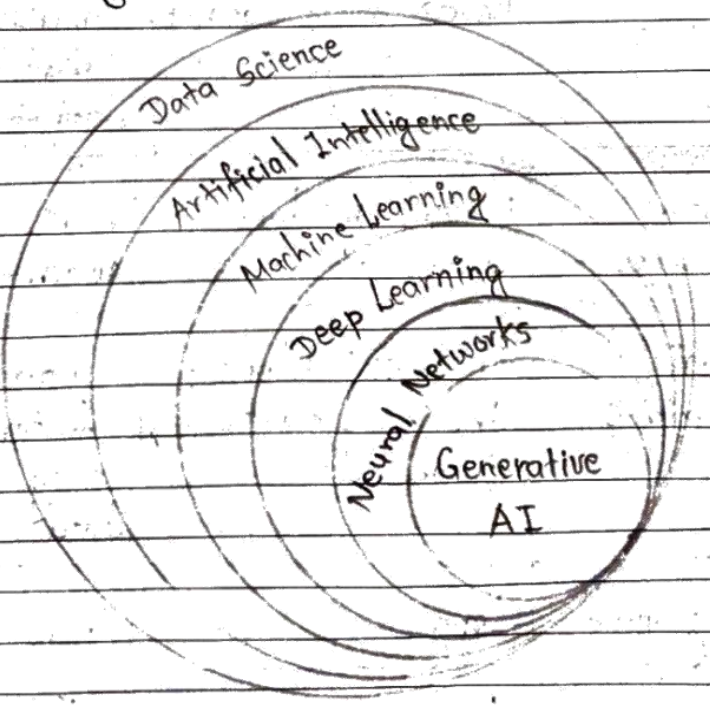
9. Deep Learning and AI

1. Deep Learning

- Deep learning is a collection of statistical techniques of machine learning feature hierarchies that are actually based on artificial neural networks.
- While the term deep learning is vague, it bases on the idea of mimicking the brain by building algorithms that resemble biological neurons' functionality in the brain.



Applications

- Computer Vision
- Natural - Language Processing (NLP)

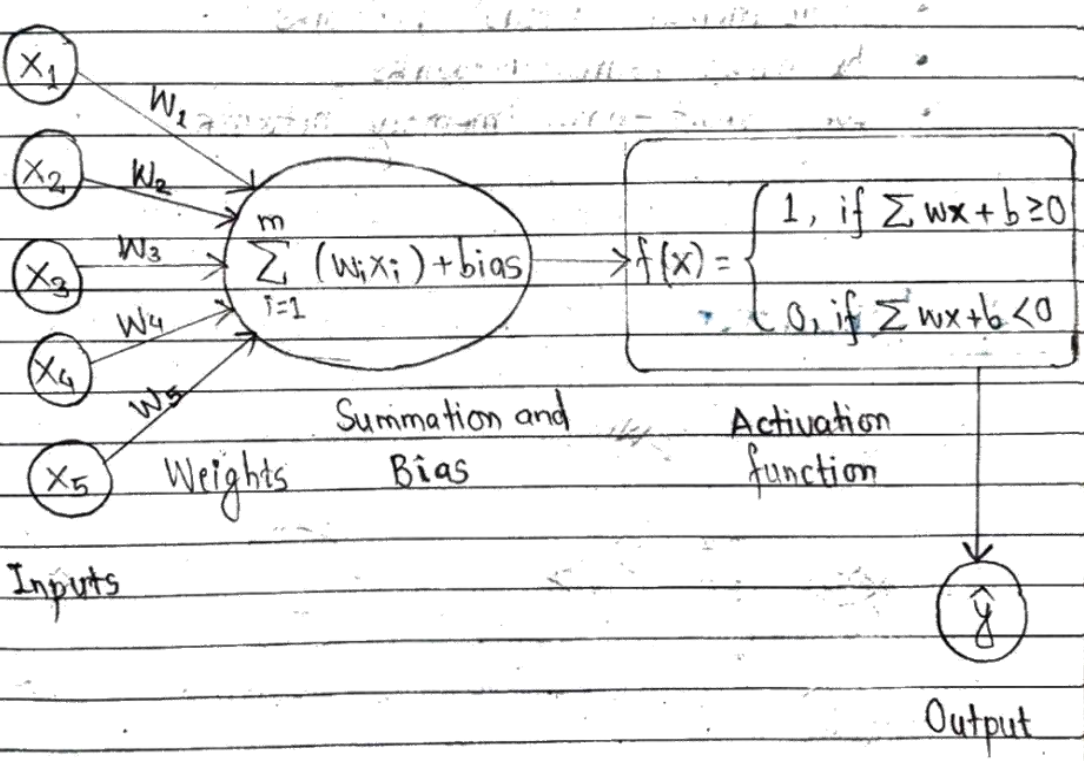
- Speech Recognization
- Health care
- Language translation
- Image detection
- Auto pilot cars

Deep Learning Vs Machine Learning

	Deep Learning	Machine Learning
Data	Needs a big dataset.	Performs well with a small to a medium dataset.
Hardware requirements	Requires machines with GPU.	Works with low-end machines.
Engineering peculiarities	Needs to understand the basic functionality of the data.	Understands the features and how they represent the data
Training time	Long	Short
Processing time	A few hours or weeks.	A few seconds or hours.
Number of algorithms	Few	Many

Data interpetation	Difficult	Some ML algorithms are easy and some are hard.

Neural Network

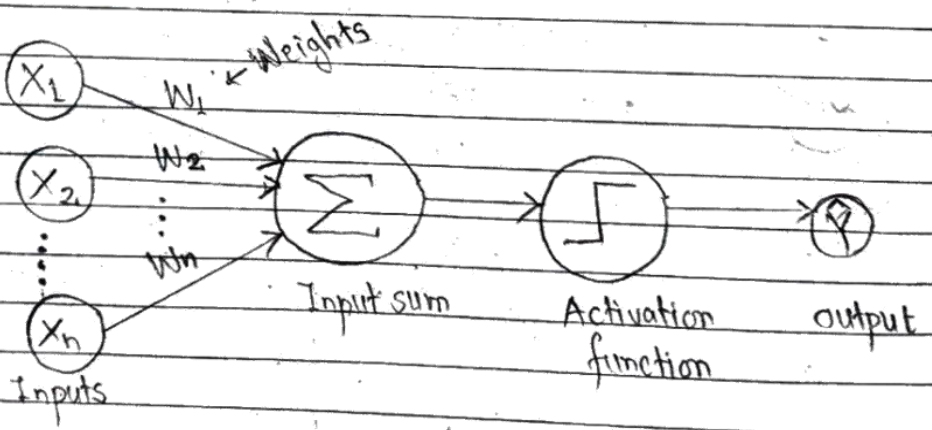


- A neural network is a type of machine learning model by the structure and function of human brain.
- It consists of interconnected units called neurons, which are organized into layers.
- **Structure :** Input layer → Hidden layer → Output layer
- **Working :** forward propagation and back - propagation

Types of Deep Learning Networks

- Perceptron (single layer)
- Feed forward networks
- Multi-layer perceptron (ANN)
- Radial based networks
- Convolutional neural networks
- Recurrent neural networks
- Long short-term memory network

2. Perceptron



A perceptron is one of the simplest type of artificial neural networks and serves as a fundamental building block for more complex neural network architectures

Working

1. Input features : The perceptron takes multiple input

features, each representing characteristics of the input data.

2. **Weights:** Each input feature is associated with a weight, which determines its influence on the output.

3. **Summation function:** The perceptron calculates the weighted sum of its input.

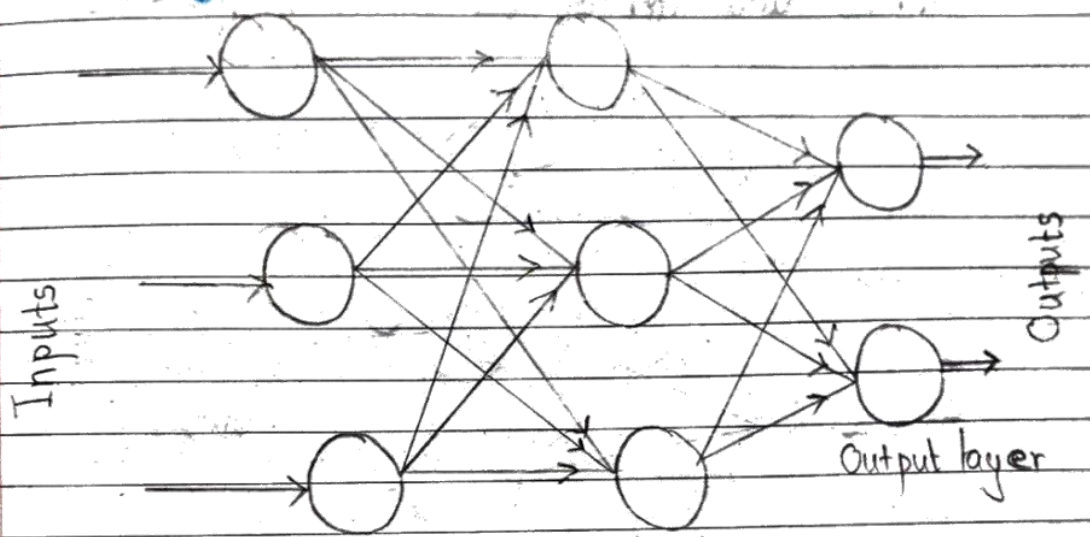
4. **Activation function:** The weighted sum is passed through an activation function (typically a step function) to produce the output. The output is usually binary (0 or 1)

Working Practically

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from mlxtend.plotting import plot_decision_regions

dataset = pd.read_csv("Suscription.csv")
dataset.head(3)

# Check for null values
dataset.isnull().sum() # No null values.

# Visualize data
plt.figure(figsize=(4,3))
```

```
sns.scatterplot(x="Age", y="Salary", data=dataset,
                hue="Purchase")
plt.show()

# Seperate input and output
x = dataset.iloc[:, :-1]
y = dataset["Purchase"]

# Train test split
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y,
        test_size = 0.2, random_state = 42)

# Perceptron model
from sklearn.linear_model import Perceptron

pr = Perceptron()
pr.fit(x_train, y_train)

pr.score(x_train, y_train), pr.score(x_test, y_test)

# Visualize decision region.
plot_decision_regions(x.to_numpy(), y.to_numpy(),
                      clf = pr)
plt.show()
"""This cannot provide high accuracy. So, instead of
this multilayer perceptron is used"""
```

3. Multilayer perceptron

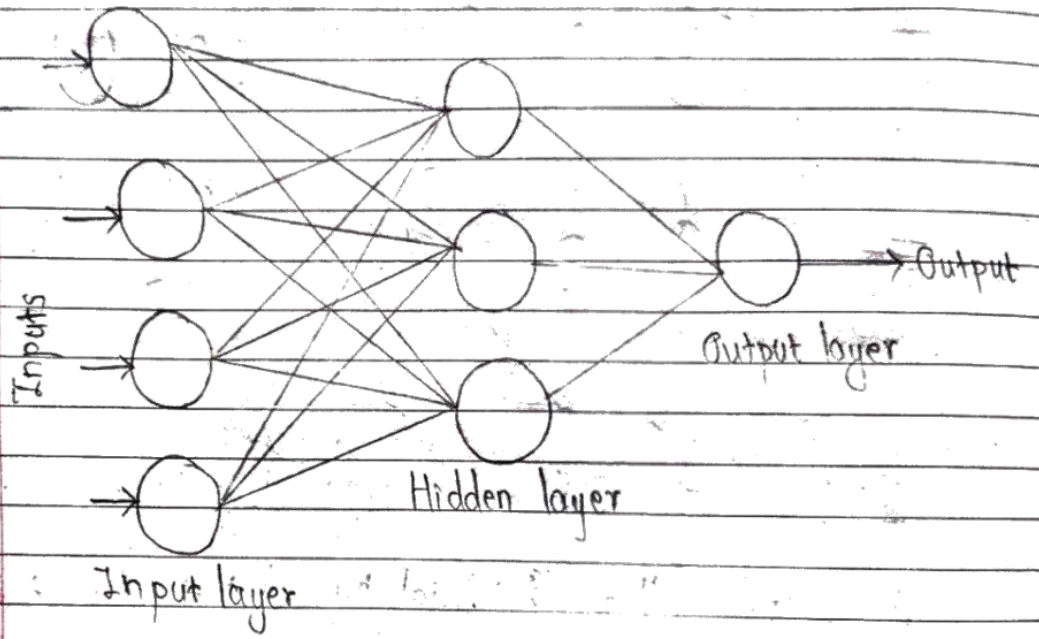


- It is also called Artificial Neural Networks (ANN)
- A multilayer perceptron (MLP) is a type of artificial neural network that consists of multiple layers of neurons, typically including an input layer, one or more hidden layers and one output layer.
- Working is based on forward propagation and back-propagation.

Forward Propagation

- Data is passed through the network from the input layer to the output layer.

- Each neuron computes a weighted sum of its inputs and applies an activation function.



Back - Propagation

- Backpropagation is a technique of ajustment of weights.
- The error between the predicted output and actual output is calculated. The error is propagated back through the network and weights are adjusted to minimize the error.
- Uses gradient descent technique (formula).

  i.e. $W_{new} = W_{old} - \lambda \left(\frac{dL}{dW} \right)$

  where λ is learning rate

Working Practically

```
import pandas as pd

dataset = pd.read_csv("ChurnModelling.csv")
dataset.head(3)

#Drop columns with object (string) type data
dataset = dataset.select_dtypes(exclude=["object"])
dataset.head(3)

#Check for null values
dataset.isnull().sum()  #No null values

#Seperate input and output data
input_data = dataset.iloc[:, :-1]
output_data = dataset.iloc[:, -1]

#Scaling the input data
from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
input_data = pd.DataFrame(ss.fit_transform(input_data),
                          columns=input_data.columns)

#Train test split
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(input_
    data, output_data, test_size=0.2, random_state=
    42)
```

```
# Create an artificial neural network
import tensorflow
import keras.layers import Dense
import keras.models import Sequential

ann = Sequential() # Object initialization

# Check the shape of training data
print(x_train.shape, y_train.shape) # Ensure shape are
# correct

# Build different layers.
ann.add(Dense(8, input_dim=x_train.shape[1],
              activation="relu")) # first hidden layer
# first parameter 8 is the number of nodes in the
# layer
ann.add(Dense(6, activation="relu")) # Second hidden
# layer
# Input dimension is needed only for first hidden layer
ann.add(Dense(4, activation="relu")) # Third hidden
# layer
ann.add(Dense(2, activation="relu")) # Fourth hidden
# layer
ann.add(Dense(1, activation="sigmoid")) # Output
# layer
# Sigmoid is used because our output is binary

# Compile the model
ann.compile(optimizer="adam", loss="binary_crossentropy",
            metrics=["accuracy"])
```

```
# Train the model
ann.fit (x_train, y_train, batch_size = 100, epochs = 50)
# Using batches helps in efficient computation and
# faster convergence.
# Batches allows the model to update it's weights
# more frequently than if it were using the entire
# dataset
# Multiple epochs are used to train the model,
# allowing it to learn and adjust its weights
# iteratively.

# Check the accuracy of our model.
from sklearn.metrics import accuracy_score

# We need prediction value of testing data.
ann.predict (x_test) # The data is not in binary form.

# Convert the data in binary form
prediction = ann.predict (x_test)
binary_prediction = []
for i in prediction:
        if i[0] > 0.5:
                binary_prediction.append (1)
        else:
                binary_prediction.append (0)
# View prediction
binary_prediction

accuracy_score (y_test, binary_prediction) * 100
# Accuracy of testing data
```

```
# For accuracy of training data
# We need prediction value of training data in
# binary format
prediction = ann.predict(x_train)
binary_prediction = []
for i in prediction:
    if i[0] > 0.5:
        binary_prediction.append(1)
    else:
        binary_prediction.append(0)
# View prediction
binary_prediction

accuracy_score(y_train, binary_prediction) * 100
# Accuracy of training data.

# To make the new predictions
import numpy as np

new_data = ss.fit_transform((([[1, 15634602, 619, 42
                2, 0.00, 1, 1, 1, 101348.88]])
new_prediction = ann.predict(new_data)
if new_prediction > 0.5:
    binary_form = 1
else:
    binary_form = 0
# View output
binary_form
```

Date
Page 210

4. Activation Functions

- An activation function decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations.

- The activation function is categorized into three main parts:

1. Binary step function
2. Linear activation function
3. Non-linear activation function.

1. Binary Step Function

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

- Mathematically, binary step function is:

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

Where,

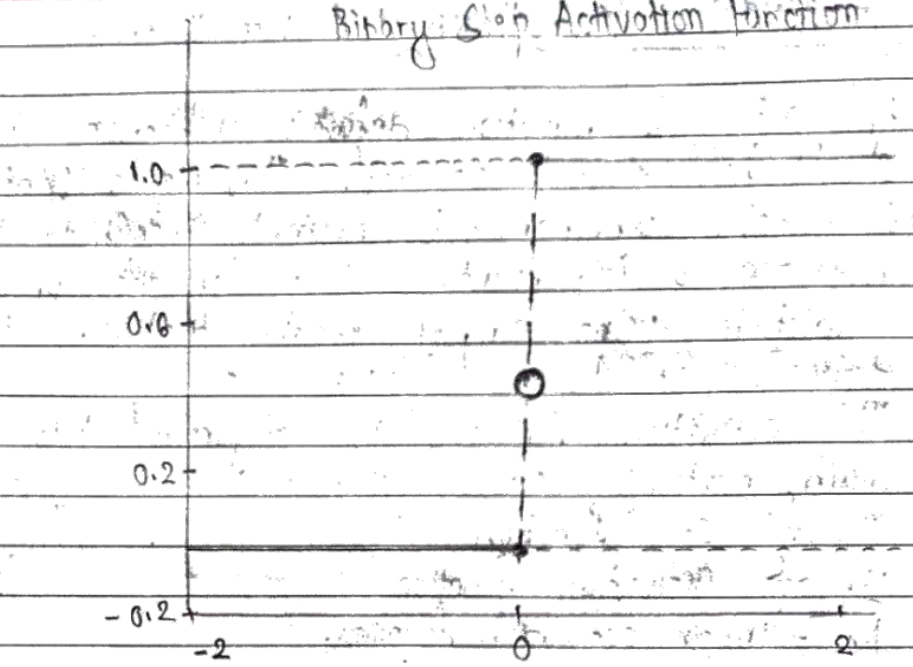
$$f(x) = \hat{y} = x_1 w_1 + x_2 w_2 + \dots + x_n w_n + b$$

where,

$\hat{y}$ is predicted output

$x_1, x_2, \dots x_n$ are input values

$w_1, w_2, \dots w_n$ are weightages.



2. Linear Activation Function

- The output of the functions is not restricted in between any range.
- Its range is specified from $-\infty$ to ∞
- Mathematically, linear activation function is:

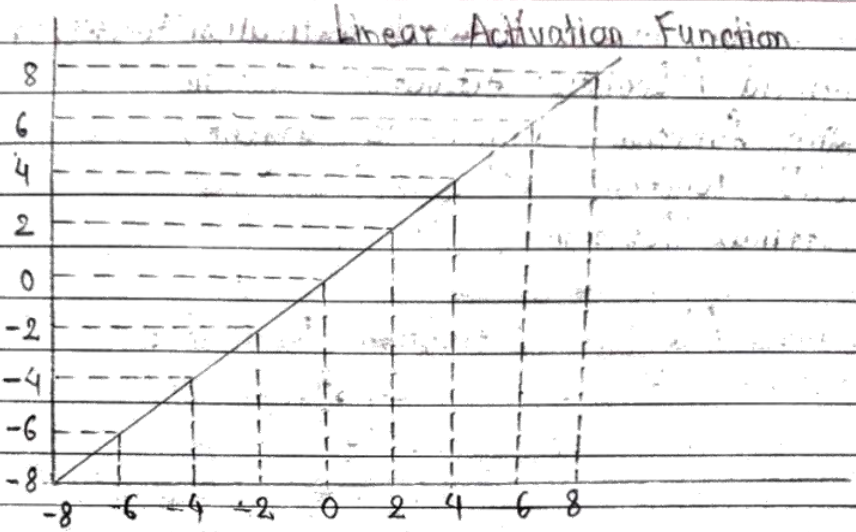
$$f(x) = x$$

i.e.

$$\hat{y} = x$$

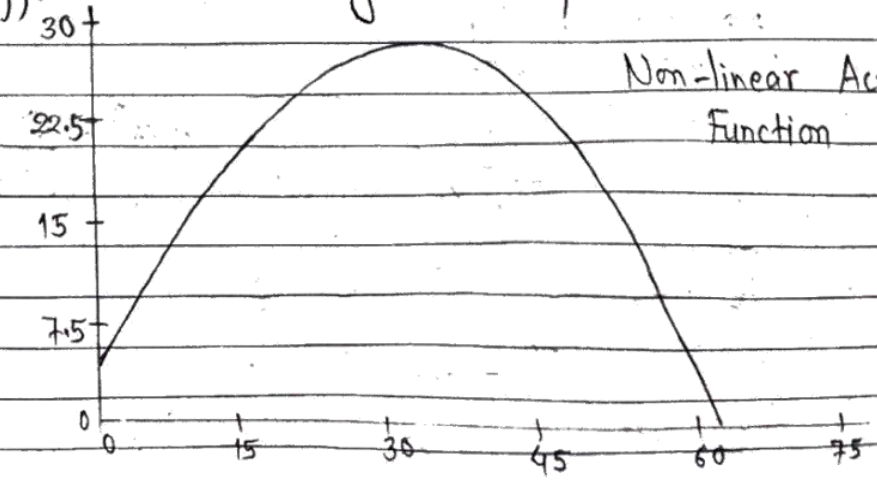
where,

$\hat{y}$ is predicted output and
x is input.



3. Non-Linear Activation Functions

- These are one of the mostly widely used activation functions.
- It helps the model in generalizing and adapting any sort of data in order to perform correct differentiation among the output.

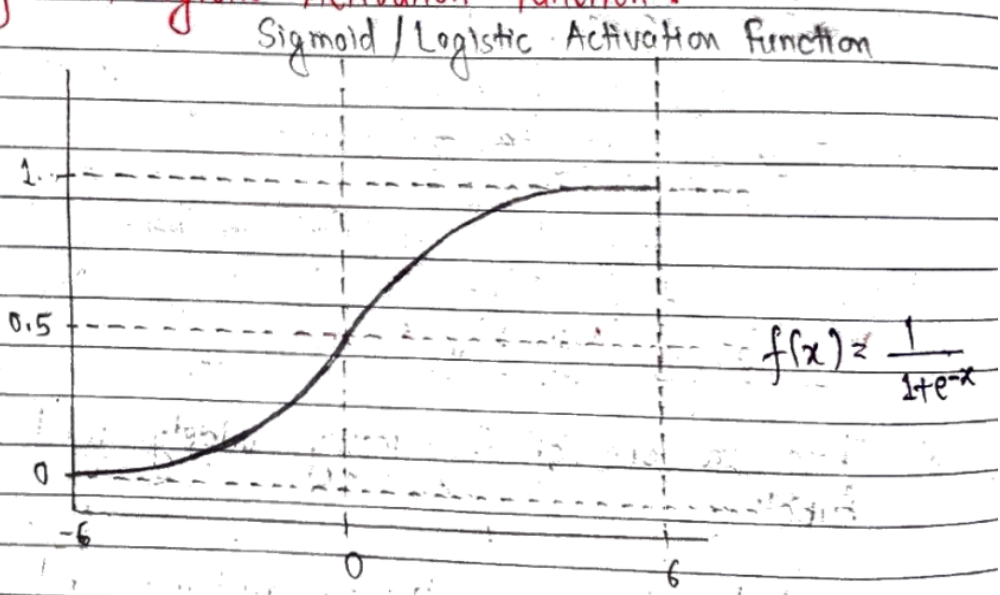


Date
Page 215

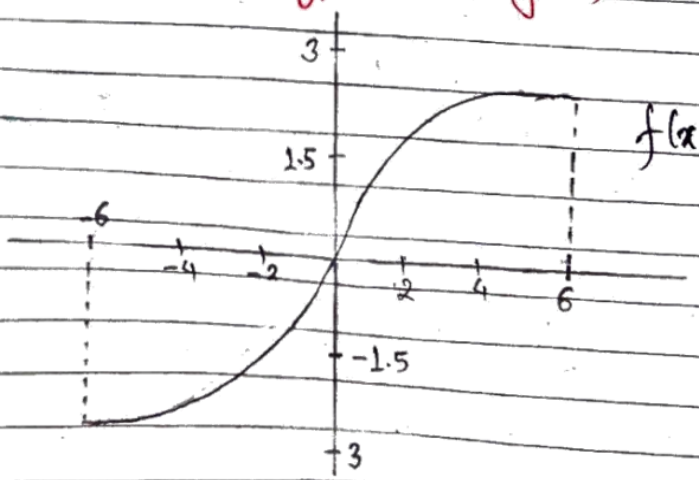
Non-linear Neural Network Activation Functions:

i. Sigmoid / Logistic Activation Function
ii. Tanh function (Hyperbolic Tangent)
iii. ReLU Function
iv. Softmax function

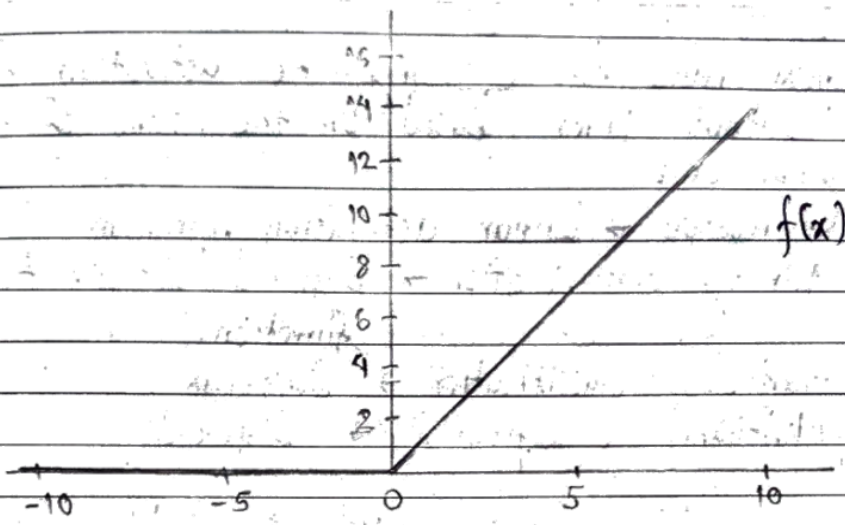
i. Sigmoid / Logistic Activation Function:



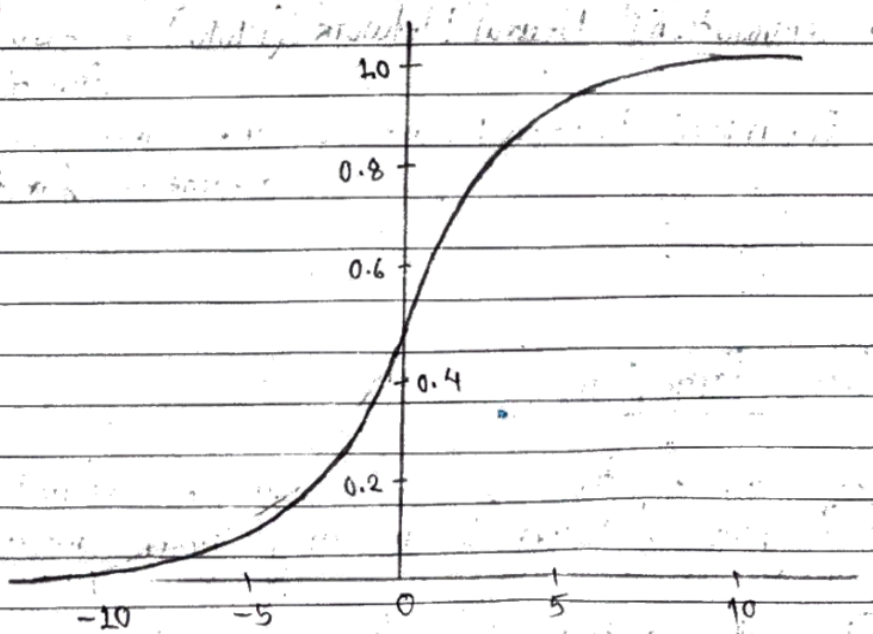
ii. Tanh Function (Hyperbolic Tangent):



iii. ReLU Function:



iv. Softmax Function:



$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Rules for Choosing Right Activation Function

- A few rules for choosing the activation function for output layer based on the type of prediction problem are:
  - Regression → linear activation function
  - Binary classification → Sigmoid / Logistic Activation function.
  - Multiclass classification → Softmax
  - Multilabel classification → Sigmoid
- The activation function used in hidden layers is typically choosen based on the type of neural network architecture:
  - Convolutional Neural Network (CNN): ReLU activation function
  - Recurrent Neural Network: Tanh and/or Sigmoid activation function

5. Loss Functions

- The loss function is a method of evaluating how well your algorithm is modeling your dataset.
- It is the mathematical function of the parameters of the machine learning algorithm.
- Remember that cost function is average of loss function of all data in dataset.

Types of Loss Functions

- **Regression :**
  - MSE (Mean Squared Error)
  - MAE (Mean Absolute Error)
  - Hubber loss
- **Classification :**
  - Binary cross-entropy / log loss
  - Categorical cross-entropy
- **Auto Encoder :**
  - KL Divergence.
- **GAN :**
  - Discriminator loss
  - Minmax GAN loss
- **Object Detection :**
  - Focal loss
- **Word Embeddings :**
  - Triplet loss

Regression Loss

1. **Mean Squared Error / Squared loss / L2 loss :**

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

2. Mean Absolute Error / L1 Loss:

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

3. Huber Loss:

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

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

where,

n = total number of data points
y = the actual value of the data point. Also known as true value.
$\hat{y}$ = 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.

Remember:

- MSE is used when there are no outliers.
- MAE is used when there are outliers but the function is not differentiable.
- Huber loss is used when you have around 30% outliers in dataset.

Classification Loss

1. Binary Cross Entropy / Log Loss :

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

Log loss works when we have binary output.
So,
When output = 0 ;

$$\text{log loss} = -\frac{1}{N}\sum_{i=1}^{N} \log(1-\hat{y}_i)$$

When output = 1 ;

$$\text{log loss} = -\frac{1}{N}\sum_{i=1}^{N} \log \hat{y}_i$$

2. Categorical Cross Entropy :

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

where,
k is number of classes in the data.

- If target column has one hot encode to classes like 001, 010, 100 then use categorical cross entropy.
- If the target column has numerical encoding to classes like 1, 2, 3, 4 .... N then use sparse categorical cross-entropy.

6. Optimizer in Neural Network

- Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and learning rate in order to reduce the losses.

- Used when we have lots of local minimums and a global minimum where algo can stuck on local minimum.

- Some Optimizers are:
  - Gradient descent
  - Stochastic gradient descent
  - Stochastic gradient descent with momentum
  - Mini-batch gradient descent
  - Adagrad
  - RMSProp
  - AdaDelta
  - Adam

7. Improve the Performance of Neural Network

- Hyperparameters Tuning:
  - No. of hidden layers
  - No. of neurons per layer
  - Learning rate
  - Optimizer
  - Batch size

- Activation function
- Epochs

↳ Solving Vanishing / Exploding Gradient :
- Optimizer manipulation
- Changing initialization weight
- Batch normalization.

↳ Use Big Data :
- Use transform learning to increase size of data.

↳ Overcome Slow Training :
- Can be achieved by hyperparameter tuning.
- Solve vanishing / exploding gradient.

↳ Control Overfitting :
- L1 and L2 regularization
- Early stopping
- Batch normalization.

8. Overfitting in Deep Learning

↳ Overfitting is a common explanation for the poor performance of a predictive model.

↳ Overfitting refers to an unwanted behaviour of a machine learning algorithm used for predictive modeling.

Minimization of Overfitting

- Cross validation
- Train with more data
- Remove features
- Early stopping
- Regularization (L1 and L2)
- Ensembling
- Hyperparameter tuning

Early Stopping

↳ Early stopping is a regularization technique in machine learning to prevent overfiting.

↳ It involves monitoring the models performance on a validation set during training and stopping the training process when the performance on the validation set starts to degrade.

Regularization

↳ Regularization is a technique used in machine learning to prevent overfitting by adding a penalty to the loss function.

↳ This penalty discourage the model from becoming too complex and helps it generalize better to new, unseen dataset.

Date
Page 222

Working Practically

```
import pandas as pd
import matplotlib.pyplot as plt

dataset = pd.read_csv("ChurnModeling.csv")
dataset.head(3)

# Drop columns with object (string) type data
dataset = dataset.select_dtypes(exclude=["object"])
dataset.head(3)

# Check for null values
dataset.isnull().sum()  # No null values

# Seperate input and output data
input_data = dataset.iloc[:, :-1]
output_data = dataset.iloc[:, -1]

# Scaling the input data
from sklearn.preprocessing import StandardScalar

ss = StandardScalar()
input_data = pd.DataFrame(ss.fit_transform(input_data),
                          columns = input_data.columns)

# Train test split
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(input_
    data, output_data, test_size = 0.2, random_state = 12)
```

```
# Create an artificial neural network
import tensorflow
from keras.layers import Dense
from keras.models import Sequential

ann = Sequential() # Object initialization

# Check the shape of training data
print(x_train.shape, y_train.shape) # Ensure shapes are
# Correct

# Build different layers
ann.add(Dense(8, input_dim = x_train.shape[1],
              activation = "relu")) # first hidden layer
# First parameter 8 is the number of nodes in the
# layer
ann.add(Dense(6, activation = "relu") # Second hidden
# layer
ann.add(Dense(4, activation = "relu") # Third hidden
# layer
ann.add(Dense(2, activation = "relu") # Fourth hidden
# layer.
ann.add(Dense(1, activation = "sigmoid")) # Output
# layer

# Compile the model
ann.compile(optimizer = "adam", loss = "binary_crossentropy"
            , metrics = ["accuracy"])

# Train the model
ann.fit(x_train, y_train, batch_size = 100, epochs = 50,
```

validation_data = (x_test, y_test)
# Validation data is for testing accuracy

View training history
ann.history.history
The history attribute of a keras model contain
the training history, including metrics like loss and
accuracy for each epoch.

# To get all the keys of history
ann.history.history.keys()

To see specific values of keys
ann.history.history["accuracy"] # For accuracy

# To get training and testing accuracy of each epoch
train_accuracy = ann.history.history["accuracy"]
test_accuracy = ann.history.history["val_accuracy"]

# Visualize the accuracy
# Get the lengths of accuracies for plotting
len(test_accuracy), len(test_accuracy)

Draw graph
plt.plot([i+1 for i in range(len(test_accuracy))],
test_accuracy, color="blue")
plt.plot([i+1 for i in range(len(train_accuracy))],
train_accuracy, color="red")
plt.show()

Check the accuracy of our model

```
from sklearn.metrics import accuracy_score

# Predictions of x_train
predict_x_train = ann.predict(x_train)
x_train_prediction = []
for i in predict_x_train:
    if i[0] > 0.5:
        x_train_predictions.append(1)
    else:
        x_train_predictions.append(0)
# View predictions
x_train_predictions

# Predictions of x_test
predict_x_test = ann.predict(x_test)
x_test_predictions = []
for i in predict_x_test:
    if i[0] > 0.5:
        x_test_predictions.append(1)
    else:
        x_test_predictions.append(0)
# View predictions
x_test_predictions

# Get accuracy
train_data_accuracy = accuracy_score(y_train, x_train_predictions) * 100
test_data_accuracy = accuracy_score(y_test, y_test_predictions) * 100
train_data_accuracy, test_data_accuracy
```

""" If the training accuracy is more than testing accuracy, the model is overfitted.
To avoid overfitting we use various techniques like Early Stopping and Regularization.
These methods help in minimizing the gaps between the accuracies. """

Early Stopping:

```
from keras.callbacks import EarlyStopping

# Train the model
ann.fit(x_train, y_train, batch_size=100, epochs=50,
        validation_data=(x_test, y_test), callbacks=
            EarlyStopping())
# EarlyStopping stops training if epoch is going towards
# overfitting

# To get training and testing accuracy
train_accuracy = ann.history.history["accuracy"]
test_accuracy = ann.history.history["val_accuracy"]

# Visualize the accuracy
plt.plot([i+1 for i in range(len(test_accuracy))],
         test_accuracy, color="blue")
plt.plot([i+1 for i in range(len(train_accuracy))],
         train_accuracy, color="red")
plt.show()
```

Regularization:

```
from keras.regularizers import L2

# Build different layers
ann.add(Dense(8, input_dim = x_train.shape[1],
              activation = "relu",
              kernel_regularizer = L2(l2 = 0.01)))
# kernel_regularizer stops the training if it is going
# towards overfitting using given regularization
# technique.
ann.add(Dense(6, activation = "relu", kernel_regularizer =
                L2(l2 = 0.01)))
# kernel_regularizer can be used in different layers
# differently if you want or you can simply skip as
# below
ann.add(Dense(4, activation = "relu"))
ann.add(Dense(2, activation = "relu"))
ann.add(Dense(1, activation = "sigmoid"))

# Compile the model
ann.compile(optimizer = "adam", loss = "binary_
            crossentropy", metrics = ["accuracy"])

# Train the model
ann.fit(x_train, y_train, batch_size = 100, epochs = 50,
        validation_data = (x_test, y_test))

# To get training and testing accuracy
train_accuracy = ann.history.history["accuracy"].
test_accuracy = ann.history.history["val_accuracy"]
train_accuracy, test_accuracy
```

```
#Visualize the accuracy
plt.plot([i+1 for i in range(len(test_accuracy))],
         test_accuracy, color = "blue")
plt.plot([i+1 for i in range(len(train_accuracy))],
         train_accuracy, color = "red")
plt.show()
```

9. Batch Normalization

- Batch normalization is a technique used to improve the training of deep neuron networks by normalizing the inputs of each layer
- This helps to stabilize and accelerate the training process.

Working Practically

```
import pandas as pd
import matplotlib.pyplot as plt

dataset = pd.read_csv("ChurnModelling.csv")
dataset.head(3)

#Drop columns with object (string) type data
dataset = dataset.select_dtypes(exclude = ["object"])
dataset.head(3)
```

```
# Check for null values
dataset.isnull().sum() # No null values

# Seperate input and output data
input_data = dataset.iloc[:, :-1]
output_data = dataset.iloc[:, -1]

# Scaling the input data
from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
input_data = pd.DataFrame(ss.fit_transform(input_data),
                          columns = input_data.columns)

# Train test split
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(
        input_data, output_data, test_size = 0.2,
        random_state = 42)

# Create an artificial neural network
import tensorflow
from keras.layers import Dense
from keras.models import Sequential
from keras.callbacks import EarlyStopping
from keras.regularizers import L2

ann = Sequential() # Object initialization

# Build different layers.
```

Date
Page 230

```
ann.add(Dense(8, input_dim=x_train.shape[1],
              activation="relu", kernel_regularizer=
              L2(l2=0.01))) # first hidden
# layer.
ann.add(Dense(6, activation="relu", kernel_regularizer
              =L2(l2=0.01)))
ann.add(Dense(4, activation="relu", kernel_regularizer=
              L2(l2=0.01)))
ann.add(Dense(2, activation="relu", kernel_regularizer=
              L2(l2=0.01)))
ann.add(Dense(1, activation="sigmoid"))

# Compile the model
ann.compile(optimizer="adam", loss="binary_crossentropy",
            metrics=["accuracy"])

# Train the model
ann.fit(x_train, y_train, batch_size=100, epochs=50,
        validation_data=(x_test, y_test), callbacks=
        EarlyStopping())

# Training and testing accuracy of each epoch
train_accuracy = ann.history.history["accuracy"]
test_accuracy = ann.history.history["val_accuracy"]
train_accuracy, test_accuracy

# Visualize the accuracy
plt.plot([i+1 for i in range(len(test_accuracy))],
         test_accuracy, color="blue")
plt.plot([i+1 for i in range(len(train_accuracy))],
         train_accuracy, color="red")
```

```
plt.show()

# Check the accuracy of our model
from sklearn.metrics import accuracy_score

# Predictions of x_train
predict_x_train = ann.predict(x_train)
x_train_predictions = []
for i in predict_x_train:
        if i[0] > 0.5:
                x_train_predictions.append(1)
        else:
                x_train_predictions.append(0)
# View predictions
x_train_predictions

# Predictions of x_test
predict_x_test = ann.predict(x_test)
x_test_predictions = []
for i in predict_x_test:
        if i[0] > 0.5:
                x_test_predictions.append(1)
        else:
                x_test_predictions.append(0)
# View predictions
x_test_predictions

# Get accuracy
train_data_accuracy = accuracy_score(y_train, x_train_
                                     predictions)* 100
test_data_accuracy = accuracy_score(y_test, x_test_
```

predictions)* 100
train-data-accuracy, test-data-accuracy

""" If the training accuracy is more than testing accuracy, the model is overfitted.
To avoid overfitting we use Batch Normalization to minimize the gaps between the accuracies. """

Batch Normalization:

```
from keras.layers import BatchNormalization

# Build different layers
ann.add (Dense (8, input_dim = x_train.shape[1],
                activation = "relu",
                kernel_regularization = L2 (l2 = 0.01)))
ann.add (BatchNormalization ()) # Applying batch
# normalization on first hidden layer output
ann.add (Dense (6, activation = "relu",
                kernel_regularization = L2 (l2 = 0.01)))
ann.add (BatchNormalization ()) # Applying batch
# normalization on second hidden layer output.
ann.add (Dense (4, activation = "relu",
                kernel_regularization = L2 (l2 = 0.01)))
ann.add (BatchNormalization ()) # Applying batch
# normalization on third hidden layer output.
ann.add (Dense (2, activation = "relu",
                kernel_regularization = L2 (l2 = 0.01)))
ann.add (BatchNormalization ()) # Applying batch
# normalization on fourth hidden layer.
ann.add (Dense (1, activation = "sigmoid"))
```

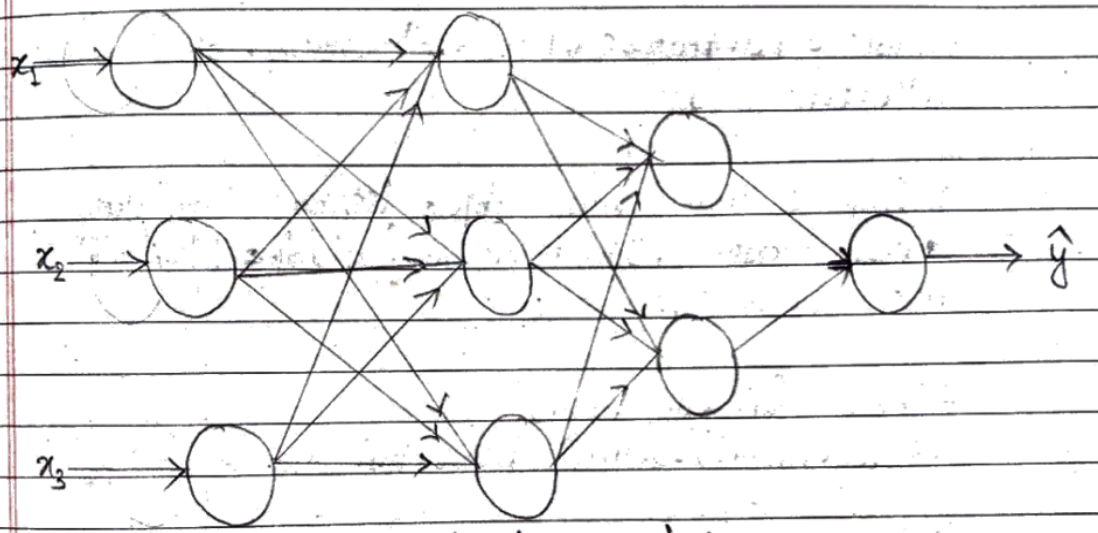
```
# Batch normalization is not required in output
# layer

# Compile the model
ann.compile(optimizer="adam", loss="binary-
            crossentropy", metrics=["accuracy"])

# Train the model
ann.fit(x_train, y_train, batch_size=100, epochs=50,
        validation_data=(x_test, y_test),
        callbacks=EarlyStopping())

# Training and testing accuracy of each epoch
train_accuracy = ann.history.history["accuracy"]
test_accuracy = ann.history.history["val_accuration"]
train_accuracy, test_accuracy

# Visualize the accuracy
plt.plot([i+1 for i in range(len(test_accuracy))],
         test_accuracy, color="blue")
plt.plot([i+1 for i in range(len(train_accuracy))],
         train_accuracy, color="red")
plt.show()
```

10. Dropout Layer

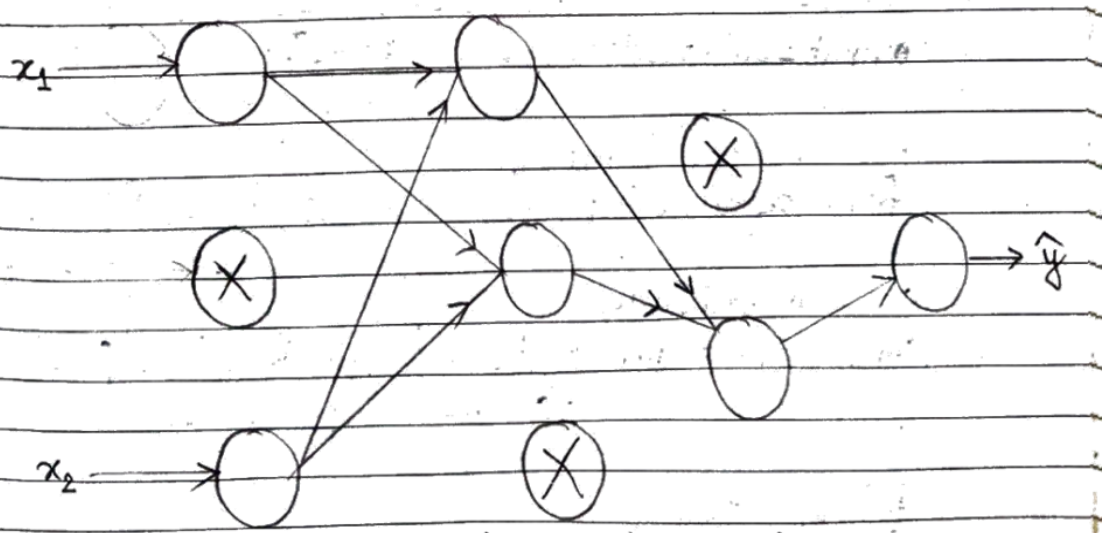
↳ All the forward and backward connections with a dropped node are temporarily removed, thus creating a new network architecture out of

the parent network.

6) The nodes are dropped by a dropout probability of p.



(a) Standard Neural Net



(b) After applying dropout layer

Working Practically

```
import pandas as pd
import matplotlib.pyplot as plt

dataset = pd.read_csv("ChurnModeling.csv")
dataset.head(3)

#Drop columns with object (string) type data
dataset = dataset.select_dtypes(exclude=["object"])
dataset.head(3)

#Check for null values
dataset.isnull().sum() #No null values

#Seperate input and output data.
input_data = dataset.iloc[:, :-1]
output_data = dataset.iloc[:, -1]

#Scaling the input data
from sklearn.preprocessing import StandardScaler

ss = StandardScaler()
input_data = pd.DataFrame(ss.fit_transform(input_data),
                          columns = input_data.columns)

#Train test split
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(
        input_data, output_data, test_size=0.2,
```

random_state = 42)

```
# Create an artificial neural network
import tensorflow
from keras.layers import Dense, BatchNormalization
from keras.models import Sequential
from keras.callbacks import EarlyStopping
from keras.regularizers import L2

ann = Sequential() # Object initialization

# Build different layers
ann.add(Dense(8, input_dim = x_train.shape[1],
              activation = "relu",
              kernel_regularizer = L2(12 = 0.01)))
ann.add(BatchNormalization())
ann.add(Dense(6, activation = "relu",
              kernel_regularizer = L2(12 = 0.01)))
ann.add(BatchNormalization())
ann.add(Dense(4, activation = "relu",
              kernel_regularizer = L2(12 = 0.01)))
ann.add(BatchNormalization())
ann.add(Dense(2, activation = "relu",
              kernel_regularizer = L2(12 = 0.01)))
ann.add(BatchNormalization())
ann.add(Dense(1, activation = "sigmoid"))

# Compile the model
ann.compile(optimizer = "adam", loss = "binary-
            crossentropy", metrics = ["accuracy"])
```

```
# Train the model
ann.fit(x_train, y_train, batch_size=100, epochs=
        50, validation_data=(x_test, y_test),
        callbacks=EarlyStopping())

# Training and testing accuracy of each epoch
train_accuracy = ann.history.history["accuracy"]
test_accuracy = ann.history.history["val_accuracy"]
train_accuracy, test_accuracy

# Visualize the accuracy
plt.plot([i+1 for i in range(len(test_accuracy))],
         test_accuracy, color="blue")
plt.plot([i+1 for i in range(len(train_accuracy))],
         train_accuracy, color="red")
plt.show()

# Check the accuracy of our model
from sklearn.metrics import accuracy_score

# Predictions of x_train
predict_x_train = ann.predict(x_train)
x_train_predictions = []
for i in predict_x_train:
    if i[0] > 0.5:
        x_train_predictions.append(1)
    else:
        x_train_predictions.append(0)
# View predictions
x_train_predictions
```

```
# Prediction of x-test
predict_x_test = ann.predict(x_test)
x_test_predictions = []
for i in predict_x_test:
    if i[0] > 0.5:
        x_test_predictions.append(1)
    else:
        x_test_predictions.append(0)
# View predictions
x_test_predictions

# Get accuracy
train_data_accuracy = accuracy_score(y_train,
                      x_train_predictions)*100
test_data_accuracy = accuracy_score(y_test,
                     x_test_predictions)*100
train_data_accuracy, test_data_accuracy
```

""" If the training accuracy is more than testing accuracy then the model is overfitted.
To avoid overfitting, we can use Dropout layer to minimize the gap between the accuracies. """

Dropout Layer:

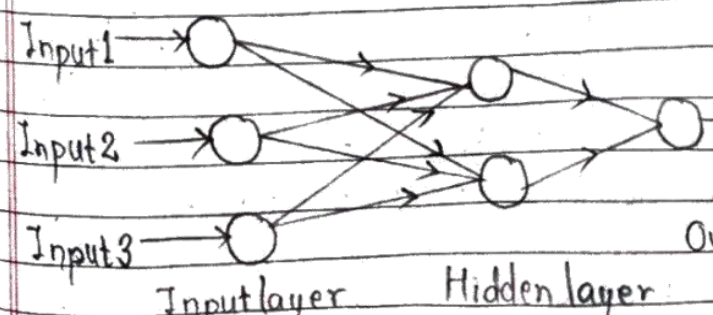
```
from keras.layers import Dropout

# Build different layers
ann.add(BatchNormalization())
ann.add(Dense(8, input_dim = x_train.shape[1],
        activation = "relu",
```

```
                kernel_regularizer = L2 (l2 = 0.01)))
ann.add (Dropout (0.3)) # Dropout layer with 30%
# to first layer
ann.add (Batch Normalization ())
ann.add (Dense (6, activation = "relu",
                kernel_regularizer = L2 (l2 = 0.01)))
ann.add (Dropout (0.3)) # Dropout layer to second
# layer
ann.add (Batch Normalization ())
ann.add (Dense (4, activation = "relu",
                kernel_regularizer = L2 (l2 = 0.01)))
ann.add (Dropout (0.3)) # Dropout layer to third
# layer
ann.add (Batch Normalization ())
ann.add (Dense (2, activation = "relu",
                kernel_regularizer = L2 (l2 = 0.01)))
ann.add (Dropout (0.3)) # Dropout layer to fourth
# layer
ann.add (Batch Normalization ())
ann.add (1, activation = "sigmoid"))

# Compile the model
ann.compile (optimizer = "adam", loss = "binary-
             crossentropy", metrics = ["accuracy"])

# Train the model
ann.fit (x_train, y_train, batch_size = 100, epochs = 50,
         validation_data = (x_test, y_test), callbacks =
         EarlyStopping ())

# Training and testing accuracy of each epoch
```

```
train_accuracy = ann.history.history["accuracy"]
test_accuracy = ann.history.history["val_accuracy"]
train_accuracy, test_accuracy

#Visualize the accuracy
plt.plot([i+1 for i in range(len(test_accuracy))],
         test_accuracy, color="blue")
plt.plot([i+1 for i in range(len(train_accuracy))],
         train_accuracy, color="red")
plt.show()
```

11. Vanishing Gradient Problem

- The vanishing gradient problem is encountered when training artificial neural networks with gradient-based learning methods and backpropagation.
- In such methods, during each iteration of training each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight.



Solutions for Vanishing Gradient Problem

1. Proper weight initialization
2. Using non-saturating activation functions (relu)
3. Batch normalization
4. Gradient clipping

Working Practically

""" While working with deep learning algorithms if you are facing a problem in which the loss from each or many number of epohs comes constant.
This is vanishing gradient problem as no an gradient is seen.
The above given methods can be used in such time to overcome the problem. """

12. Hyperparameter Tuning

- Hyperparameter tuning is a crucial step in optimizing deep learning models.
- It involves finding the best set of hyperparameters that result in the most effictive model.
- It is used to increase accuracy.

Key Hyperparameters

1. **Learning Rate:** Controls how much to change the model in response to the estimated error each time the model weights are updated.

2. **Batch Size:** Number of training examples utilized in one iteration.

3. **Number of Epochs:** Number of complete passes through the training dataset.

4. **Number of Layers and Neurons:** Structure of the neural network, including the number of layers and the number of neurons in each layer.

5. **Activation Function:** Function that introduce non-linearity to the model.

6. **Optimizer:** Algorithm used to update the weights of network (e.g. Adam, SGD)

7. **Dropout Rate:** Fraction of input units to drop during training to prevent overfitting.

8. **Regularization Parameters:** Parameters like L2, L1. L1 regularization to prevent overfitting.

Methods for Hyperparameter Tuning

1. Grid Search: Exhaustively searches through a manually specified subset of the hyperparameter space.

2. Random Search: Randomly samples the hyperparameter space and performs better than grid search in practice.

3. Bayesian Optimization: Uses a probabilistic model to find the best hyperparameters.

4. Hyperband: Combines random search with early stopping to efficiently find the best hyperparameters.

5. Genetic Algorithm: Uses principles of natural selection to search for optimal hyperparameters

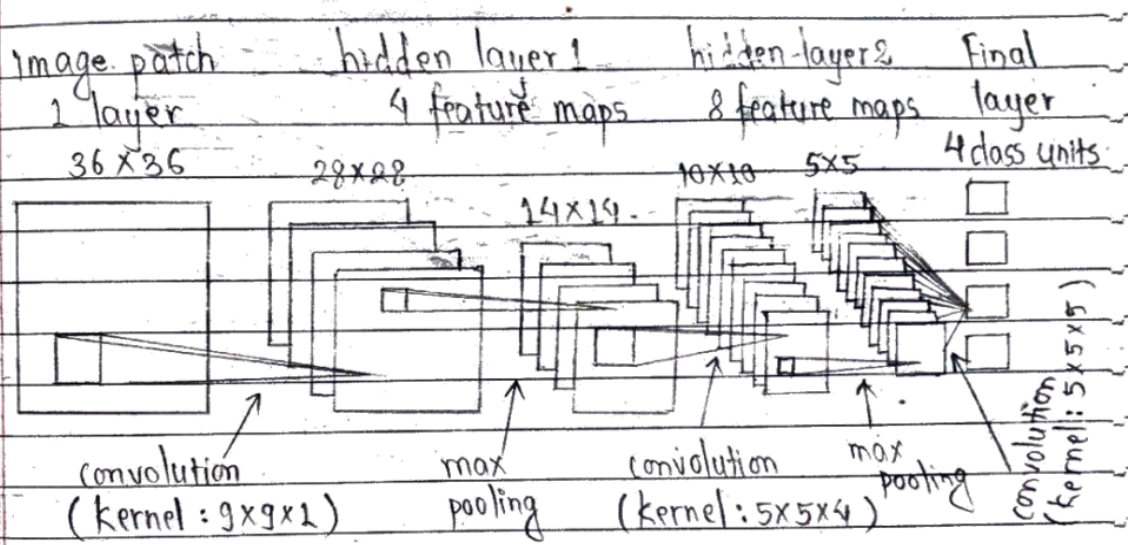
13. Convolutional Neural Network (CNN)

↳ CNN is a type of artificial neural network, which is widely used for image/object recognition and classification.

↳ Deep learning thus recognizes objects in an image by using a CNN.

↳ This works like our eyes for capturing data and brain for predictions.

Convolution → Max Pooling → Flattening → Fully Connected layers



1. Convolution

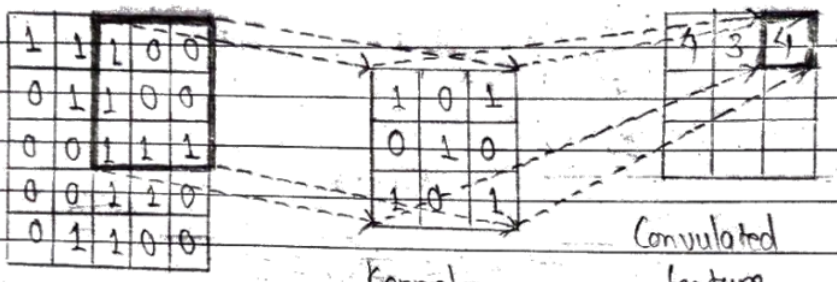
- Now in mathematics convolution is a mathematical operation on two functions that produces a third function that expresses how the shape of one is modified by the other.
- Mathematically,

$$c = (f * g)(t) = \int_0^t f(\tau)\, g(t-\tau)\, d\tau$$

Where,

c = convolution output function
f = original output function
g = function that is shifted over the input

function.
t = variable representing range of shifting
τ = shifting against t



Input data

Kernel

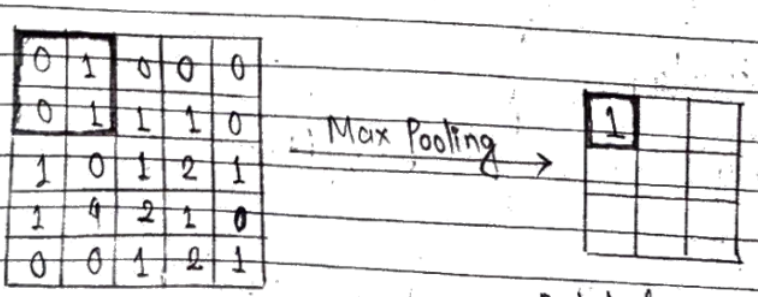
Convulated feature

1*1 = 1
0*0 = 0
0*1 = 0
1*0 = 0
1*1 = 1
0*0 = 0
1*1 = 1
1*0 = 0
+ 1*1 = 1
4

2. Pooling

10

- A pooling layer is another building block of a CNN.
- It's function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network.



Feature map

Pooled feature map

max (0, 1, 0, 1)
= 1

Date
Page 246

3. Flattening

- Flattening is converting the data into a 1-dimensional array for inputing it to next layer.
- Flatten output of the convolutional layers to create a single long feature vector.
- It is connected to the final classification model, which is called a fully connected layer.

1	1	0
4	2	1
0	2	1

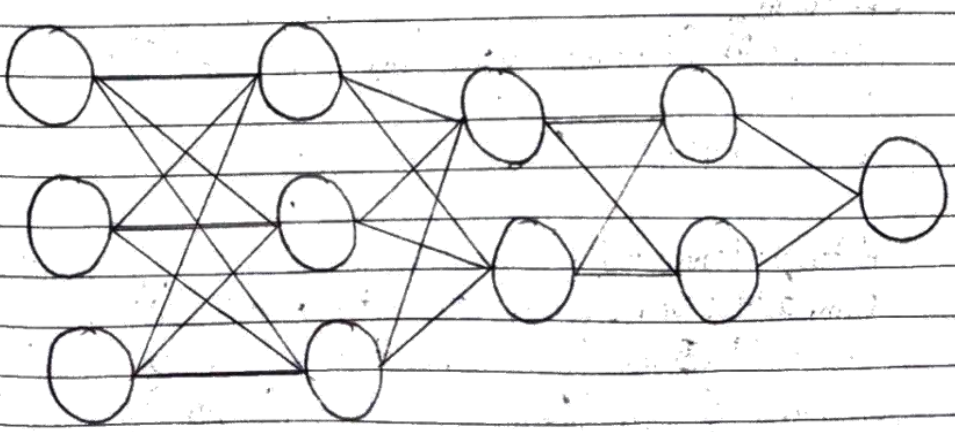
pooled feature

flattening →

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	y
1	1	0	4	2	1	0	2	1	-

4. Fully Connected Layers

- It is an Artificial Neural Network (ANN)



Working Practically

```
import tensorflow
from keras.layers import Dense, Conv2D, MaxPooling2D,
                         Flatten
from keras.models import Sequential
from tensorflow.keras.preprocessing.image import
                         ImageDataGenerator

#Create a CNN network.
cnn = Sequential()

#Convolution
cnn.add(Conv2D(filters=32, kernel_size=(3,3),
               input_shape=(64, 64, 3),
               activation="relu"))
#Pooling
cnn.add(MaxPooling2D(pool_size=(2,2)))
#Convolution
cnn.add(Conv2D(filters=32, kernel_size=(3,3),
               activation="relu"))
#Pooling
cnn.add(MaxPooling2D(pool_size=(2,2)))
#Flatten
cnn.add(Flatten())

#Fully Connected Layers
cnn.add(Dense(64, activation="relu"))
cnn.add(Dense(32, activation="relu"))
cnn.add(Dense(16, activation="relu"))
cnn.add(Dense(8, activation="relu"))
```

```
cnn.add(Dense(4, activation="relu"))
cnn.add(Dense(2, activation="relu"))
cnn.add(Dense(1, activation="sigmoid"))

# Compile model
cnn.compile(optimizer="adam", loss="binary_crossentropy")

# Create an instance of ImageDataGenerator
train_datagen = ImageDataGenerator(
        rescale=1.0/255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True
)
# Same for test_datagen
test_datagen = ImageDataGenerator(rescale=1.0/255)

# Generate batches of augmented image data from
#the specified directory.
train_generator = train_datagen.flow_from_directory(
        r".\Pictures\Train",
        target_size=(64,64),
        batch_size=32,
        class_mode="binary"
)
# Same for test_generator
test_generator = test_datagen.flow_from_directory(
        r".\Pictures\Test",
        target_size=(64,64),
        batch_size=32,
        class_mode="binary"
```

```
)

# Train the model using fit method
cnn.fit(
    train_generator,
    steps_per_epoch = 20,
    epochs = 5,
    validation_data = test_generator
)

# Evaluate the model on the test data
results = cnn.evaluate(test_generator)
results

# Making predictions (new)
from keras.preprocessing import image

# Load image
img = image.load_img(r".\Pictures\Predict\Dogs\
    PredictDog1.jpg", target_size = (64, 64))
img

# Convert the image to array
img = image.img_to_array(img)
img

# Change the dimensions of image
import numpy as np

img = np.expand_dims(img, axis=0)
```

```
cnn.predict(img)  # New prediction

# To get the output in desired form
p = cnn.predict(img)
if p[0][0] < 0.5:
        print("Dog")
else:
        print("Cat")

""" Predict the multiple values from the folder """

# Create an ImageGenerator for prediction
predict_datagen = ImageGenerator(rescale = 1.0/225)

# Generate batches of images from the specified
# directory for prediction
predict_generator = predict_datagen.flow_from_directory(
        r"..\Pictures\Predict",
        target_size = (64, 64), batch_size = 32,
        class_mode = None,
        shuffle = False
)

# Make predictions
predictions = cnn.predict(predict_generator)

# Convert predictions to class labels
threshold = 0.5
predicted_classes = np.where(predictions > threshold,
                             "Dog", "Cat")
```

```
# Get the filenames
filenames = predict_generator.filenames

# Combine files and predictions
results = list(zip(filenames, predicted_classes))

# Print the results
for filename, prediction in results:
    print(f"File: {filename}, Predicted Class:
          {prediction}")
```