Date of Performance :

Date of Submission :

EXPERIMENT NUMBER: 1

Aim: Implement Mc-Culloch Pitts model for AND and XOR logic gates.

Objective:

To implement basic neural network models for simulating logic gate

Software Used :Python

Theory:

Mc-Culloch Pitt Model :The McCulloch-Pitts (M-P) model, introduced in 1943 by Warren McCulloch and Walter Pitts, is a foundational model of artificial neural networks. The model is based on the concept of binary neurons, which are either on (1) or off (0), and the model defines a set of logical rules that determine the state of a neuron based on the states of its inputs.

In the M-P model, a neuron is represented as a threshold gate, which receives inputs from other neurons and outputs a binary value depending on whether the sum of the inputs exceeds a threshold value. The threshold gate acts as a simple decision maker, allowing a representation of logical statements such as "if A and B are true, then output 1."

One important aspect of the M-P model is that it allows for the creation of a network of neurons that can work together to solve a problem. For example, multiple neurons can be connected to recognize patterns in input data, making it possible to use the M- P model for applications such as image recognition or decision making.

Despite its simplicity, the M-P model was a major contribution to the development of artificial neural networks and provided a basis for further research in the field. The M-P model inspired the development of more sophisticated neural network models with continuous activation functions and multi-layer networks, which expanded the capability of artificial neural networks from simple decision making to more complex pattern recognition and data analysis.

In conclusion, the McCulloch-Pitts model is a simple yet powerful model of artificial neurons that laid the foundation for the development of artificial intelligence and artificial neural networks. It introduced the idea of binary neurons and showed how a network of neurons can work together to solve problems, paving the way for the development of more advanced neural network models.

Algorithm:

Here is a high-level pseudocode for a McCulloch-Pitts (M-P) model:

```
Input: binary inputs x1, x2, ..., xn
Output: binary output y

Step 1: Initialize weights w1, w2, ..., wn and threshold valueθ
Step 2: Calculate the weighted sum of inputs:
    weighted_sum = w1 * x1 + w2 * x2 + ... + wn * xn
Step 3: Compare weighted_sum to thresholdθ :
    if weighted_sum >θ :
        y = 1
    else:
        y = 0
Step 4: Return output y
```

Note that this is just a high-level pseudocode and not a complete implementation of the M-P model. A complete implementation would typically involve training the network to optimize the weights and threshold based on a set of input-output examples, and would involve additional steps and algorithms for weight update and error calculation.

Program:

*********Mc-Culloch Pitts model for AND gate*****************

```
import numpy as np
def ANDThreshold(v):
  if v > 1:
    return 1
  else:
    return 0
def MCPModel(x, w):
  v = np.dot(w, x)
  y = ANDThreshold(v)
  return y
def ANDGate(x):
  w = np.array([1, 1])
return MCPModel(x, w)
test1 = np.array([0, 0])
test2 = np.array([0, 1])
test3 = np.array([1, 0])
test4 = np.array([1, 1])
print(f"AND({0}, {0}) = { ANDGate(test1)}")
print(f"AND({0}, {1}) = { ANDGate(test2)}")
print(f"AND({1}, {0}) = { ANDGate(test3)}")
print(f"AND({1}, {1}) = { ANDGate(test4)}")
```

*********Mc-Culloch Pitts model for XOR gate*****************

```
def XORThreshold(v):
  if v == 1:
    return 1
  else:
```

```
    return 0
def MCPModel(x, w):
  v = np.dot(w, x)
  y = XORThreshold(v)
  return y
def XORGate(x):
  w = np.array([1, 1])
return MCPModel(x, w)
test1 = np.array([0, 0])
test2 = np.array([0, 1])
test3 = np.array([1, 0])
test4 = np.array([1, 1])
print(f"XOR({0}, {0}) = { XORGate(test1)}")
print(f"XOR({0}, {1}) = { XORGate(test2)}")
print(f"XOR({1}, {0}) = { XORGate(test3)}")
print(f"XOR({1}, {1}) = { XORGate(test4)}")
```

Output:

Mc-Culloch Pitts model for AND gate

Mc-Culloch Pitts model for OR gate

Conclusion/Outcome:

Thus we have implemented Mc-Culloch Pitts model for AND and XOR logic gates
We also understood that implement basic neural network models for simulating logic gate

Marks & Signature:

R1 (4 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (3 Mark)	Total (15 Marks)	Signature

Date of Performance :

Date of Submission :

EXPERIMENT NUMBER: 2

Aim: Implement Perceptron algorithm to simulate AND gate.

Objective:

To implement basic neural network models for simulating logic gate.

Software Used :Python

Theory:

Perceptron algorithm to simulate AND gate

The Perceptron algorithm is used to model the decision-making process of a logic gate by using a set of weights, bias, and a threshold to produce a binary output based on the inputs.

The Perceptron algorithm takes in two binary inputs and uses a set of weights and a bias to produce a single binary output. The inputs are multiplied by the weights and summed with the bias to produce a scalar value. This value is then compared to a threshold, and if it is above the threshold, the output is 1 (True), and if it is below the threshold, the output is 0 (False).

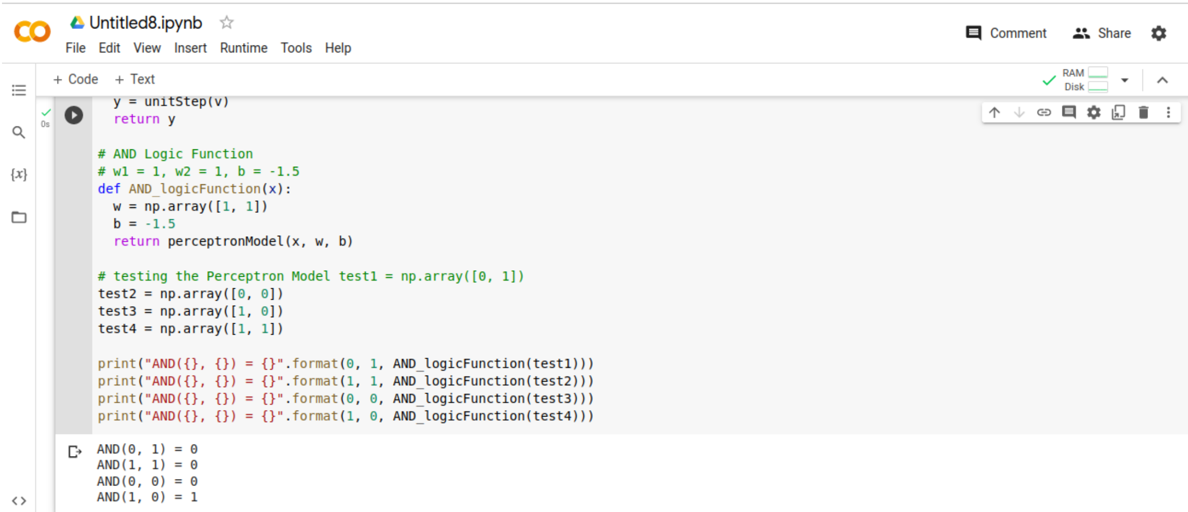
In the case of simulating an AND gate, the weights and bias are chosen such that if both inputs are 1, the output is 1 (True), and if either input is 0 (False), the output is 0 (False). The Perceptron algorithm can be used to simulate other logic gates, such as OR and NOT, by choosing appropriate values for the weights and bias.

Algorithm:
Step 1: Define unit step function
Step 2: Assume w and b value
Step 3: Find net value using wx+b
Step 4: Find output value by using unit step function.
Step 5: Find error between actual and desired.
Step 6: If error is not equal to 0 update weight and bias value and go to step 5, if error is zero, go to the next step.
Step 7: Perception model is ready for further testing.

Program:

```
*********Perceptron algorithm to simulate AND gate***************** #
importing Python library
import numpy as np

# define Unit Step Function
def unitStep(v):
  if v >= 0:
    return 1
  else:
    return 0

# design Perceptron Model
def perceptronModel(x, w, b):
  v = np.dot(w, x) + b
  y = unitStep(v)
  return y

# AND Logic Function
# w1 = 1, w2 = 1, b = -1.5
def AND_logicFunction(x):
  w = np.array([1, 1])
  b = -1.5
  return perceptronModel(x, w, b)

# testing the Perceptron Model
test1 = np.array([0, 1])
test2 = np.array([0, 0])
test3 = np.array([1, 0])
test4 = np.array([1, 1])

print("AND({}, {}) = {}".format(0, 1, AND_logicFunction(test1)))
print("AND({}, {}) = {}".format(1, 1, AND_logicFunction(test2)))
print("AND({}, {}) = {}".format(0, 0, AND_logicFunction(test3)))
print("AND({}, {}) = {}".format(1, 0, AND_logicFunction(test4)))
```

Output:



Conclusion/Outcome:

Thus we have implemented the Perceptron algorithm to simulate AND gate.

We also understood how to implement basic neural network models for simulating logic gate.

Marks & Signature:

R1 (4 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (3 Mark)	Total (15 Marks)	Signature

Date of Performance :

Date of Submission :

EXPERIMENT NUMBER: 3

Aim:Apply Adam Learning GD algorithms to learn the parameters of the supervised single layer feed forward neural network.

Objective:

To implement various training algorithms for feedforward neural networks.

Software Used :Python

Theory:

(a) Feedforward neural networks:

A Feedforward neural network is a type of artificial neural network in which data flows through the network in only one direction, from input layer to output layer, without looping back. It consists of an input layer, hidden layers, and an output layer. The hidden layers perform computation and feature extraction, while the output layer produces the desired outputs. The network is trained using supervised learning algorithms, where the parameters of the network are updated based on the error between the predicted outputs and the ground truth.

(b) Adam Learning GD

Adam is an optimization algorithm commonly used for training neural networks. It is an extension of the traditional Stochastic Gradient Descent (SGD) optimization method. Adam combines the benefits of SGD with the Adaptive Gradient Algorithm to provide a more robust and efficient optimization process. It uses moving averages of the parameters to provide a running estimate of the second raw moments of the gradients; the mean and variance. These moving averages are updated using exponential decay, allowing Adam to react to changes in the distribution of the gradients and adjust the learning rate accordingly. This makes the optimization process more stable and helps to avoid getting stuck in sub-optimal solutions or saddle points.

Algorithm:

Feedforward neural network:

Input : ProblemSize, InputPatterns iterationsmax , learnrate) Output Network
Network ConstructNetworkLayer () ;
Networkweight InitializeWeights (Network, ProblemSize);
for i = 1 to iterations max do
    Patterni < — SelectInputPattern InputPatterns)
    Outputi < —ForwardPropagate Pattern, Network)
    BackwardPropagateError (Pattern, Output, Network)
    UpdateWeight (Pattern, Output, Network, learnrate)
end
return network;

Adam Learning GD

- $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \eta = 10^{-8}$ (Defaults)

$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$i \leftarrow 0$ (Initialize step)
while Θ_i not converged **do**
  $i \leftarrow i+1$
  $g_i \leftarrow \nabla_\Theta f_i(\Theta_{i-1})$ (Get gradients at step i)
  $m_i \leftarrow \beta_1 \cdot m_{i-1} + (1-\beta_1) \cdot g_i$ (Update biased first moment estimate)
  $v_i \leftarrow \beta_2 \cdot v_{i-1} + (1-\beta_2) \cdot g_i^2$ (Update biased second raw moment estimate)
  $\hat{m}_i \leftarrow m_i / (1-\beta_1^i)$ (Compute bias-corrected first moment estimate)
  $\hat{v}_i \leftarrow v_i / (1-\beta_2^i)$ (Compute bias-corrected second raw moment estimate)
  $\Theta_i \leftarrow \Theta_{i-1} - \alpha \cdot \hat{m}_i / (\sqrt{\hat{v}_i} + \eta)$ (Update parameters)
end while
return Θ_i (resulting parameters)

Program:

*********supervised single layer feed forward neural network using Adam Learning GD algorithm*****

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

X = tf.constant(tf.range(0,100,1))
Y = X + 100
X , Y
X.shape , Y.shape
tf.random.set_seed(42)
model1 = tf.keras.Sequential([
    tf.keras.layers.Dense(100),
    tf.keras.layers.Dense(10),
    tf.keras.layers.Dense(1),
])

model1.compile(loss = tf.keras.losses.MAE,
              optimizer = tf.keras.optimizers.Adam(lr = 0.01),
              metrics = ['mae'])

model1.fit(tf.expand_dims(X , axis = -1) ,Y , epochs = 100)

model1.evaluate(X , Y)

model1.predict([7])

predict = model1.predict(X)
predict = tf.round(predict)
predict
```

Output

```
WARNING:absl:`lr` is deprecated, please use `learning_rate` instead, or use the legacy optimizer, e.g.,tf.keras.op
Epoch 1/100
4/4 [==============================] - 1s 5ms/step - loss: 151.5043 - mae: 151.5043
Epoch 2/100
4/4 [==============================] - 0s 3ms/step - loss: 142.3857 - mae: 142.3857
Epoch 3/100
4/4 [==============================] - 0s 3ms/step - loss: 133.2857 - mae: 133.2857
Epoch 4/100
4/4 [==============================] - 0s 3ms/step - loss: 123.7735 - mae: 123.7735
Epoch 5/100
4/4 [==============================] - 0s 4ms/step - loss: 114.2244 - mae: 114.2244
Epoch 6/100
4/4 [==============================] - 0s 3ms/step - loss: 104.3139 - mae: 104.3139
Epoch 7/100
4/4 [==============================] - 0s 4ms/step - loss: 93.6704 - mae: 93.6704
Epoch 8/100
4/4 [==============================] - 0s 3ms/step - loss: 82.4385 - mae: 82.4385
Epoch 9/100
4/4 [==============================] - 0s 7ms/step - loss: 70.2431 - mae: 70.2431
Epoch 10/100
4/4 [==============================] - 0s 6ms/step - loss: 57.1377 - mae: 57.1377
Epoch 11/100
4/4 [==============================] - 0s 7ms/step - loss: 45.5755 - mae: 45.5755
Epoch 12/100
4/4 [==============================] - 0s 5ms/step - loss: 41.0548 - mae: 41.0548
Epoch 13/100
4/4 [==============================] - 0s 5ms/step - loss: 42.1556 - mae: 42.1556
Epoch 14/100
4/4 [==============================] - 0s 5ms/step - loss: 43.4758 - mae: 43.4758
Epoch 15/100
```

```
array([[ 8.],
       [ 10.],
       [ 13.],
       [ 15.],
       [ 17.],
       [ 20.],
       [ 22.],
       [ 24.],
       [ 27.],
       [ 29.],
       [ 32.],
       [ 34.],
       [ 36.],
       [ 39.],
       [ 41.],
       [ 44.],
       [ 46.],
       [ 48.],
       [ 51.],
       [ 53.],
       [ 56.],
       [ 58.],
       [ 60.],
       [ 63.],
       [ 65.],
       [ 68.],
       [ 70.],
       [ 72.],
       [ 75.],
       [ 77.],
       [ 80.],
```

Conclusion/Outcome:

Thus we have implemented supervised single layer feed forward neural networks using Adam Learning GD algorithm.

Marks & Signature:

R1 (4 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (3 Mark)	Total (15 Marks)	Signature

Date of Performance:

Date of Submission :

EXPERIMENT NUMBER: 4

Aim: Implement a Backpropagation algorithm to train a DNN with at least 2 hidden layers.

Objective:

To implement various training algorithms for feedforward neural networks.

Software Used :Python

Theory:

(a) Backpropagation algorithm

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.

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.

Algorithm:

Step 1: Inputs X, arrive through the preconnected path.

Step 2: The input is modeled using true weights W. Weights are usually chosen randomly.

Step 3: Calculate the output of each neuron from the input layer to the hidden layer to the output layer. Step

4: Calculate the error in the output

Step 5: From the output layer, go back to the hidden layer to adjust the weights to reduce the error. Step

6: Repeat the process until the desired output is achieve

Program:

```
import tensorflow as tf

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

X = tf.constant(tf.range(0,100,1))

Y = X + 100

X , Y

X.shape , Y.shape
 tf.random.set_seed(101)
 model3 = tf.keras.Sequential([
     tf.keras.layers.Dense(100),
     tf.keras.layers.Dense(10),
     tf.keras.layers.Dense(1),
 ])

 model3.compile(loss = tf.keras.losses.MAE,
          optimizer =
 tf.keras.optimizers.Adam(lr = 0.02),
          metrics = ['mae'])

 history = model3.fit(tf.expand_dims(X , axis
 = -1) ,Y , epochs = 150          , verbose = 0)


 model3.evaluate(X,Y
```

Output:

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

X = tf.constant(tf.range(0,100,1))
Y = X + 100
X , Y

X.shape , Y.shape
tf.random.set_seed(101)
model3 = tf.keras.Sequential([
    tf.keras.layers.Dense(100),
    tf.keras.layers.Dense(10),
    tf.keras.layers.Dense(1),
])

model3.compile(loss = tf.keras.losses.MAE,
               optimizer =
tf.keras.optimizers.Adam(lr = 0.02),
metrics = ['mae'])

history = model3.fit(tf.expand_dims(X , axis
= -1) ,Y , epochs = 150 , verbose = 0)
model3.evaluate(X,Y)
```

```
WARNING:absl:`lr` is deprecated, please use `learning_rate` instead, or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.Adam.
4/4 [==============================] - 0s 4ms/step - loss: 32.8506 - mae: 32.8506
[32.8505973815918, 32.8505973815918]
```

Conclusion/Outcome:

Thus we have implemented a Backpropagation algorithm to train a DNN with at least 2 hidden layers.

Marks & Signature:

R1 (4 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (3 Mark)	Total (15 Marks)	Signature

Date of Performance :

Date of Submission :

EXPERIMENT NUMBER: 5

Aim: Design the architecture and implement the autoencoder model for Image denoising.

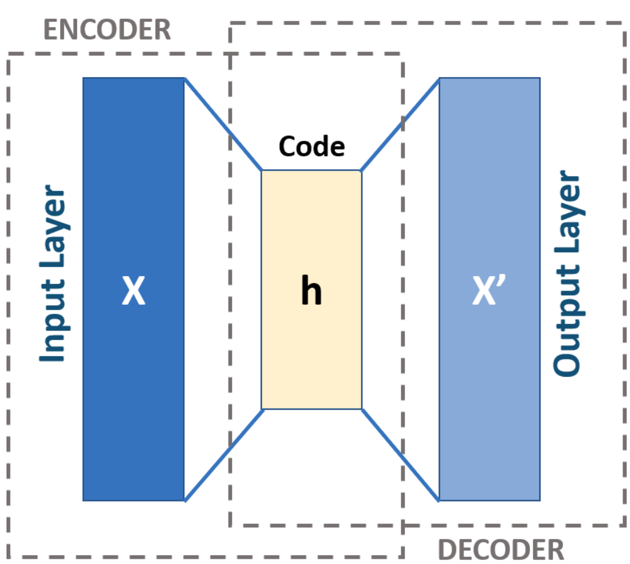
Objective:

- To acquire knowledge of advanced concepts of Auto encoders.

Software Used : Python

Theory:

Auto encoder
An **autoencoder** is a type of artificial neural network used to learn efficient codings of unlabeled data (unsupervised learning). An autoencoder learns two functions: an encoding function that transforms the input data, and a decoding function that recreates the input data from the encoded representation. The autoencoder learns an efficient representation (encoding) for a set of data, typically for dimensionality reduction.



There are 4 hyperparameters that we need to set before training an autoencoder:

- Code size: number of nodes in the middle layer. Smaller size results in more compression.
- Number of layers: the autoencoder can be as deep as we like. In the figure above we have 2 layers in both the encoder and decoder, without considering the input and output.
- Number of nodes per layer: the autoencoder architecture we're working on is called a *stacked autoencoder* since the layers are stacked one after another. Usually stacked autoencoders look like a "sandwitch". The number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the decoder. Also the decoder is symmetric to the encoder in terms of layer structure. As noted above this is not necessary and we have total control over these parameters.
- Loss function: we either use *mean squared error (mse)* or *binary crossentropy*. If the input values are in the range [0, 1] then we typically use crossentropy, otherwise we use the mean squared error.

Algorithm:

1) Download f`ashion_mnist.load_data() from keras`
2) Splitt data in training and testing part.
3) Plotting Noisy Training Images
4) Add noise to testing images.
5) Train autoencoder model
6) Find binary cross entropy loss. set learning rate and find accuracy.
7) Printing the Denoised Images from Model (Model Output)

Program:

```
import tensorflow as tf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import random
(x_train,y_train),(x_test,y_test)= tf.keras.datasets.fashion_mnist.load_data()
print(x_train.shape)
print(y_train.shape)
print(x_test.shape)
print(y_test.shape)
labels={0:"T-Shirt",1:"Trousers",2:"Pullover",3:"Dress",4:"Coat",5:"Sandal",6:"T
-Shirt",7:"Sneakers",8:"Bag",9:"Ankle Boot"}
a=np.random.randint(low=0,high=1000,size=100)
fig=plt.figure(figsize=(15,18))
c=1
for i in a:
   fig.add_subplot(10,10,c)
   plt.xticks([])
   plt.yticks([])
   plt.imshow(x_train[i],cmap='gray')
   plt.title(labels[y_train[i]],color="green",fontsize=12)
   c+=1
```

#Data Preprocessing

```
x_train=x_train/255.
x_test=x_test/255.
noise_factor=0.3
noise_train=[]

for img in x_train:
   noisy_img = img + noise_factor* np.random.randn(*img.shape)
   noisy_img = np.clip(noisy_img , 0, 1)
   noise_train.append(noisy_img)
```

#Plotting Noisy Training Images

```
a=np.random.randint(low=0,high=1000,size=50)
fig=plt.figure(figsize=(15,8))
c=1
for i in a:
   fig.add_subplot(5,10,c)
```

```
   plt.xticks([])
   plt.yticks([])
   plt.imshow(noise_train[i],cmap='gray')
   plt.title(labels[y_train[i]],color="green",fontsize=12)
   c+=1
```

#ADD NOISE TO TESTING IMAGES

```
noise_factor=0.3
noise_test=[]

for img in x_test:
   noisy_img = img + noise_factor* np.random.randn(*img.shape)
   noisy_img = np.clip(noisy_img , 0, 1)
   noise_test.append(noisy_img)

a=np.random.randint(low=0,high=1000,size=25)
fig=plt.figure(figsize=(7,8))
c=1
for i in a:
   fig.add_subplot(5,5,c)
   plt.xticks([])
   plt.yticks([])
   plt.imshow(noise_test[i],cmap='gray')
   plt.title(labels[y_test[i]],color="green",fontsize=12)
   c+=1
noise_test=np.array(noise_test)
noise_train=np.array(noise_train)
```

#AUTO-ENCODER MODEL

```
autoencoder= tf.keras.Sequential([

tf.keras.layers.Conv2D(32,(3,3),strides=2,padding='same',input_shape=(28,28,1)),
   tf.keras.layers.Conv2D(16,(3,3),strides=2,padding='same'),
   tf.keras.layers.Conv2D(16,(3,3),strides=1,padding='same'),
   tf.keras.layers.Conv2DTranspose(32,(3,3),strides=2,padding='same'),

tf.keras.layers.Conv2DTranspose(1,(3,3),strides=2,padding='same',activation='sig
moid'),
])
autoencoder.summary()
autoencoder.compile(loss='binary_crossentropy',optimizer=tf.keras.optimizers.Ada
m(learning_rate=0.001),metrics=['accuracy'])
valid_noise=noise_test[:5000,:]
valid_y=x_test[:5000,:]
noise_test = noise_test[5000:,:]
x_test = x_test[5000:,:]
```

```
print(valid_noise.shape,valid_y.shape,noise_test.shape,x_test.shape)
autoencoder.fit(noise_train.reshape(-1,28,28,1),
              x_train.reshape(-1,28,28,1),
              epochs=15,
              batch_size=64,

validation_data=(valid_noise.reshape(-1,28,28,1),valid_y.reshape(-1,28,28,1)))

metric =
autoencoder.evaluate(noise_test.reshape(-1,28,28,1),x_test.reshape(-1,28,28,1))
print("Test Accuracy is {:.2f} and loss is
{:.3f}".format(metric[1]*100,metric[0]))
```

#Printing the Denoised Images from Model (Model Output)

```
a=np.random.randint(low=0,high=1000,size=100)
fig=plt.figure(figsize=(15,18))
c=1
for i in a:
   pred=autoencoder.predict(noise_test[i].reshape(1,28,28,1))
   fig.add_subplot(10,10,c)
   plt.xticks([])
   plt.yticks([])
   plt.imshow(pred.reshape(28,28),cmap='gray')
   plt.title(labels[y_test[5000:][i]],color="green",fontsize=12)
   c+=1
```

#Printing the Original Images (Denoised)

```
fig=plt.figure(figsize=(15,18))
c=1
for i in a:
   fig.add_subplot(10,10,c)
   plt.xticks([])
   plt.yticks([])
   plt.imshow(x_test[i].reshape(28,28),cmap='gray')
   plt.title(labels[y_test[5000:][i]],color="green",fontsize=12)
   c+=1
```

Output:

Conclusion/Outcome:

Thus we have implemented Auto encoder for Image denoising.

Marks & Signature:

R1 (4 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (3 Mark)	Total (15 Marks)	Signature

Date of Performance :

Date of Submission :

EXPERIMENT NUMBER: 6

Aim: **Design and implement a CNN model for digit recognition application.**

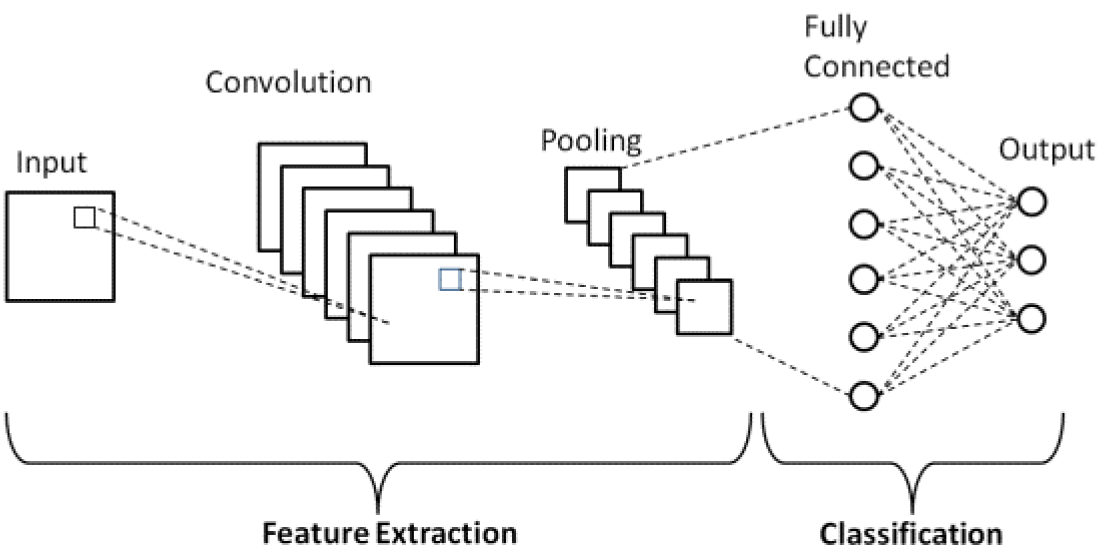
Objective:

- Acquired knowledge of advanced concepts of Convolution Neural Networks

Software Used : Python

Theory:

- Convolution Neural Networks (CNN)



1) Convolutional layer:This layer is the first layer that is used to extract the various features from the input images. In this layer, the mathematical operation of convolution is performed between the input image and a filter of a particular size MxM. By sliding the filter over the input image, the dot product is taken between the filter and the parts of the input image with respect to the size of the filter (MxM).
2) Pooling Layer:The primary aim of this layer is to decrease the size of the convolved feature map to reduce the computational costs. This is performed by decreasing the connections between layers and independently operates on each feature map. Depending upon method used, there are several types of Pooling operations like max pooling, average pooling

3) Fully Connected Layer: The Fully Connected (FC) layer consists of the weights and biases along with the neurons and is used to connect the neurons between two different layers. These layers are usually placed before the output layer and form the last few layers of a CNN Architecture. in this, the input image from the previous layers are flattened and fed to the FC layer. The flattened vector then undergoes few more FC layers where the mathematical functions operations usually take place. In this stage, the classification process begins to take place.
4) Dropout: Usually, when all the features are connected to the FC layer, it can cause overfitting in the training dataset. Overfitting occurs when a particular model works so well on the training data causing a negative impact in the model's performance when used on a new data. o overcome this problem, a dropout layer is utilised wherein a few neurons are dropped from the neural network during training process resulting in reduced size of the model.
5) Activation Functions: It decides which information of the model should fire in the forward direction and which ones should not at the end of the network. It adds non-linearity to the network. There are several commonly used activation functions such as the ReLU, Softmax, tanH and the Sigmoid functions. Each of these functions have a specific usage. For a binary classification CNN model, sigmoid and softmax functions are preferred an for a multi-class classification, generally softmax us used.

Algorithm:

1) download `mnist_dataset from keras`
2) Split data in training and testing part.
3) Save image parameters to the constants that we will use later for data re-shaping and for model traning.
4) Normalized data
5) Add convolutional layer
6) Add max pooling layer
7) Add convolutional layer
8) Add max pooling layer
9) Add Flatten layer
10) Add dropout layer and fully connected layer
11) Train model and find accuracy from confusion matrix

Program:

```
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sn
import numpy as np
import pandas as pd
import math
import datetime
import platform

print('Python version:', platform.python_version())
print('Tensorflow version:', tf.__version__)
print('Keras version:', tf.keras.__version__)
%load_ext tensorboard
# Clear any logs from previous runs.
!rm -rf ./.logs/
mnist_dataset = tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test) = mnist_dataset.load_data()
print('x_train:', x_train.shape)
print('y_train:', y_train.shape)
print('x_test:', x_test.shape)
print('y_test:', y_test.shape)
# Save image parameters to the constants that we will use later for data re-
shaping and for model traning.
(_, IMAGE_WIDTH, IMAGE_HEIGHT) = x_train.shape
IMAGE_CHANNELS = 1

print('IMAGE_WIDTH:', IMAGE_WIDTH);
print('IMAGE_HEIGHT:', IMAGE_HEIGHT);
print('IMAGE_CHANNELS:', IMAGE_CHANNELS);
pd.DataFrame(x_train[0])
plt.imshow(x_train[0], cmap=plt.cm.binary)
plt.show()
numbers_to_display = 25
num_cells = math.ceil(math.sqrt(numbers_to_display))
plt.figure(figsize=(10,10))
for i in range(numbers_to_display):
   plt.subplot(num_cells, num_cells, i+1)
   plt.xticks([])
   plt.yticks([])
   plt.grid(False)
```

```
    plt.imshow(x_train[i], cmap=plt.cm.binary)
plt.xlabel(y_train[i])
plt.show()
x_train_with_chanels = x_train.reshape(
    x_train.shape[0],
    IMAGE_WIDTH,
    IMAGE_HEIGHT,
    IMAGE_CHANNELS
)

x_test_with_chanels = x_test.reshape(
    x_test.shape[0],
    IMAGE_WIDTH,
    IMAGE_HEIGHT,
    IMAGE_CHANNELS
)
print('x_train_with_chanels:', x_train_with_chanels.shape)
print('x_test_with_chanels:', x_test_with_chanels.shape)
x_train_normalized = x_train_with_chanels / 255
x_test_normalized = x_test_with_chanels / 255
x_train_normalized[0][18]
x_train_normalized[0][18]
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Convolution2D(
    input_shape=(IMAGE_WIDTH, IMAGE_HEIGHT, IMAGE_CHANNELS),
    kernel_size=5,
    filters=8,
    strides=1,
    activation=tf.keras.activations.relu,
    kernel_initializer=tf.keras.initializers.VarianceScaling()
))

model.add(tf.keras.layers.MaxPooling2D(
    pool_size=(2, 2),
    strides=(2, 2)
))

model.add(tf.keras.layers.Convolution2D(
    kernel_size=5,
    filters=16,
    strides=1,
    activation=tf.keras.activations.relu,
    kernel_initializer=tf.keras.initializers.VarianceScaling()
))
```

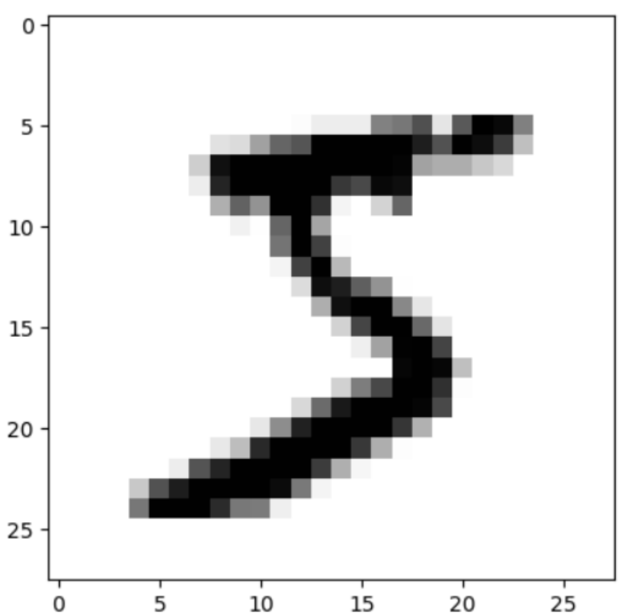
```
model.add(tf.keras.layers.MaxPooling2D(
   pool_size=(2, 2),
   strides=(2, 2)
))

model.add(tf.keras.layers.Flatten())

model.add(tf.keras.layers.Dense(
   units=128,
   activation=tf.keras.activations.relu
));

model.add(tf.keras.layers.Dropout(0.2))

model.add(tf.keras.layers.Dense(
   units=10,
   activation=tf.keras.activations.softmax,
   kernel_initializer=tf.keras.initializers.VarianceScaling()
))
model.summary()
tf.keras.utils.plot_model(
   model,
   show_shapes=True,
   show_layer_names=True,
)
adam_optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

model.compile(
   optimizer=adam_optimizer,
   loss=tf.keras.losses.sparse_categorical_crossentropy,
   metrics=['accuracy']
)
log_dir=".logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir,
histogram_freq=1)

training_history = model.fit(
   x_train_normalized,
   y_train,
   epochs=10,
   validation_data=(x_test_normalized, y_test),
   callbacks=[tensorboard_callback]
)
```

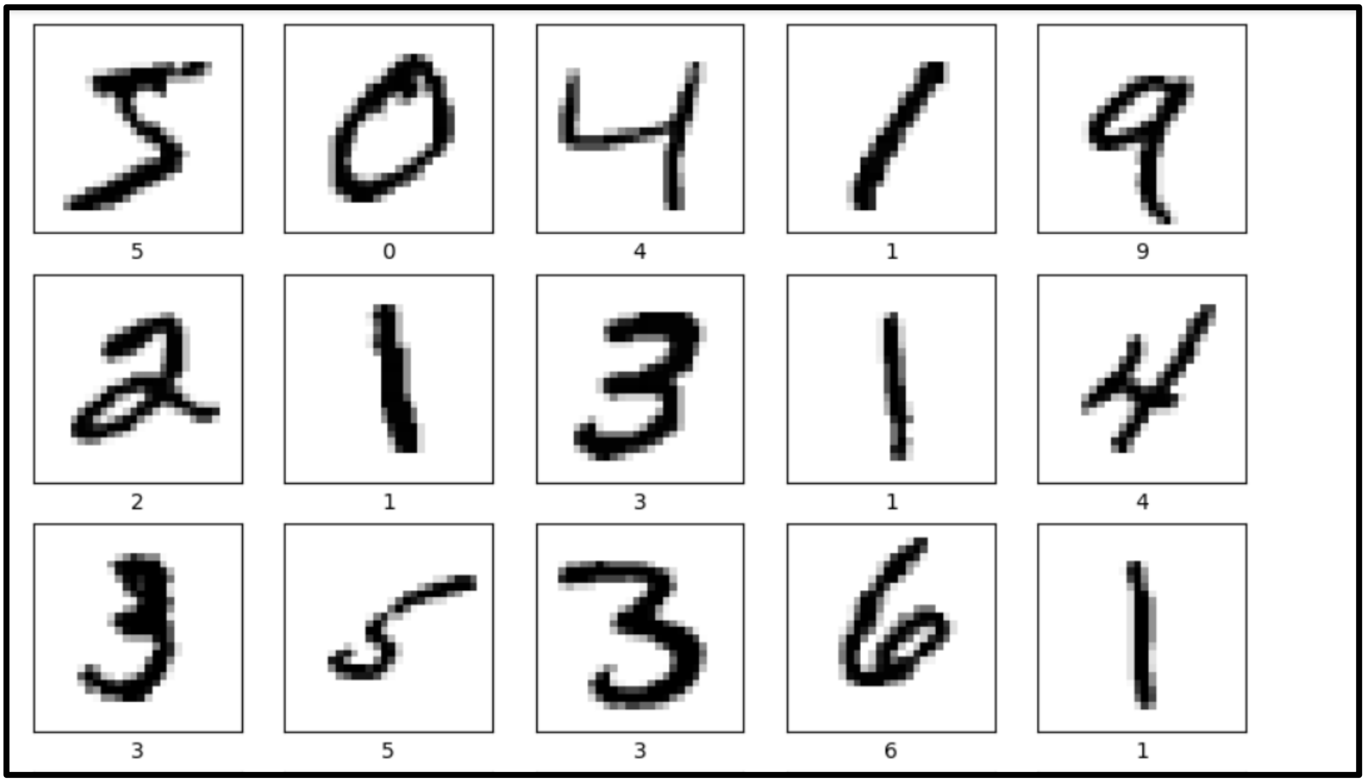
```
plt.xlabel('Epoch Number')
plt.ylabel('Loss')
plt.plot(training_history.history['loss'], label='training set')
plt.plot(training_history.history['val_loss'], label='test set')
plt.legend()
plt.xlabel('Epoch Number')
plt.ylabel('Accuracy')
plt.plot(training_history.history['accuracy'], label='training set')
plt.plot(training_history.history['val_accuracy'], label='test set')
plt.legend()
%%capture
train_loss, train_accuracy = model.evaluate(x_train_normalized, y_train)
print('Training loss: ', train_loss)
print('Training accuracy: ', train_accuracy)
%%capture
validation_loss, validation_accuracy = model.evaluate(x_test_normalized, y_test)
print('Validation loss: ', validation_loss)
print('Validation accuracy: ', validation_accuracy)
model_name = 'digits_recognition_cnn.h5'
model.save(model_name, save_format='h5')
loaded_model = tf.keras.models.load_model(model_name)
predictions_one_hot = loaded_model.predict([x_test_normalized])
print('predictions_one_hot:', predictions_one_hot.shape)
# Predictions in form of one-hot vectors (arrays of probabilities).
pd.DataFrame(predictions_one_hot)
predictions = np.argmax(predictions_one_hot, axis=1)
pd.DataFrame(predictions)
print(predictions[0])
plt.imshow(x_test_normalized[0].reshape((IMAGE_WIDTH, IMAGE_HEIGHT)),
cmap=plt.cm.binary)
plt.show()
numbers_to_display = 196
num_cells = math.ceil(math.sqrt(numbers_to_display))
plt.figure(figsize=(15, 15))

for plot_index in range(numbers_to_display):
   predicted_label = predictions[plot_index]
   plt.xticks([])
   plt.yticks([])
   plt.grid(False)
   color_map = 'Greens' if predicted_label == y_test[plot_index] else 'Reds'
   plt.subplot(num_cells, num_cells, plot_index + 1)
   plt.imshow(x_test_normalized[plot_index].reshape((IMAGE_WIDTH,
```

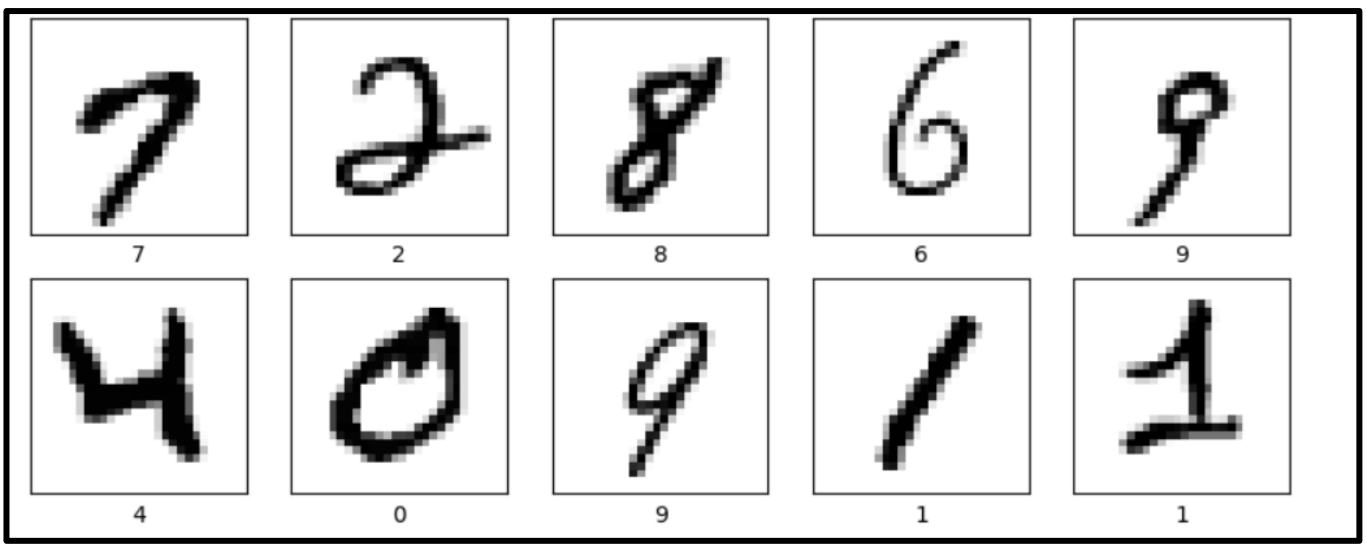
```
IMAGE_HEIGHT)), cmap=color_map)

   plt.xlabel(predicted_label)

plt.subplots_adjust(hspace=1, wspace=0.5)
plt.show()
confusion_matrix = tf.math.confusion_matrix(y_test, predictions)
f, ax = plt.subplots(figsize=(9, 7))
sn.heatmap(
   confusion_matrix,
   annot=True,
   linewidths=.5,
   fmt="d",
   square=True,
   ax=ax
)
plt.show()
```

output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 1s 0us/step
x_train: (60000, 28, 28)
y_train: (60000,)
x_test: (10000, 28, 28)
y_test: (10000,)
IMAGE_WIDTH: 28
IMAGE_HEIGHT: 28
IMAGE_CHANNELS: 1
```

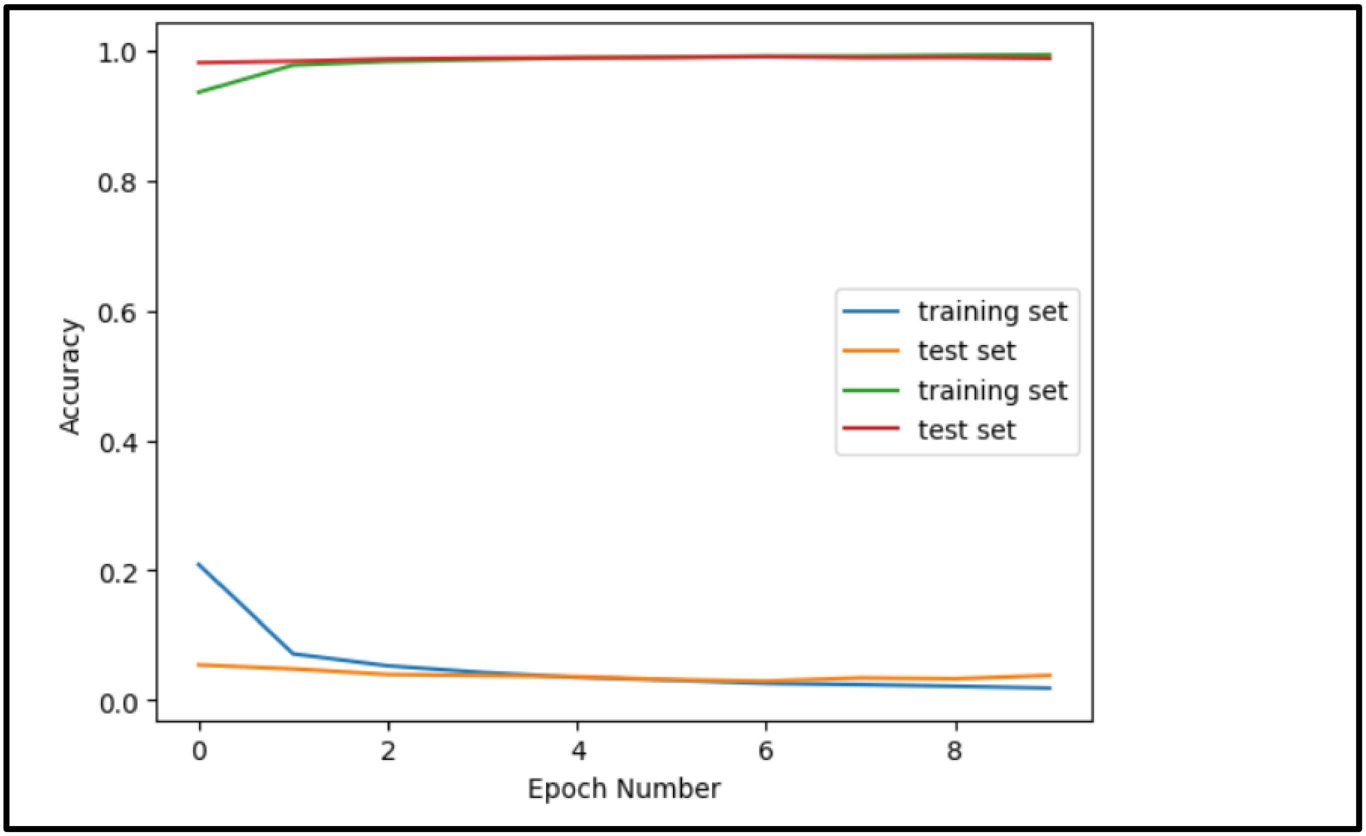



5 0 4 1 9
2 1 3 1 4
3 5 3 6 1


7 2 8 6 9
4 0 9 1 1

```
x_train_with_chanels: (60000, 28, 28, 1)
x_test_with_chanels: (10000, 28, 28, 1)
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 24, 24, 8)         208

 max_pooling2d (MaxPooling2D  (None, 12, 12, 8)        0
 )

 conv2d_1 (Conv2D)           (None, 8, 8, 16)          3216

 max_pooling2d_1 (MaxPooling  (None, 4, 4, 16)         0
 2D)

 flatten (Flatten)           (None, 256)               0

 dense (Dense)               (None, 128)               32896

 dropout (Dropout)           (None, 128)               0

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 37,610
Trainable params: 37,610
Non-trainable params: 0
```

```
Total params: 37,610
Trainable params: 37,610
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
1875/1875 [==============================] - 71s 36ms/step - loss: 0.2090 - accuracy: 0.9358 - val_loss: 0.0545 - val_accuracy: 0.9815
Epoch 2/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.0715 - accuracy: 0.9778 - val_loss: 0.0483 - val_accuracy: 0.9838
Epoch 3/10
1875/1875 [==============================] - 41s 22ms/step - loss: 0.0530 - accuracy: 0.9835 - val_loss: 0.0397 - val_accuracy: 0.9872
Epoch 4/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.0428 - accuracy: 0.9862 - val_loss: 0.0380 - val_accuracy: 0.9886
Epoch 5/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.0351 - accuracy: 0.9888 - val_loss: 0.0363 - val_accuracy: 0.9893
Epoch 6/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.0308 - accuracy: 0.9903 - val_loss: 0.0316 - val_accuracy: 0.9895
Epoch 7/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.0259 - accuracy: 0.9917 - val_loss: 0.0294 - val_accuracy: 0.9909
Epoch 8/10
1875/1875 [==============================] - 40s 21ms/step - loss: 0.0239 - accuracy: 0.9918 - val_loss: 0.0343 - val_accuracy: 0.9892
Epoch 9/10
1875/1875 [==============================] - 38s 20ms/step - loss: 0.0214 - accuracy: 0.9930 - val_loss: 0.0330 - val_accuracy: 0.9895
Epoch 10/10
1875/1875 [==============================] - 39s 21ms/step - loss: 0.0185 - accuracy: 0.9940 - val_loss: 0.0382 - val_accuracy: 0.9881
```



Conclusion/Outcome:

Thus we have implemented CNN model for digit recognition

Marks & Signature:

R1 (4 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (3 Mark)	Total (15 Marks)	Signature

Date of Performance :

Date of Submission :

EXPERIMENT NUMBER: 7

Aim: Design and implement LSTM for Sentiment Analysis

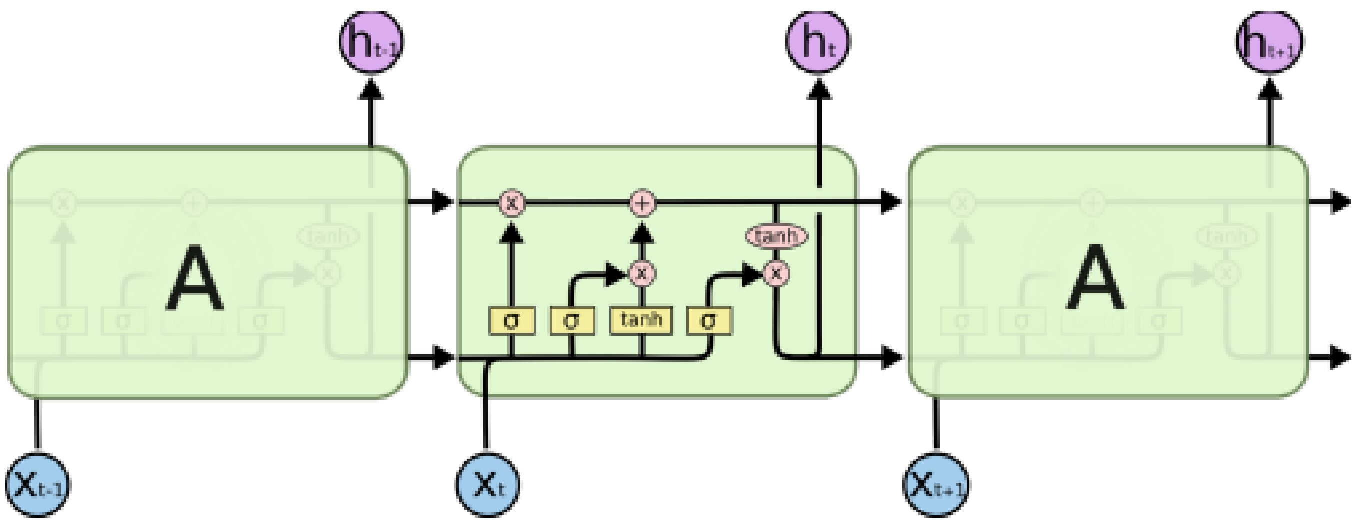
Objective:

Students will be designed appropriate DNN model for supervised, unsupervised and sequence learning applications.

Software Used : Python

Theory:

- Long Short Term Memory networks (LSTM)



LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn.LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way.
LSTM has three gates:
Forget gate: In a cell of the LSTM neural network, the first step is to decide whether we should keep the information from the previous time step or forget it.

Input Gate: The input gate is used to quantify the importance of the new information carried by the input.

Output Gate: Its value will also lie between 0 and 1 because of this sigmoid function. If you need to take the output of the current timestamp, just apply the SoftMax activation on hidden state Ht.

Algorithm:

1) download `IMDB Dataset from keras`
2) Split data in training and testing part.
3) Save image parameters to the constants that we will use later for data re-shaping and for model traning.
4) Normalized data
5) Add convolutional layer
6) Add max pooling layer
7) Add convolutional layer
8) Add max pooling layer
9) Add Flatten layer
10) Add dropout layer and fully connected layer
11) Train model and find accuracy from confusion matrix

Program:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
import warnings
# warnings.filterwarnings("ignore")
from google.colab import drive
drive.mount("/content/gdrive")
df = pd.read_csv("/content/gdrive/My Drive/Colab
Notebooks/IMDB Dataset.csv")
df.head(10)
print("Summary statistics of numerical features : \n",
df.describe())
print("======================================================
=======================") print("\nTotal number of
reviews: ",len(df))
print("======================================================
=======================") print("\nTotal number of
Sentiments: ", len(list(set(df['sentiment']))))
df['sentiment'] = np.where(df['sentiment'] ==
"positive", 1, 0) df
df = df.sample(frac=0.1, random_state=0) #uncomment to
```

```
use full set of data
# Drop missing values
df.dropna(inplace=True)
df
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
train_test_split(df['review'], df['sentiment'], \
test_size=0.1,
random_state=0)
print('Load %d training examples and %d validation
examples. \n' %(X_train.shape[0],X_test.shape[0]))
print('Show a review in the training set : \n',
X_train.iloc[10])
X_train,y_train
def cleanText(raw_text, remove_stopwords=False,
stemming=False, split_text=False, \
):
'''
Convert a raw review to a cleaned review
'''
text = BeautifulSoup(raw_text,
'html.parser').get_text()
letters_only = re.sub("[^a-zA-Z]", " ", text)
words = letters_only.lower().split()
if remove_stopwords:
stops = set(stopwords.words("english"))
words = [w for w in words if not w in stops]
if stemming==True:
stemmer = SnowballStemmer('english')
words = [stemmer.stem(w) for w in words]
if split_text==True:
return (words)
return( " ".join(words))
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from nltk.stem import SnowballStemmer,
```

```
WordNetLemmatizer
from nltk import sent_tokenize, word_tokenize, pos_tag
from bs4 import BeautifulSoup
import logging
from wordcloud import WordCloud
from gensim.models import word2vec
from gensim.models import Word2Vec
from gensim.models.keyedvectors import KeyedVectors
X_train_cleaned = []
X_test_cleaned = []
for d in X_train:
X_train_cleaned.append(cleanText(d))
print('Show a cleaned review in the training set : \n',
X_train_cleaned[10])
for d in X_test:
X_test_cleaned.append(cleanText(d))
from sklearn.feature_extraction.text import
CountVectorizer,TfidfVectorizer from
sklearn.naive_bayes import BernoulliNB, MultinomialNB
countVect = CountVectorizer()
X_train_countVect =
countVect.fit_transform(X_train_cleaned) print("Number
of features : %d \n"
%len(countVect.get_feature_names_out())) #6378
print("Show some feature names : \n",
countVect.get_feature_names_out()[::1000])
# Train MultinomialNB classifier
mnb = MultinomialNB()
mnb.fit(X_train_countVect, y_train)
import pickle
pickle.dump(countVect,open('countVect_imdb.pkl','wb'))
from sklearn import metrics
from sklearn.metrics import
accuracy_score,roc_auc_score
def modelEvaluation(predictions):
'''
Print model evaluation to predicted result
'''
```

```
print ("\nAccuracy on validation set:
{:.4f}".format(accuracy_score(y_test, predictions)))
print("\nAUC score :
{:.4f}".format(roc_auc_score(y_test, predictions)))
print("\nClassification report : \n",
metrics.classification_report(y_test, predictions))
print("\nConfusion Matrix : \n",
metrics.confusion_matrix(y_test, predictions))
predictions =
mnb.predict(countVect.transform(X_test_cleaned))
modelEvaluation(predictions)
import pickle
pickle.dump(mnb,open('Naive_Bayes_model_imdb.pkl','wb')
)
from sklearn.linear_model import LogisticRegression
tfidf = TfidfVectorizer(min_df=5) #minimum document
frequency of 5 X_train_tfidf =
tfidf.fit_transform(X_train)
print("Number of features : %d \n"
%len(tfidf.get_feature_names_out())) #1722 print("Show
some feature names : \n",
tfidf.get_feature_names_out()[::1000])
# Logistic Regression
lr = LogisticRegression()
lr.fit(X_train_tfidf, y_train)
feature_names = np.array(tfidf.get_feature_names_out())
sorted_coef_index = lr.coef_[0].argsort()
print('\nTop 10 features with smallest coefficients
:\n{}\n'.format(feature_names[sorted_coef_index[:10]]))
print('Top 10 features with largest coefficients :
\n{}'.format(feature_names[sorted_coef_index[:-11:-1]])
)
predictions =
lr.predict(tfidf.transform(X_test_cleaned))
modelEvaluation(predictions)
from sklearn.model_selection import GridSearchCV
from sklearn import metrics
from sklearn.metrics import roc_auc_score,
```

```
accuracy_score from sklearn.pipeline import Pipeline
estimators = [("tfidf", TfidfVectorizer()), ("lr",
LogisticRegression())] model = Pipeline(estimators)
params = {"lr__C":[0.1, 1, 10],
"tfidf__min_df": [1, 3],
"tfidf__max_features": [1000, None],
"tfidf__ngram_range": [(1,1), (1,2)],
"tfidf__stop_words": [None, "english"]}
grid = GridSearchCV(estimator=model, param_grid=params,
scoring="accuracy", n_jobs=-1)
grid.fit(X_train_cleaned, y_train)
print("The best paramenter set is : \n",
grid.best_params_)
# Evaluate on the validaton set
predictions = grid.predict(X_test_cleaned)
modelEvaluation(predictions)
import nltk
nltk.download('punkt')
tokenizer =
nltk.data.load('tokenizers/punkt/english.pickle') def
parseSent(review, tokenizer, remove_stopwords=False):
raw_sentences = tokenizer.tokenize(review.strip())
sentences = []
for raw_sentence in raw_sentences:
if len(raw_sentence) > 0:
sentences.append(cleanText(raw_sentence,
remove_stopwords, split_text=True))
return sentences
# Parse each review in the training set into sentences
sentences = []
for review in X_train_cleaned:
sentences += parseSent(review,
tokenizer,remove_stopwords=False)
print('%d parsed sentence in the training set\n'
%len(sentences)) print('Show a parsed sentence in the
training set : \n', sentences[10]) from wordcloud
import WordCloud
from gensim.models import word2vec
```

```
from gensim.models.keyedvectors import KeyedVectors
num_features = 300 #embedding dimension
min_word_count = 10
num_workers = 4
context = 10
downsampling = 1e-3
print("Training Word2Vec model ...\n")
w2v = Word2Vec(sentences, workers=num_workers,
vector_size=num_features, min_count = min_word_count,\
window = context, sample = downsampling)
w2v.init_sims(replace=True)
w2v.save("w2v_300features_10minwordcounts_10context")
#save trained word2vec model
#print("Number of words in the vocabulary list : %d
\n"
%len(w2v.wv.index_to_word)) #4016
#print("Show first 10 words in the vocalbulary list
vocabulary list: \n", w2v.wv.index_to_word[0:10])
def makeFeatureVec(review, model, num_features):
'''
Transform a review to a feature vector by averaging
feature vectors of words appeared in that review and in
the volcabulary list created
'''
featureVec = np.zeros((num_features,),dtype="float32")
nwords = 0.
index2word_set = set(model.wv.index2word) #index2word
is the volcabulary list of the Word2Vec model
isZeroVec = True
for word in review:
if word in index2word_set:
nwords = nwords + 1.
featureVec = np.add(featureVec, model[word])
isZeroVec = False
if isZeroVec == False:
featureVec = np.divide(featureVec, nwords)
return featureVec
def getAvgFeatureVecs(reviews, model, num_features):
```

```
'''
Transform all reviews to feature vectors using
makeFeatureVec() '''
counter = 0
reviewFeatureVecs =
np.zeros((len(reviews),num_features),dtype="float32")
for review in reviews:
reviewFeatureVecs[counter] = makeFeatureVec(review,
model,num_features) counter = counter + 1
return reviewFeatureVecs
import nltk
nltk.download('stopwords')
X_train_cleaned = []
for review in X_train:
X_train_cleaned.append(cleanText(review,
remove_stopwords=True, split_text=True))
trainVector = getAvgFeatureVecs(X_train_cleaned, w2v,
num_features) print("Training set : %d feature vectors
with %d dimensions" %trainVector.shape)
# Get feature vectors for validation set
X_test_cleaned = []
for review in X_test:
X_test_cleaned.append(cleanText(review,
remove_stopwords=True, split_text=True))
testVector = getAvgFeatureVecs(X_test_cleaned, w2v,
num_features) print("Validation set : %d feature
vectors with %d dimensions" %testVector.shape)
```

LSTM

```
#Step 1 : Prepare X_train and X_test to 2D tensor.

#Step 2 : Train a simple LSTM (embeddign layer => LSTM layer =>
dense layer). #Step 3 : Compile and fit the model using log loss
function and ADAM optimizer. !pip install matplotlib-venn

!apt-get -qq install -y libfluidsynth1

!apt-get -qq install -y libarchive-dev && pip install -U libarchive
import libarchive
```

```
!apt-get -qq install -y graphviz && pip install pydot

import pydot

!pip install cartopy

import cartopy

from keras.preprocessing import sequence

from keras.utils import np_utils

from keras.models import Sequential

from keras.layers.core import Dense, Dropout, Activation, Lambda
##from keras.layers.embeddings import Embedding

from tensorflow.keras.layers import Embedding

from keras.layers import LSTM

from keras.layers import SimpleRNN

from keras.layers import GRU

#from keras.layers.recurrent import LSTM, SimpleRNN, GRU

from keras.preprocessing.text import Tokenizer

from collections import defaultdict

from keras.layers.convolutional import Convolution1D

from keras import backend as K

##from keras.layers.embeddings import Embedding

top_words = 40000

maxlen = 200

batch_size = 62

nb_classes = 4

nb_epoch = 6

from keras.utils import pad_sequences

# Vectorize X_train and X_test to 2D tensor
```

```
tokenizer = Tokenizer(nb_words=top_words) #only consider top 20000
words in the corpse

tokenizer.fit_on_texts(X_train)

# tokenizer.word_index #access word-to-index dictionary of trained
tokenizer

sequences_train = tokenizer.texts_to_sequences(X_train)

sequences_test = tokenizer.texts_to_sequences(X_test)

X_train_seq = pad_sequences(sequences_train, maxlen=maxlen) X_test_seq
= pad_sequences(sequences_test, maxlen=maxlen)

# one-hot encoding of y_train and y_test

y_train_seq = np_utils.to_categorical(y_train, nb_classes) y_test_seq
= np_utils.to_categorical(y_test, nb_classes)

print('X_train shape:', X_train_seq.shape)

print("========================================")

print('X_test shape:', X_test_seq.shape)

print("========================================")

print('y_train shape:', y_train_seq.shape)

print("========================================")

print('y_test shape:', y_test_seq.shape)

print("========================================")

model1 = Sequential()

model1.add(Embedding(top_words, 128))

model1.add(Dropout(0.2))

model1.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model1.add(Dense(nb_classes))

model1.add(Activation('softmax'))

model1.summary()
```

```
model1.compile(loss='binary_crossentropy',

optimizer='adam',

metrics=['accuracy'])

model1.fit(X_train_seq, y_train_seq, batch_size=batch_size,
epochs=nb_epoch, verbose=1)

# Model evluation

score = model1.evaluate(X_test_seq, y_test_seq, batch_size=batch_size)


print('Test loss : {:.4f}'.format(score[0]))

print('Test accuracy : {:.4f}'.format(score[1]))

len(X_train_seq),len(y_train_seq)

print("Size of weight matrix in the embedding layer : ", \

model1.layers[0].get_weights()[0].shape)

# get weight matrix of the hidden layer

print("Size of weight matrix in the hidden layer : ", \

model1.layers[0].get_weights()[0].shape)

# get weight matrix of the output layer

print("Size of weight matrix in the output layer : ", \

model1.layers[2].get_weights()[0].shape)

import pickle

pickle.dump(model1,open('model1.pkl','wb'))

v2swe = Word2Vec.load("w2v_300features_10minwordcounts_10context")

embedding_matrix = w2v.wv.vectors

print("Shape of embedding matrix : ", embedding_matrix.shape)
top_words = embedding_matrix.shape[0] #4016

maxlen = 300
```

```
batch_size = 62

nb_classes = 4

nb_epoch = 7

# Vectorize X_train and X_test to 2D tensor

tokenizer = Tokenizer(nb_words=top_words) #only consider top 20000
words in the corpse

tokenizer.fit_on_texts(X_train)

# tokenizer.word_index #access word-to-index dictionary of trained
tokenizer sequences_train = tokenizer.texts_to_sequences(X_train)


sequences_test = tokenizer.texts_to_sequences(X_test)

X_train_seq1 = pad_sequences(sequences_train, maxlen=maxlen)
X_test_seq1 = pad_sequences(sequences_test, maxlen=maxlen)

# one-hot encoding of y_train and y_test

y_train_seq1 = np_utils.to_categorical(y_train, nb_classes)
y_test_seq1 = np_utils.to_categorical(y_test, nb_classes)

print('X_train shape:', X_train_seq1.shape)

print("========================================") print('X_test
shape:', X_test_seq1.shape)

print("========================================") print('y_train
shape:', y_train_seq1.shape)

print("========================================") print('y_test
shape:', y_test_seq1.shape)

print("========================================")
len(X_train_seq1),len(y_train_seq1)

embedding_layer = Embedding(embedding_matrix.shape[0], #4016
embedding_matrix.shape[1], #300

weights=[embedding_matrix])
```

```
model2 = Sequential()

model2.add(embedding_layer)

model2.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model2.add(Dense(nb_classes))

model2.add(Activation('softmax'))

model2.summary()

model2.compile(loss='binary_crossentropy',

optimizer='adam',

metrics=['accuracy'])

model2.fit(X_train_seq1, y_train_seq1, batch_size=batch_size,
epochs=nb_epoch, verbose=1)

# Model evaluation

score = model2.evaluate(X_test_seq1, y_test_seq1,
batch_size=batch_size) print('Test loss : {:.4f}'.format(score[0]))

print('Test accuracy : {:.4f}'.format(score[1]))

print("Size of weight matrix in the embedding layer : ", \

model2.layers[0].get_weights()[0].shape)

print("Size of weight matrix in the hidden layer : ", \

model2.layers[1].get_weights()[0].shape)

print("Size of weight matrix in the output layer : ", \

model2.layers[2].get_weights()[0].shape)
```

Output:

```
4500 parsed sentence in the training set

Show a parsed sentence in the training set :
 ['the', 'crimson', 'rivers', 'is', 'one', 'of', 'the', 'most', 'over', 'directed', 'over', 'the', 'top', 'over', 'e
verything', 'mess', 'i', 've', 'ever', 'seen', 'come', 'out', 'of', 'france', 'there', 's', 'nothing', 'worse', 'tha
n', 'a', 'french', 'production', 'trying', 'to', 'out', 'do', 'films', 'made', 'in', 'hollywood', 'and', 'cr', 'is',
'a', 'perfect', 'example', 'of', 'such', 'a', 'wannabe', 'horror', 'action', 'buddy', 'flick', 'i', 'almost', 'stopp
ed', 'it', 'halfway', 'through', 'because', 'i', 'knew', 'it', 'wouldn', 't', 'amount', 'to', 'anything', 'but', 'fr
ench', 'guys', 'trying', 'to', 'show', 'off', 'the', 'film', 'starts', 'off', 'promisingly', 'like', 'some', 'sort',
'of', 'expansive', 'horror', 'film', 'but', 'it', 'quickly', 'shifts', 'genres', 'from', 'horror', 'to', 'action', '
to', 'x', 'files', 'type', 'to', 'buddy', 'flick', 'that', 'in', 'the', 'end', 'cr', 'is', 'all', 'of', 'it', 'and',
'also', 'none', 'of', 'it', 'it', 's', 'so', 'full', 'of', 'clich', 's', 'that', 'at', 'one', 'point', 'i', 'thought
', 'the', 'whole', 'thing', 'was', 'a', 'comedy', 'the', 'painful', 'dialogue', 'and', 'those', 'silent', 'pauses',
'with', 'fades', 'outs', 'and', 'fades', 'ins', 'just', 'at', 'the', 'right', 'expositionary', 'moments', 'made', 'm
e', 'groan', 'i', 'thought', 'only', 'films', 'made', 'in', 'hollywood', 'used', 'this', 'hackneyed', 'technique', '
```

```
Classification report :
              precision    recall  f1-score   support

           0       0.87      0.87      0.87       249
           1       0.87      0.88      0.87       251

    accuracy                           0.87       500
   macro avg       0.87      0.87      0.87       500
weighted avg       0.87      0.87      0.87       500


Confusion Matrix :
 [[216  33]
 [ 31 220]]
```

Conclusion/Outcome: LSTM implemented for Sentiment Analysis

Marks & Signature:

R1 (4 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (3 Marks)	Total (15 Marks)	Signature

Date of Performance :

Date of Submission :

EXPERIMENT NUMBER: 8

Aim: **Implement a GAN model for Image generation or video generation.**

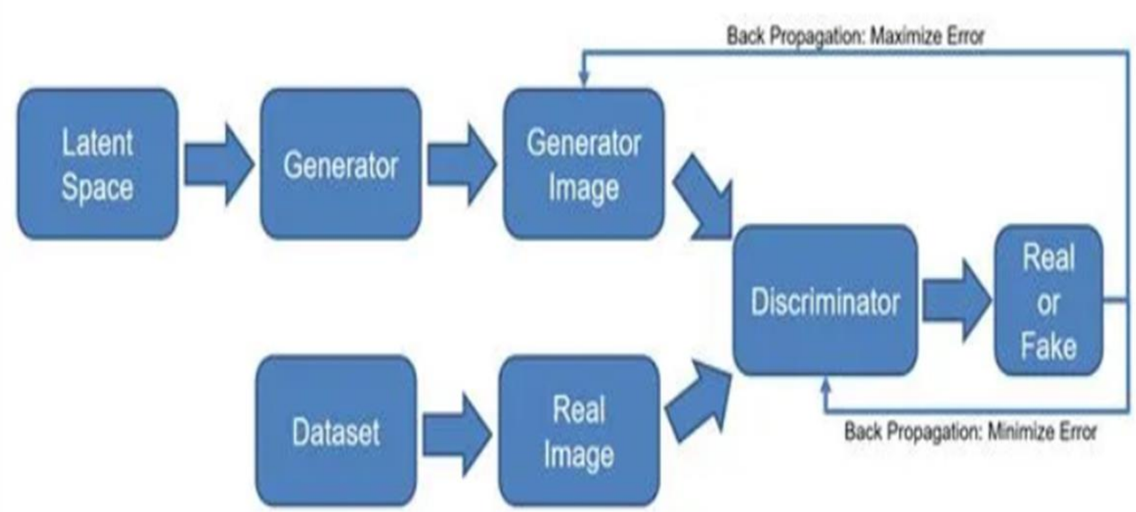
Objective:

Students will be gain familiarity with recent trends and applications of Deep Learning.

Software Used : Python

Theory:

- Generative Adversarial Network (GAN)



•The two neural networks that make up a GAN are referred to as the *generator* and the *discriminator*.

•The goal of the generator is to artificially manufacture outputs that could easily be mistaken for real data.

•The goal of the discriminator is to identify which of the outputs it receives have been artificially created.

•Generative models create their own training data. While the generator is trained to produce false data, the discriminator network is taught to distinguish between the generator's manufactured data and true examples.

• If the discriminator rapidly recognizes the fake data that the generator produces -- such as an image that isn't a human face -- the generator suffers a penalty.

• As the feedback_loop between the adversarial networks continues, the generator will begin to produce higher-quality and more believable output and the discriminator will become better at flagging data that has been artificially created.

Algorithm:

1) Load BigGAN generator module from TF Hub
2) sample truncated normal distribution based on seed and truncation parameter
3) using vectors of noise seeds and category labels, generate images
4) Create a TensorFlow session and initialize variables
5) Create video or images of interpolated BigGAN generator samples

Program:

```
# basics
import io
import os
import numpy as np

# deep learning
from scipy.stats import truncnorm
import tensorflow as tf
import tensorflow_hub as hub

# visualization
from IPython.core.display import HTML
#!pip install imageio
import imageio
import base64

# check that tensorflow GPU is enabled
tf.test.gpu_device_name() # returns empty string if using CPU
!pip install tensorflow==2.5.0
# comment out the TF Hub module path you would like to use
# module_path = 'https://tfhub.dev/deepmind/biggan-128/1'  # 128x128 BigGAN
# module_path = 'https://tfhub.dev/deepmind/biggan-256/1'  # 256x256 BigGAN
module_path = 'https://tfhub.dev/deepmind/biggan-512/1'  # 512x512 BigGAN
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()
tf.compat.v1.reset_default_graph()
##tf.reset_default_graph()
print('Loading BigGAN module from:', module_path)
module = hub.Module(module_path)
inputs = {k: tf.placeholder(v.dtype, v.get_shape().as_list(), k)
          for k, v in module.get_input_info_dict().items()}
output = module(inputs)
input_z = inputs['z']
input_y = inputs['y']
input_trunc = inputs['truncation']

dim_z = input_z.shape.as_list()[1]
vocab_size = input_y.shape.as_list()[1]
```

```
# sample truncated normal distribution based on seed and truncation parameter
def truncated_z_sample(truncation=1., seed=None):
    state = None if seed is None else np.random.RandomState(seed)
    values = truncnorm.rvs(-2, 2, size=(1, dim_z), random_state=state)
    return truncation * values

# convert `index` value to a vector of all zeros except for a 1 at `index`
def one_hot(index, vocab_size=vocab_size):
    index = np.asarray(index)
    if len(index.shape) == 0: # when it's a scale convert to a vector of size 1
        index = np.asarray([index])
    assert len(index.shape) == 1
    num = index.shape[0]
    output = np.zeros((num, vocab_size), dtype=np.float32)
    output[np.arange(num), index] = 1
    return output

def one_hot_if_needed(label, vocab_size=vocab_size):
    label = np.asarray(label)
    if len(label.shape) <= 1:
        label = one_hot(label, vocab_size)
    assert len(label.shape) == 2
    return label

# using vectors of noise seeds and category labels, generate images
def sample(sess, noise, label, truncation=1., batch_size=8,
vocab_size=vocab_size):
    noise = np.asarray(noise)
    label = np.asarray(label)
    num = noise.shape[0]
    if len(label.shape) == 0:
        label = np.asarray([label] * num)
    if label.shape[0] != num:
        raise ValueError('Got # noise samples ({}) != # label samples ({})'
                         .format(noise.shape[0], label.shape[0]))
    label = one_hot_if_needed(label, vocab_size)
    ims = []
    for batch_start in range(0, num, batch_size):
        s = slice(batch_start, min(num, batch_start + batch_size))
        feed_dict = {input_z: noise[s], input_y: label[s], input_trunc:
truncation}
```

```
        ims.append(sess.run(output, feed_dict=feed_dict))
    ims = np.concatenate(ims, axis=0)

    assert ims.shape[0] == num
    ims = np.clip(((ims + 1) / 2.0) * 256, 0, 255)
    ims = np.uint8(ims)
    return ims

def interpolate(a, b, num_interps):
    alphas = np.linspace(0, 1, num_interps)
    assert a.shape == b.shape, 'A and B must have the same shape to
interpolate.'
    return np.array([(1-x)*a + x*b for x in alphas])

def interpolate_and_shape(a, b, steps):
    interps = interpolate(a, b, steps)
    return (interps.transpose(1, 0, *range(2,
len(interps.shape)))).reshape(steps, -1))
initializer = tf.global_variables_initializer()
sess = tf.Session()
sess.run(initializer)
# category options: https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a
category = 947 # mushroom

# important parameter that controls how much variation there is
truncation = 0.2 # reasonable range: [0.02, 1]

seed_count = 10
clip_secs = 36

seed_step = int(100 / seed_count)
interp_frames = int(clip_secs * 30 / seed_count)  # interpolation frames

cat1 = category
cat2 = category
all_imgs = []

for i in range(seed_count):
    seed1 = i * seed_step # good range for seed is [0, 100]
    seed2 = ((i+1) % seed_count) * seed_step
```

```
    z1, z2 = [truncated_z_sample(truncation, seed) for seed in [seed1, seed2]]
    y1, y2 = [one_hot([category]) for category in [cat1, cat2]]

    z_interp = interpolate_and_shape(z1, z2, interp_frames)
    y_interp = interpolate_and_shape(y1, y2, interp_frames)


    imgs = sample(sess, z_interp, y_interp, truncation=truncation)

    all_imgs.extend(imgs[:-1])

# save the video for displaying in the next cell, this is way more space efficient than the gif animation
imageio.mimsave('gan.mp4', all_imgs, fps=30)
%%HTML
<video autoplay loop>
  <source src="gan.mp4" type="video/mp4">
</video>
```

Output:

Conclusion/Outcome:

Thus, we have implemented GAN model for Video generation

Marks & Signature:

R1 (4 Marks)	R2 (4 Marks)	R3 (4 Marks)	R4 (3 Mark)	Total (15 Marks)	Signature