extension (like a tree branch) receive messages for the cell.

So, in a nutshell, we can summarize that the biological neurons communicate with each other by sending chemicals, called neurotransmitters, across a tiny space, called a synapse, between the axons and dendrites of adjacent neurons.

Biological Neural Network: The ***biological neural network*** is also composed of several processing pieces known as ***neurons*** that are linked together via ***synapses***. These neurons accept either external input or the results of other neurons. The generated output from the individual neurons propagates its effect on the entire network to the last layer, where the results can be displayed to the outside world.

Every synapse has a processing value and weight recognized during network training. The performance and potency of the network fully depend on the neuron numbers in the network, how they are connected to each other (i.e., topology), and the weights assigned to every synapse.

Advantages and Disadvantages of Biological Neural Network: There are various advantages and disadvantages of the biological neural network. Some advantages and disadvantages of the biological neural network are as follows:

Advantages:

1. It can handle extremely complex parallel inputs.
2. The input processing element is the synapses.

Disadvantages:

1. As it is complex, the processing speed is slow.
2. There is no controlling mechanism in this network.

2. Illustrate Idea of computational units?

Deep Learning Activation Functions: Activation functions are a core concept to understand in deep learning.

They are what allow neurons in a neural network to communicate with each other through their synapses.

In this section, you will learn to understand the importance and functionality of activation functions in deep learning.

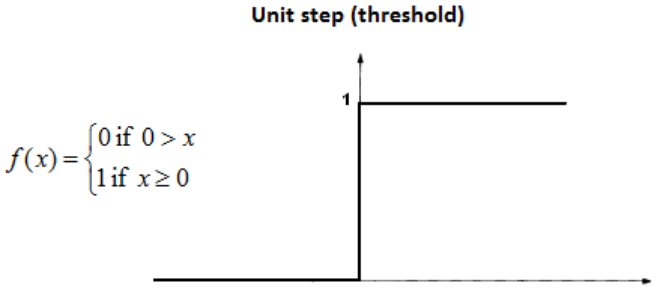
Activation Functions in Deep Learning: A weighted sum of these signals is fed into the neuron's activation function, and then the activation function's output is passed onto the next layer of the network.

There are four main types of activation functions:

1. Threshold functions
2. Sigmoid functions
3. Rectifier functions, or ReLUs
4. Hyperbolic Tangent functions

Threshold Functions: Threshold functions compute a different output signal depending on whether or not its input lies above or below a certain threshold. Remember, the input value to an activation function is the weighted sum of the input values from the preceding layer in the neural network.

Mathematically speaking, here is the formal definition of a deep learning threshold function:



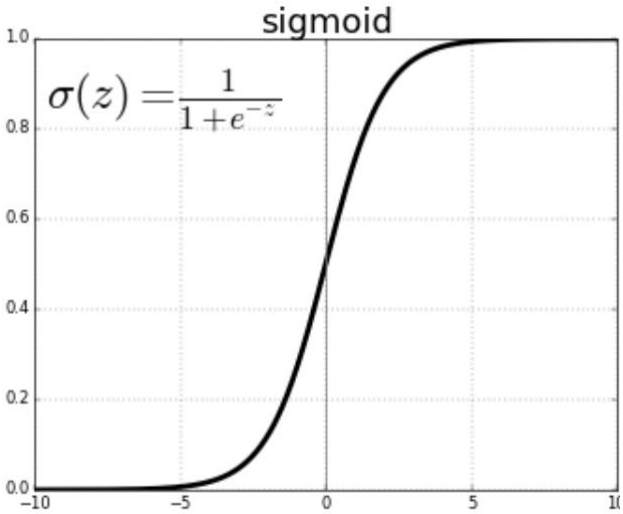
As the image above suggests, the threshold function is sometimes also called a unit step function.

Threshold functions are similar to Boolean variables in computer programming. Their computed value is either 1 (similar to True) or 0 (equivalent to False).

The Sigmoid Function: The sigmoid function is well-known among the data science community because of its use in logistic regression, one of the core deep learning techniques used to solve classification problems.

The sigmoid function can accept any value, but always computes a value between 0 and 1.

Here is the mathematical definition of the sigmoid function:



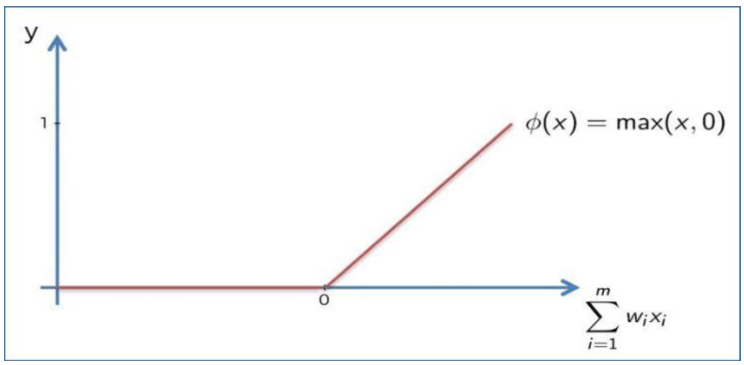
One benefit of the sigmoid function over the threshold function is that its curve is smooth. This means it is possible to calculate derivatives at any point along the curve.

The Rectifier Function: The rectifier function does not have the same smoothness property as the sigmoid function from the last section. However, it is still very popular in the field of deep learning.

The rectifier function is defined as follows:

- If the input value is less than 0, then the function outputs 0
- If not, the function outputs its input value

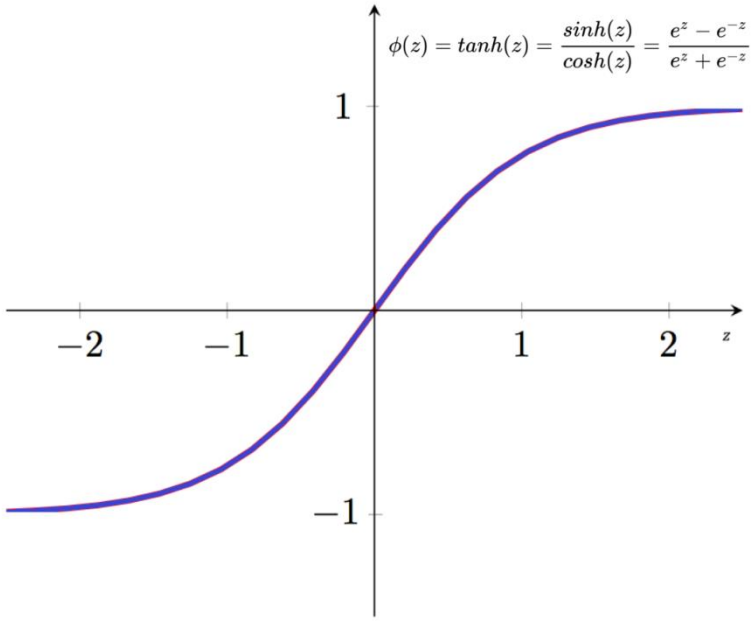
Here is this concept explained mathematically:



Rectifier functions are often called Rectified Linear Unit activation functions, or ReLUs for short.

The Hyperbolic Tangent Function: The hyperbolic tangent function is the only activation function, that is based on a trigonometric identity.

It’s mathematical definition is below:



The hyperbolic tangent function is similar in appearance to the sigmoid function, but its output values are all shifted downwards.

3. Explain about liner Perceptron?

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

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

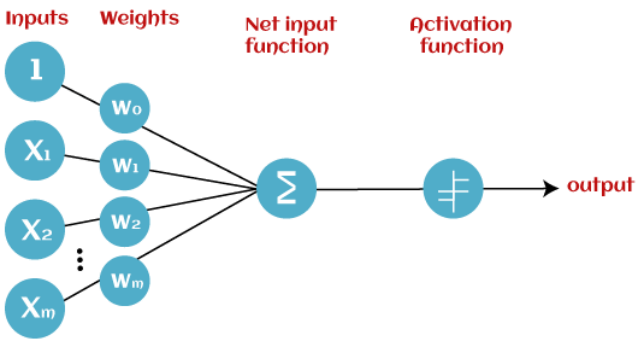
Binary classifier in Deep Learning?

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

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

Basic Components of Perceptron

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



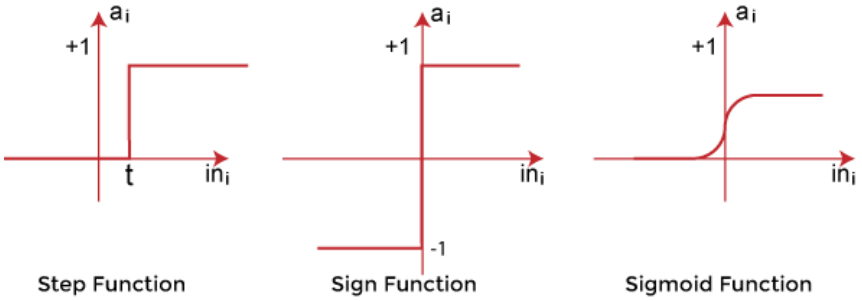
Input Nodes or Input Layer: This is the primary component of Perceptron which accepts the initial data into the system for further processing. Each input node contains a real numerical value.

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

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

Types of Activation functions:

- Sign function
- Step function, and
- Sigmoid function



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

4. Explain about Perceptron learning algorithm?

Perceptron is a linear supervised machine learning algorithm. It is used for binary classification. The perceptrons, which forms the basis for the most popular machine learning models nowadays – the n

eural networks.

Perceptron Learning Algorithm is also understood as an Artificial Neuron or neural network unit that helps to detect certain input data computations in business intelligence. The perceptron learning algorithm is treated as the most straightforward Artificial Neural network. It is a supervised learning algorithm of binary classifiers. Hence, it is a single-layer neural network with four main parameters, **i.e.**, input values, weights and Bias, net sum, and an activation function.

There are four significant steps in a perceptron learning algorithm:

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

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

   Add another essential term called bias 'b' to the weighted sum to improve the model performance.

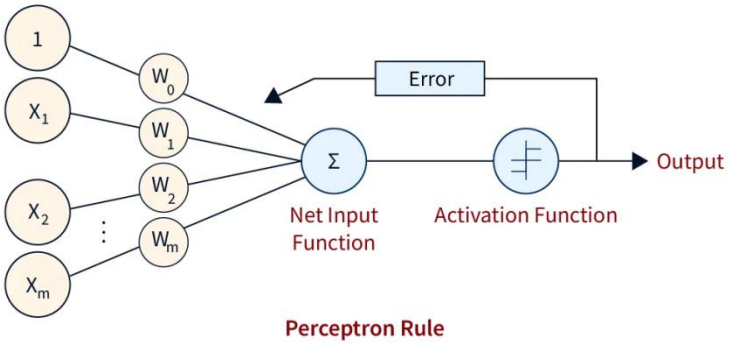
   $wi*xi+b$.

2. Next, an activation function is applied to this weighed sum, producing a binary or a continuous-valued output. $Y=f(\sum wi*xi+b)$

3. Next, the difference between this output and the actual target value is computed to get the error term, E, generally in terms of mean squared error. The steps up to this form the forward propagation part of the algorithm.
   $E=(Y-Yactual)2$

4. We optimize this error (loss function) using an optimization algorithm. Generally, some form of gradient descent algorithm is used to find the optimal values of the hyperparameters like learning rate, weight, Bias, etc. This step forms the backward propagation part of the algorithm.

An overview of this algorithm is illustrated in the following Figure:



Perceptron Rule

In a more standardized notation, the perceptron learning algorithm is as follows:

```
P <-- inputs with label 1
N <-- inputs with label 0
Initialise w randomly;
while !converge do:
$\hspace{2em}$ Pick random x $\in P \cup N;$
$\hspace{2em}$ if x $\in$ P and w.x $<$ 0 then
$\hspace{3em}$ w = w+x
$\hspace{2em}$ end
$\hspace{3em}$ if x $\in$ N and w.x $\ge$ 0 then
$\hspace{3em}$ w = w-x
$\hspace{2em}$ end
end
```

Basic Components of Perceptron:

1. **Input Nodes or Input Layer:** Primary component of Perceptron learning algorithm, which accepts the initial input data into the model. Each input node contains an actual value.
2. **Weight and Bias:** The weight parameter represents the strength of the connection between units. Bias can be considered as the line of intercept in a linear equation.
3. **Activation Function:** Final and essential components help determine whether the neuron will fire. The activation function can be primarily considered a step function. There are various types of activation functions used in a perceptron learning algorithm. Some of them are the sign function, step function, sigmoid function, etc.

4.Elobarate Convergence Theorem for perceptron learning?

Perceptron Convergence Theorem:

In the classification of linearly separable patterns belonging to two classes only, the training task for the classifier was to find the weight w such that.

(w^tx>0\hspace{0.4cm} for\hspace{0.2cm}each \hspace{0.2cm}x\in X_1\ w^tx<0\hspace{0.4cm} for\hspace{0.2cm}each \hspace{0.2cm}x\in X_2\)

Completion of training with the fixed correction training rule for any initial weight vector and any correction increment constant leads to the following weights:

w∗=wk0=wk0+1=wk0+2.....

with w∗ as the solution vector for equation.

Integer k0 is the training step number starting at which no more misclassification occurs, and thus no right adjustments take place for (k_0>=0)

This theorem is called as the "Perceptron Convergence Theorem".

Perceptron Convergence theorem states that a classifier for two linearly separable classes of patterns is always trainable in a finite number of training steps.

In summary, the training of a single discrete perceptron two class classifier requires a change of weights if and only if a misclassification occurs.

In the reason for misclassification is (w^tx<0\) then all weights are increased in proportion wo xi.

If \(w^tx>0) then all weights are decreased in proportion to xi

Summary of the Perceptron Convergence Algorithm:

Variables and Parameters: x(n)=(m+1) by 1 input vector

=[+1,x1(n),x2(n),.....xm(n)]T

w(n)=(m+1)= by 1 weight vector

=[b(n),w1(n),w2(n),.....wm(n)]T

b(n)= bias

y(n)= actual response

d(n)= desired response

η= learning rate parameter, a +ve constant less than unity

1. ***Initialization:*** Set w(0)=0 , then perform the following computations for time step n=1,2
2. ***Activation:*** At time step n, activate the perceptron by applying input vector x(n) and desired response d(n).
3. ***Computation of actual response:*** Compute the actual response of the perceptron:

   y(n)=sgn[wT(x)x(n)]
4. ***Adaptation of weight vector:*** Update the weight vector of the perceptron:

   w(n+1)=w(n)+η[d(n)−y(n)]x(n)
5. ***Continuation***: Increment time step n by 1, go to step 1

5.Explain about Linear seperability?

Linear Separability: Linear separability is an important concept in machine learning, particularly in the field of supervised learning. It refers to the ability of a set of data points to be separated into distinct categories using a linear decision boundary. In other words, if there exists a straight line that can cleanly divide the data into two classes, then the data is said to be linearly separable.

Linear separability is a concept in machine learning that refers to the ability to separate data points in binary classification problems using a linear decision boundary. If the data points can be separated using a line, linear function, or flat hyperplane, they are considered linearly separable. Linear separability is an important concept in neural networks, and it is introduced in the context of linear algebra and optimization theory.

In the context of machine learning, linear separability is an important property because it makes classification problems easier to solve. If the data is linearly separable, we can use a linear classifier, such as logistic regression or support vector machines (SVMs), to accurately classify new instances of data.

Linearly separable data points can be separated using a line, linear function, or flat hyperplane. In practice, there are several methods to determine whether data is linearly separable. One method is linear programming, which defines an objective function subjected to constraints that satisfy linear separability. Another method is to train and test on the same data - if there is a line that separates the data points, then the

accuracy or AUC should be close to 100%. If there is no such line, then training and testing on the same data will result in at least some error. Multi-layer neural networks can learn hidden features and patterns in data that linear classifiers cannot

To understand the concept of linear separability, it is helpful to first consider a simple two-dimensional example. Imagine we have a set of data points in a two-dimensional space, where each point is labeled either "red" or "blue". If these data points can be separated by a straight line, such that all the red points are on one side of the line and all the blue points are on the other side, then the data is linearly separable.

Python provides several methods to determine whether data is linearly separable. One method is linear programming, which defines an objective function subjected to constraints that satisfy linear separability. Another method is clustering, where if two clusters with cluster purity of 100% can be found using some clustering methods such as k-means, then the data is linearly separable.

However, not all data sets are linearly separable. In some cases, it may be impossible to draw a straight line that can separate the data into distinct categories. For example, imagine a set of data points that are arranged in a circular pattern, with red and blue points interspersed throughout. In this case, it is impossible to draw a straight line that separates the data into two classes.

When faced with data that is not linearly separable, machine learning algorithms must use more complex decision boundaries to accurately classify the data. For example, a decision tree or a neural network may be able to accurately classify data that is not linearly separable.

Linear separability is not only important in the context of machine learning, but it also has applications in other fields such as physics, biology, and economics. For example, in physics, linear separability can be used to analyze the relationship between two physical quantities. In biology, it can be used to study the behavior of animals or to analyze genetic data. In economics, it can be used to analyze the relationship between two economic variables.

Example:

One way to test for linear separability is to use linear programming. Linear programming defines an objective function subject to constraints that satisfy linear separability. The scipy.optimize.linprog() function in Python can be used to solve linear programming problems. Here's an example of using scipy.optimize.linprog() to test for linear separability:

```
import numpy as np
from scipy.optimize import linprog
 # Define the data points
X = np.array([[1, 2], [2, 3], [3, 1], [4, 3]])
y = np.array([1, 1, -1, -1])
# Define the objective function and constraints
c = np.zeros(X.shape[1] + 1)
c[-1] = 1
A = np.zeros((X.shape, X.shape[1] + 1))
A[:, :-1] = -y[:, np.newaxis] * X
A[:, -1] = -y
b = -np.ones(X.shape)
# Solve the linear programming problem
res = linprog(c, A_ub=A, b_ub=b)
 if res.success:
    print("The data is linearly separable.")
else:
```

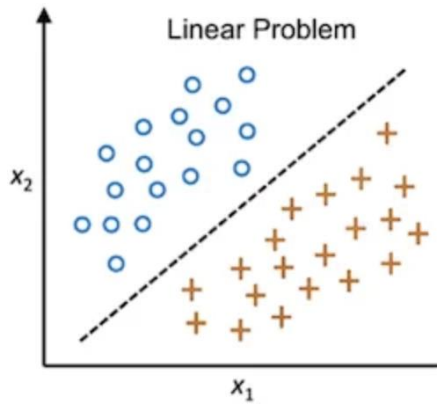
```
print("The data is not linearly separable.")
```

6.**Explain about Linear & Non-Linear Classification?**

Introduction: Linear Classification refers to categorizing a set of data points to a discrete class based on a linear combination of its explanatory variables. On the other hand, Non-Linear Classification refers to separating those instances that are not linearly separable.

Linear Classification:

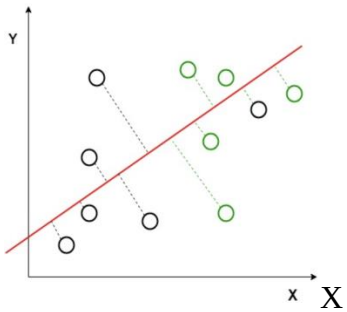
Linear Classification refers to categorizing a set of data points into a discrete class based on a linear combination of its explanatory variables. Some of the classifiers that use linear functions to separate classes are *Linear Discriminate Classifier, Naive Bayes, Logistic Regression, Perceptron, SVM (linear kernel)*.



In the figure above, we have two classes, namely 'O' and '+.' To differentiate between the two classes, an arbitrary line is drawn, ensuring that both the classes are on distinct sides. Since we can tell one class apart from the other, these classes are called 'linearly-separable.' However, an infinite number of lines can be drawn to distinguish the two classes. The exact location of this plane/hyperplane depends on the type of the linear classifier.

Linear Discriminate Classifier

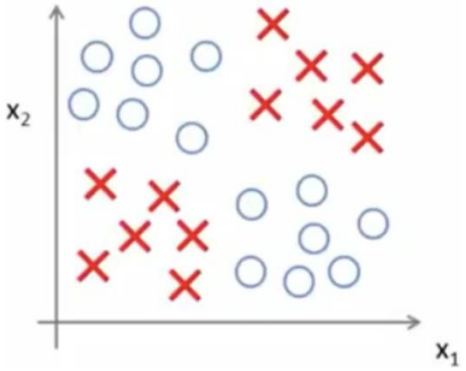
→ It is a dimensionality reduction technique in the domain of Supervised Machine Learning.
→ It is crucial in modeling differences between two groups, i.e., classes.
→ It helps project features in a high dimensions space in a lower-dimensional space.
→ Technique - Linear Discriminate Analysis (LDA) is used, which reduced the 2D graph into a 1D graph by creating a new axis. This helps to maximize the distance between the two classes for differentiation.



In the above graph, we notice that a new axis is created, which maximizes the distance between the mean of the two classes. As a result, variation within each class is also minimized. However, the problem with LDA is that it would fail in case the means of both the classes are the same. This would mean that we would not be able to generate a new axis for differentiating the two.

Non-Linear Classification:

Non-Linear Classification refers to categorizing those instances that are not linearly separable. Some of the classifiers that use non-linear functions to separate classes are Quadratic Discriminant Classifier, Multi-Layer Perceptron (MLP), Decision Trees, Random Forest, and K-Nearest Neighbours (KNN).



In the figure above, we have two classes, namely 'O' and 'X.' To differentiate between the two classes, it is impossible to draw an arbitrary straight line to ensure that both the classes are on distinct sides. We notice that even if we draw a straight line, there would be points of the first-class present between the data points of the second class. In such cases, piece-wise linear or non-linear classification boundaries are required to distinguish the two classes.

7. Discuss about Perceptron and its models?

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

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

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

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

Single Layer Perceptron Model:

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

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

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

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

Multi-Layered Perceptron Model:

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

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

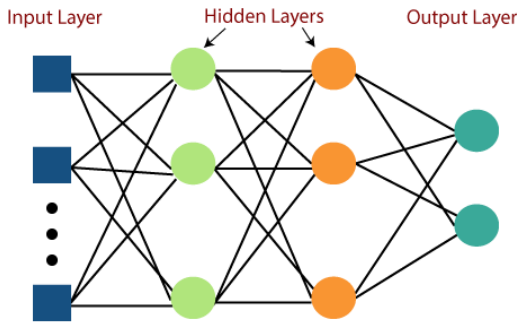
- **Forward Stage:** Activation functions start from the input layer in the forward stage and terminate on the output layer.
- **Backward Stage:** In the backward stage, weight and bias values are modified as per the model's requirement. In this stage, the error between actual output and demanded originated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

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

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

Multi-layer Perceptron in TensorFlow: Multi-Layer perceptron defines the most complex architecture of artificial neural networks. It is substantially formed from multiple layers of the perceptron. TensorFlow is a very popular deep learning framework released by, and this notebook will guide to build a neural network with this library. If we want to understand what is a Multi-layer perceptron, we have to develop a multi-layer perceptron from scratch using Numpy.

The pictorial representation of multi-layer perceptron learning is as shown below-

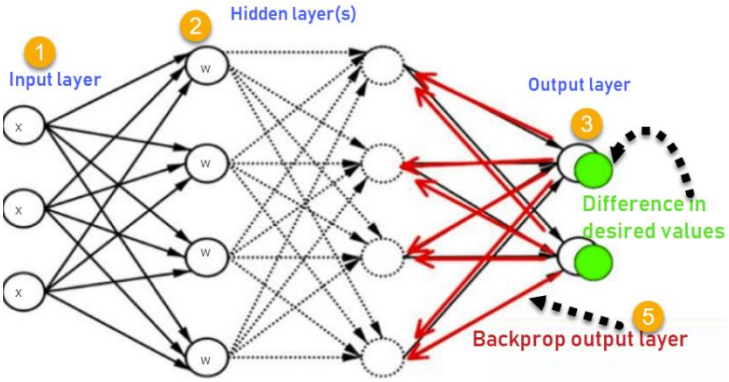


8. Explain about Back Propagation?

Backpropagation: is the essence of neural network training. It is the method of fine-tuning the weights of a neural network based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows you to reduce error rates and make the model reliable by increasing its generalization.
Backpropagation in neural network is a short form for "backward propagation of errors." It is a standard method of training artificial neural networks. This method helps calculate the gradient of a loss function with respect to all the weights in the network.

How Backpropagation Algorithm Works

The Back propagation algorithm in neural network computes the gradient of the loss function for a single weight by the chain rule. It efficiently computes one layer at a time, unlike a native direct computation. It computes the gradient, but it does not define how the gradient is used. It generalizes the computation in the delta rule.
Consider the following Back propagation neural network example diagram to understand:



How Backpropagation Algorithm Works

1. Inputs X, arrive through the preconnected path
2. Input is modeled using real weights W. The weights are usually randomly selected.
3. Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.
4. Calculate the error in the outputs

$Error_B$= Actual Output – Desired Output

5. Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased.

Keep repeating the process until the desired output is achieved

Why We Need Backpropagation?

Most prominent advantages of Backpropagation are:

- Backpropagation is fast, simple and easy to program
- It has no parameters to tune apart from the numbers of input
- It is a flexible method as it does not require prior knowledge about the network
- It is a standard method that generally works well
- It does not need any special mention of the features of the function to be learned.

What is a Feed Forward Network?

A feedforward neural network is an artificial neural network where the nodes never form a cycle. This kind of neural network has an input layer, hidden layers, and an output layer. It is the first and simplest type of artificial neural network.

Types of Backpropagation Networks

Two Types of Backpropagation Networks are:

- Static Back-propagation
- Recurrent Backpropagation

Static back-propagation:

It is one kind of backpropagation network which produces a mapping of a static input for static output. It is useful to solve static classification issues like optical character recognition.

Recurrent Backpropagation:

Recurrent Back propagation in data mining is fed forward until a fixed value is achieved. After that, the error is computed and propagated backward.
The main difference between both of these methods is: that the mapping is rapid in static back-propagation while it is nonstatic in recurrent backpropagation.

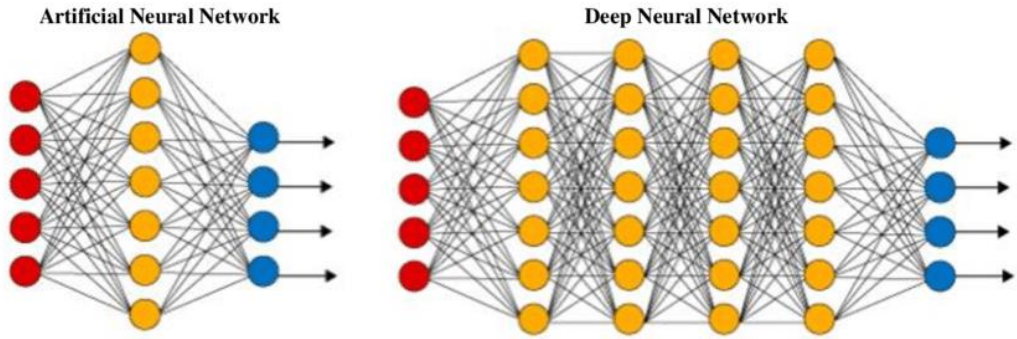
Unit – III

1. Give Introduction to Deep Neural Networks?

A deep neural network (DNN) is an ANN with multiple hidden layers between the input and output layers. Similar to shallow ANNs, DNNs can model complex non-linear relationships.

These networks usually consist of an input layer, one to two hidden layers, and an output layer. While it is possible to solve easy mathematical questions, and computer problems, including basic gate structures with their respective truth tables, it is tough for these networks to solve complicated image processing, computer vision, and natural language processing tasks.

For these problems, we utilize **deep neural networks**, which often have a complex hidden layer structure with a wide variety of different layers, such as a convolutional layer, max-pooling layer, dense layer, and other unique layers. These additional layers help the model to understand problems better and provide optimal solutions to complex projects. A deep neural network has more layers (more depth) than ANN and each layer adds complexity to the model while enabling the model to process the inputs concisely for outputting the ideal solution.



ANN vs DNN - Image Source

Deep neural networks have garnered extremely high traction due to their high efficiency in achieving numerous varieties of deep learning projects. Explore the differences **between machine learning vs deep learning** in a separate article.

Need of DNN: After training a well-built deep neural network, they can achieve the desired results with high accuracy scores. They are popular in all aspects of deep learning, including computer vision, natural language processing, and transfer learning.

The premier examples of the prominence of deep neural networks are their utility in object detection with models such as **YOLO** (You Only Look Once), language translation tasks with BERT (Bidirectional Encoder Representations from Transformers) models, transfer learning models, such as VGG-19, RESNET-50, efficient net, and other similar networks for image processing projects.

Applications for Deep Learning Software

Image recognition is one of the most common applications for deep learning software. By training a system on a large dataset of images, it can learn to recognize objects within them. This technology is being used in facial recognition systems as well as to enable object detection in autonomous vehicles.

Natural language processing (NLP) is another popular application for deep learning software. NLP allows computers to understand human language and respond appropriately to text or voice commands entered by users. This technology is behind virtual assistants such as Alexa and Siri, as well as many automated customer service agents.

OCR (Optical Character Recognition) is a technology that is capable of recognizing text from an image or video by matching it to stored templates. Businesses can use OCR to process large amounts of data very quickly.

Rossum uses deep learning to expand the capabilities of OCR and provide businesses with a more efficient and accurate method of capturing data from invoices or other documents.

By training a neural network to "read" a document like a human can, Rossum offers AI that is capable of accurately extracting the needed information from various document formats without requiring pre-programmed parameters.

Predictive analytics is another area where deep learning software is often applied. By analyzing past data sets, deep learning systems can identify patterns and trends that can predict future outcomes or behaviors with high accuracy levels.

Marketing professionals sometimes use this technology to target customers more effectively based on their past behavior or preferences. In addition, investors use it to create financial forecasting models that help them make better investment decisions based on historical market performance data.

2) What are Deep Learning Platforms?

A deep learning platform typically consists of several components, such as an artificial intelligence (AI) engine, data management tools, development frameworks for building custom models from scratch or fine-tuning existing ones, and deployment options for deploying trained models into production.

- Data management tools provide an efficient way of managing the datasets used to train deep learning models.
- Development frameworks allow users to easily build custom models from scratch or fine-tune existing ones according to their specific needs.
- Deployment options enable users to deploy their trained models into production environments such as cloud services or on-premise servers.

Deep learning platforms also include libraries of pre-trained models containing ready-to-use neural networks that are ready to deploy without any further training or customization. This makes it easy for developers and researchers to get started quickly without worrying about building their own model from the ground up.

Platforms for deep learning use artificial neural networks to enable digital systems to learn and make decisions based on unstructured, unlabeled data.

Artificial neural networks are composed of interconnected nodes that process information in a similar way as the neurons in the human brain. The nodes are organized into layers, and each layer is responsible for a specific task such as pattern recognition or decision-making.

When these networks are trained using large amounts of data, they can learn to recognize patterns and make decisions without being explicitly programmed. This allows them to process unstructured, unlabeled data and draw accurate conclusions from it.

As deep learning has become more popular and widely used, a number of different platforms have emerged to facilitate its development. The most popular deep learning platforms include:

- **Google's TensorFlow** – ideal for large-scale machine learning projects due to its scalability and flexibility
- **Microsoft's Cognitive Toolkit (CNTK)** – commercial-grade distributed deep learning
- **Amazon Web Services (AWS) Deep Learning AMIs** – optimized for cloud computing
- **Apache MXNet** – supports a wide range of programming languages
- **PyTorch** – designed specifically for research purposes

Each of these deep learning libraries offers unique features and capabilities that are suited for different types of tasks. When comparing deep learning platforms, there are several factors to consider. First, it's important to assess the platform's scalability (how easily it can handle large datasets or complex models) as well as its performance in terms of speed and accuracy.

Additionally, developers should look at the platform's ease of use (how quickly they can get up and running with their project) as well as the availability of deep learning tutorials or other resources that can help them learn how to use the platform effectively.

3) What are Deep Learning Tools and Libraries?

Constructing neural networks from scratch helps programmers to understand concepts and solve trivial tasks by manipulating these networks. However, building these networks from scratch is time-consuming and requires enormous effort. To make deep learning simpler, we have several tools and libraries at our disposal to yield an effective deep neural network model capable of solving complex problems with a few lines of code.

The most popular deep learning libraries and tools utilized for constructing deep neural networks are TensorFlow, Keras, and PyTorch. The Keras and TensorFlow libraries have been linked synonymously since the start of TensorFlow 2.0. This integration allows users to develop complex neural networks with high-level code structures using Keras within the TensorFlow network etc

Deep learning is a subfield of machine learning involving artificial neural networks, which are algorithms inspired by the structure of the human brain. Deep learning has many applications and is used in many of today's AI technologies, such as self-driving cars, news aggregation tools, natural language processing (NLP), virtual assistants, visual recognition, and much more.

In recent years, Python has proven to be an incredible tool for deep learning. Because the code is concise and readable, it makes it a perfect match for deep learning applications. Its simple syntax also enables applications to be developed faster when compared to other programming languages. Another major reason for using Python for deep learning is that the language can be integrated with other systems coded in different programming languages. This makes it easier to blend it with AI projects written in other languages.

1. TensorFlow

TensorFlow is widely considered one of the best Python libraries for deep learning applications. Developed by the Google Brain Team, it provides a wide range of flexible tools, libraries, and community resources. Beginners and professionals alike can use TensorFlow to construct deep learning models, as well as neural networks.

TensorFlow has an architecture and framework that are flexible, enabling it to run on various computational platforms like CPU and GPU. With that said, it performs best when operated on a tensor processing unit (TPU). The Python library is often used to implement reinforcement learning in deep learning models, and you can directly visualize the machine learning models. Some of the main features of TensorFlow:

- Flexible architecture and framework.
- Runs on a variety of computational platforms.
- Abstraction capabilities
- Manages deep neural networks.

2. Pytorch

Another one of the most popular Python libraries for deep learning is Pytorch, which is an open-source library created by Facebook's AI research team in 2016. The name of the library is derived from Torch, which is a deep learning framework written in the **Lua** programming language.

PyTorch enables you to carry out many tasks, and it is especially useful for deep learning applications like NLP and computer vision.

Some of the best aspects of PyTorch include its high speed of execution, which it can achieve even when handling heavy graphs. It is also a flexible library, capable of operating on simplified processors or CPUs and GPUs. PyTorch has powerful APIs that enable you to expand on the library, as well as a natural language toolkit. Some of the main features of PyTorch:

- Statistical distribution and operations
- Control over datasets
- Development of deep learning models
- Highly flexible

3. NumPy

One of the other well-known Python libraries, NumPy can be seamlessly utilized for large multi-dimensional array and matrix processing. It relies on a large set of high-level mathematical functions, which makes it especially useful for efficient fundamental scientific computations in deep learning.

NumPy arrays require a lot less storage area than other Python lists, and they are faster and more convenient to use. The data can be manipulated in the matrix, transposed, and reshaped with the library. NumPy is a great option to increase the performance of deep learning models without too much complex work required. Some of the main features of NumPy:

- Shape manipulation
- High-performance N-dimensional array object
- Data cleaning/manipulation
- Statistical operations and linear algebra

4. Scikit-Learn

Scikit-Learn was originally a third-party extension to the SciPy library, but it is now a standalone Python library on Github. Scikit-Learn includes DBSCAN, gradient boosting, support vector machines, and random forests within the classification, regression, and clustering methods.

One of the greatest aspects of Scikit-Learn is that it's easily interoperable with other SciPy stacks. It is also user-friendly and consistent, making it easier to share and use data. Some of the main features of Scikit-learn:

- Data classification and modeling
- End-to-end machine learning algorithms
- Pre-processing of data
- Model selection

5. SciPy

SciPy is one of the best Python libraries out there thanks to its ability to perform scientific and technical computing on large datasets. It is accompanied by embedded modules for array optimization and linear algebra.

The programming language includes all of NumPy's functions, but it turns them into user-friendly, scientific tools. It is often used for image manipulation and provides basic processing features for high-level, non-scientific mathematical functions. Some of the main features of SciPy:

- User-friendly
- Data visualization and manipulation
- Scientific and technical analysis
- Computes large data sets

6. Pandas

One of the open-source Python libraries mainly used in data science and deep learning subjects is Pandas. The library provides data manipulation and analysis tools, which are used for analyzing data. The library relies on its powerful data structures for manipulating numerical tables and time series analysis.

The Pandas library offers a fast and efficient way to manage and explore data by providing Series and DataFrames, which represent data efficiently while also manipulating it in different ways. Some of the main features Pandas:

- Indexing of data
- Data alignment
- Merging/joining of datasets
- Data manipulation and analysis

7. Microsoft CNTK: Another Python library for deep learning applications is Microsoft CNTK (Cognitive Toolkit), which is formerly known as Computational Network ToolKit. The open-source deep-learning library is used to implement distributed deep learning and machine learning tasks.

CNTK enables you to combine predictive models like convolutional neural networks (CNNs), feed-forward deep neural networks (DNNs), and recurrent neural networks (RNNs), with the CNTK framework. This enables the effective implementation of end-to-end deep learning tasks. Some of the main features of CNTK:

- Open-source
- Implement distributed deep learning tasks
- Combine predictive models with CNTK framework
- End-to-end deep learning tasks

8. Keras: Kears is yet another notable open-source Python library used for deep learning tasks, allowing for rapid deep neural network testing. Keras provides you with the tools needed to construct models, visualize graphs, and analyze datasets. On top of that, it also includes prelabeled datasets that can be directly imported and loaded.

The Keras library is often preferred due to it being modular, extensible, and flexible. This makes it a user-friendly option for beginners. It can also integrate with objectives, layers, optimizers, and activation functions. Keras operates in various environments and can run on CPUs and GPUs. It also offers one of the widest ranges for data types. Some of the main features of Keras:

- Developing neural layers
- Data pooling
- Builds deep learning and machine learning models
- Activation and cost functions

9. Theano: Theano, a numerical computation Python library specifically developed for machine learning and deep libraries. With this tool, you will achieve efficient definition, optimization, and evaluation of mathematical expressions and matrix calculations. All of this enables Theano to be used for the employment of dimensional arrays to construct deep learning models.

Theano is used by a lot of deep learning developers and programmers thanks to it being a highly specific library. It can be used with a graphics processing unit (GPU) instead of a central processing unit (CPU). Some of the main features of Theano:

- Built-in validation and unit testing tools
- High-performing mathematical computations
- Fast and stable evaluations
- Data-intensive calculations

10. MXNet: Closing out our list of the 10 best Python libraries for deep learning is MXNet, which is a highly scalable open-source deep learning framework. MXNet was designed to train and deploy deep neural networks, and it can train models extremely fast.

MXNet supports many programming languages, such as Python, Julia, C, C++, and more. One of the best aspects of MXNet is that it offers incredibly fast calculation speeds and resource utilization on GPU. Some of the main features of MXNet:

- Highly-scalable
- Open-source
- Train and deploy deep learning neural networks
- Trains models fast
- Fast calculation speeds

4) What is Tensorflow for deep learning?

TensorFlow is a powerful and versatile open-source software library for machine learning. It was developed by Google and released under the Apache 2.0 open-source license in 2015. TensorFlow provides an extensive set of APIs for building and training neural networks, as well as tools for deploying models into production systems.

TensorFlow deep learning is used by many large organizations. It's also popular among individual developers who are interested in creating their own applications or exploring the possibilities of artificial intelligence.

When it was first released, TensorFlow quickly became one of the most popular deep learning platforms, with the number of TensorFlow deep learning GitHub commits far surpassing other popular deep learning platforms.

To use TensorFlow, you need to first install it on your computer or server. Once installed, you can start coding with Python using the TensorFlow API to create your own neural networks. You can also access TensorFlow deep learning tutorials from the official website or GitHub repository to get started quickly.

The official TensorFlow GitHub repository contains many TensorFlow deep learning example projects, which can help users get up to speed quickly with how to use the library's API functions and classes correctly when developing their own projects.

There are also plenty of tutorials available online which explain how to build basic models step-by-step, so even those without much experience in AI development can get started quickly with deep learning projects using TensorFlow.

5) Explain about deep learning framework?

Deep learning frameworks enable machines to learn from data in order to make decisions and predictions. These frameworks use neural networks, which are systems of interconnected nodes that can process large amounts of data more efficiently than traditional algorithms.

These frameworks have become increasingly popular over the past few years, and deep learning frameworks in 2023 are expected to continue growing in popularity. Businesses can benefit from deep learning frameworks by using them to automate complex tasks such as customer service, fraud detection, document processing, and more.

For example, businesses can use Rossum's deep learning models to improve optical character recognition (OCR) document data capture processes. OCR is an AI-based technology that uses algorithms to recognize text from scanned documents or images.

OCR, however, is somewhat limited because it relies on manually-programmed parameters that must be re-adjusted for every single document format a business needs to process. Deep learning enables intelligent document processing, which learns how to navigate new formats on its own without human input.

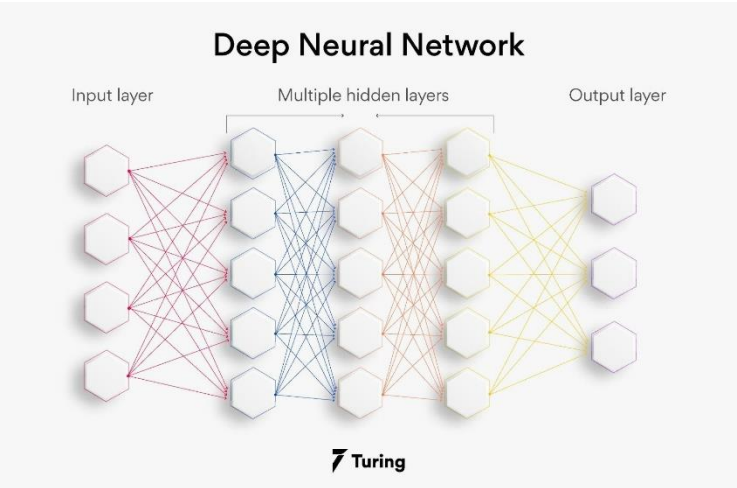
In addition to automating document data capture, businesses can also use deep learning frameworks for predictive analytics. By leveraging the power of AI-driven models, businesses can gain insights into their operations that would be difficult or time-consuming to obtain manually and use these insights to make better strategic decisions.

Ultimately, deep learning frameworks offer a wide range of benefits for businesses looking to leverage AI technologies in order to reduce costs and increase productivity. If you didn't yet take advantage of deep learning frameworks in 2022, they're well worth learning more about in 2023.

6) Explain about Deep feedforward network?

A) Deep feedforward networks, also often called **feedforward neural networks**, or **multilayer perceptrons** (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function $\mathbf{f}^*$. For example, for a classifier, $y = \mathbf{f}^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to a category y. A feedforward network defines a mapping $\boldsymbol{y} = \mathbf{f}(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation.

These models are called **feedforward** because information flows through the function being evaluated from $\boldsymbol{x}$, through the intermediate computations used to define $\mathbf{f}$, and finally to the output $\boldsymbol{y}$. There are no **feedback** connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called **recurrent neural networks.**



Feedforward networks are of extreme importance to machine learning practioners. They form the basis of many important commercial applications. For example, the convolutional networks used for object recognition from photos are a specialized kind of feedforward network. Feedforward networks are a conceptual stepping stone on the path to recurrent networks, which power many natural language applications.

Feedforward neural networks are called **networks** because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together.

For example, we might have three functions $\mathbf{f}^{(1)}$, $\mathbf{f}^{(2)}$, and $\mathbf{f}^{(3)}$ connected in a chain, to form $\mathbf{f}(\boldsymbol{x}) = \mathbf{f}^{(3)}(\mathbf{f}^{(2)}(\mathbf{f}^{(1)}(\boldsymbol{x})))$. These chain structures are the most commonly used structures of neural networks. In this case, $\mathbf{f}^{(1)}$ is called the **first layer** of the network, $\mathbf{f}^{(2)}$ is called the **second layer**, and so on. The overall length of the chain gives the **depth** of the model. It is from this terminology that the name "deep learning" arises. The final layer of a feedforward network is called the **output layer**. During neural network training, we drive $\mathbf{f}(\boldsymbol{x})$ to match $\mathbf{f}^{*}(\boldsymbol{x})$.

The training data provides us with noisy, approximate examples of $\mathbf{f}^{*}(\boldsymbol{x})$ evaluated at different training points. Each example $\boldsymbol{x}$ is accompanied by a label $y \approx \mathbf{f}^{*}(\boldsymbol{x})$. The training examples specify directly what the output layer must do at each point $\boldsymbol{x}$; it must produce a value that is close to y. The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data does not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of $\mathbf{f}^{*}$. Because the training data does not show the desired output for each of these layers, these layers are called **hidden layers**.

Finally, these networks are called *neural* because they are loosely inspired by neuroscience. Each hidden layer of the network is typically vector-valued. The dimensionality of these hidden layers determines the **width** of the model. Each element of the vector may be interpreted as playing a role analogous to a neuron. Rather than thinking of the layer as representing a single vector-to-vector function, we can also think of the layer as consisting of many **units** that act in parallel, each representing a vector-to-scalar function. Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value. The idea of using many layers of vector-valued representation is drawn from neuroscience.

The choice of the functions $\mathbf{f}^{(i)}(\boldsymbol{x})$ used to compute these representations is also loosely guided by neuroscientific observations about the functions that biological neurons compute. However, modern neural network research is guided by many mathematical and engineering disciplines, and the goal of neural networks is not to perfectly model the brain. It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.

One way to understand feedforward networks is to begin with linear models and consider how to overcome their limitations. Linear models, such as logistic regression and linear regression, are appealing because they may be fit efficiently and reliably, either in closed form or with convex optimization. Linear models also have the obvious defect that the model capacity is limited to linear functions, so the model cannot understand the interaction between any two input variables.

To extend linear models to represent nonlinear functions of $\boldsymbol{x}$, we can apply the linear model not to $\boldsymbol{x}$ itself but to a transformed input $\varphi(\boldsymbol{x})$, where φ is a nonlinear transformation.

7) What are advantages and applications of Deep feed forward networks?

A) **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.

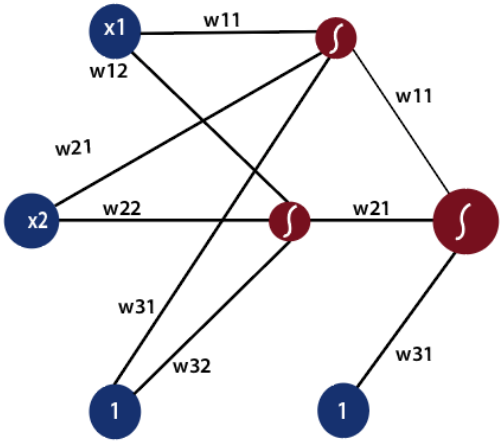
Applications of feed forward neural networks: There are many applications for these neural networks. The following are a few of them.

- **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.

8) Explain about the math behind the Deep Feed Forward networks?

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

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



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

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

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

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

2) Each input is multiplied by weight with respect to the first and second model to obtain their probability of being in the positive region in each model.
So, we will multiply our inputs by a matrix of weight using matrix multiplication.

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

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

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

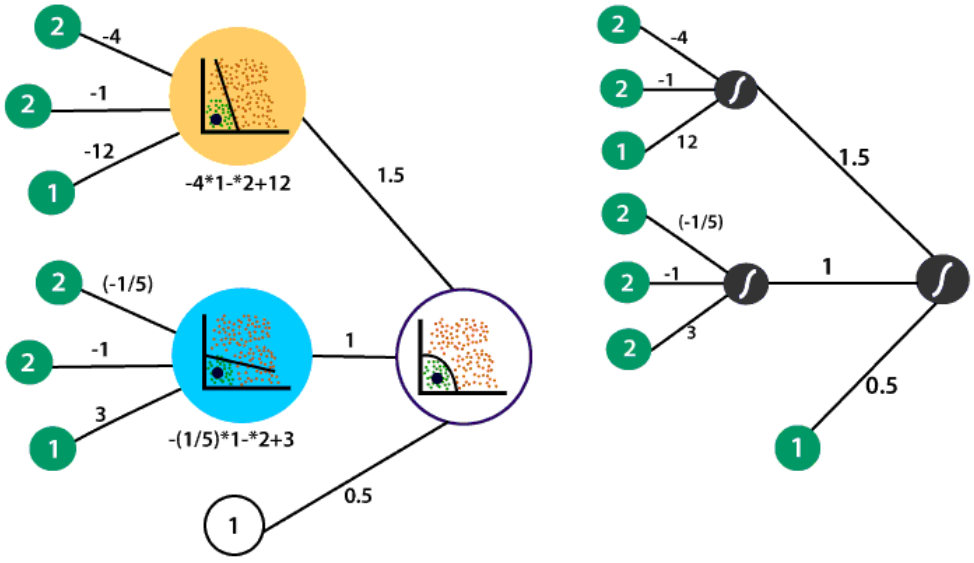
4) We multiply the probability which we have obtained from the previous step with the second set of weights. We always include a bias of one whenever taking a combination of inputs.

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

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

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

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



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

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

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

$$-4x_1 - x_2 + 12$$

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

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

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

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

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

$$[2 \quad 0.6]$$

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

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

The second layer contains the weights which dictated the combination of the linear models in the first layer to obtain the non-linear model in the second layer. The weights are 1.5, 1, and a bias value of 0.5.
Now, we have to multiply our probabilities from the first layer with the second set of weights as

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

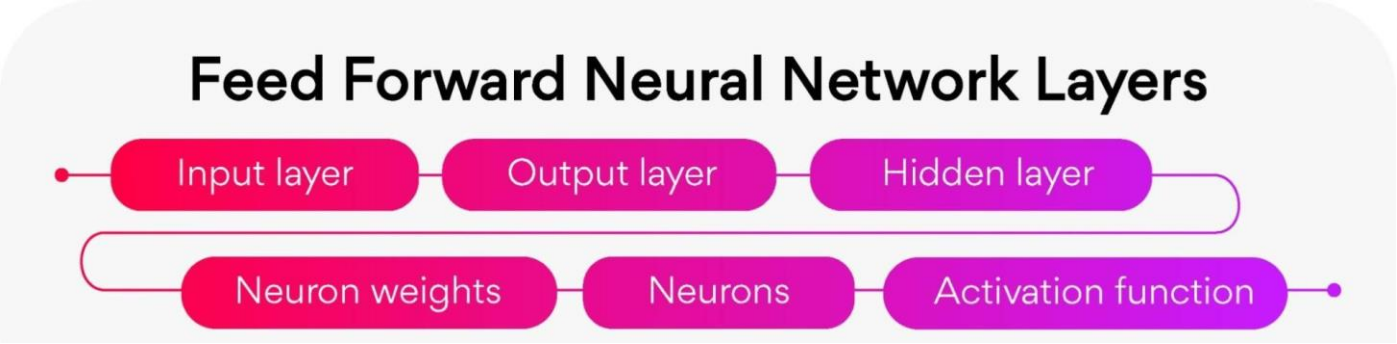
Now, we will take the sigmoid of our final score

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

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

9) Explain about various Layers and functions in Deep Feed forward networks?

A) **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).

Function in feed forward neural network: The various functions used in deep feed forward networks are

1. Cost function
2. Loss function
3. Gradient Learning
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.

10) Give an example of Learning XOR?

A) To make the idea of a feedforward network more concrete, we begin with an example of a fully functioning feedforward network on a very simple task: learning the XOR function.

The XOR function ("exclusive or") is an operation on two binary values, x_1 and x_2. When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0. The XOR function provides the target function $y = f^*(x)$ that we want to learn. Our model provides a function $y = f(x;\theta)$ and our learning algorithm will adapt the parameters θ to make f as similar as possible to f^*.

In this simple example, We want our network to perform correctly on the four points $X = \{[0,0]^T, [0,1]^T, [1,0]^T$, and $[1,1]^T\}$. We will train the network on all four of these points. The only challenge is to fit the training set.

We can treat this problem as a regression problem and use a mean squared error loss function. We choose this loss function to simplify the math for this example as much as possible. In practical applications, MSE is usually not an appropriate cost function for modeling binary data. Evaluated on our whole training set, the MSE loss function is

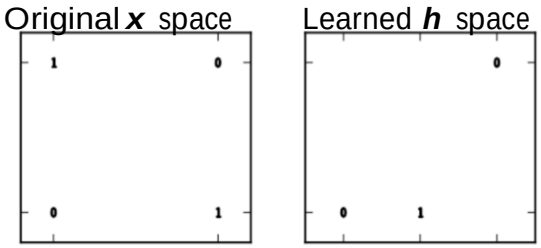
$$J(\theta) = \frac{1}{4}\sum_{x\in X}(f*(x) - f(x;\theta))2$$

Now we must choose the form of our model, $f(x;\theta)$. Suppose that we choose a linear model, with θ consisting of w and b. Our model is defined to be

$$f(x;w,b) = x^T w + b.$$

We can minimize $J(\theta)$ in closed form with respect to w and b using the normal equations.

After solving the normal equations, we obtain $w = 0$ and $b = 1/2$. The linear model simply outputs 0.5 everywhere. The following Figure shows how a linear model is not able to represent the XOR function. One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.



Specifically, we can introduce a very simple feedforward network with one hidden layer containing two hidden units.

This feedforward network has a vector of hidden units h that are computed by a function $f^{(1)}(x; W, c)$. The values of these hidden units are then used as the input for a second layer. The second layer is the output layer of the network. The output layer is still just a linear regression model, but now it is applied to h rather than to x. The network now contains two functions chained together: $h = f^{(1)}(x; W,$

$\boldsymbol{c}$) and $y = \mathbf{f}^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$, with the complete model being $\mathbf{f}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \mathbf{f}^{(2)}(\mathbf{f}^{(1)}(\boldsymbol{x}))$. The solution we described to the XOR problem is at a global minimum of the loss function.

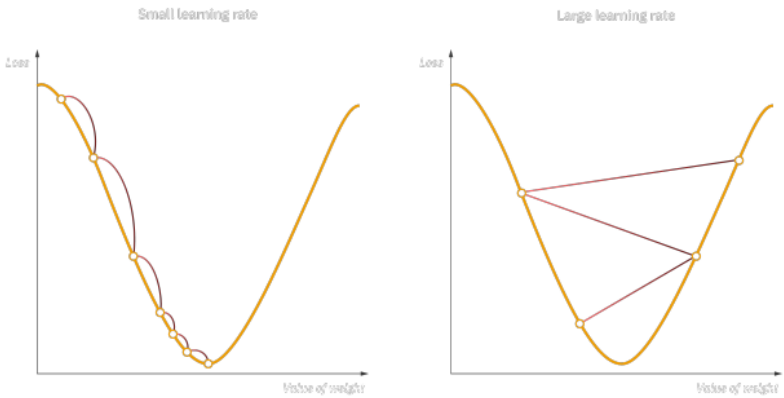
11) Explain about gradient based Learning?

A) **Gradient-based learning** is a type of machine learning in which the optimization algorithm uses gradients to update the model parameters during training. This approach is commonly used in deep learning and neural networks because it allows the model to learn complex representations of the input data.

Gradient Descent: Gradient descent is an optimization algorithm which is commonly-used to train machine learning models and neural networks. Training data helps these models learn over time, and the cost function within gradient descent specifically acts as a barometer, gauging its accuracy with each iteration of parameter updates. Until the function is close to or equal to zero, the model will continue to adjust its parameters to yield the smallest possible error. Once machine learning models are optimized for accuracy, they can be powerful tools for artificial intelligence (AI) and computer science applications.

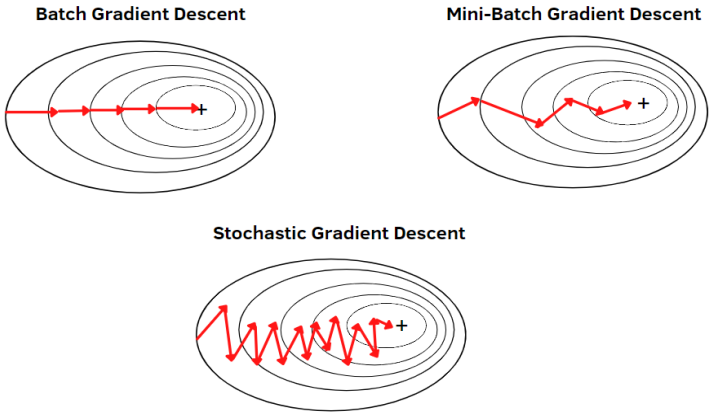
Working of gradient descent: The goal of gradient descent is to minimize the cost function, or the error between predicted and actual y. In order to do this, it requires two data points—a direction and a learning rate. These factors determine the partial derivative calculations of future iterations, allowing it to gradually arrive at the local or global minimum (i.e. point of convergence).

- **Learning rate** (also referred to as step size or the alpha) is the size of the steps that are taken to reach the minimum. This is typically a small value, and it is evaluated and updated based on the behavior of the cost function. High learning rates result in larger steps but risks overshooting the minimum. Conversely, a low learning rate has small step sizes. While it has the advantage of more precision, the number of iterations compromises overall efficiency as this takes more time and computations to reach the minimum.
- **The cost (or loss) function** measures the difference, or error, between actual y and predicted y at its current position. This improves the machine learning model's efficacy by providing feedback to the model so that it can adjust the parameters to minimize the error and find the local or global minimum. It continuously iterates, moving along the direction of steepest descent (or the negative gradient) until the cost function is close to or at zero. At this point, the model will stop learning. Additionally, while the terms, cost function and loss function, are considered synonymous, there is a slight difference between them. It's worth noting that a loss function refers to the error of one training example, while a cost function calculates the average error across an entire training set.



Types of gradient descent: There are three types of gradient descent learning algorithms:

1. Batch gradient descent,
2. Stochastic gradient descent
3. Mini-batch gradient descent.



BATCH GRADIENT DESCENT: Batch gradient descent, also known as vanilla gradient descent, calculates the error for each example within the training dataset. Still, the model is not changed until every training sample has been assessed. The entire procedure is referred to as a cycle and a training epoch.

Some benefits of batch are its computational efficiency, which produces a stable error gradient and a stable convergence. Some drawbacks are that the stable error gradient can sometimes result in a state of convergence that isn't the best the model can achieve. It also requires the entire training dataset to be in memory and available to the algorithm.

Advantages

1. Fewer model updates mean that this variant of the steepest descent method is more computationally efficient than the stochastic gradient descent method.
2. Reducing the update frequency provides a more stable error gradient and a more stable convergence for some problems.
3. Separating forecast error calculations and model updates provides a parallel processing-based algorithm implementation.

Disadvantages

1. A more stable error gradient can cause the model to prematurely converge to a suboptimal set of parameters.
2. End-of-training epoch updates require the additional complexity of accumulating prediction errors across all training examples.
3. The batch gradient descent method typically requires the entire training dataset in memory and is implemented for use in the algorithm.
4. Large datasets can result in very slow model updates or training speeds.
5. Slow and require more computational power.

STOCHASTIC GRADIENT DESCENT: By contrast, stochastic gradient descent (SGD) changes the parameters for each training sample one at a time for each training example in the dataset. Depending on the issue, this can make SGD faster than batch gradient descent. One benefit is that the regular updates give us a fairly accurate idea of the rate of improvement.

However, the batch approach is less computationally expensive than the frequent updates. The frequency of such updates can also produce noisy gradients, which could cause the error rate to fluctuate rather than gradually go down.

Advantages

1. You can instantly see your model's performance and improvement rates with frequent updates.
2. This variant of the steepest descent method is probably the easiest to understand and implement, especially for beginners.
3. Increasing the frequency of model updates will allow you to learn more about some issues faster.
4. The noisy update process allows the model to avoid local minima (e.g., premature convergence).
5. Faster and require less computational power.
6. Suitable for the larger dataset.

Disadvantages

1. Frequent model updates are more computationally intensive than other steepest descent configurations, and it takes considerable time to train the model with large datasets.
2. Frequent updates can result in noisy gradient signals. This can result in model parameters and cause errors to fly around (more variance across the training epoch).
3. A noisy learning process along the error gradient can also make it difficult for the algorithm to commit to the model's minimum error.

MINI-BATCH GRADIENT DESCENT: Since mini-batch gradient descent combines the ideas of batch gradient descent with SGD, it is the preferred technique. It divides the training dataset into manageable groups and updates each separately. This strikes a balance between batch gradient descent's effectiveness and stochastic gradient descent's durability.

Mini-batch sizes typically range from 50 to 256, although, like with other machine learning techniques, there is no set standard because it depends on the application. The most popular kind in deep learning, this method is used when training a neural network.

Advantages

1. The model is updated more frequently than the stack gradient descent method, allowing for more robust convergence and avoiding local minima.
2. Batch updates provide a more computationally efficient process than stochastic gradient descent.
3. Batch processing allows for both the efficiency of not having all the training data in memory and implementing the algorithm.

Disadvantages

1. Mini-batch requires additional hyper parameters "mini-batch size" to be set for the learning algorithm.
2. Error information should be accumulated over a mini-batch of training samples, such as batch gradient descent.
3. it will generate complex functions.

Challenges with gradient descent

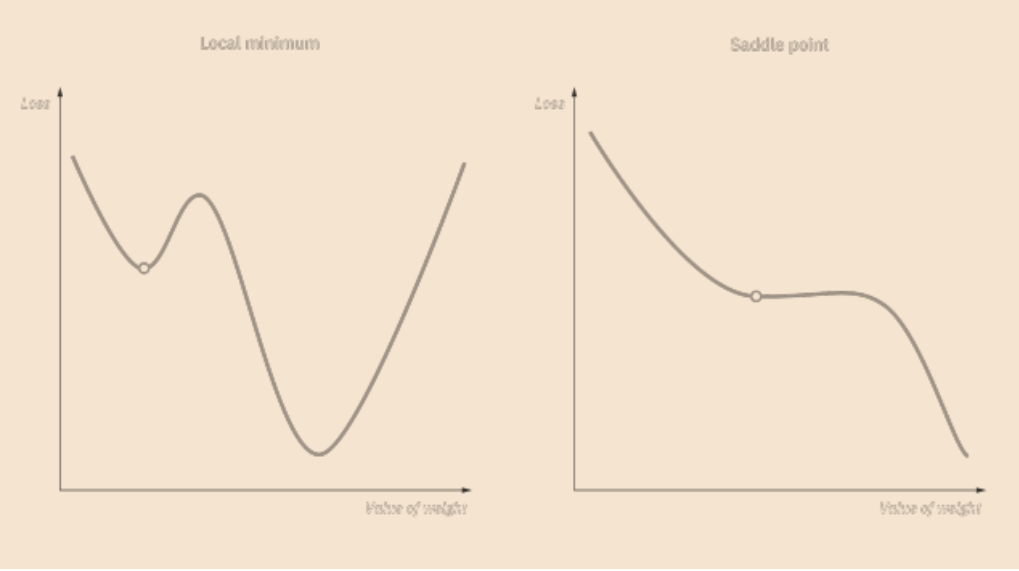
While gradient descent is the most common approach for optimization problems, it does come with its own set of challenges. Some of them include:

Local minima and saddle points: For convex problems, gradient descent can find the global minimum with ease, but as non convex problems emerge, gradient descent can struggle to find the global minimum, where the model achieves the best results.

When the slope of the cost function is at or close to zero, the model stops learning. A few scenarios beyond the global minimum can also yield this slope, which are local minima and saddle points. Local minima mimic the shape of a global minimum, where the slope of the cost function increases on either side of the current point. However, with saddle points, the negative gradient only exists on one side of the point, reaching a local maximum on one side and a local minimum on the other. Its name inspired by that of a horse's saddle. Noisy gradients can help the gradient escape local minimums and saddle points.

Vanishing and Exploding Gradients: In deeper neural networks, particular recurrent neural networks, we can also encounter two other problems when the model is trained with gradient descent and backpropagation.

- **Vanishing gradients:** This occurs when the gradient is too small. As we move backwards during backpropagation, the gradient continues to become smaller, causing the earlier layers in the network to learn more slowly than later layers. When this happens, the weight parameters update until they become insignificant—i.e. 0—resulting in an algorithm that is no longer learning.
- **Exploding gradients:** This happens when the gradient is too large, creating an unstable model. In this case, the model weights will grow too large, and they will eventually be represented as NaN. One solution to this issue is to leverage a dimensionality reduction technique, which can help to minimize complexity within the model.



12) Explain about Architecture Design?

A) Architecture Design: The key design consideration for neural networks is determining the architecture. The word **architecture** refers to the overall structure of the network: how many units it should have and how these units should be connected to each other.

Most neural networks are organized into groups of units called layers. Most neural network architectures arrange these layers in a chain structure, with each layer being a function of the layer that preceded it. In this structure, the first layer is given by

$$\boldsymbol{h}^{(1)} = g^{(1)\,\mathrm{T}}\boldsymbol{W}^{(1)T}\boldsymbol{x} + \boldsymbol{b}^{(1)\mathrm{T}},$$

the second layer is given by

$$\boldsymbol{h}^{(2)} = g^{(2)\,T}\boldsymbol{W}^{(2)T}\boldsymbol{h}^{1)} + \boldsymbol{b}^{(2)T},$$

and so on.

In these chain-based architectures, the main architectural considerations are to choose the depth of the network and the width of each layer. A network with even one hidden layer is sufficient to fit the training set.

Deeper networks often are able to use far fewer units per layer and far fewer parameters and often generalize to the test set, but are also often harder to optimize. The ideal network architecture for a task must be found via experimentation guided by monitoring the validation set error.

Universal Approximation Properties and Depth

Feedforward networks with hidden layers provide a universal approximation framework. The **universal approximation theorem** states that a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units.

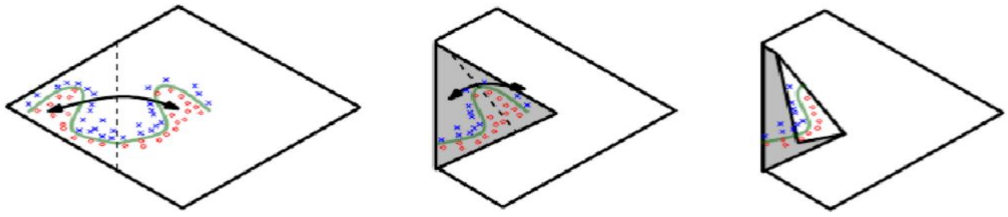
The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well

The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to *represent* this function.

The universal approximation theorem says that there exists a network large enough to achieve any degree of accuracy we desire, but the theorem does not say how large this network will be

The universal approximation theorem says that there exists a network large enough to achieve any degree of accuracy we desire, but the theorem does not say how large this network will be. There exist families of functions which can be approximated efficiently by an architecture with depth greater than some value d, but which require a much larger model if depth is restricted to be less than or equal to d.

The below picture illustrates how a network with absolute value rectification creates mirror images of the function computed on top of some hidden unit, with respect to the input of that hidden unit. Each hidden unit specifies where to fold the input space in order to create mirror responses (on both sides of the absolute value nonlinearity). By composing these folding operations, we obtain an exponentially large number of piecewise linear regions which can capture all kinds of regular (e.g., repeating) patterns.



Another key consideration of architecture design is exactly how to connect a pair of layers to each other. In the default neural network layer described by a linear transformation via a matrix $\boldsymbol{W}$, every input unit is connected to every output unit. Many specialized networks in the chapters ahead have fewer connections, so that each unit in the input layer is connected to only a small subset of units in the output layer. These strategies for reducing the number of connections reduce the number of parameters and the amount of computation required to evaluate the network, but are often highly problem-dependent.

13) Explain about Regularization in DNN?

Regularization: we can define regularization as "any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error." There are many regularization strategies. In the context of deep learning, most regularization strategies are based on regularizing estimators. Regularization of an estimator works by trading increased bias for reduced variance. An effective regularizer is one that makes a profitable trade, reducing variance significantly while not overly increasing the bias.

Types of regularization techniques: Regularization is a technique used to address overfitting by directly changing the architecture of the model by modifying the model's training process. The following are the commonly used regularization techniques:

1. L2 regularization
2. L1 regularization
3. Dropout regularization

L2 regularization: According to regression analysis, L2 regularization is also called ridge regression. In this type of regularization, the squared magnitude of the coefficients or weights multiplied with a regularizer term is added to the loss or cost function. L2 regression can be represented with the following mathematical equation.

Loss:

$$\underbrace{\sum_{i=1}^{n}\left(y_i - \sum_{j=1}^{p} x_{ij} b_j\right)^2}_{Loss\ term} + \underbrace{\lambda \cdot \left(\sum_{j=1}^{p} b_j\right)^2}_{Regularizer\ term}$$

In the above equation,

$y_i \ - \ labels$
$x_{ij} \ - \ features$
$b_j \ - \ weights$
$\lambda \ - \ regularizer\, term\ (lambda)$
$i \ - \ no\, of\, training\, samples$

We can see that a fraction of the sum of squared values of weights is added to the loss function. Thus, when gradient descent is applied on loss, the weight update seems to be consistent by giving almost equal emphasis on all features.

- Lambda is the hyperparameter that is tuned to prevent overfitting i.e. penalize the insignificant weights by forcing them to be small but not zero.
- L2 regularization works best when all the weights are roughly of the same size, i.e., input features are of the same range.
- This technique also helps the model to learn more complex patterns from data without overfitting easily.

L1 regularization: L1 regularization is also referred to as lasso regression. In this type of regularization, the absolute value of the magnitude of coefficients or weights multiplied with a regularizer term is added to the loss or cost function. It can be represented with the following equation.

Loss:

$$\underbrace{\sum_{i=1}^{n}\left(y_i - \sum_{j=1}^{p} x_{ij}b_j\right)^2}_{Loss\ Term} + \underbrace{\lambda \cdot \sum_{j=1}^{p}|b_j|}_{Regularizer\ term}$$

In the above equation,

$$y_i \ - \ labels$$
$$x_{ij} \ - \ features$$
$$b_j \ - \ weights$$
$$\lambda \ - \ regularizer\ term\ (lambda)$$
$$i \ - \ no\ of\ training\ samples$$

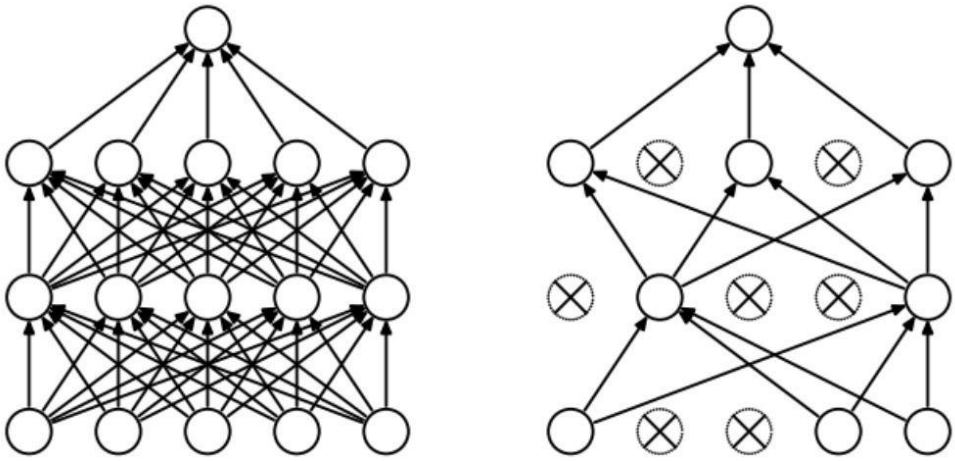
A fraction of the sum of absolute values of weights to the loss function is added in the L1 regularization. In this way, you will be able to eliminate some coefficients with lesser values by pushing those values towards 0. You can observe the following by using L1 regularization:

- Since the L1 regularization adds an absolute value as a penalty to the cost function, the feature selection will be done by retaining only some important features and eliminating the lower or unimportant features.
- This technique is also robust to outliers, i.e., the model will be able to easily learn about outliers in the dataset.
- This technique will not be able to learn complex patterns from the input data.

Dropout regularization: Dropout regularization is the technique in which some of the neurons are randomly disabled during the training such that the model can extract more useful robust features from the model. This prevents overfitting. You can see the dropout regularization in the following diagram:

- In figure (a), the neural network is fully connected. If all the neurons are trained with the entire training dataset, some neurons might memorize the patterns occurring in training data. This leads to overfitting since the model is not generalizing well.

- In figure (b), the neural network is sparsely connected, i.e., only some neurons are active during the model training. This forces the neurons to extract robust features/patterns from training data to prevent overfitting.



(a) Standard Neural Net (b) After applying dropout.

The following are the characteristics of dropout regularization:

- Dropout randomly disables some percent of neurons in each layer. So for every epoch, different neurons will be dropped leading to effective learning.
- Dropout is applied by specifying the 'p' values, which is the fraction of neurons to be dropped.
- Dropout reduces the dependencies of neurons on other neurons, resulting in more robust model behavior.
- Dropout is applied only during the model training phase and is not applied during the inference phase.
- When the model receives complete data during the inference time, you need to scale the layer outputs 'x' by 'p' such that only some parts of data will be sent to the next layer. This is because the layers have seen less amount of data as specified by dropout.

These are some of the most popular regularization techniques that are used to reduce overfitting during model training. They can be applied according to the use case or dataset being considered for more accurate model performance on the testing data.

Need of Regularisation: In a general learning algorithm, the dataset is divided into a **training set** and a **test set.** After each epoch of the algorithm, the parameters are updated accordingly after understanding the dataset. Finally, this trained model is applied to the test set.

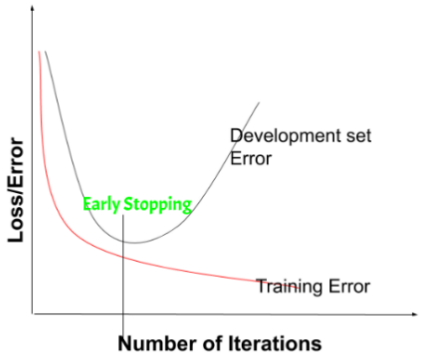
Generally, the training set error will be less compared to the test set error. This is because of overfitting whereby the algorithm memorizes the training data and produces the right results on the training set. So, the model becomes highly exclusive to the training set and fails to produce accurate results for other datasets including the test set. Regularization techniques are used in such situations to **reduce overfitting** and **increase the model's performance** on any general dataset.

14) What is Early Stopping?

In Regularization by Early Stopping, we stop training the model when the performance on the validation set is getting worse- increasing loss decreasing accuracy, or poorer scores of the scoring metric. By

plotting the error on the training dataset and the validation dataset together, both the errors decrease with a number of iterations until the point where the model starts to overfit. After this point, the training error still decreases but the validation error increases.

So, even if training is continued after this point, early stopping essentially returns the set of parameters that were used at this point and so is equivalent to stopping training at that point. So, the final parameters returned will enable the model to have low variance and better generalization. The model at the time the training is stopped will have a better generalization performance than the model with the least training error.



Early stopping can be thought of as **implicit regularization**, contrary to regularization via weight decay. This method is also efficient since it requires less amount of training data, which is not always available. Due to this fact, early stopping requires lesser time for training compared to other regularization methods. Repeating the early stopping process many times may result in the model overfitting the validation dataset, just as similar as overfitting occurs in the case of training data.

The number of iterations(i.e. epoch) taken to train the model can be considered a **hyperparameter**. Then the model has to find an optimum value for this hyperparameter (by hyperparameter tuning) for the best performance of the learning model.

Benefits of Early Stopping:

- Helps in reducing overfitting
- It improves generalisation
- It requires less amount of training data
- Takes less time compared to other regularisation models
- It is simple to implement

Limitations of Early Stopping:

- If the model stops too early, there might be risk of underfitting
- It may not be beneficial for all types of models
- If validation set is not chosen properly, it may not lead to the most optimal stopping

To summarize, early stopping can be best used to prevent overfitting of the model, and saving resources. It would give best results if taken care of few things like – parameter tuning, preventing the model from overfitting, and ensuring that the model learns enough from the data.

Algorithm : The early stopping meta-algorithm for determining the best amount of

time to train. This meta-algorithm is a general strategy that works well with a variety of training algorithms and ways of quantifying error on the validation set.

Let n be the number of steps between evaluations.
Let p be the "patience," the number of times to observe worsening validation set error before giving up.
Let $\boldsymbol{\theta}_o$ be the initial parameters.
$\boldsymbol{\theta} \rightarrow \boldsymbol{\theta}_o$

$i \rightarrow 0$

$j \rightarrow 0$ $v \rightarrow$
∞ $\boldsymbol{\theta}^* \rightarrow \boldsymbol{\theta}$
$i^* \rightarrow i$

while $j < p$ **do**

Update $\boldsymbol{\theta}$ by running the training algorithm for n steps.
$i \rightarrow i + n$

$v^T \rightarrow$ ValidationSetError($\boldsymbol{\theta}$)
if $v^T < v$ **then**

$j \rightarrow 0$ $\boldsymbol{\theta}^* \rightarrow$
$\boldsymbol{\theta}$ $i^* \rightarrow i$ v
$\rightarrow v^T$

else

$j \rightarrow j + 1$

end if end while

Best parameters are $\boldsymbol{\theta}^*$, best number of training steps is i^*

15) What Are Optimizers in Deep Learning? Explain about Adagard and Adam Optimizers?

In deep learning, optimizers are algorithms that adjust the model's parameters during training to minimize a loss function. They enable neural networks to learn from data by iteratively updating weights and biases Each optimizer has specific update rules, learning rates, and momentum to find optimal model parameters for improved performance.

An optimizer is a function or an algorithm that adjusts the attributes of the neural network, such as weights and learning rates. Thus, it helps in reducing the overall loss and improving accuracy. The problem of choosing the right weights for the model is a daunting task, as a deep learning model generally consists of millions of parameters. This various deep-learning optimizers are Gradient Descent, Stochastic Gradient Descent, Stochastic Gradient descent with momentum, Mini-Batch Gradient Descent, Adagrad, RMSProp, AdaDelta, and Adam.

Adagrad (Adaptive Gradient Descent) Deep Learning Optimizer: The adaptive gradient descent algorithm is slightly different from other gradient descent algorithms. This is because it uses different

learning rates for each iteration. The change in learning rate depends upon the difference in the parameters during training. The more the parameters get changed, the more minor the learning rate changes. This modification is highly beneficial because real-world datasets contain sparse as well as dense features. So it is unfair to have the same value of learning rate for all the features. The Adagrad algorithm uses the below formula to update the weights. Here the alpha(t) denotes the different learning rates at each iteration, n is a constant, and E is a small positive to avoid division by 0.

$$\mathrm{w_t} = \mathrm{w_{t-1}} - \eta'_t \frac{\partial \mathrm{L}}{\partial \mathrm{w(t-1)}} \qquad \eta'_t = \frac{\eta}{\mathrm{sqrt}(\alpha_t + \epsilon)}$$

The benefit of using Adagrad is that it abolishes the need to modify the learning rate manually. It is more reliable than gradient descent algorithms and their variants, and it reaches convergence at a higher speed.

One downside of the AdaGrad optimizer is that it decreases the learning rate aggressively and monotonically. There might be a point when the learning rate becomes extremely small. This is because the squared gradients in the denominator keep accumulating, and thus the denominator part keeps on increasing. Due to small learning rates, the model eventually becomes unable to acquire more knowledge, and hence the accuracy of the model is compromised.

Benefits of using AdaGrad

- **Easy to use**– It's a reasonably straightforward optimization technique and may be applied to various models.
- **No need for manual**– There is no need to manually tune hyperparameters since this optimization method automatically adjusts the learning rate for each parameter.
- **Adaptive learning rate**– Modifies the learning rate for each parameter depending on the parameter's past gradients. This implies that for parameters with big gradients, the learning rate is lowered, while for parameters with small gradients, the learning rate is raised, allowing the algorithm to converge quicker and prevent overshooting the ideal solution.
- **Adaptability to noisy data**– This method provides the ability to smooth out the impacts of noisy data by assigning lesser learning rates to parameters with strong gradients owing to noisy input.
- **Handling sparse data efficiently**– It is particularly good at dealing with sparse data, which is prevalent in NLP and recommendation systems. This is performed by giving sparse parameters faster learning rates, which may speed convergence.

Adam Optimizer in Deep Learning: Adam optimizer, short for Adaptive Moment Estimation optimizer, is an optimization algorithm commonly used in deep learning. It is an extension of the stochastic gradient descent (SGD) algorithm and is designed to update the weights of a neural network during training.

The name "Adam" is derived from "adaptive moment estimation," highlighting its ability to adaptively adjust the learning rate for each network weight individually. Unlike SGD, which maintains a single learning rate throughout training, Adam optimizer dynamically computes individual learning rates based on the past gradients and their second moments.

Adam optimizer is an optimization algorithm that extends SGD by dynamically adjusting learning rates based on individual weights. It combines the features of AdaGrad and RMSProp to provide efficient and adaptive updates to the network weights during deep learning training.

Adam Optimizer Formula: The adam optimizer has several benefits, due to which it is used widely. It is adapted as a benchmark for deep learning papers and recommended as a default optimization algorithm. Moreover, the algorithm is straightforward to implement, has a faster running time, low memory requirements, and requires less tuning than any other optimization algorithm.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \left[\frac{\delta L}{\delta w_t} \right] v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left[\frac{\delta L}{\delta w_t} \right]^2$$

The above formula represents the working of adam optimizer. Here B1 and B2 represent the decay rate of the average of the gradients.

Advantages:

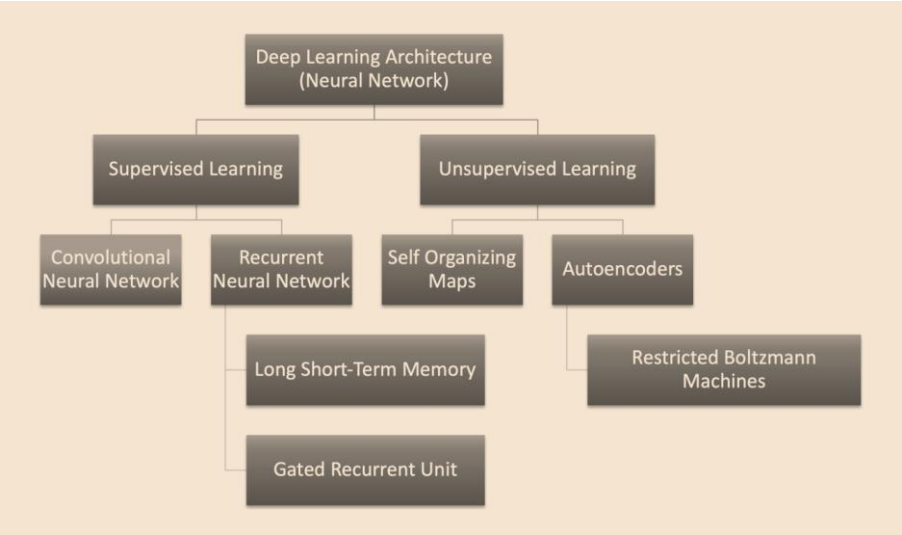
1. The method is too fast and converges rapidly.
2. Rectifies vanishing learning rate, high variance.

Disadvantages:

Computationally costly.

1) What are Deep learning architectures? And give brief note on CNN and its operation?

A) The number of architectures and algorithms that are used in deep learning is wide and varied. This section explores six of the deep learning architectures spanning the past 20 years. Notably, long short-term memory (LSTM) and convolutional neural networks (CNNs) are two of the oldest approaches in this list but also two of the most used in various applications.



Supervised deep learning: Supervised learning refers to the problem space wherein the target to be predicted is clearly labelled within the data that is used for training. The most popular supervised deep learning architectures are

- Convolutional neural networks
- Recurrent neural networks.

Convolutional neural networks: Convolutional networks (LeCun, 1989), also known as convolutional neural networks or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. The first CNN was created by Yann LeCun; The architecture is particularly useful in image-processing applications. at the time, the architecture focused on handwritten character recognition, such as postal code interpretation.

A Convolutional Neural Network, also known as CNN or ConvNet, is a class of neural networks that specializes in processing data that has a grid-like topology, such as an image. A digital image is a binary representation of visual data. It contains a series of pixels arranged in a grid-like fashion that contains pixel values to denote how bright and what color each pixel should be.

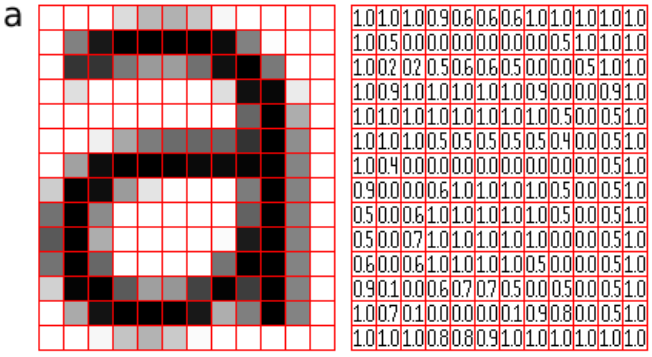
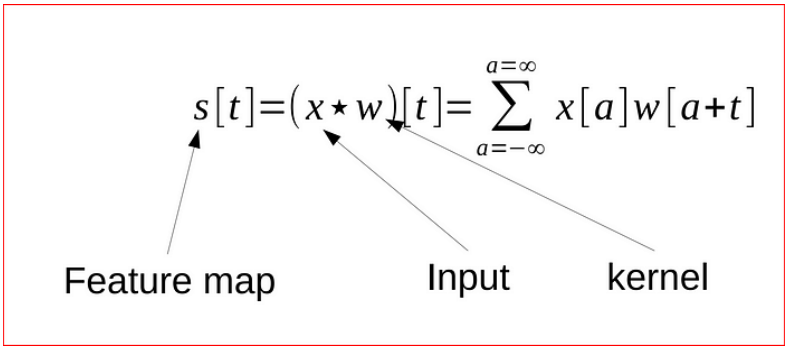


Figure 1: Representation of image as a grid of pixels

The human brain processes a huge amount of information the second we see an image. Each neuron works in its own receptive field and is connected to other neurons in a way that they cover the entire visual field. Just as each neuron responds to stimuli only in the restricted region of the visual field called the receptive field in the biological vision system, each neuron in a CNN processes data only in its receptive field as well. The layers are arranged in such a way so that they detect simpler patterns first (lines, curves, etc.) and more complex patterns (faces, objects, etc.) further along. By using a CNN, one can enable sight to computers.

Convolutional operation: Convolution is a mathematical operation where we have an input I, and an argument, kernel K to produce an output that expresses how the shape of one is modified by another. For example, We have an image "x", which is a 2D array of pixels with different color channels (Red,Green and Blue-RGB) and we have a **feature detector or kernel "w"** then the output we get after applying a mathematical operation is called a **feature map**



Convolution function

The mathematical operation helps compute similarity of two signals. we may have a feature detector or filter for identifying edges in the image, so convolution operation will help us identify the edges in the image when we use such a filter on the image. we usually assume that convolution functions are zero everywhere but the finite set of points for which we store the values. This means that in practice we can implement the infinite summation as a summation over a finite number of array elements.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

I is 2D array and K is kernel-Convolution function

Since convolution is commutative, we can rewrite the equation pictured above as shown below. We do this for ease of implementation in Machine Learning, as there is less variation in range of valid values for m and n. This is **cross correlation** function which most neural networks use.

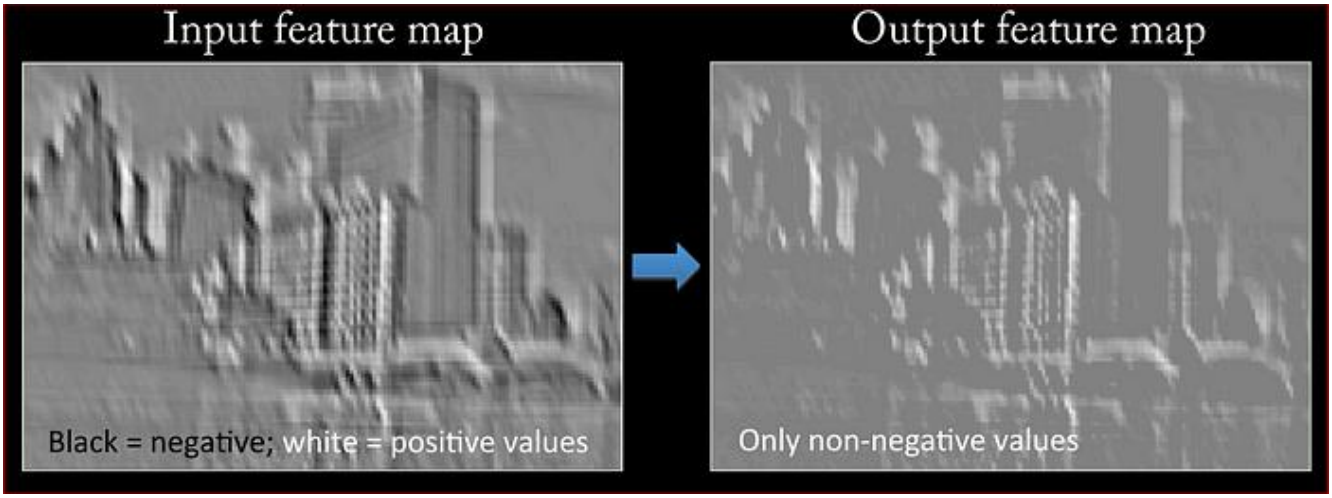
$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

Cross Correlation function

Implementing in CNN: The way we implement this is through **Convolutional Layer.** Convolutional layer is core building block of CNN, it helps with **feature detection.** Kernel K is a set of learnable filters and is small spatially compared to the image but extends through the full depth of the input image. An easy way to understand this is if you were a detective and you are came across a large image or a picture in dark, you will use you flashlight and scan across the entire image. This is exactly what we do in convolutional layer.

Kernel K, which is a feature detector is equivalent of the flashlight on image I, and we are trying to detect feature and create multiple feature maps to help us identify or classify the image. we have multiple feature detector to help with things like edge detection, identifying different shapes, bends or different colors etc.

After every convolution operation which is a linear function, we apply ReLU activation function. ReLU activation function introduces non linearity in convolutional layer. It replaces all negative pixel values with zero values in the feature map. Below figure shows the feature map transformation after applying the ReLU activation function.



2) Explain about Convolutional architecture?

A) **Convolutional Neural Network Architecture:** A CNN typically has three layers: a convolutional layer, a pooling layer, and a fully connected layer.

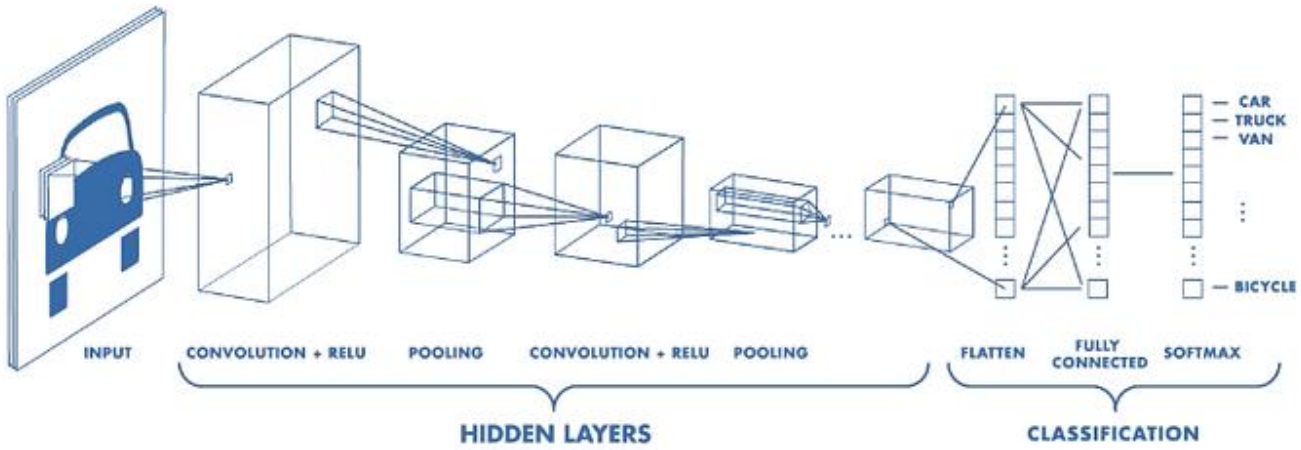


Figure : Architecture of a CNN

Convolution Layer: The convolution layer is the core building block of the CNN. It carries the main portion of the network's computational load. This layer performs a dot product between two matrices, where one matrix is the set of learnable parameters otherwise known as a kernel, and the other matrix is the restricted portion of the receptive field. The kernel is spatially smaller than an image but is more in-depth. This means that, if the image is composed of three (RGB) channels, the kernel height and width will be spatially small, but the depth extends up to all three channels.

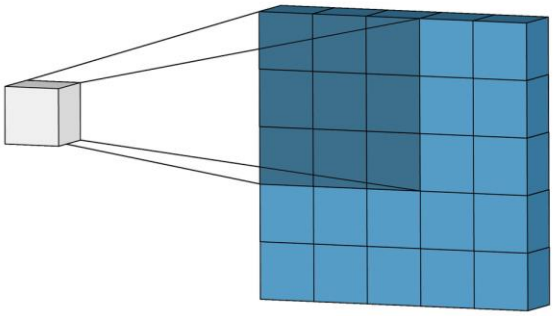


Fig: Illustration of Convolution Operation

During the forward pass, the kernel slides across the height and width of the image-producing the image representation of that receptive region. This produces a two-dimensional representation of the image known as an activation map that gives the response of the kernel at each spatial position of the image. The sliding size of the kernel is called a stride.

If we have an input of size W x W x D and Dout number of kernels with a spatial size of F with stride S and amount of padding P, then the size of output volume can be determined by the following formula:

$$W_{out} = \frac{W - F + 2P}{S} + 1$$

Fig: Formula for Convolution Layer

This will yield an output volume of size ***Wout* x *Wout* x *Dout.***

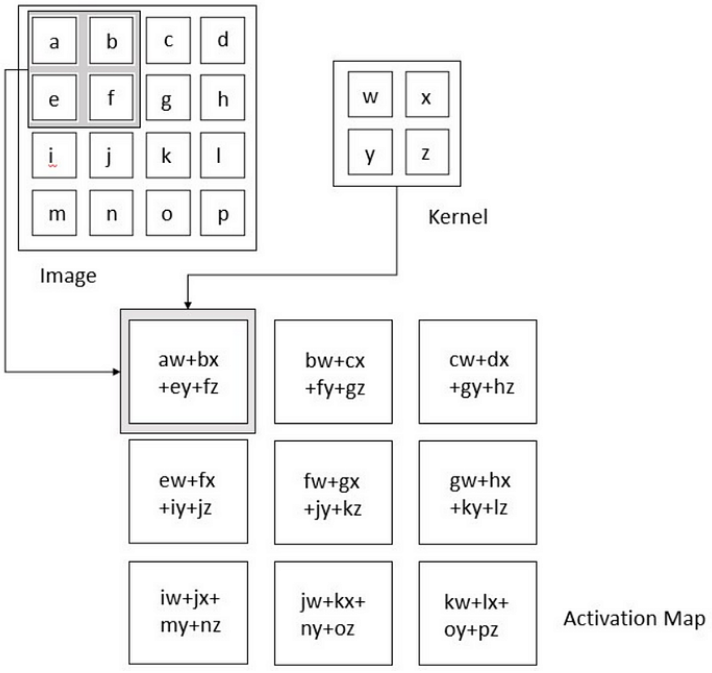


Figure: Convolution Operation

Pooling Layer: The pooling layer replaces the output of the network at certain locations by deriving a summary statistic of the nearby outputs. This helps in reducing the spatial size of the representation, which decreases the required amount of computation and weights. The pooling operation is processed on every slice of the representation individually.

There are several pooling functions such as the average of the rectangular neighborhood, L2 norm of the rectangular neighborhood, and a weighted average based on the distance from the central pixel. However, the most popular process is max pooling, which reports the maximum output from the neighborhood.

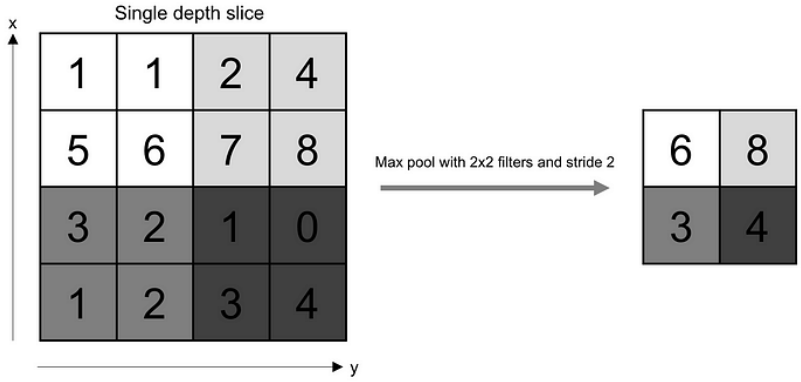


Figure: Pooling Operation

If we have an activation map of size *W* x *W* x *D*, a pooling kernel of spatial size *F*, and stride *S*, then the size of output volume can be determined by the following formula:

$$W_{out} = \frac{W - F}{S} + 1$$

Formula for Padding Layer

This will yield an output volume of size ***Wout* x *Wout* x *D*.** In all cases, pooling provides some translation invariance which means that an object would be recognizable regardless of where it appears on the frame.

Fully Connected Layer: Neurons in this layer have full connectivity with all neurons in the preceding and succeeding layer as seen in regular FCNN. This is why it can be computed as usual by a matrix multiplication followed by a bias effect. The FC layer helps to map the representation between the input and the output.

Since convolution is a linear operation and images are far from linear, non-linearity layers are often placed directly after the convolutional layer to introduce non-linearity to the activation map. There are several types of non-linear operations, the popular ones being:**1. Sigmoid, 2. Tanh, 3. ReLU**

3) Give Brief note on Motivations behind Convolution?

Motivation behind Convolution: Convolution leverages three important ideas that motivated computer vision researchers, they are

- Sparse interactions
- Parameter sharing
- Equivariant representations

Spare interaction: Sparse interaction or sparse weights is implemented by using kernels or feature detector smaller than the input image. If we have an input image of the size 256 by 256 then it becomes difficult to

detect edges in the image may occupy only a smaller subset of pixels in the image. If we use smaller feature detectors then we can easily identify the edges as we focus on the local feature identification. one more advantage is computing output requires fewer operations making it statistically efficient.

Parameter Sharing: Parameter Sharing is used to control the number of parameters or weights used in CNN. In traditional neural networks each weight is used exactly once however in CNN we assume that if the one feature detector is useful to compute one spatial position, then it can be used to compute a different spatial position. As we share parameters across the CNN, it reduces the number of parameters to be learnt and also reduces the computational needs.

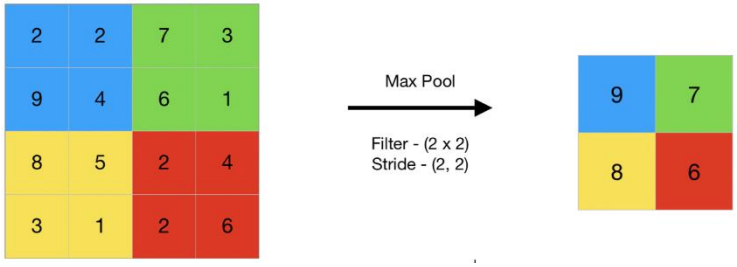
Equivariant representation: It means that object detection is invariant to the changes in illumination, change of position, but internal representation is equivariance to these changes represent(rose) = represent(transform(rose)).

4) Explain Pooling in CNN?

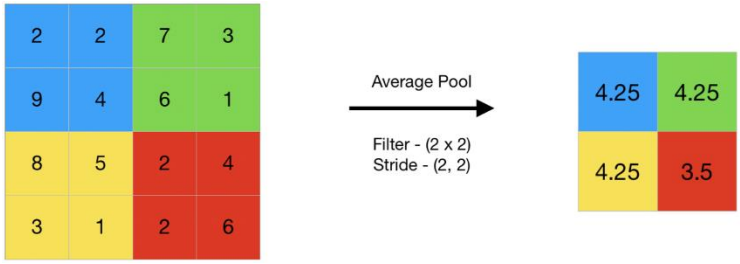
A) In convolutional neural networks (CNNs), the pooling layer is a common type of layer that is typically added after convolutional layers. The pooling layer is used to reduce the spatial dimensions (i.e., the width and height) of the feature maps, while preserving the depth (i.e., the number of channels).

The pooling layer works by dividing the input feature map into a set of non-overlapping regions, called pooling regions. Each pooling region is then transformed into a single output value, which represents the presence of a particular feature in that region. There are four types of pooling layers - **Max, Min, Average, and Global Pooling**. The most common types of pooling operations are max pooling and average pooling.

Max Pooling: In max pooling, the output value for each pooling region is simply the maximum value of the input values within that region. This has the effect of preserving the most salient features in each pooling region, while discarding less relevant information. Max pooling is often used in CNNs for object recognition tasks, as it helps to identify the most distinctive features of an object, such as its edges and corners.



Average Pooling: In average pooling, the output value for each pooling region is the average of the input values within that region. This has the effect of preserving more information than max pooling, but may also dilute the most salient features. Average pooling is often used in CNNs for tasks such as image segmentation and object detection, where a more fine-grained representation of the input is required.



Min Pooling: The Min Pooling layer summarizes the features in a region represented by the minimum value in that region. Contrary to Max Pooling in CNN, this type is mainly used for images with a light background to focus on darker pixels.

Global Pooling: The pooling technique reduces each feature map channel to a single value. This value depends on the type of global pooling, which can be any of the previously explained pooling types. In other words, applying global pooling is similar to using a filter of the exact dimensions of the feature map.

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

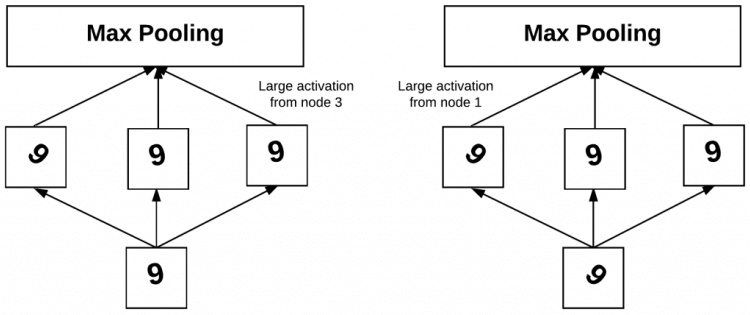
Advantages of Pooling Layer:

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

Disadvantages of Pooling Layer:

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

Example of using Max pooling to learn invariances in images: A CNN *as a whole* can *learn* filters that fire when a pattern is presented at a particular orientation. For example, consider following **Figure**,



CNN as a whole learns filters that will fire when a pattern is presented at a particular orientation. On the *left*, the digit *9* has been rotated $\approx 10^{\circ}$. This rotation is similar to node three, which has learned what the digit *9* looks like when rotated in this manner. This node will have a higher activation than the other two nodes — the max pooling operation will detect this. On the *right* we have a second example, only this time the *9* has been rotated $\approx -45^{\circ}$, causing the first node to have the highest activation

Here, we see the digit *"9"* (bottom) presented to the CNN along with a set of filters the CNN has learned (middle). Since there is a filter inside the CNN that has "learned" what a *"9"* looks like, rotated by 10 degrees, it fires and emits a strong activation. This large activation is captured during the pooling stage and ultimately reported as the final classification.

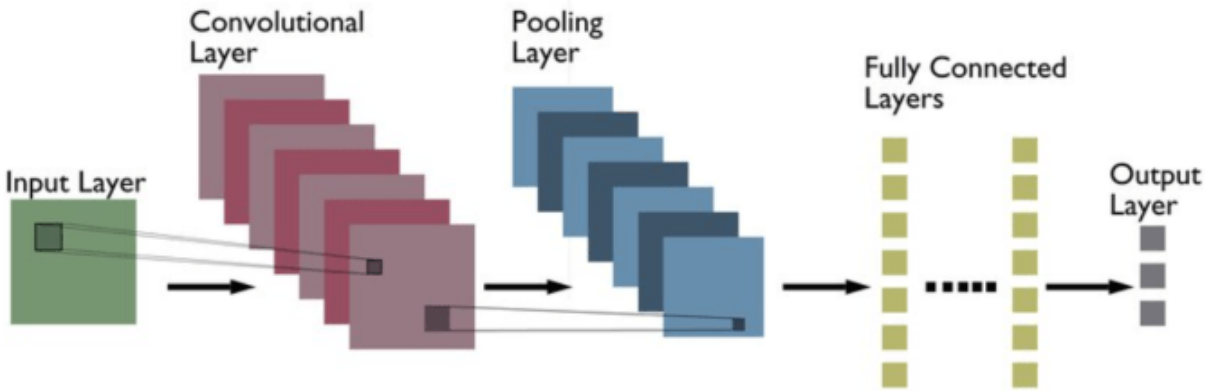
The same is true for the second example (**Figure**, *right*). Here we see the *"9"* rotated by −45 degrees, and since there is a filter in the CNN that has learned what a *"9"* looks like when it is rotated by −45 degrees, the neuron activates and fires. Again, these filters themselves are *not* rotation invariant — it's just that the CNN has learned what a *"9"* looks like under *small rotations* that exist in the training set.

Unless your training data includes digits that are rotated across the full 360-degree spectrum, your CNN is *not* truly rotation invariant.

5) Give the list of various CNN architectures?

A) Convolutional Neural Networks, commonly referred to as CNNs, are a specialized kind of neural network architecture that is designed to process data with a grid-like topology. This makes them particularly well-suited for **dealing with spatial and temporal data**, like **images** and **videos**, that maintain a high degree of correlation between adjacent elements.

CNNs are similar to other neural networks, but they have an added layer of complexity due to the fact that they use a **series of convolutional layers**. Convolutional layers perform a **mathematical operation** called **convolution**, a sort of **specialized matrix multiplication**, on the input data. The convolution operation helps to preserve the spatial relationship between pixels by learning image features using small squares of input data. . The picture below represents a typical CNN architecture

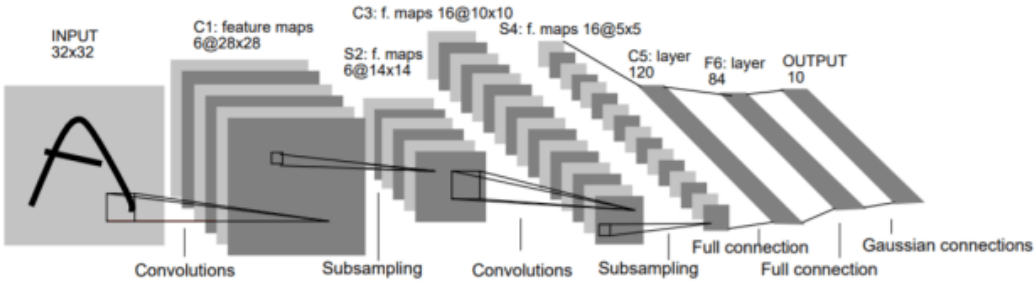


Different types of CNN Architectures: The following is a list of different types of CNN architectures:

- **LeNet**
- **AlexNet**
- **ZF Net**
- **GoogLeNet**
- **VGGNet**

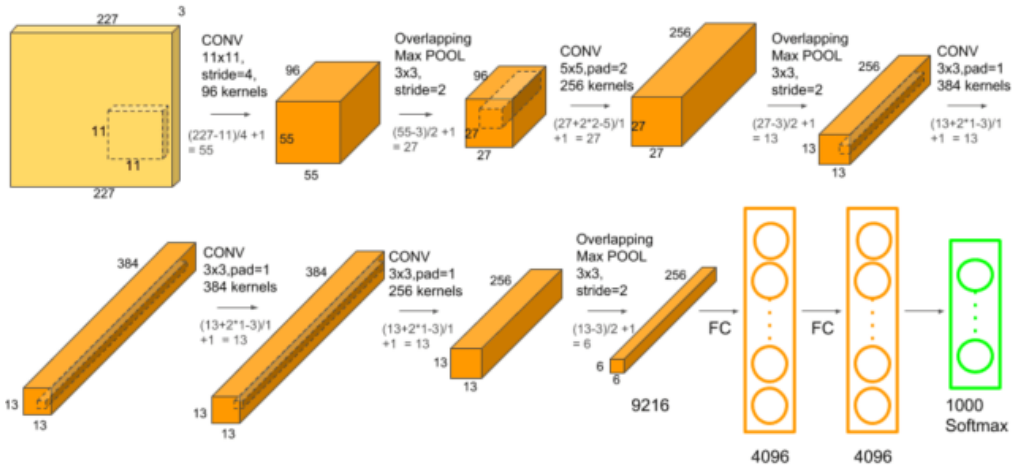
- **ResNet**
- **MobileNets**
- **GoogLeNet_DeepDream**

LeNet: LeNet is the first CNN architecture. It was developed in 1998 by Yann LeCun, Corinna Cortes, and Christopher Burges for handwritten digit recognition problems. LeNet was one of the first successful CNNs and is often considered the "Hello World" of deep learning. It is one of the earliest and most widely-used CNN architectures and has been successfully applied to tasks such as handwritten digit recognition. The LeNet architecture consists of multiple convolutional and pooling layers, followed by a fully-connected layer. The model has five convolution layers followed by two fully connected layers. LeNet was the beginning of CNNs in deep learning for computer vision problems. However, LeNet could not train well due to the vanishing gradients problem. To solve this issue, a shortcut connection layer known as max-pooling is used between convolutional layers to reduce the spatial size of images which helps prevent overfitting and allows CNNs to train more effectively. The diagram below represents LeNet-5 architecture.



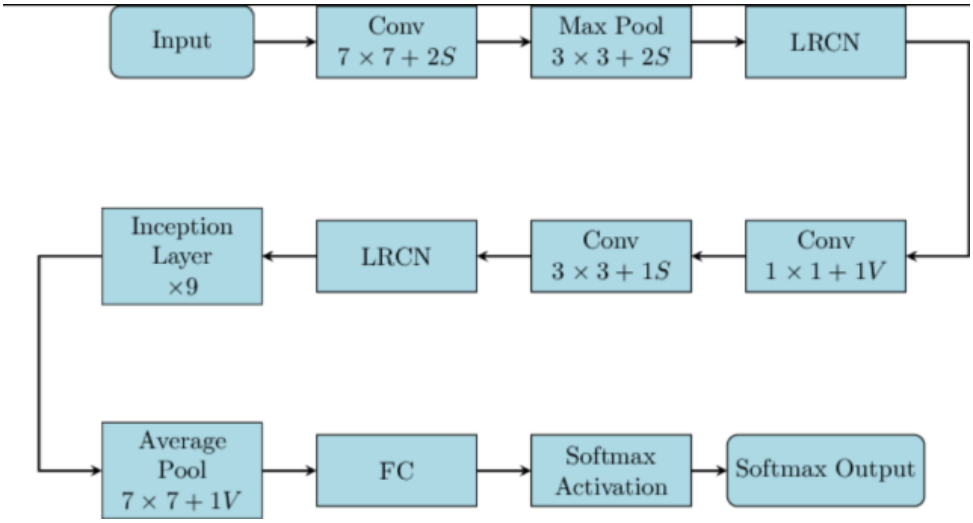
The LeNet CNN is a simple yet powerful model that has been used for various tasks such as handwritten digit recognition, traffic sign recognition, and face detection. Although LeNet was developed more than 20 years ago, its architecture is still relevant today and continues to be used.

AlexNet: AlexNet is the deep learning architecture that popularized CNN. It was developed by Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. AlexNet network had a very similar architecture to LeNet, but was deeper, bigger, and featured Convolutional Layers stacked on top of each other. AlexNet was the first large-scale CNN and was used to win the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. The AlexNet architecture was designed to be used with large-scale image datasets and it achieved state-of-the-art results at the time of its publication. AlexNet is composed of 5 convolutional layers with a combination of max-pooling layers, 3 fully connected layers, and 2 dropout layers. The activation function used in all layers is **Relu.** The activation function used in the output layer is **Softmax**. The total number of parameters in this architecture is around 60 million.



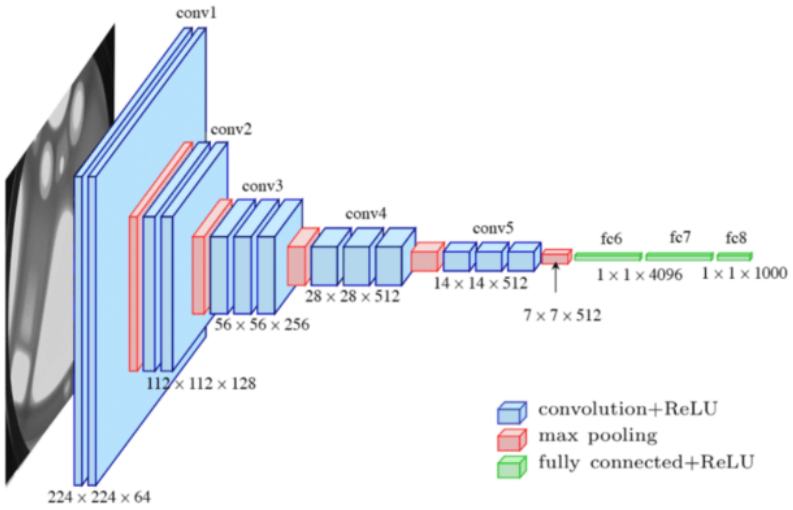
ZF Net: ZFnet is the CNN architecture that uses a combination of fully-connected layers and CNNs. ZF Net was developed by Matthew Zeiler and Rob Fergus. It was the ILSVRC 2013 winner. The network has relatively fewer parameters than AlexNet, but still outperforms it on ILSVRC 2012 classification task by achieving top accuracy with only 1000 images per class. It was an improvement on AlexNet by tweaking the architecture hyperparameters, in particular by expanding the size of the middle convolutional layers and making the stride and filter size on the first layer smaller. It is based on the Zeiler and Fergus model, which was trained on the ImageNet dataset. ZF Net CNN architecture consists of a total of seven layers: Convolutional layer, max-pooling layer (downscaling), concatenation layer, convolutional layer with linear activation function, and stride one, dropout for regularization purposes applied before the fully connected output. This CNN model is computationally more efficient than AlexNet by introducing an approximate inference stage through deconvolutional layers in the middle of CNNs.

GoogLeNet: GoogLeNet is the CNN architecture used by Google to win ILSVRC 2014 classification task. It was developed by Jeff Dean, Christian Szegedy, Alexandro Szegedy et al.. It has been shown to have a notably reduced error rate in comparison with previous winners AlexNet (Ilsvrc 2012 winner) and ZF-Net (Ilsvrc 2013 winner). In terms of error rate, the error is significantly lesser than VGG (2014 runner up). It achieves deeper architecture by employing a number of distinct techniques, including 1×1 convolution and global average pooling. GoogleNet CNN architecture is computationally expensive. To reduce the parameters that must be learned, it uses heavy unpooling layers on top of CNNs to remove spatial redundancy during training and also features shortcut connections between the first two convolutional layers before adding new filters in later CNN layers. Real-world applications/examples of GoogLeNet CNN architecture include Street View House Number (SVHN) digit recognition task, which is often used as a proxy for roadside object detection. Below is the simplified block diagram representing GoogLeNet CNN architecture:



VGGNet: VGGNet is the CNN architecture that was developed by Karen Simonyan, Andrew Zisserman et al. at Oxford University. VGGNet is a 16-layer CNN with up to 95 million parameters and trained on over one billion images (1000 classes). It can take large input images of 224 x 224-pixel size for which it has 4096 convolutional features. CNNs with such large filters are expensive to train and require a lot of data, which is the main reason why CNN architectures like GoogLeNet (AlexNet architecture) work better than VGGNet for most image classification tasks where input images have a size between 100 x 100-pixel and 350 x 350 pixels. Real-world applications/examples of VGGNet CNN architecture include the ILSVRC 2014 classification task, which was also won by GoogleNet CNN architecture. The VGG CNN model is computationally efficient and serves as a strong baseline for many applications in computer vision due to its

applicability for numerous tasks including object detection. Its deep feature representations are used across multiple neural network architectures like YOLO, SSD, etc. The diagram below represents the standard VGG16 network architecture diagram:



ResNet: ResNet is the CNN architecture that was developed by Kaiming He et al. to win the ILSVRC 2015 classification task with a top-five error of only 15.43%. The network has 152 layers and over one million parameters, which is considered deep even for CNNs because it would have taken more than 40 days on 32 GPUs to train the network on the ILSVRC 2015 dataset. CNNs are mostly used for image classification tasks with 1000 classes, but ResNet proves that CNNs can also be used successfully to solve natural language processing problems like sentence completion or machine comprehension, where it was used by the Microsoft Research Asia team in 2016 and 2017 respectively. Real-life applications/examples of ResNet CNN architecture include Microsoft's machine comprehension system, which has used CNNs to generate the answers for more than 100k questions in over 20 categories. The CNN architecture ResNet is computationally efficient and can be scaled up or down to match the computational power of GPUs.

MobileNets: MobileNets are CNNs that can be fit on a mobile device to classify images or detect objects with low latency. MobileNets have been developed by Andrew G Trillion et al.. They are usually very small CNN architectures, which makes them easy to run in real-time using embedded devices like smartphones and drones. The architecture is also flexible so it has been tested on CNNs with 100-300 layers and it still works better than other architectures like VGGNet. Real-life examples of MobileNets CNN architecture include CNNs that is built into Android phones to run Google's Mobile Vision API, which can automatically identify labels of popular objects in images.

GoogLeNet_DeepDream: GoogLeNet_DeepDream is a deep dream CNN architecture that was developed by Alexander Mordvintsev, Christopher Olah, etc. It uses the Inception network to generate images based on CNN features. The architecture is often used with the ImageNet dataset to generate psychedelic images or create abstract artworks using human imagination at the ICLR 2017 workshop by David Ha, et al.

6) Give Brief notes on CNN Visualization Techniques?

A) CNN Visualization Techniques: Convolutional Neural Network (CNN) visualizations refer to techniques for visually representing the learned features or patterns captured by different layers of a CNN. These visualizations can help to gain insights into how the network is processing and classifying input images.

There are several types of visualizations for CNNs, including feature map visualization, activation maximization, integrated gradients, saliency maps, etc.

1. **Feature maps** show the output of each filter in a given layer of the network for a particular input image. These visualizations can help to understand the features that the network is detecting at each layer.
2. **Activation maximization** involves generating an image that maximally activates a particular neuron or group of neurons in a given layer of the network. This technique can be used to visualize the features learned by the network at different levels of abstraction.
3. **Integrated Gradients** is a technique used to measure the importance of each input feature (e.g., pixel values) for a given output class of a deep neural network. The technique involves computing the gradient of the output class with respect to the input features, and then integrating this gradient over a path from a baseline input (e.g., an all-zero image) to the actual input image. The resulting integrated gradient map highlights the important input features that contributed to the output class.
4. **Saliency maps** highlight the regions of an input image that are most important for a particular output class. This technique can help to identify which parts of the input image the network is focusing on when making its classification decision.

CNN visualizations are powerful tools for understanding how these complex neural networks are processing and classifying images and can be useful for tasks such as network debugging, feature engineering, and model interpretability.

7) Sequence Modeling and Recurrent Neural Networks?

A) Sequence Models: Sequence models are the machine learning models that input or output sequences of data. Sequential data includes text streams, audio clips, video clips, time-series data and etc. Recurrent Neural Networks (RNNs) is a popular algorithm used in sequence models.

Applications of Sequence Models:

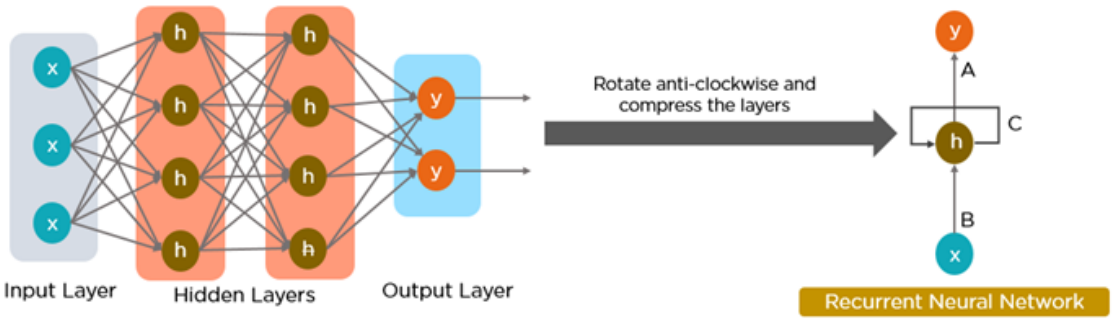
1. Speech recognition: In speech recognition, an audio clip is given as an input and then the model has to generate its text transcript. Here both the input and output are sequences of data.

2. Sentiment Classification: In sentiment classification opinions expressed in a piece of text is categorized. Here the input is a sequence of words.

3. Video Activity Recognition: In video activity recognition, the model needs to identify the activity in a video clip. A video clip is a sequence of video frames, therefore in case of video activity recognition input is a sequence of data.

Recurrent Neural Networks (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 behavior. Simply said, recurrent neural networks can anticipate sequential data in a way that other algorithms can’t.



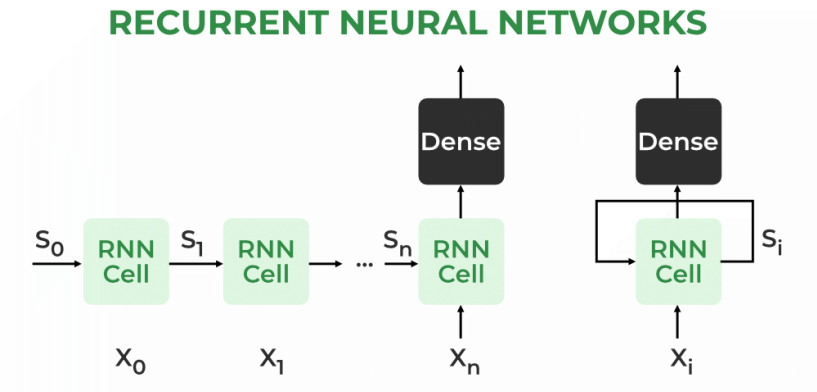
All of the inputs and outputs in standard neural networks are independent of one another, however in some circumstances, such as when predicting the next word of a phrase, the prior words are necessary, and so the previous words must be remembered. As a result, RNN was created, which used a Hidden Layer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

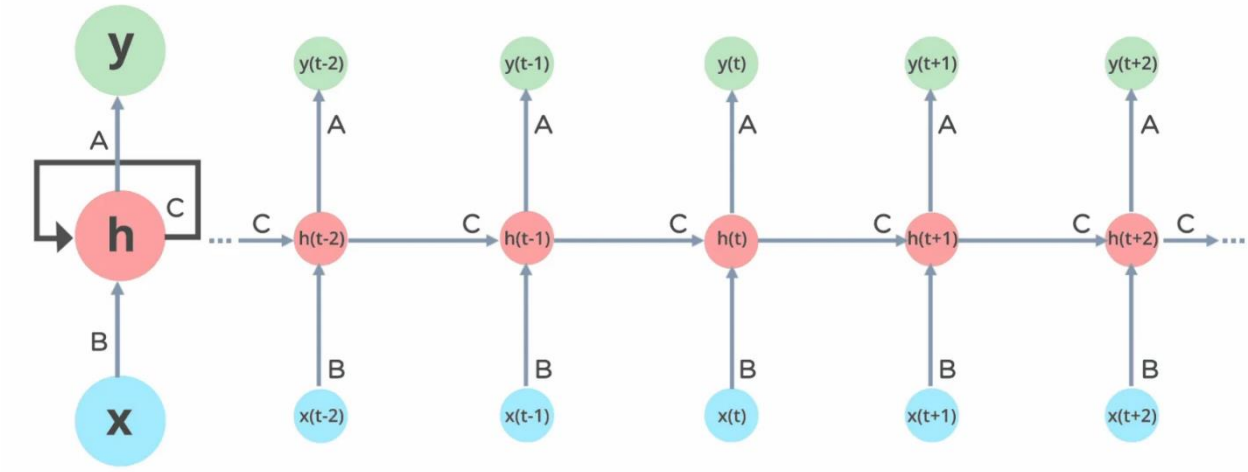
Architecture Of Recurrent Neural Network : RNNs have the same input and output architecture as any other deep neural architecture. However, differences arise in the way information flows from input to output. Unlike Deep neural networks where we have different weight matrices for each Dense network in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

$$h = \sigma(UX + Wh_{-1} + B)$$
$$Y = O(Vh + C) \text{ Hence}$$
$$Y = f(X, h, W, U, V, B, C)$$

Here S is the State matrix which has element si as the state of the network at timestep i The parameters in the network are W, U, V, c, b which are shared across timestep



RNN working: In Recurrent Neural networks, the information cycles through a loop to the middle hidden layer.



The input layer 'x' takes in the input to the neural network and processes it and passes it onto the middle layer. The middle layer 'h' can consist of multiple hidden layers, each with its own activation functions and weights and biases. If you have a neural network where the various parameters of different hidden layers are not affected by the previous layer, ie: the neural network does not have memory, then you can use a recurrent neural network.

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

The formula for calculating the current state:

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

where:

h_t -> current state

h_{t-1} -> previous state

x_t -> input state

Formula for applying Activation function(tanh):

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

where:

w_{hh} -> weight at recurrent neuron

w_{xh} -> weight at input neuron

The formula for calculating output:

$$y_t = W_{hy}h_t$$

Y_t -> output

W_{hy} -> weight at output layer

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

Training through RNN

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

Advantages of Recurrent Neural Network

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

Disadvantages of Recurrent Neural Network

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

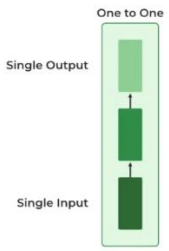
Applications of Recurrent Neural Network

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

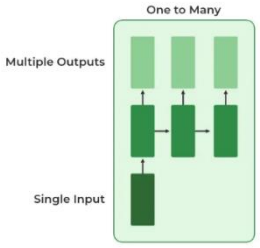
Types Of RNN: There are four types of RNNs based on the number of inputs and outputs in the network.

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

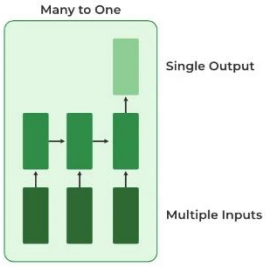
One to One : This type of RNN behaves the same as any simple Neural network it is also known as Vanilla Neural Network. In this Neural network, there is only one input and one output.



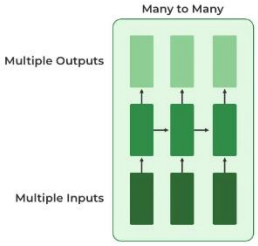
One To Many : In this type of RNN, there is one input and many outputs associated with it. One of the most used examples of this network is Image captioning where given an image we predict a sentence having Multiple words.



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



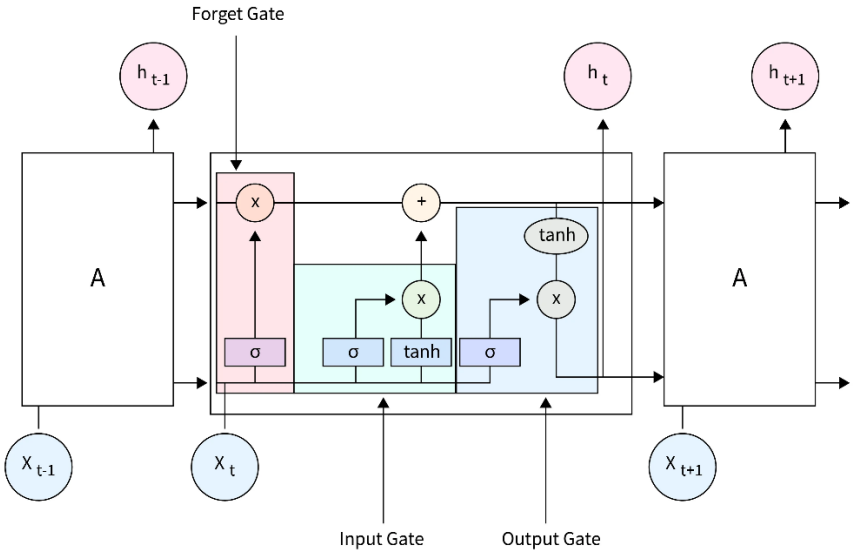
Many to Many : In this type of neural network, there are multiple inputs and multiple outputs corresponding to a problem. One Example of this Problem will be language translation. In language translation, we provide multiple words from one language as input and predict multiple words from the second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



8) Describe about LSTM?

A) Long Short-Term Memory (LSTM): LSTM is a type of Neural Network used in the field of Deep Learning. LSTM stands for **Long-Short-term-Memory**. LSTM is an improved version of the RNN (Recurrent Neural Network). LSTM is mainly used in **Time series** and **Sequence** data because RNN doesn't perform efficiently as the gap length rises. LSTM differs from conventional Feedforward Networks as it uses previous data and its output to affect the current predictions. LSTM is also better at retaining information for longer periods when compared with RNN. Long Short-Term Memory uses Gated Cells to remember or forget previous information.

LSTM was designed by **Hochreiter & Schmidhuber**. It's a challenging task to get your head around LSTM as it belongs to the complex area of Deep Learning. LSTM deals with algorithms to uncover the underlying relationships in the given sequential data. The **chain structure LSTM** contains four neural networks and different memory blocks called **cells**. The LSTM may keep information for a long time by default, and information is retained by the cells, and the three gates do the memory manipulations.



Need of LSTM: LSTM was introduced to tackle the problems and challenges in Recurrent Neural Networks. It touches on the topic of RNN. RNN is a type of Neural Network that stores the previous output to help improve its future predictions. Vanilla RNN has a "short-term" memory. The input at the beginning of the sequence doesn't affect the output of the Network after a while, maybe 3 or 4 inputs. This is called a **long-term dependency issue**.

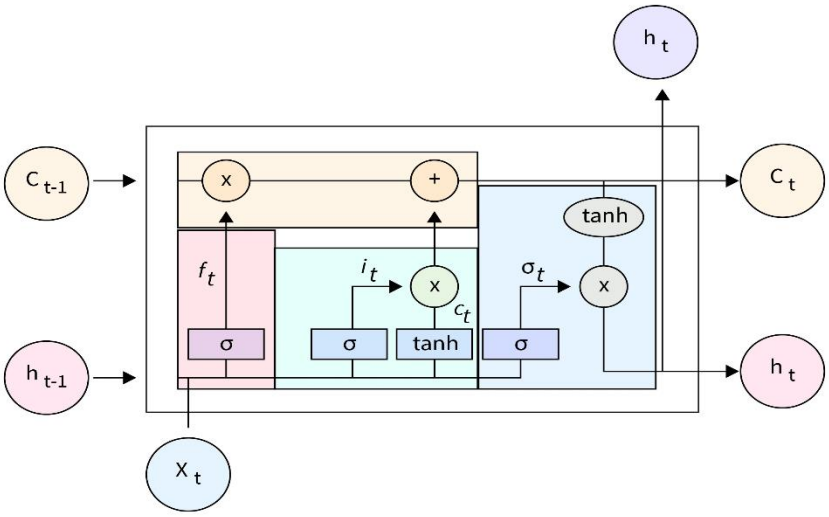
Example:
Let's take this sentence.
The Sun rises in the _____.
An RNN could easily return the correct output that the sun rises in the East as all the necessary information is nearby.

Example2:
I was born in Japan, …… and I speak fluent _____.

In this sentence, the RNN would be unable to return the correct output as it requires remembering the word Japan for a long duration. Since RNN only has a "Short-term" memory, it doesn't work well. LSTM solves this problem by enabling the Network to remember **Long-term dependencies**.

The other RNN problems are the **Vanishing Gradient** and **Exploding Gradient**. It arises during the Backpropagation of the Neural Network. For example, suppose the gradient of each layer is contained between 0 and 1. As the value gets multiplied in each layer, it gets smaller and smaller, ultimately, a value very close to 0. This is the Vanishing gradient problem. The converse, when the values are greater than 1, exploding gradient problem occurs, where the value gets really big, disrupting the training of the Network. Again, these problems are tackled in LSTMs.

Structure of LSTM: LSTM is a cell that consists of 3 gates. A **forget gate**, **input gate**, and **output gate**. The gates decide which information is important and which information can be forgotten. The cell has two states **Cell State** and **Hidden State**. They are continuously updated and carry the information from the previous to the current time steps. The cell state is the "long-term" memory, while the hidden state is the "short-term" memory.



Forget Gate: Forget gate is responsible for deciding what information should be **removed from the cell state**. It takes in the hidden state of the previous time-step and the current input and passes it to a **Sigma** Activation Function, which outputs a value between 0 and 1, where 0 means forget and 1 means keep.

Forget Gate

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

Input Gate: The Input Gate considers the current input and the hidden state of the previous time step. The input gate is used to **update the cell state** value. It has two parts. The first part contains the **Sigma activation** function. Its purpose is to decide what percent of the information is required. The second part passes the two values to a **Tanh activation** function. It aims to map the data between -1 and 1. To obtain the relevant information required from the output of Tanh, we multiply it by the output of the Sigma function. This is the output of the Input gate, which updates the cell state.

Input Gate

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

$$C_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_f)$$

Output Gate: The output gate returns the hidden state for the next time stamp. The output gate has two parts. The first part is a **Sigma function**, which serves the same purpose as the other two gates, to decide the percent of the relevant information required. Next, the newly updated cell state is passed through a **Tanh function** and multiplied by the output from the sigma function. This is now the **new hidden state**.

Output Gate

$$\sigma_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

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

Cell State: The forget gate and input gate update the cell state. The cell state of the previous state is multiplied by the output of the forget gate. The output of this state is then summed with the output of the input gate. This value is then used to calculate **hidden state** in the output gate.

Cell State

$$C_t = f_t * C_{t-1}, i_t * \bar{C}_t$$

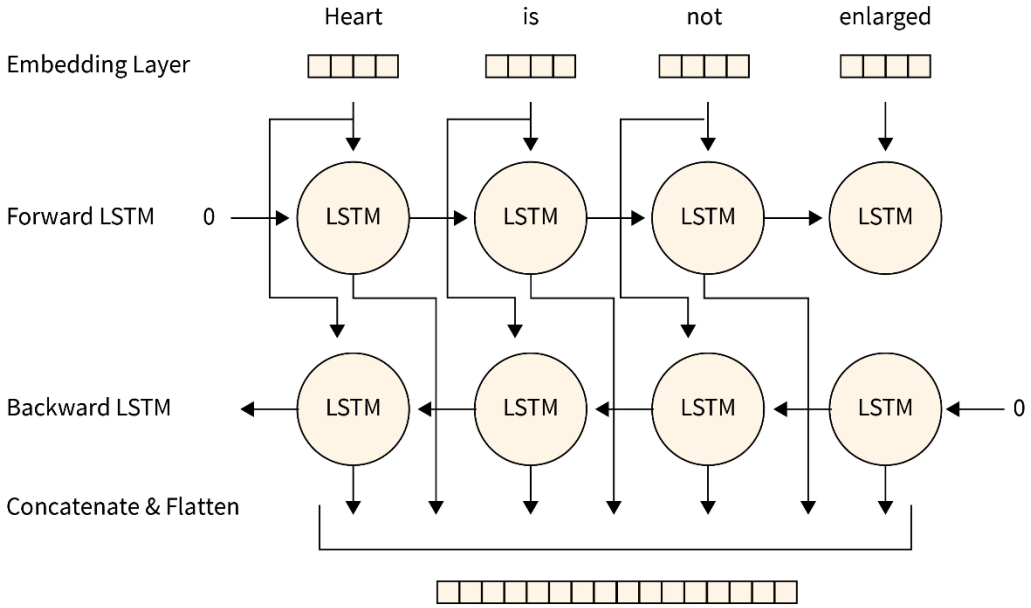
LSTM Working: The LSTM architecture is similar to RNN, but instead of the feedback loop has an LSTM cell. The sequence of LSTM cells in each layer is fed with the **output of the last cell**. This enables the cell to get the previous inputs and sequence information. A cyclic set of steps happens in each LSTM cell

- The Forget gate is computed.
- The Input gate value is computed.
- The Cell state is updated using the above two outputs.
- The output (hidden state) is computed using the output gate.

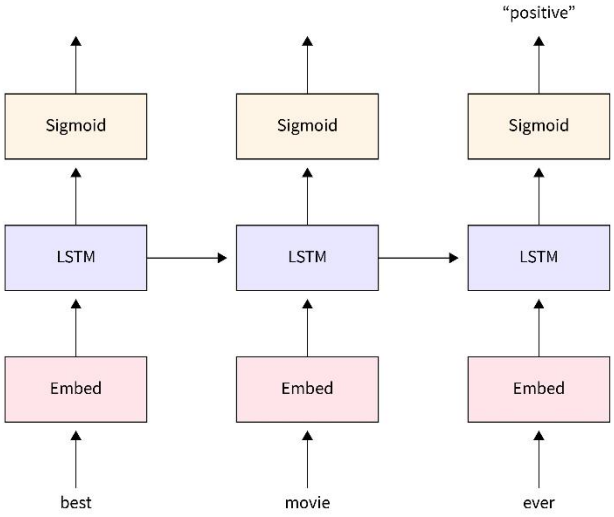
These series of steps occur in every LSTM cell. The intuition behind LSTM is that the Cell and Hidden states carry the previous information and pass it on to future time steps. The Cell state is **aggregated** with all the past data information and is the **long-term** information retainer. The Hidden state carries the output of the last cell, i.e. **short-term memory**. This combination of Long term and short-term memory techniques enables LSTM's to perform well In time series and sequence data.

Applications of LSTM:

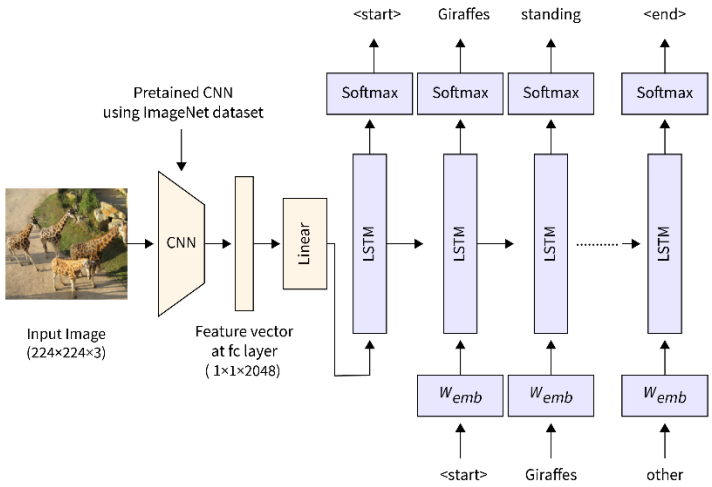
- **Language Modeling:** LSTMs have been used to build language models that can generate natural language text, such as in machine translation systems or chatbots.



- **Time series prediction:** LSTMs have been used to model time series data and predict future values in the series. For example, LSTMs have been used to predict stock prices or traffic patterns.
- **Sentiment analysis:** LSTMs have been used to analyze text sentiments, such as in social media posts or customer reviews.



- **Speech recognition:** LSTMs have been used to build speech recognition systems that can transcribe spoken language into text.
- **Image captioning:** LSTMs have been used to generate descriptive captions for images, such as in image search engines or automated image annotation systems.



9) Describe about Bidirectional RNN and Bidirectional LSTM?

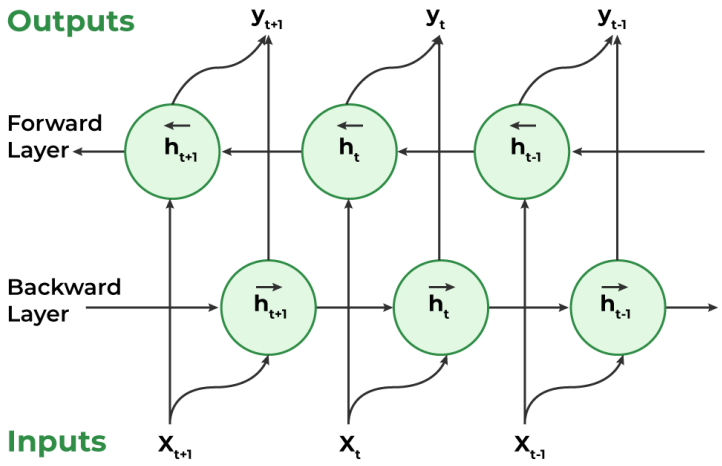
A) **Bi-directional Recurrent Neural Network:** An architecture of a neural network called a bidirectional recurrent neural network (BRNN) is made to process sequential data. In order for the network to use information from both the past and future context in its predictions, BRNNs process input sequences in both the forward and backward directions. This is the main distinction between BRNNs and conventional recurrent neural networks.

A BRNN has two distinct recurrent hidden layers, one of which processes the input sequence forward and the other of which processes it backward. After that, the results from these

hidden layers are collected and input into a prediction-making final layer. Any recurrent neural network cell, such as Long Short-Term Memory (LSTM) or Gated Recurrent Unit, can be used to create the recurrent hidden layers.

The BRNN functions similarly to conventional recurrent neural networks in the forward direction, updating the hidden state depending on the current input and the prior hidden state at each time step. The backward hidden layer, on the other hand, analyses the input sequence in the opposite manner, updating the hidden state based on the current input and the hidden state of the next time step.

In order to update the model parameters, the gradients are computed for both the forward and backward passes of the backpropagation through the time technique that is typically used to train BRNNs. The input sequence is processed by the BRNN in a single forward pass at inference time, and predictions are made based on the combined outputs of the two hidden layers. layers.



Working of Bidirectional Recurrent Neural Network

1. **Inputting a sequence:** A sequence of data points, each represented as a vector with the same dimensionality, are fed into a BRNN. The sequence might have different lengths.
2. **Dual Processing:** Both the forward and backward directions are used to process the data. On the basis of the input at that step and the hidden state at step t-1, the hidden state at time step t is determined in the forward direction. The input at step t and the hidden state at step t+1 are used to calculate the hidden state at step t in a reverse way.
3. **Computing the hidden state:** A non-linear activation function on the weighted sum of the input and previous hidden state is used to calculate the hidden state at each step. This creates a memory mechanism that enables the network to remember data from earlier steps in the process.
4. **Determining the output:** A non-linear activation function is used to determine the output at each step from the weighted sum of the hidden state and a number of output weights. This output has two options: it can be the final output or input for another layer in the network.
5. **Training:** The network is trained through a supervised learning approach where the goal is to minimize the discrepancy between the predicted output and the actual output. The network adjusts its weights in the input-to-hidden and hidden-to-output connections during training through backpropagation.

To calculate the output from an RNN unit, we use the following formula:

H_t (Forward) = A(X_t * W_{XH} (forward) + H_{t-1} (Forward) * W_{HH} (Forward) + b_H (Forward)
H_t (Backward) = A(X_t * W_{XH} (Backward) + H_{t+1} (Backward) * W_{HH} (Backward) + b_H (Backward)

where,
A = activation function,
W = weight matrix
b = bias

The hidden state at time t is given by a combination of H_t (Forward) and H_t (Backward). The output at any given hidden state is :

$$\mathbf{Y_t = H_t * W_{AY} + b_y}$$

The training of a BRNN is similar to backpropagation through a time algorithm. BPTT algorithm works as follows:

- Roll out the network and calculate errors at each iteration
- Update weights and roll up the network.

However, because forward and backward passes in a BRNN occur simultaneously, updating the weights for the two processes may occur at the same time. This produces inaccurate outcomes. Thus, the following approach is used to train a BRNN to accommodate forward and backward passes individually.

Applications of Bidirectional Recurrent Neural Network

Bi-RNNs have been applied to various natural language processing (NLP) tasks, including:

1. **Sentiment Analysis:** By taking into account both the prior and subsequent context, BRNNs can be utilized to categorize the sentiment of a particular sentence.
2. **Named Entity Recognition:** By considering the context both before and after the stated thing, BRNNs can be utilized to identify those entities in a sentence.
3. **Part-of-Speech Tagging:** The classification of words in a phrase into their corresponding parts of speech, such as nouns, verbs, adjectives, etc., can be done using BRNNs.
4. **Machine Translation:** BRNNs can be used in encoder-decoder models for machine translation, where the decoder creates the target sentence and the encoder analyses the source sentence in both directions to capture its context.
5. **Speech Recognition:** When the input voice signal is processed in both directions to capture the contextual information, BRNNs can be used in automatic speech recognition systems.

Advantages of Bidirectional RNN

- **Context from both past and future:** With the ability to process sequential input both forward and backward, BRNNs provide a thorough grasp of the full context of a sequence. Because of this, BRNNs are effective at tasks like sentiment analysis and speech recognition.
- **Enhanced accuracy:** BRNNs frequently yield more precise answers since they take both historical and upcoming data into account.
- **Efficient handling of variable-length sequences:** When compared to conventional RNNs, which require padding to have a constant length, BRNNs are better equipped to handle variable-length sequences.

- **Resilience to noise and irrelevant information:** BRNNs may be resistant to noise and irrelevant data that are present in the data. This is so because both the forward and backward paths offer useful information that supports the predictions made by the network.
- **Ability to handle sequential dependencies:** BRNNs can capture long-term links between sequence pieces, making them extremely adept at handling complicated sequential dependencies.

Disadvantages of Bidirectional RNN

- **Computational complexity:** Given that they analyze data both forward and backward, BRNNs can be computationally expensive due to the increased amount of calculations needed.
- **Long training time**: BRNNs can also take a while to train because there are many parameters to optimize, especially when using huge datasets.
- **Difficulty in parallelization:** Due to the requirement for sequential processing in both the forward and backward directions, BRNNs can be challenging to parallelize.
- **Overfitting:** BRNNs are prone to overfitting since they include many parameters that might result in too complicated models, especially when trained on short datasets.
- **Interpretability:** Due to the processing of data in both forward and backward directions, BRNNs can be tricky to interpret since it can be difficult to comprehend what the model is doing and how it is producing predictions.

Bidirectional LSTM (BiLSTM): Bidirectional LSTM or BiLSTM is a term used for a sequence model which contains two LSTM layers, one for processing input in the forward direction and the other for processing in the backward direction. It is usually used in NLP-related tasks. The intuition behind this approach is that by processing data in both directions, the model is able to better understand the relationship between sequences (e.g. knowing the following and preceding words in a sentence).

For example, The first statement is “Server can you bring me this dish” and the second statement is “He crashed the server”. In both these statements, the word server has different meanings and this relationship depends on the following and preceding words in the statement. The bidirectional LSTM helps the machine to understand this relationship better than compared with unidirectional LSTM. This ability of BiLSTM makes it a suitable architecture for tasks like **sentiment analysis, text classification, and machine translation**.

Architecture: The architecture of bidirectional LSTM comprises of two unidirectional LSTMs which process the sequence in both forward and backward directions. This architecture can be interpreted as having two separate LSTM networks, one gets the sequence of tokens as it is while the other gets in the reverse order. Both of these LSTM network returns a probability vector as output and the final output is the combination of both of these probabilities. It can be represented as:

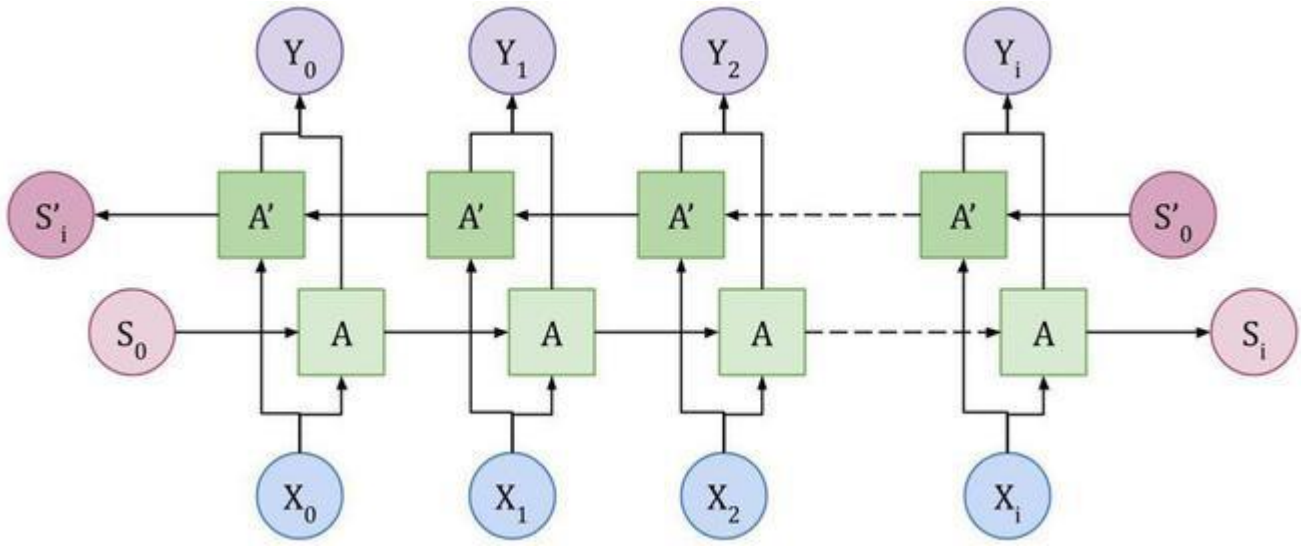
$$++p_t = p_t^f + p_t^b$$

where,

p_t : Final probability vector of the network.

p_t^f : Probability vector from the forward LSTM network.

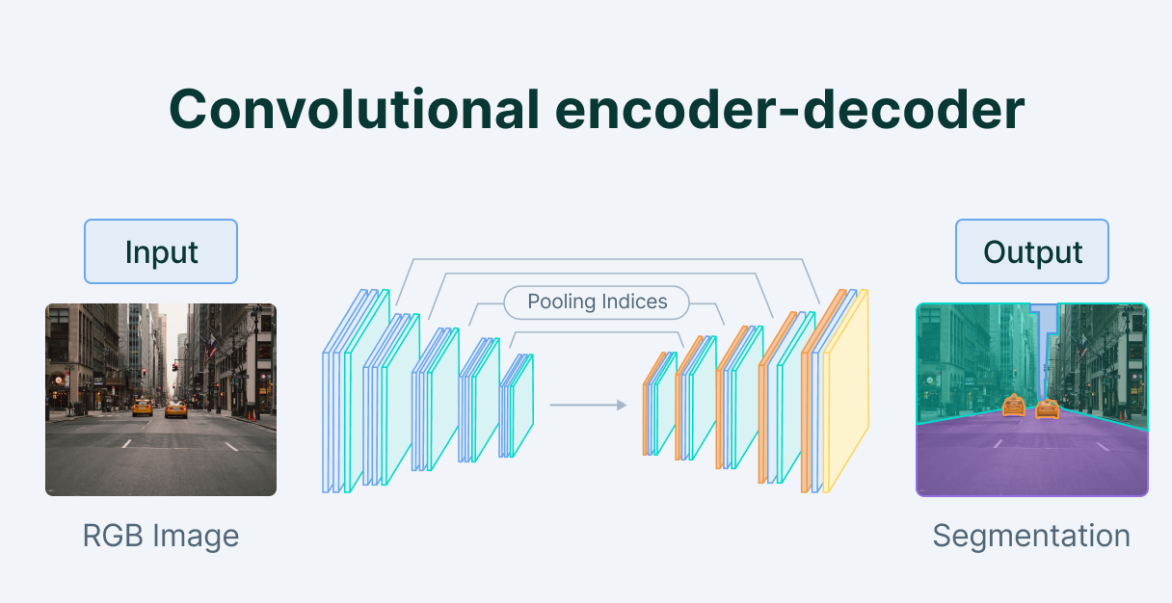
- : Probability vector from the backward LSTM network.



where X_i is the input token, Y_i is the output token, and A and A' are LSTM nodes. The final output of $Y_i$is the combination of A and A' LSTM nodes.

10) What is an autoencoder?

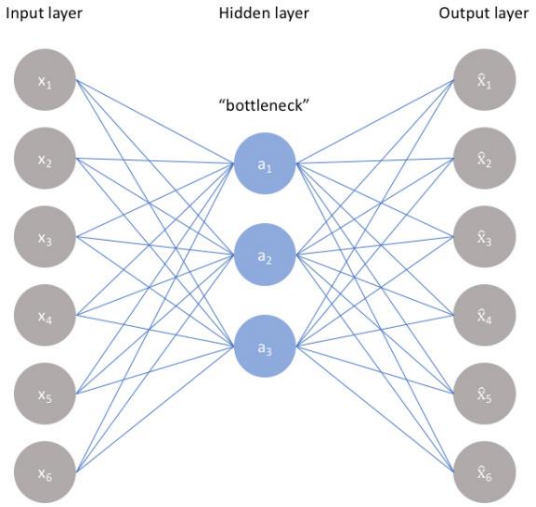
An autoencoder is a type of artificial neural network used to learn data encodings in an unsupervised manner. The aim of an autoencoder is to learn a lower-dimensional representation (encoding) for a higher-dimensional data, typically for dimensionality reduction, by training the network to capture the most important parts of the input image.



Autoencoders are self-supervised machine learning models which are used to reduce the size of input data by recreating it. These models are trained as supervised machine learning models and during inference, they work as unsupervised models that's why they are called self-supervised models. **Autoencoder is made up of 3 components namely,**

1. **Encoder**: A module that compresses the train-validate-test set input data into an encoded representation that is typically several orders of magnitude smaller than the input data.
2. **Bottleneck:** A module that contains the compressed knowledge representations and is therefore the most important part of the network.
3. **Decoder:** A module that helps the network "decompress" the knowledge representations and reconstructs the data back from its encoded form. The output is then compared with a ground truth.

The architecture as a whole look something like this:



Training autoencoders: we need to set 4 hyperparameters before *training* an autoencoder:

1. **Code size:** The code size or the size of the bottleneck is the most important hyperparameter used to tune the autoencoder. The bottleneck size decides how much the data has to be compressed. This can also act as a regularization term.
2. **Number of layers:** Like all neural networks, an important hyperparameter to tune autoencoders is the depth of the encoder and the decoder. While a higher depth increases model complexity, a lower depth is faster to process.
3. **Number of nodes per layer:** The number of nodes per layer defines the weights we use per layer. Typically, the number of nodes decreases with each subsequent layer in the autoencoder as the input to each of these layers becomes smaller across the layers.
4. **Reconstruction Loss:** The loss function we use to train the autoencoder is highly dependent on the type of input and output we want the autoencoder to adapt to. If we are working with image data, the most popular loss functions for reconstruction are MSE Loss and L1 Loss. In case the inputs and outputs are within the range [0,1], as in MNIST, we can also make use of Binary Cross Entropy as the reconstruction loss.

Types of Autoencoders

- **Under Complete Autoencoders:** Under complete autoencoders is an unsupervised neural network that you can use to generate a compressed version of the input data. It is done by taking in an image and trying to predict the same image as output, thus reconstructing the image from its compressed bottleneck region. The primary use for autoencoders like these is generating a latent space or bottleneck, which forms a compressed substitute of the input data and can be easily decompressed back with the help of the network when needed.

- **Sparse Autoencoders:** Sparse autoencoders are controlled by changing the number of nodes at each hidden layer. Since it is impossible to design a neural network with a flexible number of nodes at its hidden layers, sparse autoencoders work by penalizing the activation of some neurons in hidden layers. It means that a penalty directly proportional to the number of neurons activated is applied to the loss function. As a means of regularizing the neural network, the sparsity function prevents more neurons from being activated. There are two types of regularizes used:

    1. The L1 Loss method is a general regularizer we can use to add magnitude to the model.

2. The KL-divergence method considers the activations over a collection of samples at once rather than summing them as in the L1 Loss method. We constrain the average activation of each neuron over this collection.

- **Contractive Autoencoders:** The input is passed through a bottleneck in a contractive autoencoder and then reconstructed in the decoder. The bottleneck function is used to learn a representation of the image while passing it through. The contractive autoencoder also has a regularization term to prevent the network from learning the identity function and mapping input into output. To train a model that works along with this constraint, we need to ensure that the derivatives of the hidden layer activations are small concerning the input**.**

- **Denoising Autoencoders**: Have you ever wanted to remove noise from an image but didn't know where to start? If so, then denoising autoencoders are for you! Denoising autoencoders are similar to regular autoencoders in that they take an input and produce an output. However, they differ because they don't have the input image as their ground truth. Instead, they use a noisy version. The loss function usually used with these networks is L2 or L1 loss.

- **Variational Autoencoders:** Variational autoencoders (VAEs) are models that address a specific problem with standard autoencoders. When you train an autoencoder, it learns to represent the input just in a compressed form called the latent space or the bottleneck. However, this latent space formed after training is not necessarily continuous and, in effect, might not be easy to interpolate.

Applications: Autoencoders have various applications like:

- **Anomaly detection**: autoencoders can identify data anomalies using a loss function that penalizes model complexity. It can be helpful for anomaly detection in financial markets, where you can use it to identify unusual activity and predict market trends.
- **Data denoising image and audio:** autoencoders can help clean up noisy pictures or audio files. You can also use them to remove noise from images or audio recordings.
- **Image inpainting**: autoencoders have been used to fill in gaps in images by learning how to reconstruct missing pixels based on surrounding pixels. For example, if you're trying to restore an old photograph that's missing part of its right side, the autoencoder could learn how to fill in the missing details based on what it knows about the rest of the photo.
- **Information retrieval**: autoencoders can be used as content-based image retrieval systems that allow users to search for images based on their content.

11) What is a Boltzmann machine? Briefly explain about Deep Boltzmann machines (DBM)?

A) A Boltzmann machine is an unsupervised deep learning model in which every node is connected to every other node. It is a type of recurrent neural network, and the nodes make binary decisions with some level of bias. These machines are not deterministic deep learning models, they are stochastic or generative deep learning models. They are representations of a system.

A Boltzmann machine has two kinds of nodes

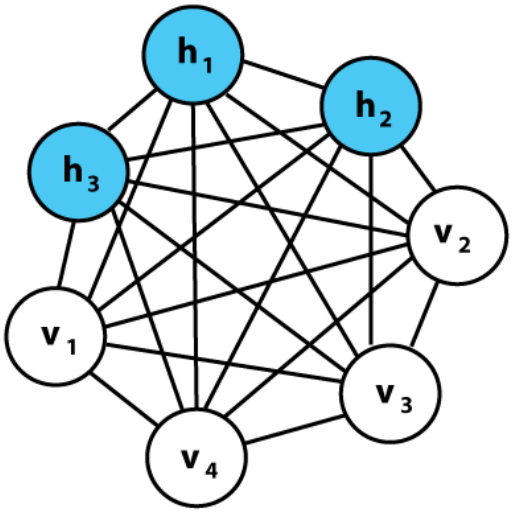
- Visible nodes: These are nodes that can be measured and are measured.
- Hidden nodes: These are nodes that cannot be measured or are not measured.

A Boltzmann machine can be called a stochastic Hopfield network which has hidden units. It has a network of units with an 'energy' defined for the overall network. Boltzmann machines seek to reach thermal equilibrium. It essentially looks to optimize global distribution of energy. But the temperature and energy of the system are relative to laws of thermodynamics and are not literal.

A Boltzmann machine is made up of a learning algorithm that enables it to discover interesting features in datasets composed of binary vectors. The learning algorithm tends to be slow in networks that have many layers of feature detectors but it is possible to make it faster by implementing a learning layer of feature detectors.

They use stochastic binary units to reach probability distribution equilibrium (to minimize energy). It is possible to get multiple Boltzmann machines to collaborate together to form far more sophisticated systems like deep belief networks.

The Boltzmann machine is named after **Ludwig Boltzmann**, an Austrian scientist who came up with the Boltzmann distribution. However, this type of network was first developed by **Geoff Hinton**, a Stanford Scientist.



Working of Boltzmann machine: Boltzmann machines are non-deterministic (stochastic) generative Deep Learning models that only have two kinds of nodes - hidden and visible nodes. They don't have any output nodes, and that's what gives them the non-deterministic feature. They learn patterns without the **typical 1 or 0 type** output through which patterns are learned and optimized using Stochastic Gradient Descent.

A major difference is that unlike other traditional networks (A/C/R) which don't have any connections between the input nodes, Boltzmann Machines have connections among the input nodes. Every node is connected to all other nodes irrespective of whether they are input or hidden nodes. This enables them to share information among themselves and self-generate subsequent data. You'd only measure what's on the visible nodes and not what's on the hidden nodes. After the input is provided, the Boltzmann machines are able to capture all the parameters, patterns and correlations among the data. It is because of this that they are known as deep generative models and they fall into the class of Unsupervised Deep Learning.

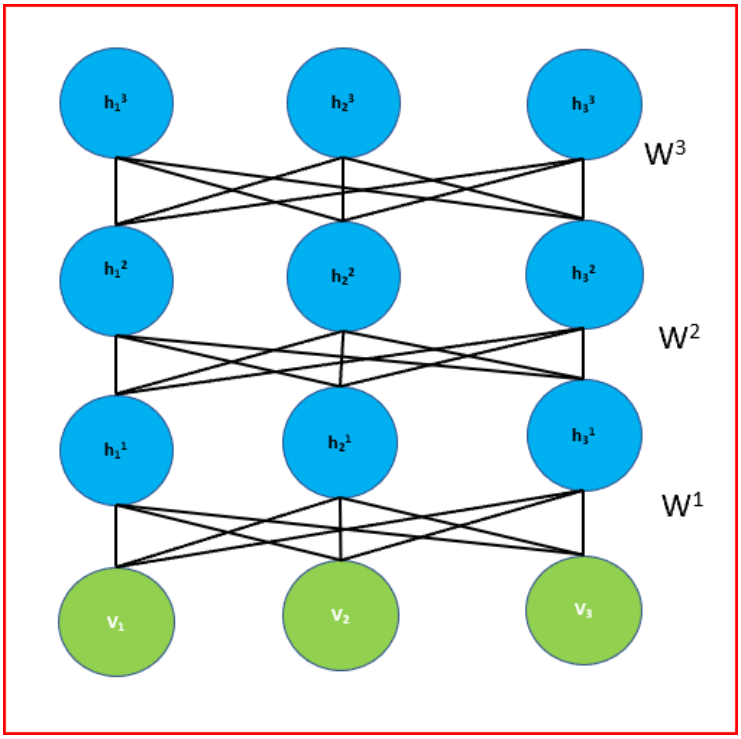
Types of Boltzmann machines: There are three types of Boltzmann machines. These are:

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

1. Restricted Boltzmann Machines (RBMs): While in a full Boltzmann machine all the nodes are connected to each other and the connections grow exponentially, an RBM has certain restrictions with respect to node connections. In a Restricted Boltzmann Machine, hidden nodes cannot be connected to each other while visible nodes are connected to each other.

2. Deep Belief Networks (DBNs): In a Deep Belief Network, you could say that multiple Restricted Boltzmann Machines are stacked, such that the outputs of the first RBM are the inputs of the subsequent RBM. The connections within individual layers are undirected, while the connections between layers are directed. However, there is an exception here. The connection between the top two layers is undirected. A deep belief network can either be trained using a Greedy Layer-wise Training Algorithm or a Wake-Sleep Algorithm.

3. Deep Boltzmann Machines (DBMs): Deep Boltzmann Machines are very similar to Deep Belief Networks. The difference between these two types of Boltzmann machines is that while connections between layers in DBNs are directed, in DBMs, the connections within layers, as well as the connections between the layers, are all undirected.



Deep Boltzmann Machines (DBMs) are a type of generative probabilistic model that belong to the family of deep learning architectures. They were introduced as a way to model complex, high-dimensional probability distributions.

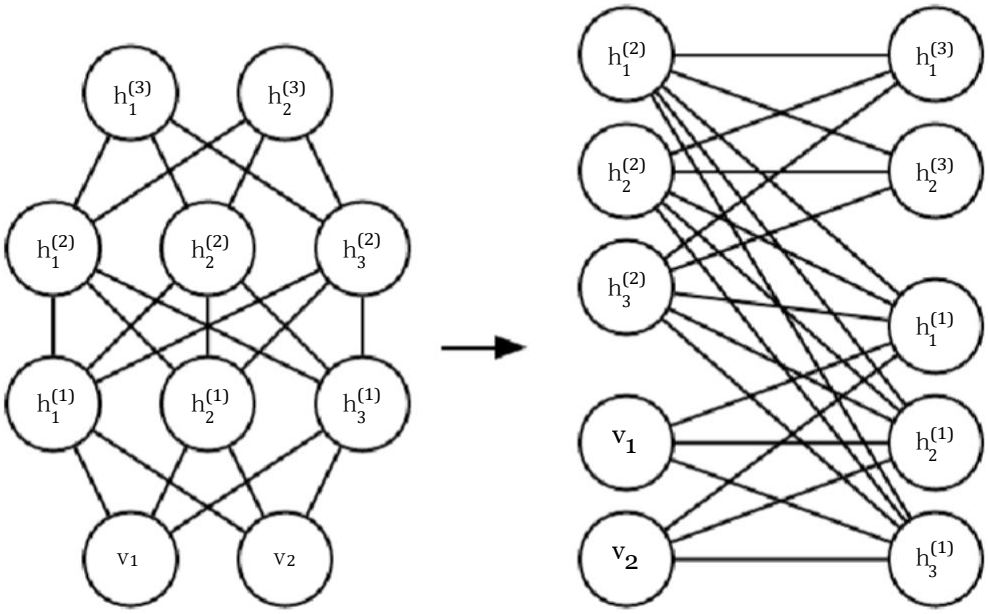


Figure: A deep Boltzmann machine, re-arranged to reveal its bipartite graph structure.

In comparison to the RBM energy function, the DBM energy function includes connections between the hidden units (latent variables) in the form of the weight matrices ($\boldsymbol{W}^{(2)}$ and $\boldsymbol{W}^{(3)}$). As we will see, these connections have significant consequences for both the model behavior as well as how we go about performing inference in the model. The energy function of a Deep Boltzmann Machine (DBM) is a crucial aspect of its probabilistic modeling framework. The energy function is used to define the joint probability distribution over the visible and hidden units in the network. Let's break down the energy function for a standard two-layer DBM, consisting of visible units (v) and two sets of hidden units (h1 and h2). The energy function (E) for a Deep Boltzmann Machine is given by:

$$E(v, h_1, h_2) = -\sum_i \sum_j w_{ij}^{(1)} v_i h_j^{(1)} - \sum_j \sum_k w_{jk}^{(2)} h_j^{(1)} h_k^{(2)} - \sum_i b_i v_i - \sum_j c_j^{(1)} h_j^{(1)} - \sum_k c_k^{(2)} h_k^{(2)}$$

Here's a breakdown of the terms in the energy function:

1. v_i represents the state (binary or real-valued) of visible unit i.
2. $h_j^{(1)}$ and $h_k^{(2)}$ represent the states of hidden units in the first and second hidden layers, respectively.
3. $w_{ij}^{(1)}$ are the weights connecting visible unit i to hidden unit j in the first hidden layer.
4. $w_{jk}^{(2)}$ are the weights connecting hidden unit j in the first hidden layer to hidden unit k in the second hidden layer.
5. $b_i, c_j^{(1)}$, and $c_k^{(2)}$ are biases for the visible units, first hidden layer units, and second hidden layer units, respectively.

The energy function determines the compatibility between different configurations of visible and hidden units. Configurations with lower energy are assigned higher probabilities in the model. The negative exponential of the energy function is used to define the probability distribution over the joint configuration space:

$$P(v, h_1, h_2) = \frac{e^{-E(v,h_1,h_2)}}{Z}$$

Where Z is the partition function, a normalization term calculated as the sum of the exponential of the negative energy over all possible configurations.

$$Z = \sum_{v,h_1,h_2} e^{-E(v,h_1,h_2)}$$

Training a DBM involves adjusting the weights and biases to maximize the likelihood of the training data. This often involves techniques like Contrastive Divergence and layer-wise pre-training.

UNIT - V

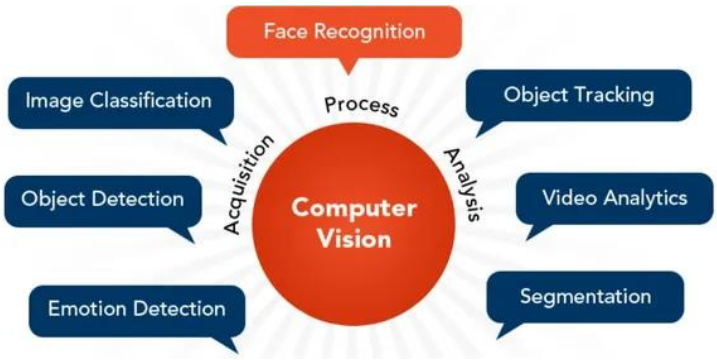
1) Give brief description of Deep Learning application?

Deep learning is a branch of artificial intelligence (AI) that teaches neural networks to learn and reason. Its capacity to resolve complicated issues and deliver cutting-edge performance in various sectors has attracted significant interest and appeal in recent years. Deep learning algorithms have revolutionized AI by allowing machines to process and comprehend enormous volumes of data. The structure and operation of the human brain inspired these algorithms.

Common Applications of Deep Learning in Artificial Intelligence: Deep learning has many uses in many fields, and its potential grows. Let's analyze a few of artificial intelligence's widespread profound learning uses.

- Image Recognition and Computer Vision
- Natural Language Processing (NLP)
- Speech Recognition and Voice Assistants
- Recommendation Systems
- Autonomous Vehicles
- Healthcare and Medical Imaging
- Fraud Detection and Cybersecurity
- Gaming and Virtual Reality

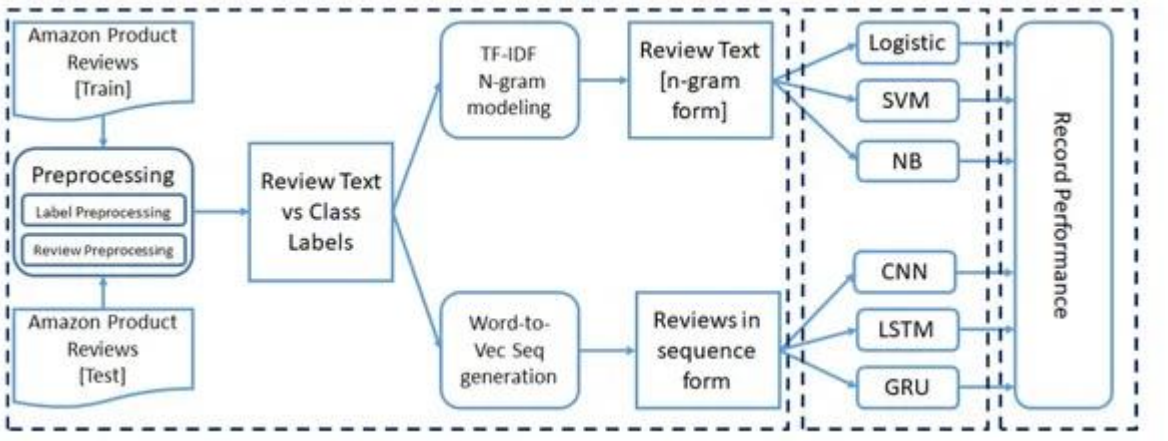
Image Recognition and Computer Vision



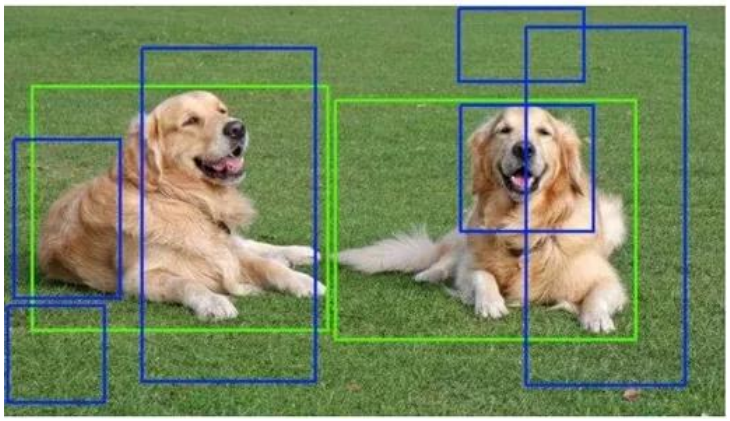
The performance of image recognition and computer vision tasks has significantly improved due to deep learning. Computers can now reliably classify and comprehend images owing to training deep neural networks on enormous datasets, opening up a wide range of applications.

A smartphone app that can instantaneously determine a dog's breed from a photo and self-driving cars that employ computer vision algorithms to detect pedestrians, traffic signs, and other roadblocks for safe navigation are two examples of this in practice.

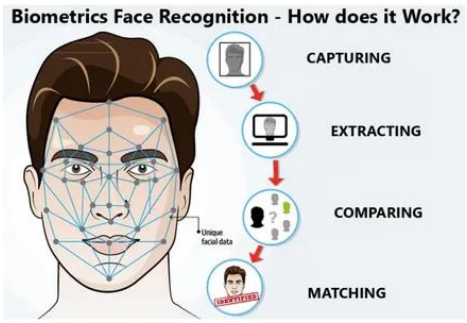
Deep Learning Models for Image Classification: The process of classifying photos entails giving them labels based on the content of the images. **Convolutional neural networks (CNNs)**, one type of deep learning model, have performed exceptionally well in this context. They can categorize objects, situations, or even specific properties within an image by learning to recognize patterns and features in visual representations.



Object Detection and Localization using Deep Learning: Object detection and localization go beyond image categorization by identifying and locating various things inside an image. Deep learning methods have recognized and localized objects in real-time, such as You Only Look Once (YOLO) and region-based convolutional neural networks (R-CNNs). This has uses in robotics, autonomous cars, and surveillance systems, among other areas.



Applications in Facial Recognition and Biometrics: Deep learning has completely changed the field of facial recognition. Hence, allowing for the precise identification of people using their facial features. Security systems, access control, monitoring, and law enforcement use facial recognition technology. Deep learning methods have also been applied in biometrics for functions including voice recognition, iris scanning, and fingerprint recognition.



Natural Language Processing (NLP)

Natural language processing (NLP) aims to make it possible for computers to comprehend, translate, and create human language. NLP has substantially advanced primarily to deep learning, making strides in several language-related activities. Virtual voice assistants like Apple's Siri and Amazon's Alexa, who can comprehend spoken orders and questions, are a practical illustration of this.

Deep Learning for Text Classification and Sentiment Analysis: Text classification entails classifying text materials into several groups or divisions. Deep learning models like **recurrent neural networks (RNNs)** and **long short-term memory (LSTM)** networks have been frequently used for text categorization tasks. To ascertain the sentiment or opinion expressed in a text, whether good, negative, or neutral, sentiment analysis is a widespread use of text categorization.

Language Translation and Generation with Deep Learning: Machine translation systems have considerably improved because of deep learning. Deep learning-based neural machine translation (NMT) models have been shown to perform better when converting text across multiple languages. These algorithms can gather contextual data and generate more precise and fluid translations. Deep learning models have also been applied to creating news stories, poetry, and other types of text, including coherent paragraphs.

Question Answering and Chatbot Systems Using Deep Learning: Deep learning is used by **chatbots** and **question-answering programs** to recognize and reply to human inquiries. Transformers and attention mechanisms, among other deep learning models, have made tremendous progress in understanding the context and semantics of questions and producing pertinent answers. Information retrieval systems, virtual assistants, and customer service all use this technology.

Speech Recognition and Voice Assistants

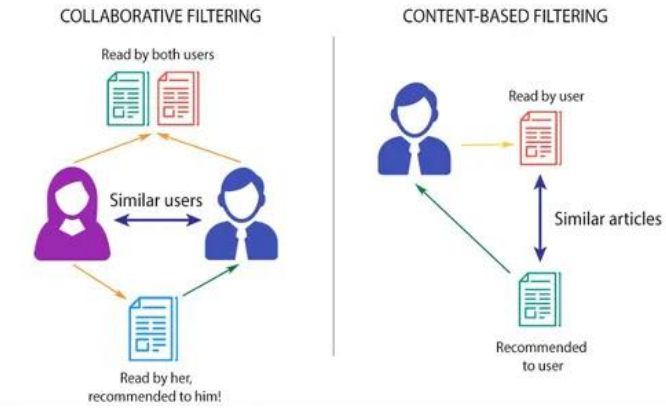
The creation of voice assistants that can comprehend and respond to human speech and the advancement of **speech recognition** systems have significantly benefited from deep learning. A real-world example is using your smartphone's voice recognition feature to dictate messages rather than typing them and asking a smart speaker to play your favorite tunes or provide the weather forecast.

Deep Learning Models for Automatic Speech Recognition: Systems for automatic speech recognition (ASR) translate spoken words into written text. Recurrent neural networks and attention-based models, in particular, have substantially improved ASR accuracy. Better voice commands, transcription services, and accessibility tools for those with speech difficulties are the outcome. Some examples are voice search features in search engines like Google, Bing, etc.

Voice Assistants Powered by Deep Learning Algorithms: Daily, we rely heavily on voice assistants like Siri, Google Assistant, and Amazon Alexa. Guess what drives them? Deep learning it is. Deep learning techniques are used by these intelligent devices to recognize and carry out spoken requests. The technology also enables voice assistants to recognize speech, decipher user intent, and deliver precise and pertinent responses thanks to deep learning models.

Applications in Transcription and Voice-Controlled Systems: Deep learning-based speech recognition has applications in transcription services, where large volumes of audio content must be accurately converted into text. Voice-controlled systems, such as smart homes and in-car infotainment systems, utilize deep learning algorithms to enable hands-free control and interaction through voice commands.

Recommendation Systems



Recommendation systems use deep learning algorithms to offer people personalized recommendations based on their tastes and behavior.

Deep Learning-Based Collaborative Filtering: A standard method used in recommendation systems to suggest products/services to users based on how they are similar to other users is collaborative filtering. Collaborative filtering has improved accuracy and performance thanks to deep learning models like matrix **factorization** and **deep autoencoders**, which have produced more precise and individualized recommendations.

Deep neural networks have been used to identify intricate links and patterns in user behavior data, allowing for more precise and individualized suggestions. Deep learning algorithms can forecast user preferences and make relevant product, movie, or content recommendations by looking at user interactions, purchase history, and demographic data. An instance of this is when streaming services recommend films or TV shows based on your interests and history.

Applications in E-Commerce and Content Streaming Platforms

Deep learning algorithms are widely employed to fuel recommendation systems in e-commerce platforms and video streaming services like Netflix and Spotify. These programs increase user pleasure and engagement by assisting users in finding new goods, entertainment, or music that suits their tastes and preferences.

Autonomous Vehicles

Deep learning has significantly impacted how well autonomous vehicles can understand and navigate their surroundings. These vehicles can analyze enormous volumes of sensor data in real-time using powerful deep learning algorithms. Thus, enabling them to make wise decisions, navigate challenging routes, and guarantee the safety of passengers and pedestrians. This game-changing technology has prepared the path for a time when driverless vehicles will completely change how we travel.

Deep Learning Algorithms for Object Detection and Tracking

Autonomous vehicles must perform crucial tasks, including object identification and tracking, to recognize and monitor objects like pedestrians, cars, and traffic signals. Convolutional and recurrent neural networks (CNNs) and other deep learning algorithms have proved essential in obtaining high accuracy and real-time performance in object detection and tracking.

Deep Reinforcement Learning for Decision-Making in Self-Driving Cars

Autonomous vehicles are designed to make complex decisions and navigate various traffic circumstances using deep reinforcement learning. This technology is profoundly used in self-driving cars manufactured by companies like Tesla. These vehicles can learn from historical driving data and adjust to changing road conditions using deep neural networks. Self-driving cars demonstrate this in practice, which uses cutting-edge sensors and artificial intelligence algorithms to navigate traffic, identify impediments, and make judgments in real time.

Applications in Autonomous Navigation and Safety Systems

The development of autonomous navigation systems that decipher sensor data, map routes, and make judgments in real time depends heavily on deep learning techniques. These systems focus on collision avoidance, generate lane departure warnings, and offer adaptive cruise control to enhance the general safety and dependability of the vehicles.

Healthcare and Medical Imaging

Deep learning has shown tremendous potential in revolutionizing healthcare and medical imaging by assisting in diagnosis, disease detection, and patient care. Revolutionizing diagnostics using AI-powered algorithms that can precisely identify early-stage tumors from medical imaging is an example of how to do this. This will help with prompt treatment decisions and improve patient outcomes.

Fraud Detection and Cybersecurity

Deep learning has become essential in detecting anomalies, identifying fraud patterns, and strengthening cybersecurity systems. In fraud prevention systems, deep neural networks have been used to recognize and stop fraudulent transactions, **credit card fraud**, and identity theft. These algorithms examine user behavior, transaction data, and historical patterns to spot irregularities and notify security staff. This enables proactive fraud prevention and shields customers and organizations from financial loss. Organizations like Visa, Mastercard, and PayPal use deep neural networks. It helps improve their fraud detection systems and guarantees secure customer transactions.

Gaming and Virtual Reality

Deep learning algorithms have produced more intelligent and lifelike video game characters. Game makers may create realistic animations, enhance character behaviors, and make **more immersive gaming experiences** by training deep neural networks on enormous datasets of motion capture data.

Experiences in augmented reality (AR) and **virtual reality (VR)** have been improved mainly due to deep learning. Deep neural networks are used by VR and AR systems to correctly track and identify objects, detect movements and facial expressions, and build real virtual worlds, enhancing the immersiveness and interactivity of the user experience.

2) Explain the process of Hand written digits recognition using Deep Learning?

A) Handwritten digit recognition is the ability of a computerto recognize the human handwritten digits from different sources like images, papers, touch screens, etc, and classify them into 10 predefined classes (0-9). This has been a topic of boundless-research in the field of deep learning. Digit recognition has many applications like number platerecognition, postal mail sorting, bank check processing, etc.

In Handwritten digit recognition, we face many challenges because of different styles of writing of different peoples as it is not an Optical character recognition. This research provides a comprehensive comparison between different machinelearning and deep learning algorithms for the purpose of handwritten digit recognition. For this, we have used Support Vector Machine, Multilayer Perceptron, and Convolutional Neural Network. The comparison between these algorithmsis carried out on the basis of their accuracy, errors, and testing-training time corroborated by plots and charts that have been constructed using matplotlib for visualization.

The accuracy of any model is paramount as more accurate models make better decisions. The models with low accuracy are not suitable for real-world applications. **Ex-** For an automated bank cheque processing system where the system recognizes the amount and date on the check, high accuracyis very critical. If the system incorrectly recognizes a digit, it can lead to major damage which is not desirable. That's why an algorithm with high accuracy is required in these real-world applications. Hence, we are providing a comparisonof different algorithms based on their accuracy so that themost accurate algorithm with the least chances of errors canbe employed in various applications of handwritten digit recognition.

METHODOLOGY:

Importing the libraries: Libraries are useful tools that can make a web developer's job more efficient. It's a set of prewritten code, that we can call while programming our own code. Basically, it's the work that's already done by someone else that you can make use of, without having to do it yourself. You can also use it in your own code. Different libraries have different restrictions on fair use, but this is a code that was designed to be used by others, instead of just standing alone.

The libraries used in this code are -

a. ***Tensorflow*** - Tensorflow's framework is open source, it is used in machine learning and other computations on various data.

b. ***OpenCV*** – OpenCV (Open-Source Computer Vision Library) is an open-source computer vision and machine learning software libraries.

c. ***NumPy*** – NumPy stands for Numerical Python.

Loading datasets: For the training and testing data, we will be using a Dataset which is present in the Tensorflow API. This specific dataset is the **MNIST** data that contains around fifty thousand images for the training data and another ten thousandimages for testing data confined to a dimension of 28X28.

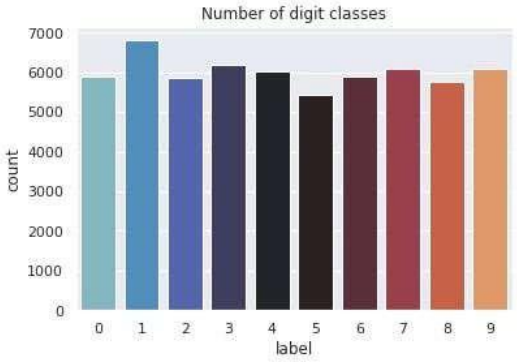


Figure 1. Bar graph illustrating the MNIST handwritten digit training dataset (Label vs Total number of training samples).

Figure 2. Plotting of some random MNIST Handwritten digits.

Creating a model: We are creating a model using a sequential neural network to recognize handwritten digits. We are using Three dense layers namely input hidden and output layers. Here we used 128, 128, 10 parameters for the previously mentioned layers respectively.

Training the Model: We are training the model using 3 dense layers, one using input layer, one using hidden layer and the last one using the output layer. The 3 dense layers take 128, 128, 10 parameters each. We flatten the pixels of the greyscale image in the input layer. Then we apply the activation functions to the weights in the hidden layer. The output layer gives the result as a prediction.

Testing the Model: Once the model is trained, we can use the loss function and accuracy function to test the model. We should have accuracy as high as possible and loss as less as possible to get the desired output (with accuracy being close to 1 and loss being close to 0). We can use “ADAM” as our optimizer, “cross-entropy” as our loss and “accuracy” as our metrics.

Getting the output: We take a set of inputs and upload them into our model. Then we use a while loop to analyze each inputindividually and give the predictions for each image. This process has been shown in a flowchart below in Fig.

Flowchart

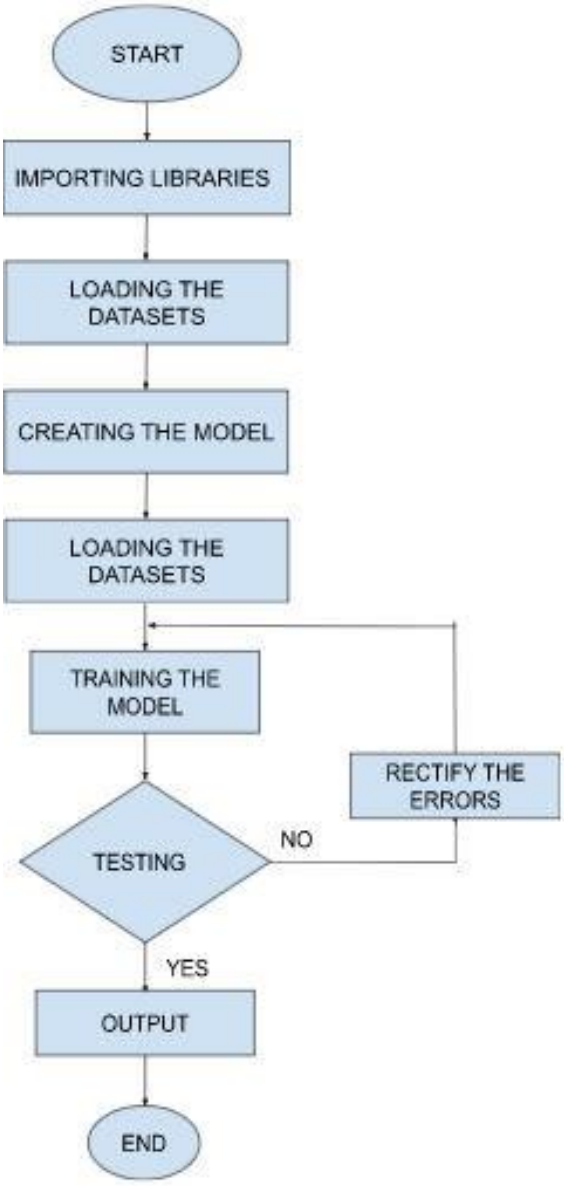


Fig. Methodology used in Implementation of Handwritten Recognition

IMPLEMENTATION: Neural network is implemented wherein the model recognizes and predicts a handwritten digit. Initially **Tensorflow and Keras** are used to form the bones of the implementation. We load the datasets from both of these open-source libraries and make our model to analyze thousands of images. The model learns all the patterns, pixel placements of the greyscale images and all the neural connections.

Keras is an API (Application Programming Interface), which is designed for machine learning and deep learning. It's an opensource library which has a lot of inbuilt data. It's the interface of Tensorflow library.

1. It follows the best practices available for reducing cognitive load.
2. It provides simple & consistent APIs (Application Programming Interface).
3. It has extensive documentation and has developer guides.
4. It minimizes the amount of user actions needed for common use cases, and it provides actionable & clear error messages.

TensorFlow framework is open source, it is used in machine learning and other computations on various

data. It has the symbol of TF. It allows the developers to create the learning algorithms on their set of data and models, and to experiment with diversealgorithms. It allows the developers to create data flow structures, which shows how the data will move through a series ofgraphs or nodes. All the datasets which we used in this program are taken from Tensorflow.

Then we use the **ADAM** optimizer for training our model.It's the best optimizer to train our neural network in less time with high efficiency. An optimizer is used to update the networkweights live during the training. Then we use the loss function and the metrics function to check the performance of our model.We use cross-entropy and accuracy respectively to check the performance. Once the model is trained a, we save it to ourcomputer and comment on all the preprocessing and training code. So, whenever the code is required, we can just load the savedmodel since all the data is stored in it. Now the final step is to give the inputs to our model. For this we scan the digits we wroteon a sheet of paper or draw the digit in MS paint tool and download the image in PNG format. Once we upload all the imagesinto our python script, then we use while loop to all each image and analyze it to predict the output.

Results And Discussion: From this implementation, we are able to identify the handwritten digits as input given to the code as shown in Fig.5, analyze it and predict the probability output as shown in Fig.6.With the code we are able to show that written data in MS paint application can be saved. Where the saved file is now loaded into the program and will run. With a while loop, we check each image and predict the value if multiple data set files are present. With this the image is analyzed by the already learnt neural connectionsand the grayscale pixels. It will now provide us with a prediction of the accurate output of recognized data. So, we can successfully recognize and digitalize our data. We are successfully able to understand and use Sequential Model, ReLU, SoftMax, adam, Cross-Entropy, Accuracy functions for image recognition. The snips of the code used in the program are shown in the Fig.7(a) and Fig.7(b).

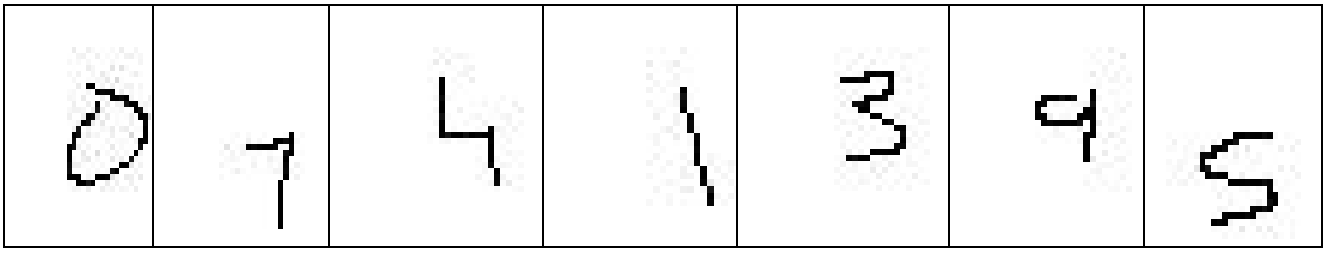


Fig.5 Handwritten digits given as input Images

```
1/1 [==============================] - 0s 363ms/step
This digit might be 0
1/1 [==============================] - 0s 24ms/step
This digit might be 7
1/1 [==============================] - 0s 22ms/step
This digit might be 4
1/1 [==============================] - 0s 21ms/step
This digit might be 1
1/1 [==============================] - 0s 35ms/step
This digit might be 3
1/1 [==============================] - 0s 26ms/step
This digit might be 9
1/1 [==============================] - 0s 47ms/step
This digit might be 5
```

Fig.6 Outputs Obtained

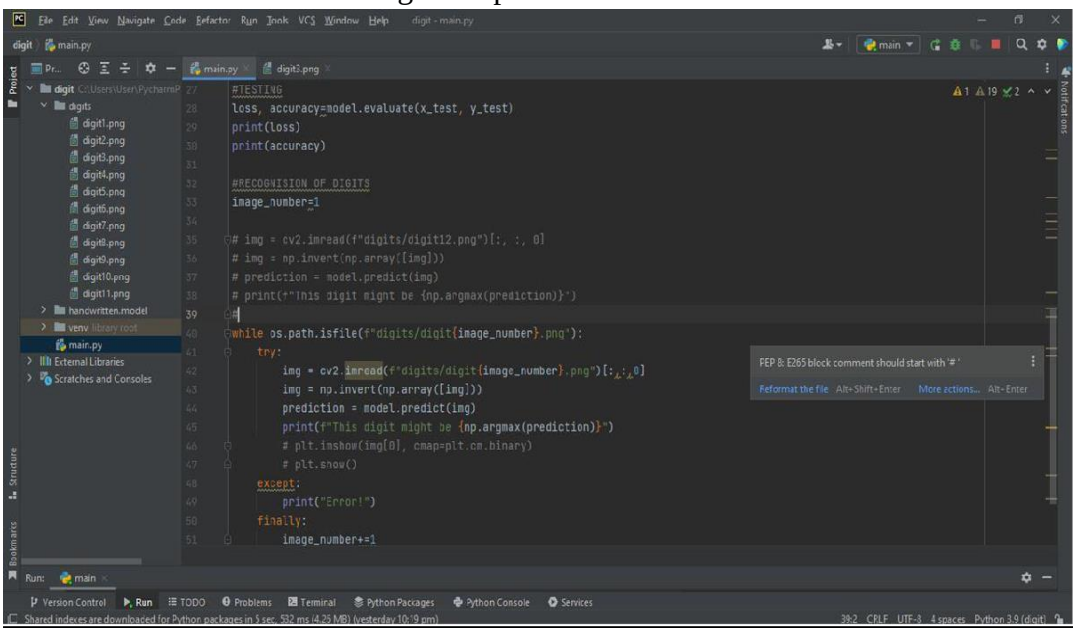


Fig.7 (a) Simulation framework of code

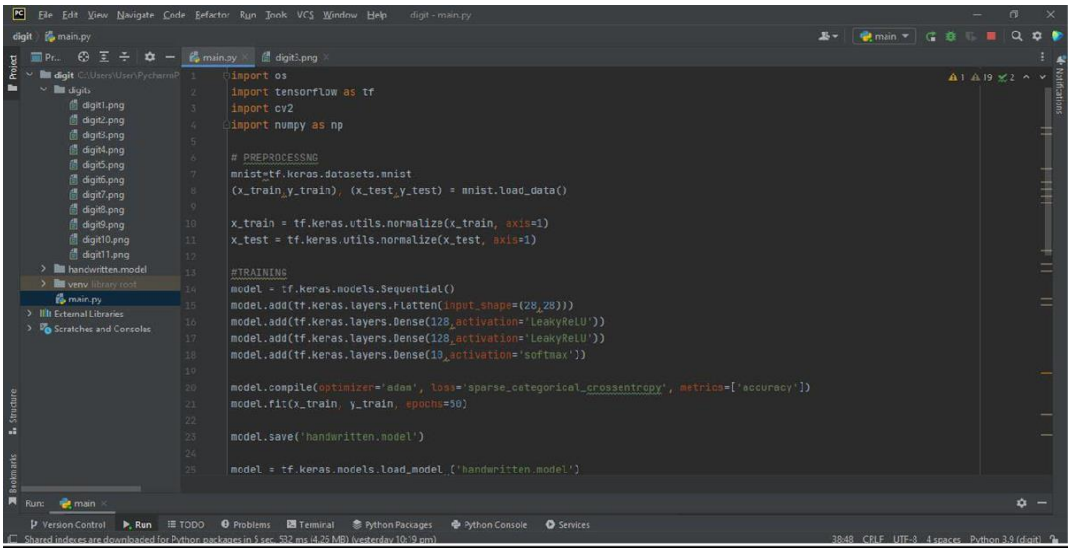


Fig.7(b) Simulation framework of code

Performance Analysis: We use the loss function and the metrics to get the loss and accuracy respectively. The loss function used for the model is **Cross- Entropy**. It tells us the amount of loss was obtained by training our model. The metrics used for our model is Accuracy. It tells ushow accurate our model is after training it.

The Accuracy should be close to 1 and the Loss should be close to 0. Then and only then, our model is close to perfection. If the loss is high compared to 0 and accuracy compared to 1, then the model is not perfect,and we need to train the model with different parameters and a greater number of epochs. The loss we got after training our model is around 3%The accuracy we got after the model is trained is around 97%, as shown in Fig.

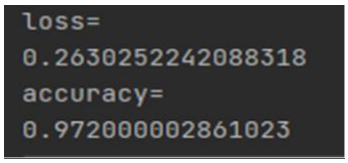


Fig. Performance Analysis- Loss and Accuracy of testing model

Conclusion: In this implementation, we created a neural network to recognize a handwritten digit using TensorFlow and Keras. using the **MNIST** dataset, our model can recognize handwritten digits which are being input to the model. The performance of CNN for the handwritten digit is accurate. The method works well, and the loss percentage is less with all those training sessions. The only difficulty here is the noise in the image, but with the training it has, it tries to achieve the best possible output.

The model's performance and accuracy were tested after training the model for 50 epochs. With the results given from this work we are more confident in finding other ways to make this better and to make it easier for complex data like converting handwritten paragraphs into text. Through this research work we understood all the mechanisms used to identify handwritten data. We understand the importance of hand recognition as it is easy for the user to write data on paper and use handwritten data recognition to convert it into text instead of the typing it on keyboard. Further it is recommended to implement on edge computing platforms like Raspberry Pi 4 system for actual usage.

3) LSTM With Keras?

A) **LSTM stands for "Long Short-Term Memory"**. An LSTM is actually a kind of RNN architecture. It is, theoretically, a more "sophisticated" Recurrent Neural Network. Instead of just having recurrence, it also has "gates" that regulate information flow through the unit as shown in the image. LSTMs were initially introduced to solve the vanishing gradient problem of RNNs. They are often used over traditional, "simple" recurrent neural networks because they are also more computationally efficient.

Creating a Simple LSTM with Keras: Using Keras and Tensorflow makes building neural networks much easier to build. It's much easier to build neural networks with these libraries than from scratch. The best reason to build a neural network from scratch is to understand how neural networks work. In practical situations, using a library like Tensorflow is the best approach. It's straight forward and simple to build a neural network with Tensorflow and Keras, let's take a look at how to use Keras to build our LSTM.

Importing the Right Modules: The first thing we need to do is import the right modules. For this example, we're going to be working with tensorflow. We don't technically *need* to do the bottom two imports, but they save us time when writing so when we add layers we don't need to type **"tf.keras.layers"**. but can rather just write layers.

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

Adding Layers to Your Keras LSTM Model: It's quite easy to build an LSTM in Keras. All that's really required for an LSTM neural network is that it has to have LSTM cells or at least one LSTM layer. If we add different types of layers and cells, we can still call our neural network an LSTM, but it would be more accurate to give it a mixed name.

To build an LSTM, the first thing we're going to do is initialize a Sequential model. Afterwards, we'll add an LSTM layer. This is what makes this an LSTM neural network. Then we'll add a batch normalization layer and a dense (fully connected) output layer. Next, we'll print it out to get an idea of what it looks like.

```
model = keras.Sequential()
model.add(layers.LSTM(64, input_shape=(None, 28)))
model.add(layers.BatchNormalization())
model.add(layers.Dense(10))
print(model.summary())
```

we'll see that the LSTM actually has WAY more parameters than the Simple RNN we built with Keras. The LSTM layer has four times the number of parameters as a simple RNN layer. This is because of the gates we talked about earlier.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 64)                23808

 batch_normalization (BatchN  (None, 64)               256
 ormalization)

 dense (Dense)               (None, 10)                650

=================================================================
Total params: 24,714
Trainable params: 24,586
Non-trainable params: 128
```

Keras LSTM parameters

Training and Testing our Keras LSTM on the MNIST Dataset: Now that we've built our LSTM let's see how it does on the MNIST digit dataset. This is the same dataset we tested the Keras RNN and the built from scratch Neural Network on. The MNIST dataset is a classic dataset to train and test neural networks on. It is a set of handwritten digits.

Load the MNIST dataset: The first thing we'll do is load up the MNIST dataset from Keras. We'll use the `load_data()` function from the MNIST dataset to load a pre-separated training and testing dataset. After loading the datasets, we'll normalize our training data by dividing by 255. This is due to the scale of 256 (0 to 255) for the image data. Finally, we'll set aside 10 test data points.

```
mnist = keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train/255.0, x_test/255.0
x_validate, y_validate = x_test[:-10], y_test[:-10]
x_test, y_test = x_test[-10:], y_test[-10:]
```

Compile the LSTM Neural Network: Now that we've created our LSTM and loaded up our data, let's compile our model. We have to compile (or build) or model before we can train or test it. In our model compilation we will specify the loss function, in this case Sparse Categorical Cross Entropy, our optimizer, stochastic gradient descent, and our metric(s), accuracy. We can specify multiple metrics, but we'll just go with accuracy for this example.

```
model.compile(
loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
optimizer="sgd",
metrics=["accuracy"],)
```

Train and Fit the Keras LSTM Model: Now that the model is compiled, let's train the model. To train the model in Keras, we just call the fit function. To use the fit function, we'll need to pass in the training data for x and y, the validation, the batch_size, and the epochs. For this example, we'll just train for 10 epochs.

```
model.fit (
x_train, y_train, validation_data=(x_test, y_test), batch_size=64, epochs=1
)
model.fit(    x_train, y_train, validation_data=(x_validate, y_validate), batch_size=64, epochs=10)
```

Test the Long Short Term Memory Keras Model: We've already trained and fit the model. The last thing to do is to test the model. We'll run our model and use it to predict the sample we set aside earlier. Then, we'll print out the sample and the correct label.

```
for i in range(10):
   result = tf.argmax(model.predict(tf.expand_dims(x_test[i], 0)), axis=1)
   print(result.numpy(), y_test[i])
```

We can see that after 1 epoch (you should really train for more, but once again this is just for an example) We can see that after 10 epochs we see a pretty good accuracy at about 96%.

```
None
Epoch 1/10
938/938 [==============================] - 12s 12ms/step - loss: 0.9447 - accuracy: 0.6961 - val_loss: 0.5518 - val_accuracy: 0.8202
Epoch 2/10
938/938 [==============================] - 11s 12ms/step - loss: 0.4125 - accuracy: 0.8745 - val_loss: 0.2904 - val_accuracy: 0.9115
Epoch 3/10
938/938 [==============================] - 11s 12ms/step - loss: 0.2651 - accuracy: 0.9205 - val_loss: 0.3372 - val_accuracy: 0.8841
Epoch 4/10
938/938 [==============================] - 12s 12ms/step - loss: 0.2074 - accuracy: 0.9372 - val_loss: 0.2361 - val_accuracy: 0.9242
Epoch 5/10
938/938 [==============================] - 11s 12ms/step - loss: 0.1748 - accuracy: 0.9465 - val_loss: 0.2044 - val_accuracy: 0.9344
Epoch 6/10
938/938 [==============================] - 15s 16ms/step - loss: 0.1540 - accuracy: 0.9528 - val_loss: 0.1846 - val_accuracy: 0.9374
Epoch 7/10
938/938 [==============================] - 15s 16ms/step - loss: 0.1364 - accuracy: 0.9585 - val_loss: 0.3032 - val_accuracy: 0.9008
Epoch 8/10
938/938 [==============================] - 13s 14ms/step - loss: 0.1238 - accuracy: 0.9635 - val_loss: 0.1490 - val_accuracy: 0.9501
Epoch 9/10
938/938 [==============================] - 11s 12ms/step - loss: 0.1130 - accuracy: 0.9656 - val_loss: 0.1043 - val_accuracy: 0.9662
Epoch 10/10
938/938 [==============================] - 13s 13ms/step - loss: 0.1036 - accuracy: 0.9682 - val_loss: 0.1113 - val_accuracy: 0.9635
[7] 7
[8] 8
[9] 9
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
```

Keras LSTM after 10 epochs

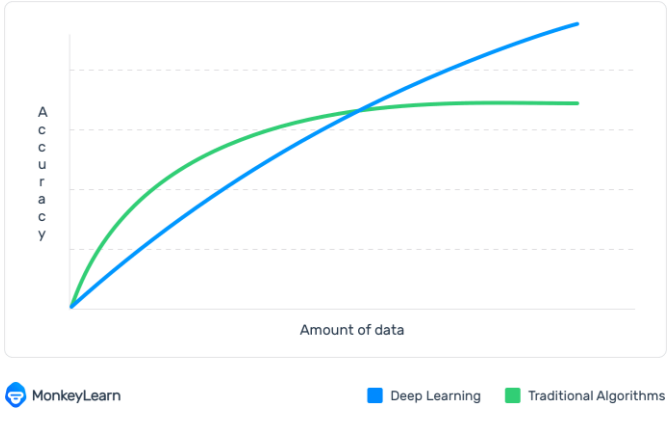
4) What Is Sentiment Analysis with Deep Learning?

A) **Sentiment analysis** is the classification of emotions (positive, negative, and neutral) within data using text analysis techniques. Harnessing the power of deep learning, sentiment analysis models can be trained to understand text beyond simple definitions, read for context, sarcasm, etc., and understand the actual mood and feeling of the writer. For example:

Based on word definitions, alone, the above tweet wouldn't give us much information. But when run through a well-trained sentiment analyzer, the program would understand that this is definitely a ***negative*** tweet. In order to exploit the full power of sentiment analysis tools, we can plug them into deep learning models. As we mentioned earlier, deep learning is a study within machine learning that uses **"artificial neural networks**" to process information much like the human brain does.

Deep learning is hierarchical machine learning that uses multiple algorithms in a progressive chain of events to solve complex problems and allows you to tackle massive amounts of data, accurately and with very little human interaction.

Deep learning and machine learning are sometimes used interchangeably. Deep learning is, indeed, machine learning, but it is more advanced. When basic machine learning makes a mistake, human input is required to correct it – to change the output and **"force"** the model to learn. In deep learning, however, the neural network can learn to correct itself through its advanced algorithm chain.



That said, the initial training of a deep learning model is extremely time-consuming and often requires millions of data points until it begins to learn on its own. To continue with the comparison to the human brain, think about how long it takes a child to build correct sentence structure or learn basic math. However, once they do, they can learn more advanced language or mathematics on their own because they have learned the essential rules and processes.

Once fully trained to effectively teach themselves, machine learning models can perform phenomenal feats. Text analysis, for example, uses **Natural Language processing (NLP**) to break down language and understand it much as a human would: subject, verb, object, etc. It's not until the computer has broken a sentence down, mathematically, can it move on to other analytical processes.

And, of course, it's much more complex than simply dissecting a sentence into subject, verb, object, and moving on. Successful NLP models have taken years to train. However, with the use of NLP, deep learning models can break sentences, paragraphs, and entire documents into individual opinion units:

I like the new update, but it seems really slow, and I can't get tech support on the phone.
[Opinion Unit 1] [Opinion Unit 2] [Opinion Unit 3]

Once broken into opinion units, the model could perform topic classification to organize each statement into predefined categories, like ***Usability (Opinion Unit 1), Functionality (Opinion Unit 2),* and *Support (Opinion Unit 3)*.**

From there, the deep learning model can perform sentiment analysis on each statement by topic: "like the new update" - ***Positive***; "seems really slow" - ***Negative***; "can't get tech support on the phone" - ***Negative***. Now we have sentiment analysis performed on our topic categories:

- Usability: Positive
- Functionality: Negative
- Support: Negative

Imagine this kind of deep learning analysis performed on thousands of customer reviews, social media posts, questionnaires, etc. You can get a broad overview or hundreds of detailed insights. There are **5 major steps** involved in the building a deep learning model for sentiment classification:

Step1: Get data.
Step 2: Generate embeddings
Step 3: Model architecture
Step 4: Model Parameters
Step 5: Train and test the model
Step 6: Run the model

Step1: Get data
Sourcing the labelled data for training a deep learning model is one of the most difficult parts of building a model. Fortunately we can use the Stanford sentiment treebank data for our purpose. The data set "dictionary.txt" consists of 239,233 lines of sentences with an index for each line. The index is used to match each of the sentences to a sentiment score in the file "labels.txt". The score ranges from 0 to 1, 0 being very negative and 1 being very positive.

The below code reads the dictionary.txt and **labels.txt** files, combines the score to each sentences. This code is found within ***train/utility_function.py***

```
def read_data(path):# read dictionary into dfdf_data_sentence = pd.read_table(path + 'dictionary.txt')df_data_sentence_processed = df_data_sentence['Phrase|Index'].str.split('|', expand=True)df_data_sentence_processed = df_data_sentence_processed.rename(columns={0: 'Phrase', 1: 'phrase_ids'})# read sentiment labels into dfdf_data_sentiment = pd.read_table(path + 'sentiment_labels.txt')df_data_sentiment_processed = df_data_sentiment['phrase ids|sentiment values'].str.split('|', expand=True)df_data_sentiment_processed = df_data_sentiment_processed.rename(columns={0: 'phrase_ids', 1: 'sentiment_values'})#combine data frames containing sentence and sentimentdf_processed_all = df_data_sentence_processed.merge(df_data_sentiment_processed, how='inner', on='phrase_ids'return df_processed_all
```

The data is split into 3 parts:

- train.csv : This is the main data which is used to train the model. This is 50% of the overall data.
- val.csv : This is a validation data set to be used to ensure the model does not overfit. This is 25% of the overall data.
- test.csv : This is used to test the accuracy of the model post training. This is 25% of the overall data.

Step 2: Generate embeddings

Prior to training this model we are going to convert each of the words into a word embedding. You can think of word embeddings as numerical representation of words to enable our model to learn. Word embeddings are vector representations that capture the context of the underlying words in relation to other words in the sentence. This transformation results in words having similar meaning being clustered closer together in the hyperplane and distinct words positioned further away in the hyperplane.

Convert each word into a word embedding: We are going to use a pre-trained word embedding model know as **GloVe**. For our model we are going to represent each word using a 100 dimension embedding. The detailed code for converting the data into word embedding is in within ***train/utility_function.py.*** This function basically replace each of the words by its respective embedding by performing a lookup from the **GloVe** pre-trained vectors. An illustration of the process is shown below, where each word is converted into an embedding and fed into a neural network.

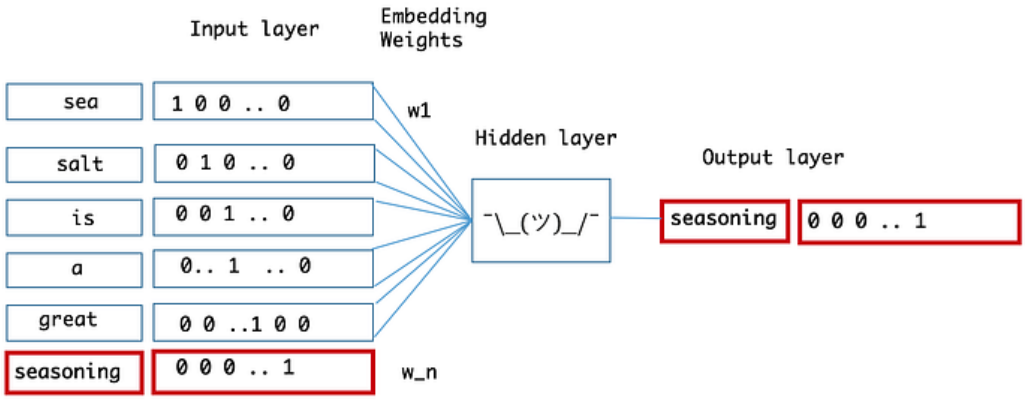


Fig: Converting Sentences to Embedding for a Neural Network

The below code is used to split the data into train, val and test sets. Also the corresponding embeddings for the data is stored in the weight_matrix variable.

```
def load_data_all(data_dir, all_data_path,pred_path, gloveFile, first_run, load_all):   numClasses = 10# Load embeddings for the filtered glove list
    if load_all == True:
        weight_matrix, word_idx = uf.load_embeddings(gloveFile)
    else:
        weight_matrix, word_idx = uf.load_embeddings(filtered_glove_path)    # create test, validation and trainng data
    all_data = uf.read_data(all_data_path)
    train_data, test_data, dev_data = uf.training_data_split(all_data, 0.8, data_dir)train_data = train_data.reset_index()
    dev_data = dev_data.reset_index()
    test_data = test_data.reset_index()    maxSeqLength, avg_words, sequence_length = uf.maxSeqLen(all_data)
  # load Training data matrix
    train_x = uf.tf_data_pipeline_nltk(train_data, word_idx, weight_matrix, maxSeqLength)
    test_x = uf.tf_data_pipeline_nltk(test_data, word_idx, weight_matrix, maxSeqLength)
    val_x = uf.tf_data_pipeline_nltk(dev_data, word_idx, weight_matrix, maxSeqLength)    # load labels data matrix
    train_y = uf.labels_matrix(train_data)
    val_y = uf.labels_matrix(dev_data)
    test_y = uf.labels_matrix(test_data)return train_x, train_y, test_x, test_y, val_x, val_y, weight_matrix
```

Step3: Model architecture

In order to train the model we are going to use a type of Recurrent Neural Network, know as LSTM (Long Short Term Memory). The main advantage of this network is that it is able to remember the sequence of past data i.e. words in our case in order to make a decision on the sentiment of the word.

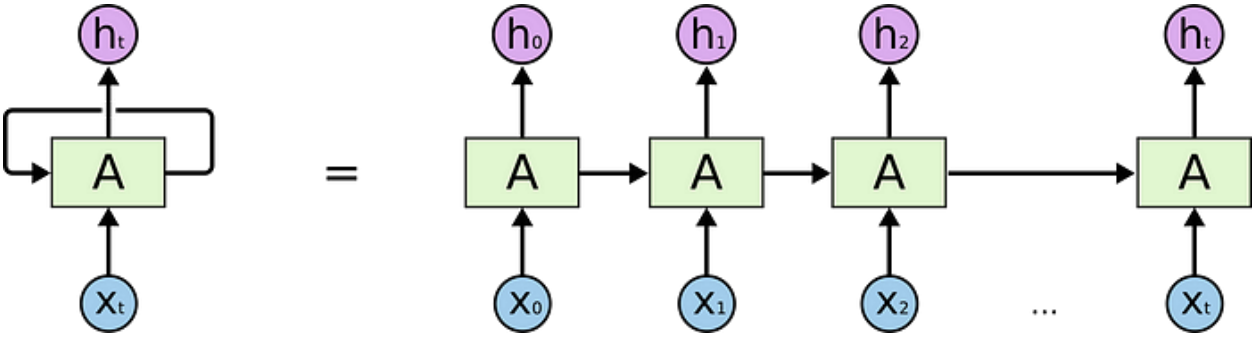


Fig: A RNN Network

As seen in the above picture it is basically a sequence of copies of the cells, where output of each cell is forwarded as input to the next. LSTM network are essentially the same but each cell architecture is a bit more complex. This complexity as seen below allows the each cells to decide which of the past information to remember and the ones to forget.

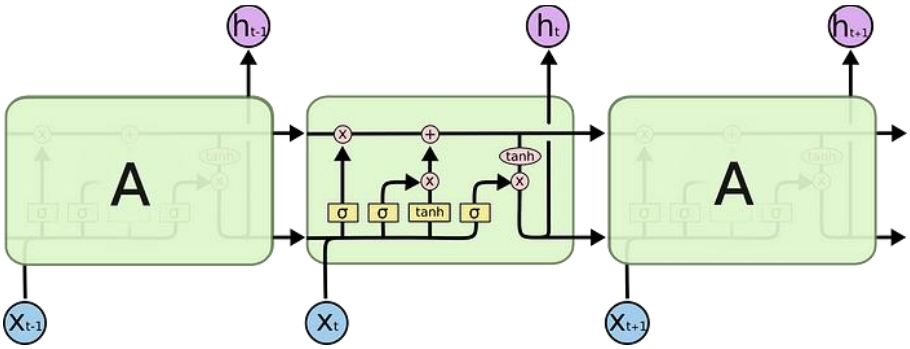


Fig: A LSTM Cell

We are going to create the network using Keras. Keras is built on tensorflow and can be used to build most types of deep learning models. We are going to specify the layers of the model as below. In order to estimate the parameters such as dropout, no of cells etc I have performed a grid search with different parameter values and chose the parameters with best performance.

Layers:

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 56, 100)	40000100
bidirectional_1 (Bidirection	(None, 256)	234496
dense_1 (Dense)	(None, 512)	131584
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 10)	5130

Total params: 40,371,310
Trainable params: 371,210
Non-trainable params: 40,000,100

Fig: Model Architecture

Layer 1: An embedding layer of a vector size of 100 and a max length of each sentence is set to 56.
Layer 2: 128 cell bi-directional LSTM layers, where the embedding data is fed to the network. We add a dropout of 0.2 this is used to prevent overfitting.
Layer 3: A 512 layer dense network which takes in the input from the LSTM layer. A Dropout of 0.5 is added here.
Layer 4: A 10 layer dense network with softmax activation, each class is used to represent a sentiment category, with class 1 representing sentiment score between 0.0 to 0.1 and class 10 representing a sentiment score between 0.9 to 1.

Code to create an LSTM model in Keras:

```
import os
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense
```

```
from keras.layers import Flatten
from keras.layers import LSTM
from keras.layers.embeddings import Embedding
from keras.layers import Bidirectional
from keras.preprocessing import sequence
from keras.layers import Dropout
from keras.models import model_from_json
from keras.models import load_model
def create_model_rnn(weight_matrix, max_words, EMBEDDING_DIM):# create the modelmodel = Sequential()
model.add(Embedding(len(weight_matrix), EMBEDDING_DIM, weights=[weight_matrix],
input_length=max_words, trainable=False))
model.add(Bidirectional(LSTM(128, dropout=0.2, recurrent_dropout=0.2)))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.50))
model.add(Dense(10, activation='softmax'))# Adam Optimiser
model.compile(loss='categorical_crossentropy',optimizer='adam', metrics=['accuracy'])return model
```

Step 4: Model Parameters:

Activation Function: I have used ReLU as the activation function. ReLU is a non-linear activation function, which helps complex relationships in the data to be captured by the model.

Optimiser: We use adam optimiser, which is an adaptive learning rate optimiser.

Loss function: We will train a network to output a probability over the 10 classes using **Cross-Entropy loss**, also called Softmax Loss. It is very useful for multi-class classification.

Step 5: Train and test the model
We start the training of the model by passing the train, validation and test data set into the function below:

```
def train_model(model,train_x, train_y, test_x, test_y, val_x, val_y, batch_size):# save the best model and early
stopping
saveBestModel = keras.callbacks.ModelCheckpoint('../best_weight_glove_bi_100d.hdf5', monitor='val_acc',
verbose=0, save_best_only=True, save_weights_only=False, mode='auto', period=1)earlyStopping =
keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0, patience=3, verbose=0, mode='auto')# Fit the
model
model.fit(train_x, train_y, batch_size=batch_size, epochs=25,validation_data=(val_x, val_y),
callbacks=[saveBestModel, earlyStopping])# Final evaluation of the model
score, acc = model.evaluate(test_x, test_y, batch_size=batch_size)
return model
```

We have run the training on a **batch size of 500** items at a time. As you increase the batch size the time for training would reduce but it will require additional computational capacity. Hence it is a trade-off between computation capacity and time for training.
The training is set to run for 25 epochs. One epoch would mean that the network has seen the entire training data once. As we increase the number of epochs there is a risk that the model will overfit to the training data.

Hence to prevent the model from overfitting I have enabled early stopping. Early stopping is a method that allows us to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out/validation dataset. The model on the test set of 10 class sentiment classification provides a result of **48.6%** accuracy. The accuracy will be much higher on a 2-class binary (positive or negative) data set.

Step 6: Run the model

Once the model is trained you can save the model in keras using the below code.

```
model.save_weights("/model/best_model.h5")
```

The next step is to use the trained model in real time to run predictions on new data. In order to do this, you will need to transform the input data to embeddings, similar to the way we treated our training data. The function ***live_test*** below performs the required pre-processing of the data and returns the result of the trained model. Here in order to ensure robustness of the results of the model I am taking the average top 3 sentiments bands from the model. This provides a better calibration for the model results.

```
def live_test(trained_model, data, word_idx):live_list = []
live_list_np = np.zeros((56,1))# split the sentence into its words and remove any punctuations.tokenizer =
RegexpTokenizer(r'\w+')
data_sample_list = tokenizer.tokenize(data)
labels = np.array(['1','2','3','4','5','6','7','8','9','10'], dtype = "int")# get index for the live stage
data_index = np.array([word_idx[word.lower()] if word.lower() in word_idx else 0 for word in data_sample_list])
data_index_np = np.array(data_index)# padded with zeros of length 56 i.e maximum length
padded_array = np.zeros(56)
padded_array[:data_index_np.shape[0]] = data_index_np
data_index_np_pad = padded_array.astype(int)
live_list.append(data_index_np_pad)
live_list_np = np.asarray(live_list)# get score from the model
score = trained_model.predict(live_list_np, batch_size=1, verbose=0)
single_score = np.round(np.argmax(score)/10, decimals=2) # maximum of the array i.e single band# weighted
score of top 3 bands
top_3_index = np.argsort(score)[0][-3:]
top_3_scores = score[0][top_3_index]
top_3_weights = top_3_scores/np.sum(top_3_scores)
single_score_dot = np.round(np.dot(top_3_index, top_3_weights)/10, decimals = 2)return single_score_dot,
single_score
```

As seen in the code below, you can specify the model path, sample data and the corresponding embeddings to the live_test function. It will return the sentiment of the sample data.

```
# Load the best model that is saved in previous step
weight_path = '/model/best_model.hdf5'
loaded_model = load_model(weight_path)# sample sentence
data_sample = "This blog is really interesting."
result = live_test(loaded_model,data_sample, word_idx)
```

Results

Let us compare the results of our deep learning model to the NLTK model by taking a sample.

LSTM Model: This sentence *"Great!! it is raining today!!"* contains negative context and our model is able to predict this as seen below. it gives it a score of 0.34.

```
data_sample = "Great!! it is raining today!!"
result = live_test(loaded_model,data_sample, word_idx) [  353    20    14 24150   373]
print (result) 0.34
```

Fig: LSTM Model

NLTK Model: The same sentence when analysed by the bi-gram NLTK model, scores it as being positive with a score of 0.74.

```
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer

data_sample = "Great!! it is raining today!!"
sid = SentimentIntensityAnalyzer()
ss = sid.polarity_scores(data_sample)
ss['compound'] 0.7405
```

Fig:NLTK Model

5) Image Dimensionality Reduction using Encoders LSTM with Keras?

A) Dimensionality Reduction: Dimensionality Reduction is the process of reducing the number of dimensions in the data either by excluding less useful features (Feature Selection) or transform the data into lower dimensions (Feature Extraction). Dimensionality reduction prevents overfitting. Overfitting is a phenomenon in which the model learns too well from the training dataset and fails to generalize well for unseen real-world data.

Types of Feature Selection for Dimensionality Reduction,

- Recursive Feature Elimination
- Genetic Feature Selection
- Sequential Forward Selection

Types of Feature Extraction for Dimensionality Reduction,

- Auto Encoders
- Principal Component Analysis (PCA)
- Linear Determinant Analysis (LDA)

AutoEncoder is an **unsupervised Artificial Neural Network** that attempts to encode the data by compressing it into the lower dimensions (bottleneck layer or code) and then decoding the data to reconstruct the original input. The bottleneck layer (or code) holds the compressed representation of the input data.

In AutoEncoder the number of output units must be equal to the number of input units since we're attempting to reconstruct the input data. AutoEncoders usually consist of an encoder and a decoder. The encoder encodes the provided data into a lower dimension which is the size of the bottleneck layer and the decoder decodes the compressed data into its original form.

The number of neurons in the layers of the encoder will be decreasing as we move on with further layers, whereas the number of neurons in the layers of the decoder will be increasing as we move on with further layers. There are three layers used in the encoder and decoder in the following example. The encoder contains 32, 16, and 7 units in each layer respectively and the decoder contains 7, 16, and 32 units in each layer respectively. The code size/ the number of neurons in bottle-neck must be less than the number of features in the data. Before feeding the data into the AutoEncoder the data must definitely be scaled between 0 and 1 using MinMaxScaler since we are going to use sigmoid activation function in the output layer which outputs values between 0 and 1.

When we are using AutoEncoders for dimensionality reduction we'll be extracting the bottleneck layer and use it to reduce the dimensions. This process can be viewed as **feature extraction**. The type of AutoEncoder that we're using is **Deep AutoEncoder**, where the encoder and the decoder are symmetrical. The Autoencoders don't necessarily have a symmetrical encoder and decoder but we can have the encoder and decoder non-symmetrical as well.

Dimensionality reduction using Keras Auto Encoder

- Prepare Data
- Design Auto Encoder
- Train Auto Encoder
- Use Encoder level from Auto Encoder
- Use Encoder to obtain reduced dimensionality data for train and test sets

```
import os
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

from numpy.random import seed
from sklearn.preprocessing import minmax_scale
from sklearn.model_selection import train_test_split
from keras.layers import Input, Dense
from keras.models import Model

print(os.listdir("../input"))
```

```
/opt/conda/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning: Conversion of the second argu
ment of issubdtype from `float` to `np.floating` is deprecated. In future, it will be treated as `np.float64 == np
.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
['train.csv', 'sample_submission.csv', 'test.csv']
```

Read train and test data

```
train = pd.read_csv('../input/train.csv')
test = pd.read_csv('../input/test.csv')
Dropping Target and ID's from train and test

target = train['target']
train_id = train['ID']
```

```
test_id = test['ID']

train.drop(['target'], axis=1, inplace=True)
train.drop(['ID'], axis=1, inplace=True)
test.drop(['ID'], axis=1, inplace=True)

print('Train data shape', train.shape)
print('Test data shape', test.shape)
```

```
Train data shape (4459, 4991)
Test data shape (49342, 4991)
```

Scaling Train and Test data for Neural Net

```
train_scaled = minmax_scale(train, axis = 0)
test_scaled = minmax_scale(test, axis = 0)
```

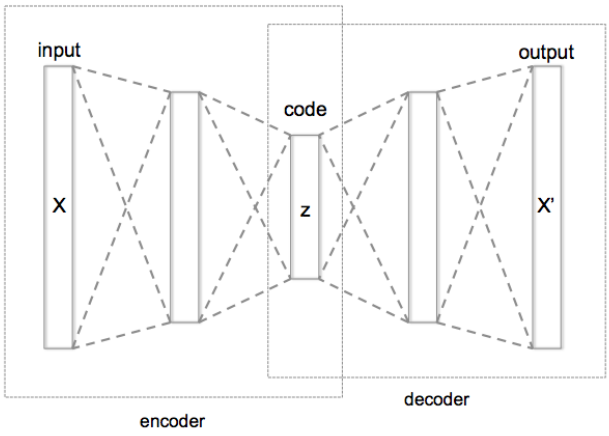
Design Auto Encoder

Auto Encoders are is a type of artificial neural network used to learn efficient data patterns in an unsupervised manner. An Auto Encoder ideally consists of an encoder and decoder. The Neural Network is designed compress data using the Encoding level. The Decoder will try to uncompress the data to the original dimension. To achieve this, the Neural net is trained using the Training data as the training features as well as target.

```
# Training a Typical Neural Net
model.fit(X_train, y_train)

# Training a Auto Encoder
model.fit(X_train, X_train)
```

These are typically used for dimensionality reduction use cases where there are more number of features.




```
# define the number of features
ncol = train_scaled.shape[1]
```

Split train data into train and validation 80:20 in ratio

```
X_train, X_test, Y_train, Y_test = train_test_split(train_scaled, target, train_size = 0.9, random_state = seed(
2017))
/opt/conda/lib/python3.6/site-packages/sklearn/model_selection/_split.py:2026: FutureWarning: From versio
n 0.21, test_size will always complement train_size unless both are specified.
  FutureWarning)

### Define the encoder dimension
encoding_dim = 200

input_dim = Input(shape = (ncol, ))

# Encoder Layers
encoded1 = Dense(3000, activation = 'relu')(input_dim)
encoded2 = Dense(2750, activation = 'relu')(encoded1)
encoded3 = Dense(2500, activation = 'relu')(encoded2)
encoded4 = Dense(2250, activation = 'relu')(encoded3)
encoded5 = Dense(2000, activation = 'relu')(encoded4)
encoded6 = Dense(1750, activation = 'relu')(encoded5)
encoded7 = Dense(1500, activation = 'relu')(encoded6)
encoded8 = Dense(1250, activation = 'relu')(encoded7)
encoded9 = Dense(1000, activation = 'relu')(encoded8)
encoded10 = Dense(750, activation = 'relu')(encoded9)
encoded11 = Dense(500, activation = 'relu')(encoded10)
encoded12 = Dense(250, activation = 'relu')(encoded11)
encoded13 = Dense(encoding_dim, activation = 'relu')(encoded12)

# Decoder Layers
decoded1 = Dense(250, activation = 'relu')(encoded13)
decoded2 = Dense(500, activation = 'relu')(decoded1)
decoded3 = Dense(750, activation = 'relu')(decoded2)
decoded4 = Dense(1000, activation = 'relu')(decoded3)
decoded5 = Dense(1250, activation = 'relu')(decoded4)
decoded6 = Dense(1500, activation = 'relu')(decoded5)
decoded7 = Dense(1750, activation = 'relu')(decoded6)
decoded8 = Dense(2000, activation = 'relu')(decoded7)
decoded9 = Dense(2250, activation = 'relu')(decoded8)
decoded10 = Dense(2500, activation = 'relu')(decoded9)
decoded11 = Dense(2750, activation = 'relu')(decoded10)
decoded12 = Dense(3000, activation = 'relu')(decoded11)
decoded13 = Dense(ncol, activation = 'sigmoid')(decoded12)

# Combine Encoder and Deocder layers
autoencoder = Model(inputs = input_dim, outputs = decoded13)

# Compile the Model
autoencoder.compile(optimizer = 'adadelta', loss = 'binary_crossentropy')
```

```
autoencoder.summary()
Total params: 101,590,191
Trainable params: 101,590,191
Non-trainable params: 0
```

Train Auto Encoder

```
autoencoder.fit(X_train, X_train, nb_epoch = 10, batch_size = 32, shuffle = False, validation_data = (X_test, X_test))
/opt/conda/lib/python3.6/site-packages/ipykernel_launcher.py:1: UserWarning: The `nb_epoch` argument in `fit` has been renamed `epochs`.
  """Entry point for launching an IPython kernel.
```

Train on 4013 samples, validate on 446 samples

```
Epoch 1/10
4013/4013 [==============================] - 31s 8ms/step - loss: 0.6861 - val_loss: 0.6668
Epoch 2/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.1167 - val_loss: 0.0160
Epoch 3/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0149 - val_loss: 0.0147
Epoch 4/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0138 - val_loss: 0.0140
Epoch 5/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0136 - val_loss: 0.0140
Epoch 6/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0135 - val_loss: 0.0139
Epoch 7/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0135 - val_loss: 0.0139
Epoch 8/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0135 - val_loss: 0.0139
Epoch 9/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0135 - val_loss: 0.0139
Epoch 10/10
4013/4013 [==============================] - 23s 6ms/step - loss: 0.0134 - val_loss: 0.0139

<keras.callbacks.History at 0x7fb8ba981d30>
```

Use Encoder level to reduce dimension of train and test data

```
encoder = Model(inputs = input_dim, outputs = encoded13)
encoded_input = Input(shape = (encoding_dim, ))
```

Predict the new train and test data using Encoder

```
encoded_train = pd.DataFrame(encoder.predict(train_scaled))
encoded_train = encoded_train.add_prefix('feature_')

encoded_test = pd.DataFrame(encoder.predict(test_scaled))
```

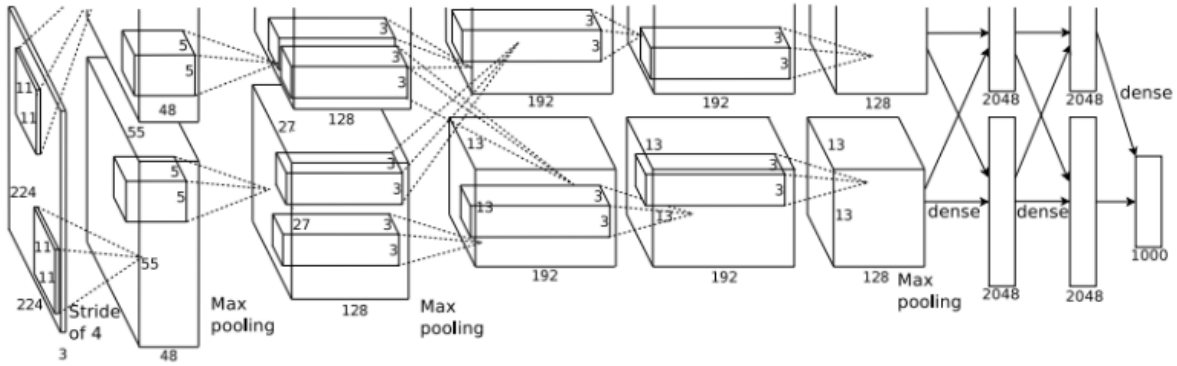
```
encoded_test = encoded_test.add_prefix('feature_')

Add target to train
encoded_train['target'] = target

print(encoded_train.shape)
encoded_train.head()
(4459, 201)
print(encoded_test.shape)
encoded_test.head()
(49342, 200)
In [17]:
linkcode
encoded_train.to_csv('train_encoded.csv', index=False)
encoded_test.to_csv('test_encoded.csv', index=False)
```

6) Briefly Describe about AlexNet and VGGNet?

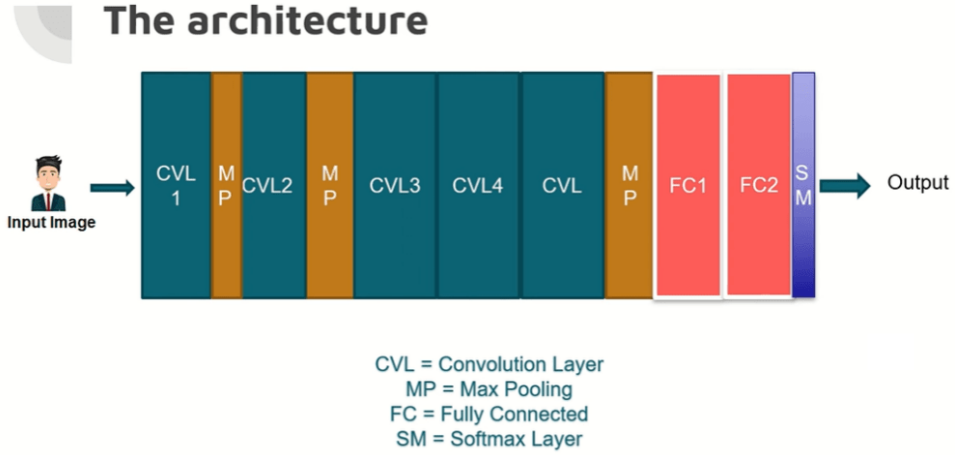
A) **AlexNet:** AlexNet is a deep learning architecture and represents a variation of the convolutional neural network. It was originally proposed by **Alex Krizhevsky** during his research, under the guidance of Geoffrey E. Hinton, a prominent figure in the field of deep learning research. In 2012, Alex Krizhevsky participated in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC2012) and utilized the AlexNet model, achieving an impressive top-5 error rate of 15.3%, surpassing the runner-up by more than 10.8 percentage points.



Architecture of AlexNet

- The AlexNet architecture consists of eight layers in total.
- The first five layers are convolutional layers.
- The sizes of the convolutional filters are 11×11, 5×5, 3×3, 3×3, and 3×3 for the respective convolutional layers.
- Some of the convolutional layers are followed by max-pooling layers, which help reduce spatial dimensions while retaining important features.
- The activation function used in the network is the Rectified Linear Unit (ReLU), known for its superior performance compared to sigmoid and tanh functions.
- After the convolutional layers, there are three fully connected layers.
- The network's parameters can be tuned based on the training performance.

- The AlexNet can be used with transfer learning, utilizing pre-trained weights on the ImageNet dataset to achieve exceptional performance. However, in this article, we will define the CNN without using pre-trained weights, following the proposed architecture.



Key Components of AlexNet Architecture

Point 1 – Convolutional Neural Network (CNN): AlexNet is a deep Convolutional Neural Network (CNN) architecture designed for image classification tasks. CNNs are specifically suited for visual recognition tasks, leveraging convolutional layers to learn features from images hierarchically.

Point 2 – Architecture: AlexNet consists of eight layers, with the first five being convolutional layers and the last three being fully connected layers. The convolutional layers are designed to extract relevant patterns and features from input images, while the fully connected layers perform the classification based on those features.

Point 3 – ReLU Activation: Rectified Linear Unit (ReLU) activation functions are used after each convolutional and fully connected layer. ReLU introduces non-linearity, enabling the network to model more complex relationships in the data.

Point 4 – Max Pooling: Max pooling layers are applied after certain convolutional layers to reduce spatial dimensions while retaining essential features. This downsampling process helps reduce computation and controls overfitting.

Point 5 – Local Response Normalization: Local Response Normalization (LRN) is implemented to enhance generalization by normalizing the output of a neuron relative to its neighbors. This creates a form of lateral inhibition, making the network more robust to variations in input data.

Point 6 – Dropout: AlexNet uses dropout regularization during training, where random neurons are dropped out during forward and backward passes. This technique prevents overfitting and improves the model's generalization performance.

Point 7 – Batch Normalization: Batch Normalization is applied to normalize the outputs of each layer within a mini-batch during training. It stabilizes and accelerates the training process, allowing for higher learning rates and deeper architectures.

Point 8 – Softmax Activation: The final layer of AlexNet uses the softmax activation function to convert the model's raw output into class probabilities. This allows the network to provide a probability distribution over the possible classes for each input image.

Point 9 – Training and Optimization: AlexNet is trained using stochastic gradient descent with momentum. The learning rate is adjusted during training, and data augmentation techniques are applied to increase the diversity of the training dataset.

Point 10 – ImageNet Competition: AlexNet achieved significant success when it participated in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. Its superior performance helped popularize deep learning and CNNs for image classification tasks.

AlexNet's emergence in 2012 marked a pivotal moment in the world of deep learning and computer vision. Its innovative architecture, depth, and novel techniques like ReLU activation, dropout, and data augmentation set a new standard for CNN models. With its impressive performance and contributions to the ImageNet competition, AlexNet laid the groundwork for future advancements in the field, paving the way for a new era of artificial intelligence applications.

VGGNet: VGG stands for Visual Geometry Group; it is a standard deep Convolutional Neural Network (CNN) architecture with multiple layers. The "deep" refers to the number of layers with VGG-16 or VGG-19 consisting of 16 and 19 convolutional layers. The VGG architecture is the basis of ground-breaking object recognition models. Developed as a deep neural network, the **VGGNet** also surpasses baselines on many tasks and datasets beyond ImageNet. Moreover, it is now still one of the most popular image recognition architectures.

The VGG model, or VGGNet, that supports 16 layers is also referred to as VGG16, which is a convolutional neural network model proposed by A. Zisserman and K. Simonyan from the University of Oxford. These researchers published their model in the research paper titled, "Very Deep Convolutional Networks for Large-Scale Image Recognition." The VGG16 model achieves almost 92.7% top-5 test accuracy in ImageNet.

ImageNet is a dataset consisting of more than 14 million images belonging to nearly 1000 classes. Moreover, it was one of the most popular models submitted to ILSVRC-2014. It replaces the large kernel-sized filters with several 3×3 kernel-sized filters one after the other, thereby making significant improvements over AlexNet.

The VGG16 model was trained using Nvidia Titan Black GPUs for multiple weeks. As mentioned above, the VGGNet-16 supports 16 layers and can classify images into 1000 object categories, including keyboard, animals, pencil, mouse, etc. Additionally, the model has an image input size of 224-by-224. Real-time object detection application built on Viso Suite.

The concept of the **VGG19** model (also VGGNet-19) is the same as the VGG16 except that it supports 19 layers. The "16" and "19" stand for the number of weight layers in the model (convolutional layers). This means that VGG19 has three more convolutional layers than VGG16.

VGG Architecture VGGNets are based on the most essential features of convolutional neural networks (CNN). The following graphic shows the basic concept of how a CNN works: The architecture of a Convolutional Neural Network: Image data is the input of the CNN; the model output provides prediction categories for input images.

The VGG network is constructed with very small convolutional filters. The VGG-16 consists of 13 convolutional layers and three fully connected layers.

The architecture of VGG:

Input: The VGGNet takes in an image input size of 224×224. For the ImageNet competition, the creators of the model cropped out the center 224×224 patch in each image to keep the input size of the image consistent.

Convolutional Layers: VGG's convolutional layers leverage a minimal receptive field, i.e., 3×3, the smallest possible size that still captures up/down and left/right. Moreover, there are also 1×1 convolution filters acting as a linear transformation of the input. This is followed by a ReLU unit, which is a huge innovation from AlexNet that reduces training time. ReLU stands for rectified linear unit activation function; it is a piecewise linear function that will output the input if positive; otherwise, the output is zero. The convolution stride is fixed at 1 pixel to keep the spatial resolution preserved after convolution (stride is the number of pixel shifts over the input matrix).

Hidden Layers: All the hidden layers in the VGG network use ReLU. VGG does not usually leverage Local Response Normalization (LRN) as it increases memory consumption and training time. Moreover, it makes no improvements to overall accuracy.

Fully-Connected Layers: The VGGNet has three fully connected layers. Out of the three layers, the first two have 4096 channels each, and the third has 1000 channels, 1 for each class. Fully Connected Layers VGG16 Architecture The number 16 in the name VGG refers to the fact that it is 16 layers deep neural network (VGGnet). This means that VGG16 is a pretty extensive network and has a total of around 138 million parameters. Even according to modern standards, it is a huge network.

However, VGGNet16 architecture's simplicity is what makes the network more appealing. Just by looking at its architecture, it can be said that it is quite uniform. There are a few convolution layers followed by a pooling layer that reduces the height and the width. If we look at the number of filters that we can use, around 64 filters are available that we can double to about 128 and then to 256 filters. In the last layers, we can use 512 filters.

The number of filters that we can use doubles on every step or through every stack of the convolution layer. This is a major principle used to design the architecture of the VGG16 network. One of the crucial downsides of the VGG16 network is that it is a huge network, which means that it takes more time to train its parameters. Because of its depth and number of fully connected layers, the VGG16 model is more than 533MB. This makes implementing a VGG network a time-consuming task.

The VGG16 model is used in several deep learning image classification problems, but smaller network architectures such as GoogLeNet and SqueezeNet are often preferable. In any case, the VGGNet is a great building block for learning purposes as it is straightforward to implement. Performance of VGG Models VGG16 highly surpasses the previous versions of models in the ILSVRC-2012 and ILSVRC-2013 competitions.

Moreover, the VGG16 result is competing for the classification task winner (GoogLeNet with 6.7% error) and considerably outperforms the ILSVRC-2013 winning submission Clarifai. It obtained 11.2% with external training data and around 11.7% without it. In terms of the single-net performance, the VGGNet-16 model achieves the best result with about 7.0% test error, thereby surpassing a single GoogLeNet by around 0.9%.