Assignment No 1:

Aim:Classification with Multilayer Perceptron using Scikit-learn(MNIST Dataset)

```
!pip install -U scikit-learn

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.datasets import fetch_openml

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler

from sklearn.neural_network import MLPClassifier

from sklearn.metrics import classification_report, confusion_matrix

# Load MNIST dataset

mnist = fetch_openml('mnist_784', version=1)

# Split data and labels

X, y = mnist["data"], mnist["target"]

# Convert labels to integers

y = y.astype(np.int8)

# Split data into training and test sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the dataset (mean=0, variance=1)

scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)

X_test_scaled = scaler.transform(X_test)

# Create MLPClassifier model

mlp = MLPClassifier(hidden_layer_sizes=(64, 64), max_iter=20, alpha=1e-4,

                    solver='adam', verbose=10, random_state=1)

# Train the model

mlp.fit(X_train_scaled, y_train)

# Predict on test data

y_pred = mlp.predict(X_test_scaled)

# Classification report

print("Classification Report:\n", classification_report(y_test, y_pred))

# Confusion matrix

conf_matrix = confusion_matrix(y_test, y_pred)
```

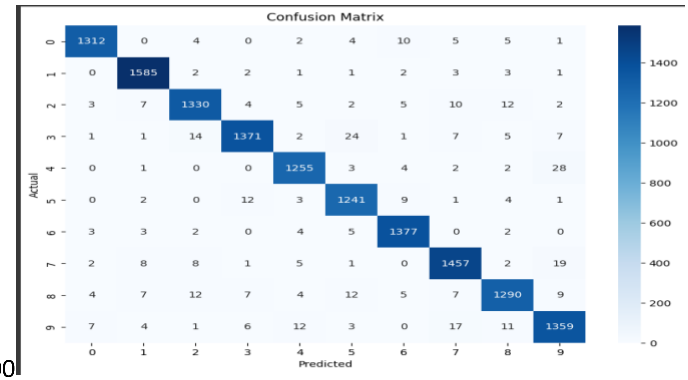
```python
# Plot confusion matrix using seaborn heatmap
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=range(10), yticklabels=range(10))
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
# Reshape the original X_test (not scaled) to (n_samples, 28, 28)
X_test_images = X_test.to_numpy().reshape(-1, 28, 28)  # Reshape to (n_samples, 28, 28)
# Plotting some test images with predictions
fig, axes = plt.subplots(2, 5, figsize=(10, 5))  # Create a 2x5 grid of subplots
for i, ax in enumerate(axes.flat):
    ax.imshow(X_test_images[i], cmap='gray')  # Display the image in grayscale
    ax.set_title(f"True: {y_test.iloc[i]}\nPred: {y_pred[i]}")  # Set the title with true and predicted labels
    ax.axis('off')  # Hide axis lines and labels
plt.show()  # Display the plot
```

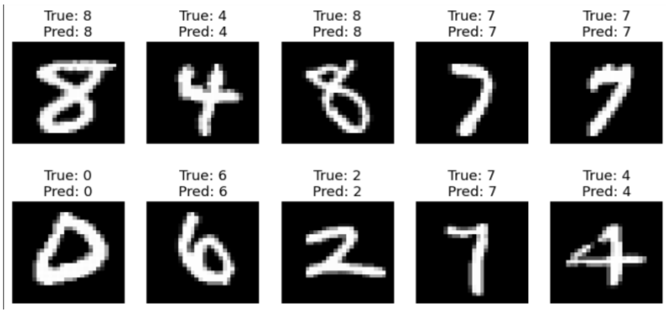
OUTPUT:

```
Iteration 1, loss = 0.38803361
Iteration 2, loss = 0.14476268
Iteration 3, loss = 0.09887330
Iteration 4, loss = 0.07339894
Iteration 5, loss = 0.05628719
Iteration 6, loss = 0.04401882
Iteration 7, loss = 0.03494394
Iteration 8, loss = 0.02853816
Iteration 9, loss = 0.02356742
Iteration 10, loss = 0.01844750
Classification Report:
```

	precision	recall	f1-score	support
0	0.98	0.98	0.98	1343
1	0.98	0.99	0.99	1600
2	0.97	0.96	0.97	1380
3	0.98	0.96	0.97	1433
4	0.97	0.97	0.97	1295
5	0.96	0.97	0.97	1273
6	0.97	0.99	0.98	1396
7	0.97	0.97	0.97	1503

8	0.97	0.95	0.96	1357
9	0.95	0.96	0.95	1420
accuracy			0.97	14000
macro avg	0.97	0.97	0.97	14000
weighted avg	0.97	0.97	0.97	14000





Assignment No 2:

Aim/Problem Statement:Fashion MNIST classification of MNIST Dataset using CNN

Import necessary libraries

import tensorflow as tf

```
from tensorflow.keras import datasets, layers, models, callbacks

import matplotlib.pyplot as plt

# Load and preprocess the MNIST dataset

(train_images, train_labels), (test_images, test_labels) = datasets.mnist.load_data()

# Normalize pixel values to between 0 and 1

train_images = train_images / 255.0

test_images = test_images / 255.0

# Reshape the images to (28, 28, 1) to match the input shape for the CNN

train_images = train_images.reshape((train_images.shape[0], 28, 28, 1))

test_images = test_images.reshape((test_images.shape[0], 28, 28, 1))

# Build the CNN model with more layers and Dropout for regularization

model = models.Sequential()

# First Convolutional Layer + MaxPooling

model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))

model.add(layers.MaxPooling2D((2, 2)))

# Second Convolutional Layer + MaxPooling

model.add(layers.Conv2D(64, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

# Third Convolutional Layer + MaxPooling

model.add(layers.Conv2D(128, (3, 3), activation='relu'))

model.add(layers.MaxPooling2D((2, 2)))

# Flatten the output before feeding into Dense layers

model.add(layers.Flatten())

# Fully connected (Dense) layers with Dropout to prevent overfitting

model.add(layers.Dense(128, activation='relu'))

model.add(layers.Dropout(0.5))  # Dropout layer with 50% dropout rate

# Output layer (softmax) for multi-class classification

model.add(layers.Dense(10, activation='softmax'))

# Compile the model with Adam optimizer, categorical crossentropy loss

model.compile(optimizer='adam',

          loss='sparse_categorical_crossentropy',

          metrics=['accuracy'])

# Early stopping callback to prevent overfitting

early_stopping = callbacks.EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)

# Train the model with validation data and early stopping

history = model.fit(train_images, train_labels, epochs=20,
```

```
             validation_data=(test_images, test_labels),
 callbacks=[early_stopping])
# Evaluate the model on test data
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Test Accuracy: {test_acc:.4f}")
# Plot training and validation accuracy over epochs
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.show()

# Optional: Test the model by making predictions on test images
predictions = model.predict(test_images)

# Display the first prediction and the corresponding actual label
print(f"Predicted label: {predictions[0].argmax()}, Actual label: {test_labels[0]}")
```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 76s 37ms/step - accuracy: 0.8207 - loss: 0.5559 - val_accuracy: 0.9787 - val_loss: 0.0682
Epoch 2/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 72s 38ms/step - accuracy: 0.9722 - loss: 0.0957 - val_accuracy: 0.9805 - val_loss: 0.0703
Epoch 3/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 75s 35ms/step - accuracy: 0.9826 - loss: 0.0631 - val_accuracy: 0.9832 - val_loss: 0.0587
Epoch 4/20
```

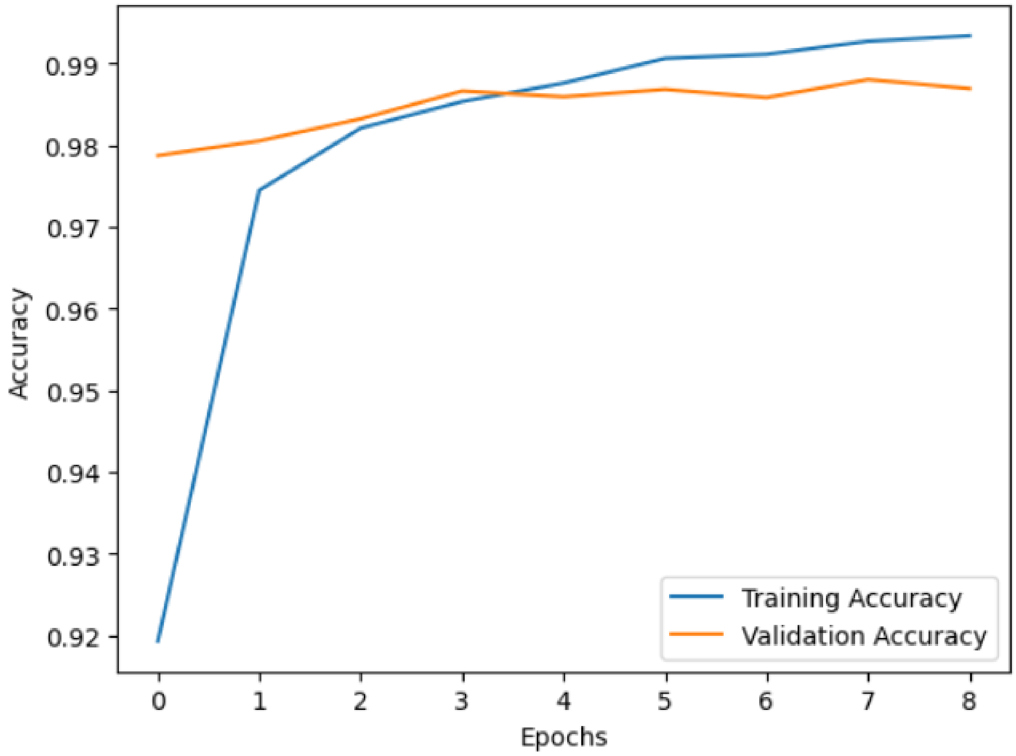
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 64s 34ms/step - accuracy: 0.9856 - loss: 0.0492 - val_accuracy: 0.9866 - val_loss: 0.0474

Epoch 5/20

1875/1875 ━━━━━━━━━━━━━━━━━━━━ 81s 34ms/step - accuracy: 0.9875 - loss: 0.0405 - val_accuracy: 0.9859 - val_loss: 0.0532

313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 15ms/step - accuracy: 0.9837 - loss: 0.0587

Test Accuracy: 0.9868



Assignment No:4

Aim/Problem Statement:Time Series Analysis with LSTM using python's keras Library

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
import matplotlib.pyplot as plt

# Generate synthetic time series data
def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
    time = np.linspace(0, 1, n_steps)
```

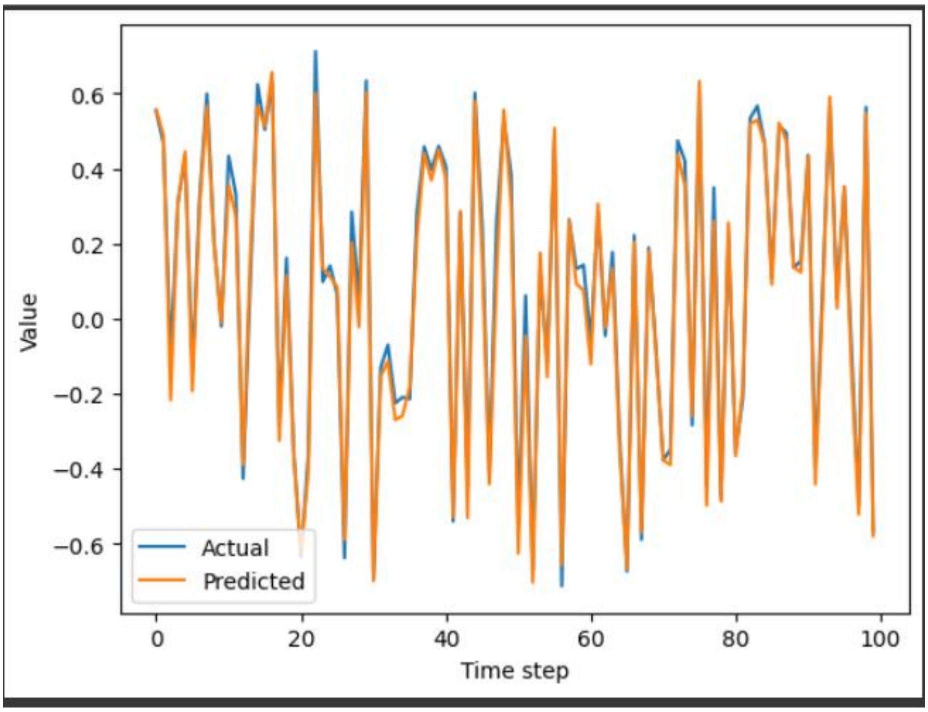
```
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10))  # wave 1
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20))  # wave 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5)    # noise
    return series[..., np.newaxis].astype(np.float32)
# Prepare the data
n_steps = 50
X_train = generate_time_series(10000, n_steps)
X_valid = generate_time_series(2000, n_steps)
X_test = generate_time_series(2000, n_steps)
# We are predicting the next value in the time series, so y is shifted by one time step.
y_train = X_train[:, -1, 0]  # predicting the last value of the series
y_valid = X_valid[:, -1, 0]
y_test = X_test[:, -1, 0]
# Build a 1D CNN model for time series prediction
model = models.Sequential()
# 1D Convolutional Layer for time series data
model.add(layers.Conv1D(filters=64, kernel_size=5, strides=1, padding='causal', activation='relu', input_shape=[n_steps, 1]))
model.add(layers.Conv1D(filters=64, kernel_size=5, strides=1, padding='causal', activation='relu'))
# Flatten the output and pass it to a Dense layer
model.add(layers.GlobalAveragePooling1D())
model.add(layers.Dense(1))
# Compile the model
model.compile(optimizer='adam', loss='mse')
# Train the model
history = model.fit(X_train, y_train, epochs=20, validation_data=(X_valid, y_valid))
# Evaluate the model
test_loss = model.evaluate(X_test, y_test)
print(f'Test Loss: {test_loss}')
# Plot the training and validation loss
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
# Make predictions on the test data
```

```
y_pred = model.predict(X_test)

# Plot the predicted vs actual values for the first time series in the test set

plt.plot(y_test[:100], label="Actual")

plt.plot(y_pred[:100], label="Predicted")

plt.xlabel('Time step')

plt.ylabel('Value')

plt.legend()

plt.show()
```

OUTPUT:

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

Epoch 1/20

313/313 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - loss: 0.1116 - val_loss: 0.0728

Epoch 2/20

313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 0.0695 - val_loss: 0.0457

Epoch 3/20

313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 0.0386 - val_loss: 0.0193

Epoch 4/20

313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 11ms/step - loss: 0.0203 - val_loss: 0.0172

Epoch 5/20

313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - loss: 0.0148 - val_loss: 0.0185

Epoch 6/20

313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - loss: 0.0133 - val_loss: 0.0101

Epoch 7/20

313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 12ms/step - loss: 0.0109 - val_loss: 0.0100

Epoch 8/20

313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 0.0093 - val_loss: 0.0128

Epoch 9/20

313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 0.0078 - val_loss: 0.0061

Epoch 10/20

313/313 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - loss: 0.0074 - val_loss: 0.0056

Epoch 11/20

313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 0.0058 - val_loss: 0.0056
```

Test Loss: 0.002182631054893136



Assignment 3:

Aim/Problem Statement:Face Recognition using Deep Learning CNN in python

Step 1:

```
import keras
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout
from keras.optimizers import Adam
from keras.callbacks import TensorBoard

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import accuracy_score
from keras.utils import np_utils
```

```
import itertools
```

Step 2:

```
#load dataset
data = np.load('ORL_faces.npz')
# load the "Train Images"
x_train = data['trainX']
#normalize every image
x_train = np.array(x_train,dtype='float32')/255
x_test = data['testX']
x_test = np.array(x_test,dtype='float32')/255

# load the Label of Images
y_train= data['trainY']
y_test= data['testY']
# show the train and test Data format
print('x_train : {}'.format(x_train[:]))
print('Y-train shape: {}'.format(y_train))
print('x_test shape: {}'.format(x_test.shape))
```

OUTPUT:

```
x_train : [[0.1882353  0.19215687 0.1764706  ... 0.18431373 0.18039216 0.18039216]
 [0.23529412 0.23529412 0.24313726 ... 0.1254902  0.13333334 0.13333334]
 [0.15294118 0.17254902 0.20784314 ... 0.11372549 0.10196079 0.11372549]
 ...
 [0.44705883 0.45882353 0.44705883 ... 0.38431373 0.3764706  0.38431373]
 [0.4117647  0.4117647  0.41960785 ... 0.21176471 0.18431373 0.16078432]
 [0.45490196 0.44705883 0.45882353 ... 0.37254903 0.39215687 0.39607844]]
Y-train shape: [ 0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  1  1
  2  2  2  2  2  2  2  2  2  2  2  2  3  3  3  3  3  3  3  3  3  3  3  3
  4  4  4  4  4  4  4  4  4  4  4  4  5  5  5  5  5  5  5  5  5  5  5  5
  6  6  6  6  6  6  6  6  6  6  6  6  7  7  7  7  7  7  7  7  7  7  7  7
  8  8  8  8  8  8  8  8  8  8  8  8  9  9  9  9  9  9  9  9  9  9  9  9
 10 10 10 10 10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11 11 11 11 11
 12 12 12 12 12 12 12 12 12 12 12 12 13 13 13 13 13 13 13 13 13 13 13 13
 14 14 14 14 14 14 14 14 14 14 14 14 15 15 15 15 15 15 15 15 15 15 15 15
```

 16 16 16 16 16 16 16 16 16 16 16 16 17 17 17 17 17 17 17 17 17 17 17 17

 18 18 18 18 18 18 18 18 18 18 18 18 19 19 19 19 19 19 19 19 19 19 19 19]

X_test shape: (160, 10304)

Step 3:

```
x_train, x_valid, y_train, y_valid= train_test_split(x_train, y_train, test_size=.05, random_state=1234,)
```

Step 3:

```
im_rows=112
im_cols=92
batch_size=512
im_shape=(im_rows, im_cols, 1)

#change the size of images
x_train = x_train.reshape(x_train.shape[0], *im_shape)
x_test = x_test.reshape(x_test.shape[0], *im_shape)
x_valid = x_valid.reshape(x_valid.shape[0], *im_shape)

print('x_train shape: {}'.format(y_train.shape[0]))
print('x_test shape: {}'.format(y_test.shape))
```

OUTPUT:

x_train shape: 228

x_test shape: (160,)

Step 4:

```
cnn_model= Sequential([
   Conv2D(filters=36, kernel_size=7, activation='relu', input_shape= im_shape),
   MaxPooling2D(pool_size=2),
   Conv2D(filters=54, kernel_size=5, activation='relu', input_shape= im_shape),
   MaxPooling2D(pool_size=2),
   Flatten(),
   Dense(2024, activation='relu'),
    Dropout(0.5),
   Dense(1024, activation='relu'),
   Dropout(0.5),
```

```
    Dense(512, activation='relu'),
    Dropout(0.5),
    #20 is the number of outputs
    Dense(20, activation='softmax')
])

cnn_model.compile(
    loss='sparse_categorical_crossentropy',#'categorical_crossentropy',
    optimizer=Adam(lr=0.0001),
    metrics=['accuracy']
)
```

```
/usr/local/lib/python3.8/dist-packages/keras/optimizers/optimizer_v2/adam.py:110: UserWarning: The `lr` argument is deprecated, use `learning_rate` instead.
  super(Adam, self).__init__(name, **kwargs)
```

```
cnn_model.summary()
```

OUTPUT:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 106, 86, 36)       1800

 max_pooling2d (MaxPooling2D  (None, 53, 43, 36)       0
 )

 conv2d_1 (Conv2D)           (None, 49, 39, 54)        48654

 max_pooling2d_1 (MaxPooling  (None, 24, 19, 54)       0
 2D)

 flatten (Flatten)           (None, 24624)             0

 dense (Dense)               (None, 2024)              49841000

 dropout (Dropout)           (None, 2024)              0
```

```
 dense_1 (Dense)           (None, 1024)           2073600

 dropout_1 (Dropout)       (None, 1024)           0

 dense_2 (Dense)           (None, 512)            524800

 dropout_2 (Dropout)       (None, 512)            0

 dense_3 (Dense)           (None, 20)             10260

=================================================================
Total params: 52,500,114
Trainable params: 52,500,114
Non-trainable params: 0
```

Step 5:

```
history=cnn_model.fit(
    np.array(x_train), np.array(y_train), batch_size=512,
    epochs=250, verbose=2,
    validation_data=(np.array(x_valid),np.array(y_valid)),
)
```

OUTPUT:

```
Epoch 1/250
1/1 - 10s - loss: 3.0025 - accuracy: 0.0439 - val_loss: 2.9890 - val_accuracy: 0.0833 - 10s/epoch - 10s/step
Epoch 2/250
1/1 - 9s - loss: 2.9947 - accuracy: 0.0702 - val_loss: 2.9775 - val_accuracy: 0.0833 - 9s/epoch - 9s/step
Epoch 3/250
1/1 - 8s - loss: 3.0263 - accuracy: 0.0658 - val_loss: 2.9745 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 4/250
1/1 - 8s - loss: 2.9759 - accuracy: 0.0789 - val_loss: 2.9739 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
Epoch 5/250
1/1 - 8s - loss: 2.9693 - accuracy: 0.1009 - val_loss: 2.9740 - val_accuracy: 0.2500 - 8s/epoch - 8s/step
Epoch 6/250
1/1 - 8s - loss: 2.9890 - accuracy: 0.0526 - val_loss: 2.9724 - val_accuracy: 0.0833 - 8s/epoch - 8s/step
```

Epoch 7/250

1/1 - 8s - loss: 2.9638 - accuracy: 0.0965 - val_loss: 2.9751 - val_accuracy: 0.0833 - 8s/epoch - 8s/step

Epoch 8/250

1/1 - 8s - loss: 2.9988 - accuracy: 0.0439 - val_loss: 2.9756 - val_accuracy: 0.0833 - 8s/epoch - 8s/step

Epoch 9/250

1/1 - 8s - loss: 2.9734 - accuracy: 0.1053 - val_loss: 2.9770 - val_accuracy: 0.0833 - 8s/epoch - 8s/step

Epoch 10/250

1/1 - 8s - loss: 2.9736 - accuracy: 0.0570 - val_loss: 2.9777 - val_accuracy: 0.0833 - 8s/epoch - 8s/step

Epoch 11/250

1/1 - 8s - loss: 2.9675 - accuracy: 0.0746 - val_loss: 2.9778 - val_accuracy: 0.0833 - 8s/epoch - 8s/step

Epoch 12/250

Step 6:

```
scor = cnn_model.evaluate( np.array(x_test),  np.array(y_test), verbose=0)

print('test los {:.4f}'.format(scor[0]))
print('test acc {:.4f}'.format(scor[1]))
```

OUTPUT:

test los 0.3214

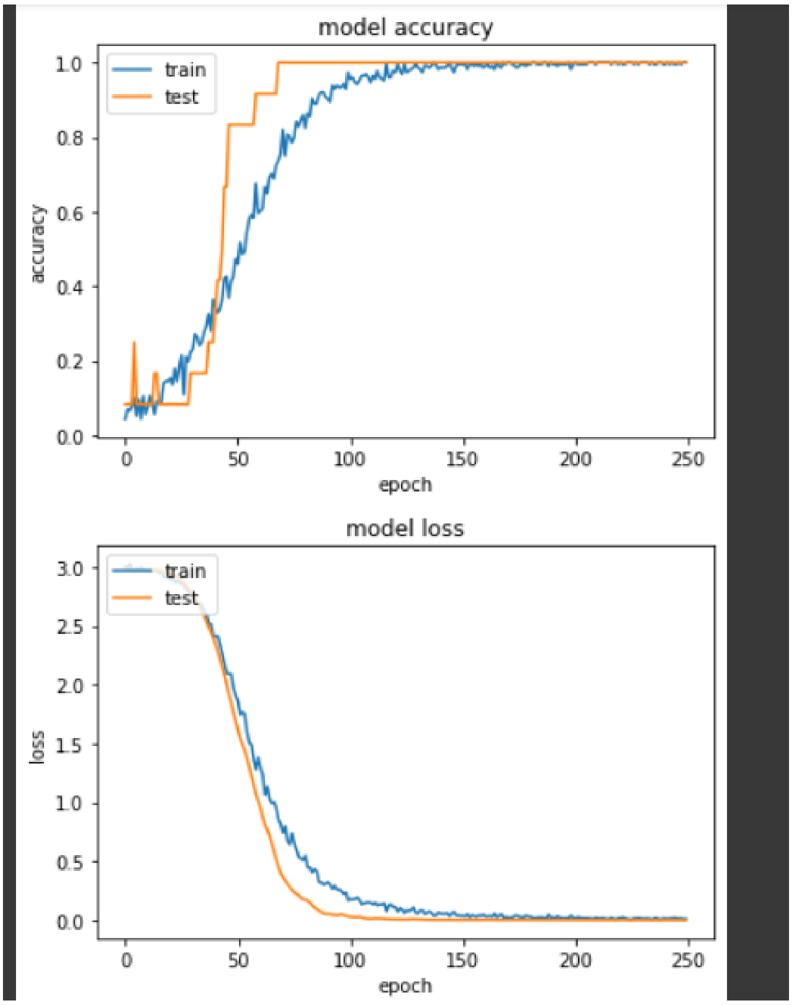
test acc 0.9500

step 7:

```
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
```

```
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

OUTPUT:



Step 8:

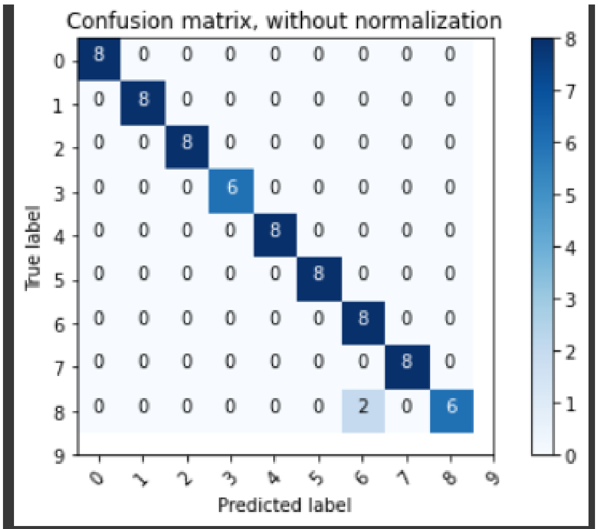
```
predicted =np.array( cnn_model.predict(x_test))
print(predicted)
print(y_test)
ynew = np.argmax(cnn_model.predict(x_test), axis=-1)
Acc=accuracy_score(y_test, ynew)
print("accuracy : ")
print(Acc)
#/tn, fp, fn, tp = confusion_matrix(np.array(y_test), ynew).ravel()
```

```
cnf_matrix=confusion_matrix(np.array(y_test), ynew)

y_test1 = np_utils.to_categorical(y_test, 20)

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        #print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
    #print(cm)
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)
fmt = '.2f' if normalize else 'd'
thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
      horizontalalignment="center",
      color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
 plt.show()
print('Confusion matrix, without normalization')
print(cnf_matrix)
plt.figure()
```

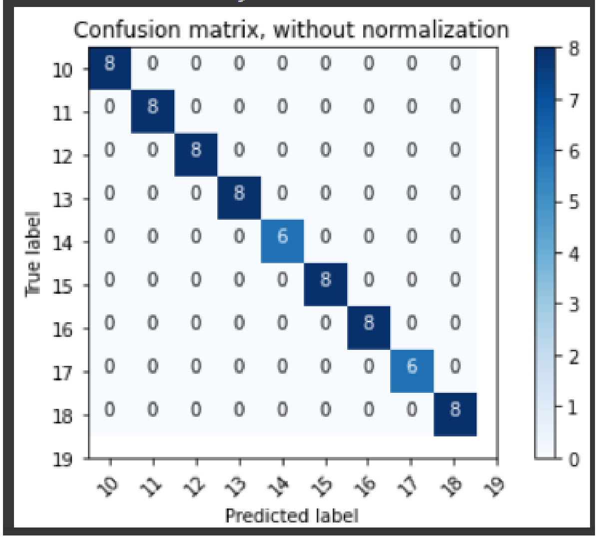
```
plot_confusion_matrix(cnf_matrix[1:10,1:10], classes=[0,1,2,3,4,5,6,7,8,9],
                      title='Confusion matrix, without normalization')
plt.figure()
plot_confusion_matrix(cnf_matrix[11:20,11:20], classes=[10,11,12,13,14,15,16,17,18,19],
                      title='Confusion matrix, without normalization')

print("Confusion matrix:\n%s" % confusion_matrix(np.array(y_test), ynew))
print(classification_report(np.array(y_test), ynew))
```

OUTPUT:

```
5/5 [==============================] - 2s 322ms/step
[[9.9546921e-01 1.9353812e-04 7.9316116e-08 ... 3.3380311e-05
  3.6349343e-03 3.7619247e-07]
 [9.6505862e-01 4.4584442e-05 4.9042995e-08 ... 2.2253296e-05
  2.3999320e-04 3.0376427e-08]
 [9.8446137e-01 2.5217005e-05 1.2214008e-06 ... 3.2328474e-04
  7.6428393e-04 7.3747998e-07]
 ...
 [3.5669402e-07 1.3151069e-05 6.9416594e-03 ... 3.9353850e-04
  8.3770874e-07 9.1356450e-01]
 [8.1324352e-08 3.1203381e-06 4.9745995e-03 ... 4.6892710e-05
  4.0027339e-07 9.8212498e-01]
 [2.4835536e-10 1.2370619e-07 2.1639163e-08 ... 2.9042346e-07
  2.9384781e-10 9.9999315e-01]]
[ 0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2
  3  3  3  3  3  3  3  3  4  4  4  4  4  4  4  4  5  5  5  5  5  5  5  5
  6  6  6  6  6  6  6  6  7  7  7  7  7  7  7  7  8  8  8  8  8  8  8  8
  9  9  9  9  9  9  9  9 10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11
 12 12 12 12 12 12 12 12 13 13 13 13 13 13 13 13 14 14 14 14 14 14 14 14
 15 15 15 15 15 15 15 15 16 16 16 16 16 16 16 16 17 17 17 17 17 17 17 17
 18 18 18 18 18 18 18 18 19 19 19 19 19 19 19 19]
5/5 [==============================] - 2s 314ms/step
accuracy :
0.95
Confusion matrix, without normalization
```



Confusion matrix, without normalization



[[8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

[0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

[0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

[0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

[0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0]

[0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 2 0 6 0 0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0]

[2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0]

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0]

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0]

[0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 6 0]

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8]]

Confusion matrix, without normalization

Confusion matrix, without normalization

Confusion matrix:

[[8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

[0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

[0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

[0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

[0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0]

[0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 2 0 6 0 0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0]

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0]

[2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0]

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0]

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0]

[0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 6 0]

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8]]

	precision	recall	f1-score	support
0	0.80	1.00	0.89	8
1	1.00	1.00	1.00	8
2	1.00	1.00	1.00	8
3	1.00	1.00	1.00	8
4	1.00	0.75	0.86	8
5	1.00	1.00	1.00	8

6	1.00	1.00	1.00	8
7	0.67	1.00	0.80	8
8	1.00	1.00	1.00	8
9	1.00	0.75	0.86	8
10	1.00	1.00	1.00	8
11	1.00	1.00	1.00	8
12	1.00	1.00	1.00	8
13	1.00	1.00	1.00	8
14	1.00	1.00	1.00	8
15	1.00	0.75	0.86	8
16	1.00	1.00	1.00	8
17	0.80	1.00	0.89	8
18	1.00	0.75	0.86	8
19	1.00	1.00	1.00	8
accuracy			0.95	160
macro avg	0.96	0.95	0.95	160
weighted avg	0.96	0.95	0.95	160

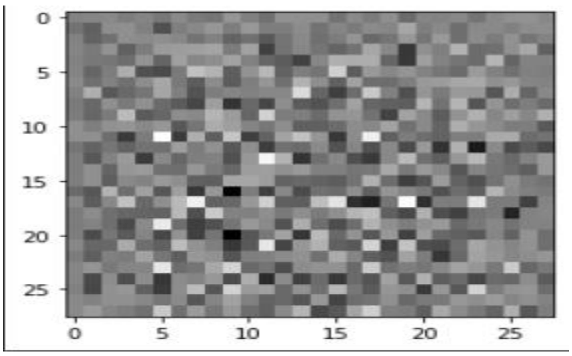
Assignment No:5

Aim/Problem Statement:To Analyze and differentiate fake and real image through GAN

```
import tensorflow as tf
from tensorflow.keras.layers import (Dense,
                                     BatchNormalization,
                                     LeakyReLU,
                                     Reshape,
                                     Conv2DTranspose,
                                     Conv2D,
                                     Dropout,
                                     Flatten)
import matplotlib.pyplot as plt

# underscore to omit the label arrays
(train_images, train_labels), (_, _) = tf.keras.datasets.mnist.load_data()

train_images = train_images.reshape(train_images.shape[0], 28, 28, 1).astype('float32')
train_images = (train_images - 127.5) / 127.5 # Normalize the images to [-1, 1]

BUFFER_SIZE = 60000
BATCH_SIZE = 256

# Batch and shuffle the data
train_dataset = tf.data.Dataset.from_tensor_slices(train_images).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)

def make_generator_model():
    model = tf.keras.Sequential()
    model.add(Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    model.add(Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size

    model.add(Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
```

```
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    model.add(Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    model.add(Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)

    return model

generator = make_generator_model()

# Create a random noise and generate a sample
noise = tf.random.normal([1, 100])
generated_image = generator(noise, training=False)
# Visualize the generated sample
plt.imshow(generated_image[0, :, :, 0], cmap='gray')
```

OUTPUT:



Step 2:

```
def make_discriminator_model():
    model = tf.keras.Sequential()

    model.add(Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[28, 28, 1]))
    model.add(LeakyReLU())
```

```
    model.add(Dropout(0.3))

    model.add(Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(LeakyReLU())
    model.add(Dropout(0.3))

    model.add(Flatten())
 model.add(Dense(1))

return model
discriminator = make_discriminator_model()
decision = discriminator(generated_image)
print (decision)
```

OUTPUT:

tf.Tensor([[-0.00014858]], shape=(1, 1), dtype=float32)

Step 3:

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

import os
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(generator_optimizer=generator_optimizer, discriminator_optimizer=discriminator_optimizer,
generator=generator, discriminator=discriminator)
EPOCHS = 120
num_examples_to_generate = 16
```

```
noise_dim = 100
seed = tf.random.normal([num_examples_to_generate, noise_dim])
@tf.function
def train_step(images):
 noise = tf.random.normal([BATCH_SIZE, noise_dim])

    # 2 - Generate images and calculate loss values
    # GradientTape method records operations for automatic differentiation.
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
      generated_images = generator(noise, training=True)

      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, training=True)

      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_output)
gradients_of_generator = gen_tape.gradient(gen_loss,
                                         generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss,
                                          discriminator.trainable_variables)
generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))

import time
from IPython import display # A command shell for interactive computing in Python.

def train(dataset, epochs):
  # A. For each epoch, do the following:
  for epoch in range(epochs):
    start = time.time()
    # 1 - For each batch of the epoch,
    for image_batch in dataset:
      # 1.a - run the custom "train_step" function
      # we just declared above
      train_step(image_batch)
```

```
    # 2 - Produce images for the GIF as we go
    display.clear_output(wait=True)
    generate_and_save_images(generator,
                             epoch + 1,
                             seed)

    # 3 - Save the model every 5 epochs as
    # a checkpoint, which we will use later
    if (epoch + 1) % 5 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)

    # 4 - Print out the completed epoch no. and the time spent
    print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

  # B. Generate a final image after the training is completed
  display.clear_output(wait=True)
  generate_and_save_images(generator,
                           epochs,
                           seed)
def generate_and_save_images(model, epoch, test_input):
  # Notice `training` is set to False.
  # This is so all layers run in inference mode (batchnorm).
  # 1 - Generate images
  predictions = model(test_input, training=False)
  # 2 - Plot the generated images
  fig = plt.figure(figsize=(4,4))
  for i in range(predictions.shape[0]):
      plt.subplot(4, 4, i+1)
      plt.imshow(predictions[i, :, :, 0] * 127.5 + 127.5, cmap='gray')
      plt.axis('off')
  # 3 - Save the generated images
  plt.savefig('image_at_epoch_{:04d}.png'.format(epoch))
  plt.show()

train(train_dataset, EPOCHS)
```

```
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))


# PIL is a library which may open different image file formats
import PIL
# Display a single image using the epoch number
def display_image(epoch_no):
  return PIL.Image.open('image_at_epoch_{:04d}.png'.format(epoch_no))
display_image(EPOCHS)
```

```
import glob # The glob module is used for Unix style pathname pattern expansion.
import imageio # The library that provides an easy interface to read and write a wide range of image data


anim_file = 'dcgan.gif'


with imageio.get_writer(anim_file, mode='I') as writer:
  filenames = glob.glob('image*.png')
  filenames = sorted(filenames)
  for filename in filenames:
    image = imageio.imread(filename)
    writer.append_data(image)
  # image = imageio.imread(filename)
  # writer.append_data(image)
```

```
display.Image(open('dcgan.gif','rb').read())
```

Assignment No:6

Aim/Problem Statement:Sentiment Analysis using LSTM and Glove Embedding

```
!pip install tensorflow --upgrade


# Importing the necessary libraries
import numpy as np
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense
from tensorflow.keras.datasets import imdb
# Load the IMDB dataset (keeping the top 5000 words)
vocab_size = 5000
maxlen = 100
embedding_dim = 128
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=vocab_size)
# Padding the sequences to a fixed length
X_train = pad_sequences(X_train, maxlen=maxlen)
X_test = pad_sequences(X_test, maxlen=maxlen)
# Build the LSTM model
model = Sequential()
# Embedding layer (vocab size, embedding dimension, input length)
model.add(Embedding(input_dim=vocab_size, output_dim=embedding_dim, input_length=maxlen))
# Adding an LSTM layer
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
# Fully connected layer
model.add(Dense(1, activation='sigmoid'))
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Print the model summary
model.summary()
# Optionally, print output shapes for each layer
```

```
# Use model.summary() for a structured output

# Train the model

model.fit(X_train, y_train, epochs=5, batch_size=64, validation_data=(X_test, y_test))


# Evaluate the model

loss, accuracy = model.evaluate(X_test, y_test)

print(f"Test Accuracy: {accuracy * 100:.2f}%")

from tensorflow.keras.utils import plot_model

plot_model(model, show_shapes=True, to_file='model.png')
```

OUTPUT:

Epoch 1/5

391/391 ━━━━━━━━━━━━━━━━━━━━ 137s 343ms/step - accuracy: 0.7005 - loss: 0.5564 - val_accuracy: 0.8200 - val_loss: 0.4087

Epoch 2/5

391/391 ━━━━━━━━━━━━━━━━━━━━ 142s 344ms/step - accuracy: 0.8501 - loss: 0.3527 - val_accuracy: 0.8345 - val_loss: 0.3722

Epoch 3/5

391/391 ━━━━━━━━━━━━━━━━━━━━ **143s** 346ms/step - accuracy: 0.8733 - loss: 0.3055 - val_accuracy: 0.8358 - val_loss: 0.3705

Epoch 4/5

391/391 ━━━━━━━━━━━━━━━━━━━━ **142s** 364ms/step - accuracy: 0.8902 - loss: 0.2701 - val_accuracy: 0.8458 - val_loss: 0.3669

Epoch 5/5

391/391 ━━━━━━━━━━━━━━━━━━━━ **195s** 345ms/step - accuracy: 0.9144 - loss: 0.2215 - val_accuracy: 0.8448 - val_loss: 0.3858

782/782 ━━━━━━━━━━━━━━━━━━━━ **52s** 67ms/step - accuracy: 0.8406 - loss: 0.3999Test Accuracy: 84.48%