AD3511 DEEP LEARNING LABORATORY MANUAL

LIST OF EXPERIMENTS:

1. Solving XOR problem using DNN
2. Character recognition using CNN
3. Face recognition using CNN
4. Language modeling using RNN
5. Sentiment analysis using LSTM
6. Parts of speech tagging using Sequence to Sequence architecture
7. Machine Translation using Encoder-Decoder model
8. Image augmentation using GANs
9. Mini-project on real world applications

TABLE OF CONTENTS

S. NO.	TITLE OF THE EXPERIMENTS	PAGE NO.
1	Solving XOR problem using DNN	1
2	Character recognition using CNN	4
3	Face recognition using CNN	10
4	Language modeling using RNN	17
5	Sentiment analysis using LSTM	22
6	Parts of speech tagging using Sequence to Sequence architecture	28
7	Machine Translation using Encoder-Decoder model	37
8	Image augmentation using GANs	43
9	Mini-project on real world applications	51

Ex. No: 1	SOLVING XOR PROBLEM USING DNN
Date :	

Aim:

To write a program to Solving XOR problem using DNN.

Algorithm:

1. Start the program.
2. Get the training data
3. To create the Number of Input units , Number of Hidden units, request.4.To send frames to server from the client side.
5. If your frames reach the server it will send ACK signal to client otherwise it willsend NACK signal to client.
6. Stop the program

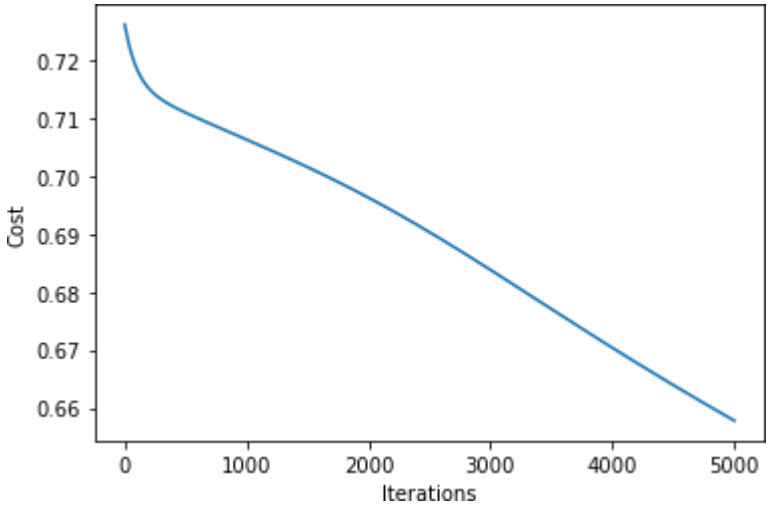
Program:

```
import numpy as np # For matrix math
import matplotlib.pyplot as plt # For plotting
import sys # For printing
# The training data.
X = np.array([
   [0, 1],
   [1, 0],
   [1, 1],
   [0, 0]
])

# The labels for the training data.
y = np.array([
   [1],
   [1],
   [0],
   [0]
])
num_i_units = 2 # Number of Input units
num_h_units = 2 # Number of Hidden units
num_o_units = 1 # Number of Output units
# The learning rate for Gradient Descent.
learning_rate = 0.01
# The parameter to help with overfitting.
reg_param = 0
# Maximum iterations for Gradient Descent.
max_iter = 5000
# Number of training examples
```

```
m = 4
np.random.seed(1)
W1 = np.random.normal(0, 1, (num_h_units, num_i_units)) # 2x2
W2 = np.random.normal(0, 1, (num_o_units, num_h_units)) # 1x2
B1 = np.random.random((num_h_units, 1)) # 2x1
B2 = np.random.random((num_o_units, 1)) # 1x1
def sigmoid(z, derv=False):
    if derv: return z * (1 - z)
    return 1 / (1 + np.exp(-z))

def forward(x, predict=False):
    a1 = x.reshape(x.shape[0], 1) # Getting the training example as a
column vector.
    z2 = W1.dot(a1) + B1 # 2x2 * 2x1 + 2x1 = 2x1
    a2 = sigmoid(z2) # 2x1
    z3 = W2.dot(a2) + B2 # 1x2 * 2x1 + 1x1 = 1x1
    a3 = sigmoid(z3)
    if predict: return a3
    return (a1, a2, a3)
dW1 = 0 # Gradient for W1
dW2 = 0 # Gradient for W2
dB1 = 0 # Gradient for B1
dB2 = 0 # Gradient for B2
cost = np.zeros((max_iter, 1)) # Column vector to record the cost of the
NN after each Gradient Descent iteration.
def train(_W1, _W2, _B1, _B2): # The arguments are to bypass
UnboundLocalError error
    for i in range(max_iter):
        c = 0
        dW1 = 0
        dW2 = 0
        dB1 = 0
        dB2 = 0

        for j in range(m):
            sys.stdout.write("\rIteration: {} and {}".format(i + 1, j + 1))
            # Forward Prop.
            a0 = X[j].reshape(X[j].shape[0], 1) # 2x1
            z1 = _W1.dot(a0) + _B1 # 2x2 * 2x1 + 2x1 = 2x1
            a1 = sigmoid(z1) # 2x1
            z2 = _W2.dot(a1) + _B2 # 1x2 * 2x1 + 1x1 = 1x1
            a2 = sigmoid(z2) # 1x1
            # Back prop.
            dz2 = a2 - y[j] # 1x1
            dW2 += dz2 * a1.T # 1x1 .* 1x2 = 1x2
            dz1 = np.multiply((_W2.T * dz2), sigmoid(a1, derv=True)) #
(2x1 * 1x1) .* 2x1 = 2x1
            dW1 += dz1.dot(a0.T) # 2x1 * 1x2 = 2x2

            dB1 += dz1 # 2x1
```

```
            dB2 += dz2 # 1x1

            c = c + (-(y[j] * np.log(a2)) - ((1 - y[j]) * np.log(1 - a2)))
            sys.stdout.flush() # Updating the text.

        _W1 = _W1 - learning_rate * (dW1 / m) + ( (reg_param / m) *
_W1)
        _W2 = _W2 - learning_rate * (dW2 / m) + ( (reg_param / m) *
_W2)

        _B1 = _B1 - learning_rate * (dB1 / m)
        _B2 = _B2 - learning_rate * (dB2 / m)
        cost[i] = (c / m) + (
            (reg_param / (2 * m)) *
            (
                np.sum(np.power(_W1, 2)) +
                np.sum(np.power(_W2, 2))
            )
        )
    return (_W1, _W2, _B1, _B2)
W1, W2, B1, B2 = train(W1, W2, B1, B2)
# Assigning the axes to the different elements.
plt.plot(range(max_iter), cost)
# Labelling the x axis as the iterations axis.
plt.xlabel("Iterations")
# Labelling the y axis as the cost axis.
plt.ylabel("Cost")
# Showing the plot.
plt.show()
```

Output:



Result:

Thus the Python program successfully to Solving XOR problem using DNN.

Ex. No: 2	CHARACTER RECOGNITION USING CNN
Date :	

Aim:

To write a python program to implement the Character recognition using CNN.

Algorithm:

1. Start the program.
2. Get the relevant packages for Recognition
3. Load the A_Z Handwritten Data.csv from the directory.
4. Reshape data for model creation
5. Train the model and Prediction on test data
6. Prediction on External Image
7. Stop the program

Program:

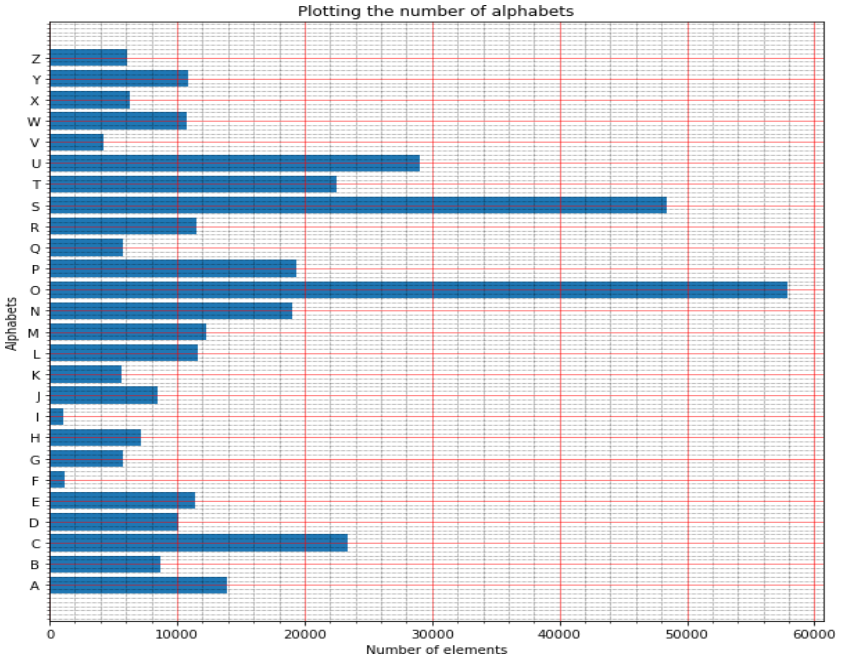
```
pip install opencv-python
pip install keras
pip install tensorflow
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPool2D, Dropout
from keras.optimizers import SGD, Adam
from keras.callbacks import ReduceLROnPlateau, EarlyStopping
from keras.utils import to_categorical
data = pd.read_csv(r"A_Z Handwritten Data.csv").astype('float32')
X = data.drop('0',axis = 1)
y = data['0']
train_x, test_x, train_y, test_y = train_test_split(X, y, test_size = 0.2)
train_x = np.reshape(train_x.values, (train_x.shape[0], 28,28))
test_x = np.reshape(test_x.values, (test_x.shape[0], 28,28))
word_dict =
{0:'A',1:'B',2:'C',3:'D',4:'E',5:'F',6:'G',7:'H',8:'I',9:'J',10:'K',11:'L',12:'M',13:'N',14:'O',
15:'P',16:'Q',17:'R',18:'S',19:'T',20:'U',21:'V',22:'W',23:'X',  24:'Y',25:'Z'}
y_int = np.int0(y)
```

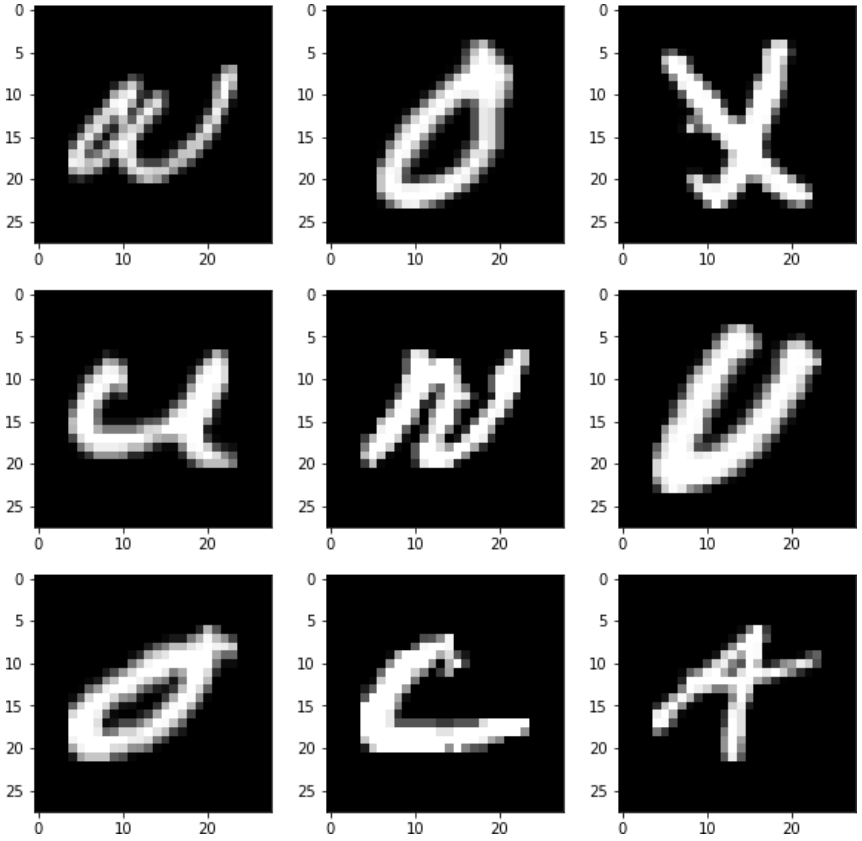
```
count = np.zeros(26, dtype='int')
for i in y_int:
    count[i] +=1

alphabets = []
for i in word_dict.values():
    alphabets.append(i)


fig, ax = plt.subplots(1,1, figsize=(10,10))
ax.barh(alphabets, count)

# naming the x axis
plt.xlabel("Number of elements")
# naming the y axis
plt.ylabel("Alphabets")
# giving a title
plt.title("Plotting the number of alphabets")
# Turn on the minor TICKS, which are required for the minor GRID
plt.minorticks_on()
# Customize the major grid
plt.grid(which='major', linestyle='-', linewidth='0.5', color='red')
# Customize the minor grid
plt.grid(which='minor', linestyle=':', linewidth='0.5', color='black')

plt.show()

uff = shuffle(train_x[:100])
fig, ax = plt.subplots(3,3, figsize = (10,10))
axes = ax.flatten()
for i in range(9):
    _, shu = cv2.threshold(shuff[i], 30, 200, cv2.THRESH_BINARY)
    axes[i].imshow(np.reshape(shuff[i], (28,28)), cmap=plt.get_cmap('gray'))
plt.show()
# Reshape data for model creation
train_X = train_x.reshape(train_x.shape[0],train_x.shape[1],train_x.shape[2],1)
print("The new shape of train data: ", train_X.shape)

test_X = test_x.reshape(test_x.shape[0], test_x.shape[1], test_x.shape[2],1)
print("The new shape of train data: ", test_X.shape)

train_yOHE = to_categorical(train_y, num_classes = 26, dtype='int')
print("The new shape of train labels: ", train_yOHE.shape)

test_yOHE = to_categorical(test_y, num_classes = 26, dtype='int')
print("The new shape of test labels: ", test_yOHE.shape)
model = Sequential()

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

```
model.add(MaxPool2D(pool_size=(2, 2), strides=2))

model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding =
'same'))
model.add(MaxPool2D(pool_size=(2, 2), strides=2))

model.add(Conv2D(filters=128, kernel_size=(3, 3), activation='relu', padding =
'valid'))
model.add(MaxPool2D(pool_size=(2, 2), strides=2))

model.add(Flatten())

model.add(Dense(64,activation ="relu"))
model.add(Dense(128,activation ="relu"))

model.add(Dense(26,activation ="softmax"))
model.compile(optimizer = Adam(learning_rate=0.001),
loss='categorical_crossentropy', metrics=['accuracy'])

history = model.fit(train_X, train_yOHE, epochs=1, validation_data =
(test_X,test_yOHE))
model.summary()
model.save(r'model_hand.h5')

print("The validation accuracy is :", history.history['val_accuracy'])
print("The training accuracy is :", history.history['accuracy'])
print("The validation loss is :", history.history['val_loss'])
print("The training loss is :", history.history['loss'])
# Prediction on test data
fig, axes = plt.subplots(3,3, figsize=(8,9))
axes = axes.flatten()

for i,ax in enumerate(axes):
    img = np.reshape(test_X[i], (28,28))
    ax.imshow(img,  cmap=plt.get_cmap('gray'))

    pred = word_dict[np.argmax(test_yOHE[i])]
    ax.set_title("Prediction: "+pred)
# Predection on External Image

img = cv2.imread(r'test_image.jpg')
img_copy = img.copy()

img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = cv2.resize(img, (400,440))


img_copy = cv2.GaussianBlur(img_copy, (7,7), 0)
img_gray = cv2.cvtColor(img_copy, cv2.COLOR_BGR2GRAY)
_, img_thresh = cv2.threshold(img_gray, 100, 255, cv2.THRESH_BINARY_INV)
```

```
img_final = cv2.resize(img_thresh, (28,28))
img_final =np.reshape(img_final, (1,28,28,1))



img_pred = word_dict[np.argmax(model.predict(img_final))]

cv2.putText(img, "Image Data", (100,25), cv2.FONT_HERSHEY_DUPLEX ,
fontScale= 1, thickness=2, color = (255,0,0))
cv2.putText(img, "Character Prediction: " + img_pred, (10,410),
cv2.FONT_HERSHEY_SIMPLEX, fontScale= 1, thickness=2, color = (0,0,255))
cv2.imshow('Character Recognition', img)

while (1):
    k = cv2.waitKey(1) & 0xFF
    if k == 27:
        break
cv2.destroyAllWindows()
```

Output:



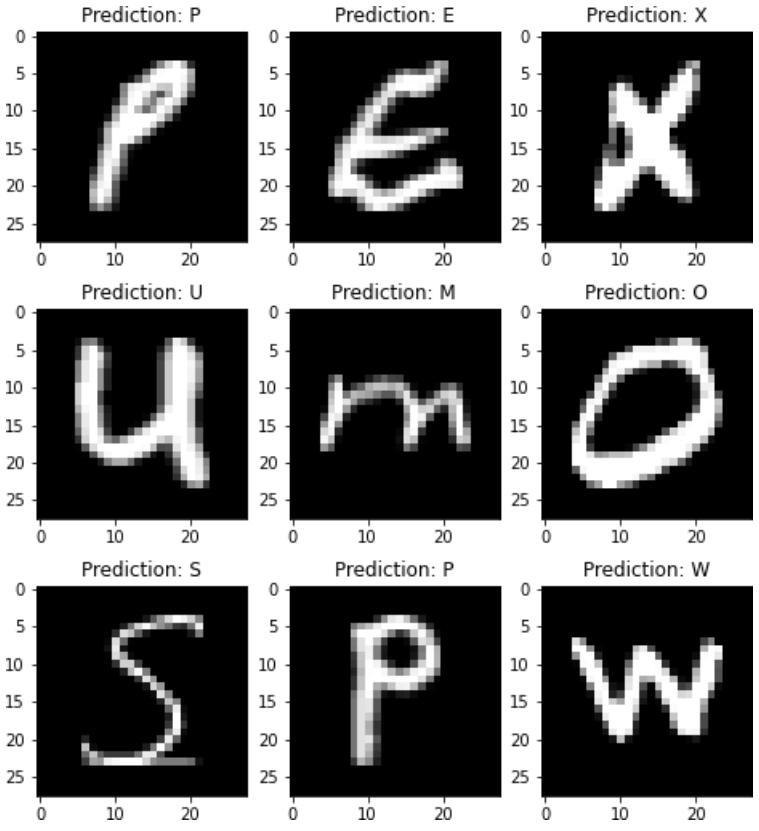


```
The new shape of train data:   (297960, 28, 28, 1)
The new shape of train data:   (74490, 28, 28, 1)
The new shape of train labels:  (297960,    26)
The new shape of test labels:   (74490, 26)
```

```
       9312/9312 [==============================] - 85s 9ms/step - loss
: 0.1440
                 - accuracy: 0.9595 - val_loss: 0.0853 - val_accuracy:
0.9761
```

```
model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 13, 13, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 6, 6, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 4, 4, 128)         73856
```

```
 max_pooling2d_2 (MaxPooling   (None, 2, 2, 128)         0
 2D)

 flatten (Flatten)             (None, 512)               0

 dense (Dense)                 (None, 64)                32832

 dense_1 (Dense)               (None, 128)               8320

 dense_2 (Dense)               (None, 26)                3354

=================================================================
Total params: 137,178
Trainable params: 137,178
Non-trainable params: 0
_________________________________________________________________
The validation accuracy is : [0.9760907292366028]
The training accuracy is : [0.9595012664794922]
The validation loss is : [0.08530429750680923]
The training loss is : [0.1440141350030899]
```




```
1/1 [==============================] - 0s 79ms/step
```

Result:

Thus written a python program to implement successfully for the Character recognition using CNN.

Ex. No: 3	FACE RECOGNITION USING CNN
Date :	

Aim:

To write a python program to implement the Face recognition using CNN.

Algorithm:

Start the program.
Get the relevant packages for Face Recognition
Load the A_Z Handwritten Data.csv from the directory.
Reshape data for model creation
Train the model and Prediction on test data
Prediction on External Image
Stop the program

Program:

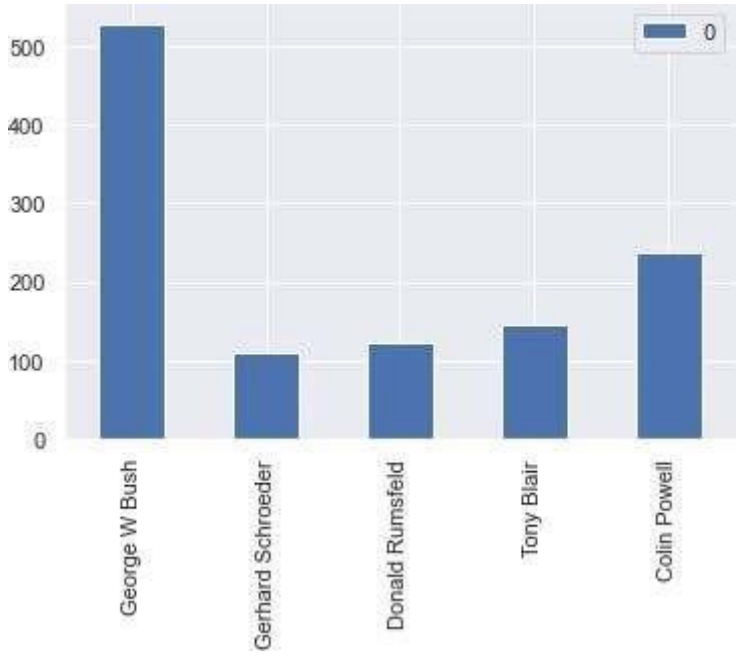
```
import numpy as np
import pandas as pd
from sklearn.datasets import fetch_lfw_people

faces = fetch_lfw_people(min_faces_per_person=100, resize=1.0, slice_=(slice(60,
188), slice(60, 188)), color=True)
class_count = len(faces.target_names)

print(faces.target_names)
print(faces.images.shape)
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

fig, ax = plt.subplots(3, 6, figsize=(18, 10))

for i, axi in enumerate(ax.flat):
   axi.imshow(faces.images[i] / 255) # Scale pixel values so Matplotlib doesn't clip
everything above 1.0
   axi.set(xticks=[], yticks=[], xlabel=faces.target_names[faces.target[i]])
from collections import Counter
counts = Counter(faces.target)
names = {}

for key in counts.keys():
   names[faces.target_names[key]] = counts[key]

df = pd.DataFrame.from_dict(names, orient='index')
df.plot(kind='bar')
```

```
mask = np.zeros(faces.target.shape, dtype=np.bool)

for target in np.unique(faces.target):
    mask[np.where(faces.target == target)[0][:100]] = 1

x_faces = faces.data[mask]
y_faces = faces.target[mask]
x_faces = np.reshape(x_faces, (x_faces.shape[0], faces.images.shape[1],
faces.images.shape[2], faces.images.shape[3]))
x_faces.shape
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split

face_images = x_faces / 255 # Normalize pixel values
face_labels = to_categorical(y_faces)

x_train, x_test, y_train, y_test = train_test_split(face_images, face_labels,
train_size=0.8, stratify=face_labels, random_state=0)
from keras.layers import Dense
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPooling2D
from keras.layers import Flatten

model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu',
input_shape=(face_images.shape[1:])))
model.add(MaxPooling2D(2, 2))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(2, 2))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(2, 2))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(class_count, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model.summary()
hist = model.fit(x_train, y_train, validation_data=(x_test, y_test), epochs=20,
batch_size=25)
acc = hist.history['accuracy']
val_acc = hist.history['val_accuracy']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, '-', label='Training Accuracy')
plt.plot(epochs, val_acc, ':', label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.plot()
```

```
from sklearn.metrics import confusion_matrix

y_predicted = model.predict(x_test)
mat = confusion_matrix(y_test.argmax(axis=1), y_predicted.argmax(axis=1))

sns.heatmap(mat.T, square=True, annot=True, fmt='d', cbar=False, cmap='Blues',
        xticklabels=faces.target_names,
        yticklabels=faces.target_names)

plt.xlabel('Predicted label')
plt.ylabel('Actual label')
import keras.utils as image

x = image.load_img('george.jpg', target_size=(face_images.shape[1:]))
plt.xticks([])
plt.yticks([])
plt.imshow(x)

x = image.img_to_array(x) / 255
x = np.expand_dims(x, axis=0)
y = model.predict(x)[0]

for i in range(len(y)):
   print(faces.target_names[i] + ': ' + str(y[i]))
```

Output:

```
['Colin Powell' 'Donald Rumsfeld' 'George W Bush' 'Gerhard Schroeder'
 'Tony Blair']
(1140, 128, 128, 3)
```




```
(500, 128, 128, 3)

Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 126, 126, 32)      896

 max_pooling2d (MaxPooling2D  (None, 63, 63, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 61, 61, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 30, 30, 64)       0
 2D)

 conv2d_2 (Conv2D)           (None, 28, 28, 64)        36928

 max_pooling2d_2 (MaxPooling  (None, 14, 14, 64)       0
 2D)

 flatten (Flatten)           (None, 12544)             0

 dense (Dense)               (None, 128)               1605760

 dense_1 (Dense)             (None, 5)                 645

=================================================================
Total params: 1,662,725
Trainable params: 1,662,725
Non-trainable params: 0
_________________________________________________________________
```

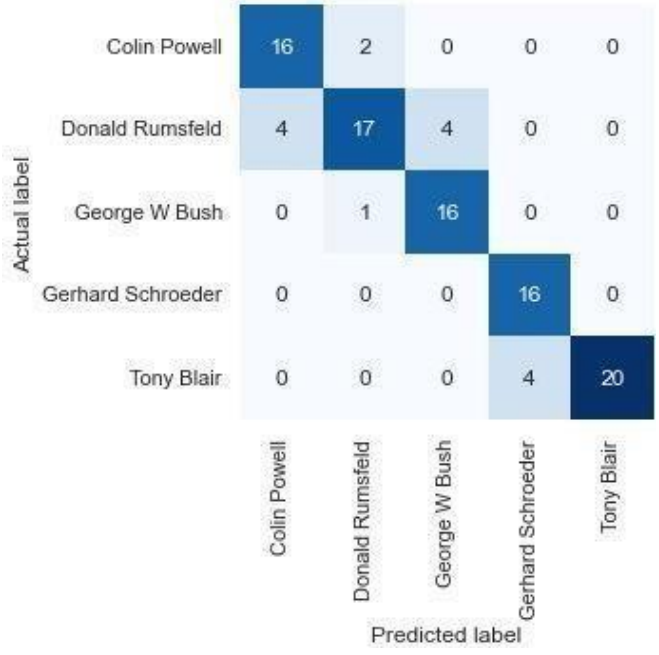
Epoch 1/20

16/16 [==============================] - 2s 123ms/step - loss: 1.6558 - accuracy: 0.1925 - val_loss: 1.6038 - val_accuracy: 0.2000
Epoch 2/20
16/16 [==============================] - 2s 110ms/step - loss: 1.5860 - accuracy: 0.3175 - val_loss: 1.5416 - val_accuracy: 0.3200
Epoch 3/20
16/16 [==============================] - 2s 112ms/step - loss: 1.4851 - accuracy: 0.3675 - val_loss: 1.3706 - val_accuracy: 0.4500
Epoch 4/20
16/16 [==============================] - 2s 110ms/step - loss: 1.1602 - accuracy: 0.5775 - val_loss: 1.0931 - val_accuracy: 0.5900
Epoch 5/20
16/16 [==============================] - 2s 112ms/step - loss: 0.8385 - accuracy: 0.7000 - val_loss: 0.8494 - val_accuracy: 0.6700
Epoch 6/20
16/16 [==============================] - 2s 111ms/step - loss: 0.5011 - accuracy: 0.8275 - val_loss: 0.8085 - val_accuracy: 0.6900
Epoch 7/20
16/16 [==============================] - 2s 111ms/step - loss: 0.3819 - accuracy: 0.8550 - val_loss: 0.7241 - val_accuracy: 0.7200
Epoch 8/20
16/16 [==============================] - 2s 110ms/step - loss: 0.3558 - accuracy: 0.8950 - val_loss: 0.5499 - val_accuracy: 0.7800
Epoch 9/20
16/16 [==============================] - 2s 114ms/step - loss: 0.1407 - accuracy: 0.9575 - val_loss: 0.7090 - val_accuracy: 0.8000
Epoch 10/20
16/16 [==============================] - 2s 115ms/step - loss: 0.0869 - accuracy: 0.9875 - val_loss: 0.6296 - val_accuracy: 0.8400
Epoch 11/20
16/16 [==============================] - 2s 111ms/step - loss: 0.0413 - accuracy: 0.9950 - val_loss: 0.5816 - val_accuracy: 0.8300
Epoch 12/20
16/16 [==============================] - 2s 110ms/step - loss: 0.0325 - accuracy: 0.9950 - val_loss: 0.5888 - val_accuracy: 0.8300
Epoch 13/20
16/16 [==============================] - 2s 110ms/step - loss: 0.0359 - accuracy: 0.9900 - val_loss: 0.6945 - val_accuracy: 0.8100
Epoch 14/20
16/16 [==============================] - 2s 110ms/step - loss: 0.0085 - accuracy: 1.0000 - val_loss: 0.5278 - val_accuracy: 0.8600
Epoch 15/20
16/16 [==============================] - 2s 111ms/step - loss: 0.0048 - accuracy: 1.0000 - val_loss: 0.5697 - val_accuracy: 0.8500
Epoch 16/20
16/16 [==============================] - 2s 111ms/step - loss: 0.0032 - accuracy: 1.0000 - val_loss: 0.6065 - val_accuracy: 0.8500
Epoch 17/20
16/16 [==============================] - 2s 110ms/step - loss: 0.0022 - accuracy: 1.0000 - val_loss: 0.6007 - val_accuracy: 0.8500
Epoch 18/20
16/16 [==============================] - 2s 112ms/step - loss: 0.0017 - accuracy: 1.0000 - val_loss: 0.6242 - val_accuracy: 0.8500
Epoch 19/20

16/16 [==============================] - 2s 118ms/step - loss: 0.0013 - accuracy: 1.0000 - val_loss: 0.6333 - val_accuracy: 0.8500
Epoch 20/20
16/16 [==============================] - 2s 111ms/step - loss: 0.0011 - accuracy: 1.0000 - val_loss: 0.6541 - val_accuracy: 0.8500


```
4/4 [==============================] - 0s 26ms/step
```

Text(89.18, 0.5, 'Actual label')




```
<matplotlib.image.AxesImage at 0x1ec80d4d910>
```

```
1/1 [==============================] - 0s 48ms/step
Colin Powell: 0.20101844
Donald Rumsfeld: 0.20214622
George W Bush: 0.2216323
Gerhard Schroeder: 0.21147959
Tony Blair: 0.16372345
```

Result:

Thus written a python program to implemented successfully for the Face recognition using CNN.

Ex. No: 4	LANGUAGE MODELING USING RNN
Date :	

Aim:

To write a python program to implement the Language modeling using RNN.

Algorithm:

Start the program.
Get the relevant packages for Language modeling
Read a file and split into lines.
Build the category_lines dictionary, a list of lines per category
Add the Random item from a list
Get a random category and random line from that category.
One-hot vector for category
Make category, input, and target tensors from a random category, line pair
Sample from a category and starting letter
Get multiple samples from one category and multiple starting letters.
Train the model and Prediction on test data
Prediction on External Image
Stop the program

Program:

```
from__future__import unicode_literals, print_function, division
from io import open
import glob
import os
import unicodedata
import string
all_letters = string.ascii_letters + " .,;'-"
n_letters = len(all_letters) + 1 # Plus EOS marker
def findFiles(path): return glob.glob(path)

# Turn a Unicode string to plain ASCII, thanks to
https://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
        and c in all_letters
    )

# Read a file and split into lines
def readLines(filename):
    with open(filename, encoding='utf-8') as some_file:
        return [unicodeToAscii(line.strip()) for line in some_file]

# Build the category_lines dictionary, a list of lines per category
category_lines = {}
all_categories = []
for filename in findFiles('data/names/*.txt'):
    category = os.path.splitext(os.path.basename(filename))[0]
```

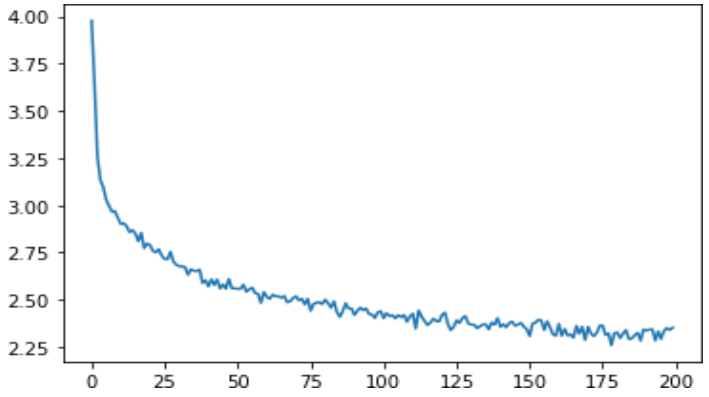
```
    all_categories.append(category)
    lines = readLines(filename)
    category_lines[category] = lines

n_categories = len(all_categories)

if n_categories == 0:
    raise RuntimeError('Data not found. Make sure that you downloaded data '
        'from https://download.pytorch.org/tutorial/data.zip and extract it to '
        'the current directory.')

print('# categories:', n_categories, all_categories)
print(unicodeToAscii("O'Néàl"))

pip install torch

import torch
import torch.nn as nn

class RNN(nn.Module):
    def __init (self, input_size, hidden_size, output_size):
        super(RNN, self). init ()
        self.hidden_size = hidden_size

        self.i2h = nn.Linear(n_categories + input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(n_categories + input_size + hidden_size, output_size)
        self.o2o = nn.Linear(hidden_size + output_size, output_size)
        self.dropout      =      nn.Dropout(0.1)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, category, input, hidden):
        input_combined = torch.cat((category, input, hidden), 1)
        hidden = self.i2h(input_combined)
        output = self.i2o(input_combined)
        output_combined = torch.cat((hidden, output), 1)
        output = self.o2o(output_combined)
        output = self.dropout(output)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
import random

# Random item from a list
def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

# Get a random category and random line from that category
def randomTrainingPair():
    category = randomChoice(all_categories)
    line = randomChoice(category_lines[category])
    return category, line
# One-hot vector for category
```

```
def categoryTensor(category):
    li = all_categories.index(category)
    tensor = torch.zeros(1, n_categories)
    tensor[0][li] = 1
    return tensor

# One-hot matrix of first to last letters (not including EOS) for input
def inputTensor(line):
    tensor = torch.zeros(len(line), 1, n_letters)
    for li in range(len(line)):
        letter = line[li]
        tensor[li][0][all_letters.find(letter)] = 1
    return tensor

# ``LongTensor`` of second letter to end (EOS) for target
def targetTensor(line):
    letter_indexes = [all_letters.find(line[li]) for li in range(1, len(line))]
    letter_indexes.append(n_letters - 1) # EOS
    return torch.LongTensor(letter_indexes)
# Make category, input, and target tensors from a random category, line pair
def randomTrainingExample():
    category, line = randomTrainingPair()
    category_tensor = categoryTensor(category)
    input_line_tensor = inputTensor(line)
    target_line_tensor = targetTensor(line)
    return category_tensor, input_line_tensor, target_line_tensor
criterion = nn.NLLLoss()
learning_rate = 0.0005

def train(category_tensor, input_line_tensor, target_line_tensor):
    target_line_tensor.unsqueeze_(-1)
    hidden = rnn.initHidden()
    rnn.zero_grad()
    loss = 0
    for i in range(input_line_tensor.size(0)):
        output, hidden = rnn(category_tensor, input_line_tensor[i], hidden)
        l = criterion(output, target_line_tensor[i])
        loss += l

    loss.backward()
    for p in rnn.parameters():
        p.data.add_(p.grad.data, alpha=-learning_rate)

    return output, loss.item() / input_line_tensor.size(0)

import time
import math

def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)
```

```
rnn = RNN(n_letters, 128, n_letters)

n_iters = 100000
print_every = 5000
plot_every = 500
all_losses = []
total_loss = 0 # Reset every ``plot_every`` ``iters``

start = time.time()
for iter in range(1, n_iters + 1):
    output, loss = train(*randomTrainingExample())
    total_loss += loss
    if iter % print_every == 0:
        print('%s (%d %d%%) %.4f' % (timeSince(start), iter, iter / n_iters * 100, loss))

    if iter % plot_every == 0:
        all_losses.append(total_loss / plot_every)
        total_loss = 0
import matplotlib.pyplot as plt

plt.figure()
plt.plot(all_losses)
max_length = 20

# Sample from a category and starting letter
def sample(category, start_letter='A'):
    with torch.no_grad(): # no need to track history in sampling
        category_tensor = categoryTensor(category)
        input = inputTensor(start_letter)
        hidden = rnn.initHidden()

        output_name = start_letter

        for i in range(max_length):
            output, hidden = rnn(category_tensor, input[0], hidden)
            topv, topi = output.topk(1)
            topi = topi[0][0]
            if topi == n_letters - 1:
                break
            else:
                letter = all_letters[topi]
                output_name += letter
            input = inputTensor(letter)

        return output_name

# Get multiple samples from one category and multiple starting letters
def samples(category, start_letters='ABC'):
    for start_letter in start_letters:
        print(sample(category, start_letter))
samples('Russian',  'RUS')
samples('German',  'GER')
samples('Spanish',  'SPA')
samples('Chinese', 'CHI')
```

Output:

```
# categories: 18 ['Arabic', 'Chinese', 'Czech', 'Dutch',
'English', 'French', 'German', 'Greek', 'Irish', 'Italian
', 'Japanese', 'Korean', 'Polish', 'Portuguese', 'Russian
', 'Scottish', 'Spanish', 'Vietnamese']
O'Neal
0m 5s (5000 5%) 2.6595
0m 11s (10000 10%) 2.9644
0m 16s (15000 15%) 3.3754
0m 22s (20000 20%) 2.0799
0m 27s (25000 25%) 2.6884
0m 33s (30000 30%) 2.2509
0m 38s (35000 35%) 2.3497
0m 43s (40000 40%) 2.5290
0m 49s (45000 45%) 2.9439
0m 54s (50000 50%) 2.7406
0m 59s (55000 55%) 3.0044
1m 4s (60000 60%) 2.5765
1m 10s (65000 65%) 2.3694
1m 15s (70000 70%) 2.2810
1m 20s (75000 75%) 2.2660
1m 26s (80000 80%) 2.1720
1m 31s (85000 85%) 2.4900
1m 36s (90000 90%) 2.0302
1m 42s (95000 95%) 1.8320
1m 47s (100000 100%) 2.4904

[<matplotlib.lines.Line2D at 0x1e56757bcd0>]
```




```
Rovonov
Uarakov
Shavanov
Gerre
Eeren
Roure
Salla
Para
Allana
Cha
Han
Iun
```

Result:

Thus written a python program to implemented successfully for the Language Modeling Using RNN.

Ex. No: 5	SENTIMENT ANALYSIS USING LSTM
Date :	

Aim:

To write a python program to implement the Sentiment analysis using LSTM.

Algorithm:

Start the program.

Get the relevant packages for Keras-PreprocessingLoad the IMDB Dataset.csv file.

Remove HTML tags, URL and non-alphanumeric charactersRead a file and split into lines.

Tuning the hyperparameters of the modelModel initialization

compile model

reviews on which we need to predict.Stop the program

Program:

```
pip install Keras-Preprocessing
import re
import pandas as pd
import numpy as np
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from keras.preprocessing.text import Tokenizer
from keras_preprocessing.sequence import pad_sequences
import keras
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
import math
import nltk
data = pd.read_csv('IMDB Dataset.csv')
data
def remove_tags(string):
    removelist = ""
    result = re.sub("","",string)          #remove HTML tags
    result = re.sub('https://.*',"",result)  #remove URLs
    result = re.sub(r'[^w'+removelist+']', ' ',result)    #remove non-alphanumeric
characters
    result = result.lower()
    return result
data['review']=data['review'].apply(lambda cw : remove_tags(cw))
nltk.download('stopwords')
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
data['review'] = data['review'].apply(lambda x: ' '.join([word for word in x.split() if
```

```
word not in (stop_words)]))
import nltk
nltk.download()
#we want to download 'wordnet' and 'omw-1.4' from nltk
w_tokenizer = nltk.tokenize.WhitespaceTokenizer()
lemmatizer = nltk.stem.WordNetLemmatizer()
def lemmatize_text(text):
    st = ""
    for w in w_tokenizer.tokenize(text):
        st = st + lemmatizer.lemmatize(w) + " "
    return st
data['review'] = data.review.apply(lemmatize_text)
data
s = 0.0
for i in data['review']:
    word_list = i.split()
    s = s + len(word_list)
print("Average length of each review : ",s/data.shape[0])
pos = 0
for i in range(data.shape[0]):
    if data.iloc[i]['sentiment'] == 'positive':
        pos = pos + 1
neg = data.shape[0]-pos
print("Percentage of reviews with positive sentiment is
"+str(pos/data.shape[0]*100)+"%")
print("Percentage of reviews with negative sentiment is
"+str(neg/data.shape[0]*100)+"%")

reviews = data['review'].values
labels = data['sentiment'].values
encoder = LabelEncoder()
encoded_labels = encoder.fit_transform(labels)

train_sentences, test_sentences, train_labels, test_labels = train_test_split(reviews,
encoded_labels, stratify = encoded_labels)

# Hyperparameters of the model
vocab_size = 3000 # choose based on statistics
oov_tok = "
embedding_dim = 100
max_length = 200 # choose based on statistics, for example 150 to 200
padding_type='post'
trunc_type='post'
# tokenize sentences
tokenizer = Tokenizer(num_words = vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(train_sentences)
word_index = tokenizer.word_index
# convert train dataset to sequence and pad sequences
train_sequences = tokenizer.texts_to_sequences(train_sentences)
train_padded = pad_sequences(train_sequences, padding='post',
```

```
maxlen=max_length)
# convert Test dataset to sequence and pad sequences
test_sequences = tokenizer.texts_to_sequences(test_sentences)
test_padded = pad_sequences(test_sequences, padding='post', maxlen=max_length)

# model initialization
model = keras.Sequential([
   keras.layers.Embedding(vocab_size, embedding_dim,
input_length=max_length),
   keras.layers.Bidirectional(keras.layers.LSTM(64)),
   keras.layers.Dense(24, activation='relu'),
   keras.layers.Dense(1, activation='sigmoid')
])
# compile model
model.compile(loss='binary_crossentropy',
            optimizer='adam',
            metrics=['accuracy'])
# model summary
model.summary()

num_epochs = 5
history = model.fit(train_padded, train_labels,
                epochs=num_epochs,  verbose=1,
                validation_split=0.1)

prediction = model.predict(test_padded)
# Get labels based on probability 1 if p>= 0.5 else 0
pred_labels = []
for i in prediction:
   if i >= 0.5:
      pred_labels.append(1)
   else:
      pred_labels.append(0)
print("Accuracy of prediction on test set : ",
accuracy_score(test_labels,pred_labels))

# reviews on which we need to predict
sentence = ["The movie was very touching and heart whelming",
          "I have never seen a terrible movie like this",
          "the movie plot is terrible but it had good acting"]
# convert to a sequence
sequences = tokenizer.texts_to_sequences(sentence)
# pad the sequence
padded = pad_sequences(sequences, padding='post', maxlen=max_length)
# Get labels based on probability 1 if p>= 0.5 else 0
prediction = model.predict(padded)
pred_labels = []
for i in prediction:
   if i >= 0.5:
      pred_labels.append(1)
```

```
    else:
        pred_labels.append(0)
for i in range(len(sentence)):
    print(sentence[i])
    if pred_labels[i] == 1:
        s = 'Positive'
    else:
        s = 'Negative'
    print("Predicted sentiment : ",s)
```

Output:

	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Mattei's "Love in the Time of Money" is...	positive
...	...	...
49995	I thought this movie did a down right good job...	positive
49996	Bad plot, bad dialogue, bad acting, idiotic di...	negative
49997	I am a Catholic taught in parochial elementary...	negative
49998	I'm going to have to disagree with the previou...	negative
49999	No one expects the Star Trek movies to be high...	negative

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\CGNANAM.ADS\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!

showing info https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/index.xml
```

Out[6]:

```
True
```

	review	sentiment
0	w w w w w w w w w w w w w w w w w w w w ...	positive
1	w w w w w w w w w w w w w w	positive
2	w w w w w w w w w w w w w w w w w w	positive
3	w w w w w w w w w w	negative
4	w w w w w w w w w w w w w w w w w w	positive

	review	sentiment
...	...	...
49995	w w w w w w w w w w w w w w w w w w w w	positive
49996	w w w w w w w w	negative
49997	w w w w w w w w w w w w	negative
49998	w w w w w w w w w w w w w w w w	negative
49999	w w w w w w w w w w w w	negative

50000 rows × 2 columns

```
Average length of each review :  18.714
Percentage of reviews with positive sentiment is 50.0%
Percentage of reviews with negative sentiment is 50.0%
```

Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 200, 100)	300000
bidirectional (Bidirectional)	(None, 128)	84480
dense (Dense)	(None, 24)	3096
dense_1 (Dense)	(None, 1)	25

Total params: 387,601
Trainable params: 387,601
Non-trainable params: 0

```
Epoch 1/5
1055/1055 [==============================] - 60s 55ms/step - loss: 0.69
32 - accuracy: 0.5021 - val_loss: 0.6925 - val_accuracy: 0.5205
Epoch 2/5
1055/1055 [==============================] - 58s 55ms/step - loss: 0.69
26 - accuracy: 0.5094 - val_loss: 0.6925 - val_accuracy: 0.5173
```

```
Epoch 3/5
1055/1055 [==============================]  - 59s 56ms/step - loss:  0.69
26 - accuracy: 0.5129 - val_loss: 0.6924 -  val_accuracy: 0.5171
Epoch 4/5
1055/1055 [==============================]  - 59s 56ms/step - loss:  0.69
23 - accuracy: 0.5166 - val_loss: 0.6927 -  val_accuracy: 0.4965
Epoch 5/5
1055/1055 [==============================]  - 58s 55ms/step - loss:  0.69
25 - accuracy: 0.5141 - val_loss: 0.6924 -  val_accuracy: 0.5173

391/391 [==============================] -  6s 14ms/step
Accuracy of prediction on test set :  0.5148

1/1 [==============================] - 0s 23ms/step
The movie was very touching and heart whelming
Predicted sentiment :  Positive
I have never seen a terrible movie like this
Predicted sentiment :  Positive
the movie plot is terrible but it had good acting
Predicted sentiment :  Positive
```

Result:

Thus written a python program to implemented successfully for the Sentiment analysis using LSTM.

Ex. No: 6	PARTS OF SPEECH TAGGING USING TO SEQUENCEARCHITECTURE
Date :	

Aim:

To write a python program to implement the parts of speech tagging using Sequence to Sequence architecture.

Algorithm:

Start the program
Get the relevant packages for functiontools
Load the Vocabulary Creation and data split in to train and test
Sort the vocabulary by occurrences of words
write the sorted vocabulary to vocab file
build a vocabulary list with only frequent words (i.e. occur no less than 3 times)
replace non-frequent words in word with <unk>
make sure the index of the current word is less than the next
build an emission and a transition dictionaries
write the emission and transition dictionaries into a json file
load txt file vocab
split dev lists (index, word and pos) to individual samples (list --> list of sublists)
initialize a dictionary that keeps track of the highest cumulative probability of each possible pos at each position of the input sentence
use viterbi to predict pos for dev
merge the list of sublists to a single list
Stop the program

Program:

```
import numpy as np
import pandas as pd
import json
import functools as fc
from sklearn.metrics import accuracy_score
```

Task 1: Vocabulary Creation

```
#train = pd.read_csv('data1\train', sep='\t', names=['index', 'word', 'POS'])
train = pd.read_csv('data1/train', sep='\t', names=['index', 'word', 'POS'])
train.head()
word = train['word'].values.tolist()
index = train['index'].values.tolist()
pos = train['POS'].values.tolist()
vocab = {}

for i in range(len(word)):
    if word[i] in vocab:
        vocab[word[i]] += 1
    else:
        vocab[word[i]] = 1
# replace rare words with <unk> (threshold = 3)
vocab2 = {}
num_unk = 0
```

```
for w in vocab:
    if vocab[w] >= 3:
        vocab2[w] = vocab[w]
    else:
        num_unk += vocab[w]

# sort the vocabulary by occurrences of words
vocab_sorted = sorted(vocab.items(), key=lambda item: item[1], reverse=True)
# write the sorted vocabulary to vocab file
#with open('recap/vocab.txt', 'w') as vocab_file:
with open('output/vocab_frequent', 'w') as vocab_file:
    # the format of the vocab is word index occurrence
    # we add <unk> to the top of the vocabulary manually
    vocab_file.write('<unk>' + '\t' + str(0) + '\t' + str(num_unk) + '\n')
    for i in range(len(vocab_sorted)):
        vocab_file.write(vocab_sorted[i][0] + '\t' + str(i+1) + '\t' + str(vocab_sorted[i][1]) +
'\n')
print(f'The total size of my vocabulary is {len(vocab_sorted)}\n')
print(f'The total occurrences of <unk> is {num_unk}\n')
```

Task 2: Model Learning

```
# build a vocabulary list with only frequent words (i.e. occur no less than 3 times)
vocab_ls = list(vocab2.keys())

# write the frequent words into a json file
#with open('recap/vocab_frequent.txt', 'w') as output:
with open('output/vocal_frequent', 'w') as output:
    for word in vocab_ls:
        output.write(word + '\n')

# replace non-frequent words in word with <unk>
            # for ss, we need to count the times that a pos tag occurs at
            the beginning# of a sequence (i.e. (s|<s>))
            for i in
                range(len(
                word)):if
                index[i]
                == 1:
                    if str(pos[i]) + '|' + '<s>' in ss:
                        ss[str(pos[i]) + '|'
                    + '<s>'] += 1else:
                        ss[str(pos[i]) + '|' + '<s>'] = 1

            # build an emission and a transition
            dictionariesemission = {}
            transition = {}

            # count
            occurrences of pos
            tagscount_pos = {}
```

```
for p in pos:
    if p in count_pos:
        count_pos[p] += 1
    else:
        count_pos[p] = 1

# don't forget to count the occurrences of <start>
count_pos['<s>'] = 0
for i in index:
    if i == 1:
        count_pos['<s>'] += 1

# emission dictionary {(s, x): count(s, x) / count(s)}
for sx_pair in sx:
    emission[sx_pair] = sx[sx_pair] / count_pos[sx_pair.split('|')[1]]

# transition dictionary {(s, s'): count(s, s') / count(s)}
for ss_pair in ss:
    transition[ss_pair] = ss[ss_pair] / count_pos[ss_pair.split('|')[1]]

print(f'There are {len(transition)} transition parameters in my HMM\n')
print(f'There are {len(emission)} emission parameters in my HMM\n')

# write the emission and transition dictionaries into a json file
emission_transition = [emission, transition]
#with open('recap/hmm.json', 'w') as output:
with open('output/hmm.json', 'w') as output:
    json.dump(emission_transition, output)
# build a list of distinct pos
pos_distinct = list(count_pos.keys())

# write the pos_distinct into a txt file
#with open('recap/pos.txt', 'w') as pos_output:
with open('output/pos.txt', 'w') as pos_output:
    for _, pos in enumerate(pos_distinct):
        pos_output.write(pos + '\n')
```

Task 3: Greedy Decoding with HMM

```
    # load txt file vocab
    vocab_frequent = []
    #with open('recap/vocab_frequent.txt', 'r') as vocab_txt:
    with open('output/vocab_frequent.txt', 'r') as vocab_txt:
        for word in vocab_txt:
            word = word.strip('\n')
            vocab_frequent.append(word)
    vocab_frequent

    # load txt file pos
    pos_distinct = []

    #with open('recap/pos.txt', 'r') as pos_txt:
    with open('output/pos.txt', 'r') as pos_txt:
        for pos in pos_txt:
            pos = pos.strip('\n')
            pos_distinct.append(pos)

# load json file hmm
#with open('recap/hmm.json', 'r') as hmm:
with open('output/hmm.json', 'r') as hmm:
    json_data = json.load(hmm)

emission, transition = json_data[0], json_data[1]

#dev = pd.read_csv('data1/dev', sep='\t', names=['index', 'word', 'POS'])
dev = pd.read_csv('data1/dev', sep='\t', names=['index', 'word', 'POS'])
dev.head()
index_dev = dev.loc[:, 'index'].values.tolist()
word_dev = dev.loc[:, 'word'].values.tolist()
pos_dev = dev.loc[:, 'POS'].values.tolist()


# split dev lists (index, word and pos) to individual samples (list --> list of sublists)
word_dev2 = []
pos_dev2 = []
word_sample = []
pos_sample = []
for i in range(len(dev)-1):
    if index_dev[i] < index_dev[i+1]:
        word_sample.append(word_dev[i])
        pos_sample.append(pos_dev[i])
    else:
        word_sample.append(word_dev[i])
        word_dev2.append(word_sample)
        word_sample = []

        pos_sample.append(pos_dev[i])
        pos_dev2.append(pos_sample)
        pos_sample = []
```

```
def greedy(sentence):
    # initialize a dictionary to keep track of the pos for each position
    pos = []

    # predict the pos of the first word in the sentence

    # we need to make sure the first word is in the vocabulary. If not, replace
    # with <unk>
    if sentence[0] not in vocab_frequent:
        sentence[0] = '<unk>'
    # predict pos based on the product of the emission and transition
    max_prob = 0
    p0 = 'UNK'

    for p in pos_distinct:
        try:
            temp = emission[sentence[0] + '|' + p] * transition[p + '|' + '<s>']
            if temp > max_prob:
                max_prob = temp
                p0 = p
        except:
            pass

    pos.append(p0)

    # predict the pos of the remaining words


    for i in range(1, len(sentence)):
        # again, we need to check the existence of the word in the vocabulary.
        if sentence[i] not in vocab_frequent:
            sentence[i] = '<unk>'

        max_prob = 0
        pi = 'UNK'
        for p in pos_distinct:
            try:
                temp = emission[sentence[i] + '|' + p] * transition[p + '|' + pos[-1]]
                if temp > max_prob:
                    max_prob = temp
                    pi = p
            except:
                pass

    pos.append(pi)

    return pos
pos_greedy = [greedy(s) for s in word_dev2]
# concatenate the list of sublists into one single list
pos_greedy = fc.reduce(lambda a, b: a + b, pos_greedy)
pos_dev = fc.reduce(lambda a, b: a + b, pos_dev2)

acc = accuracy_score(pos_dev, pos_greedy)
print('The prediction accuracy on the dev data is {:.2f}%'.format(acc * 100))
```

Task 4: Viterbi Decoding with HMM

```
# load txt file vocab
vocab_frequent = []
#with open('recap/vocab_frequent.txt', 'r') as vocab_txt:
with open('output/vocab_frequent.txt', 'r') as vocab_txt:
    for word in vocab_txt:
        word = word.strip('\n')
        vocab_frequent.append(word)
# load txt file pos
pos_distinct = []

#with open('recap/pos.txt', 'r') as pos_txt:
with open('output/pos.txt', 'r') as pos_txt:
    for pos in pos_txt:
        pos = pos.strip('\n')
        pos_distinct.append(pos)
# load json file hmm
#with open('recap/hmm.json', 'r') as hmm:
with open('output/hmm.json', 'r') as hmm:
    json_data = json.load(hmm)

emission, transition = json_data[0], json_data[1]

#dev = pd.read_csv('data1/dev', sep='\t', names=['index', 'word', 'POS'])
dev = pd.read_csv('data1/dev', sep='\t', names=['index', 'word', 'POS'])
dev.head()

index_dev = dev.loc[:, 'index'].values.tolist()
word_dev = dev.loc[:, 'word'].values.tolist()
pos_dev = dev.loc[:, 'POS'].values.tolist()

# split dev lists (index, word and pos) to individual samples (list --> list of sublists)
word_dev2 = []
pos_dev2 = []
word_sample = []
pos_sample = []
for i in range(len(dev)-1):
    if index_dev[i] < index_dev[i+1]:
        word_sample.append(word_dev[i])
        pos_sample.append(pos_dev[i])
    else:
        word_sample.append(word_dev[i])
        word_dev2.append(word_sample)
        word_sample = []

        pos_sample.append(pos_dev[i])
        pos_dev2.append(pos_sample)
        pos_sample = []
```

```
# define a function to predict the pos for an input sentence
def viterbi(sentence):
    # initialize a dictionary that keeps track of the highest cumulative probability of each
possible
    # pos at each position of the input sentence
    seq = {i:{} for i in range(len(sentence))}
    # also initialize a dictionary that keeps track of the pos of the previous pos that leads to the
    # highest cumulative probability of each possible pos at each position of the input sentence
    # for instance, for a pos of NNP at position i, we want to know which pos of position i-1
leads to
    # the highest cumulative probability of NNP at position i.
    pre_pos = {i:{} for i in range(len(sentence))}

    # for the first position, the highest cumulative probability of each possible pos would be
    # emission[sentence[0]|pos] * transition[pos|<s>]

    # check if the first word is in the vocabualry. If not, replace with '<unk>'
    if sentence[0] not in vocab_frequent:
        sentence[0] = '<unk>'

    for p in pos_distinct:
        if p + '|' + '<s>' in transition:
            try:
                seq[0][p] = transition[p + '|' + '<s>'] * \
                        emission[sentence[0] + '|' + p]
            except:
                seq[0][p] = 0
    # set <s> as the previous pos of each possible pos at the first position
    for p in seq[0].keys():
        pre_pos[0][p] = '<s>'

    # for position i > 0, the highest cumulative probability of each possible pos would be
    # emission[sentence[i]|pos[i]] * transition[pos[i]|pos[i-1]] * seq[i-1][pos]
    for i in range(1, len(sentence)):
        # still, check if the word is in the vocabulary
        if sentence[i] not in vocab_frequent:
            sentence[i] = '<unk>'
        for p in seq[i-1].keys():
            for p_prime in pos_distinct:
                if p_prime + '|' + p in transition:
                    if p_prime in seq[i]:
                        try:
                            temp = seq[i-1][p]  *  \
                                transition[p_prime + '|' + p] * \
                                emission[sentence[i] + '|' + p_prime]
                            if temp > seq[i][p_prime]:
                                seq[i][p_prime]  =  temp
                                pre_pos[i][p_prime] = p
                        except:
                            pass
```

```
                else:
                    try:
                        seq[i][p_prime] = seq[i-1][p] * \
                                          transition[p_prime + '|' + p] * \
                                          emission[sentence[i] + '|' + p_prime]
                        pre_pos[i][p_prime] = p
                    except:
                        seq[i][p_prime] = 0
    # after we get the maximum probability for every possible pos at every position of a
sentence,
    # we can trace backward to find out our prediction on the pos for the sentence.
    seq_predict = []

    # The pos of the last word in the sentence is the one with the highest probability
    # after predicting the pos of the last word in the sentence, we can iterate through pre_pos to
predict
    # the pos of the remaining words in the input sentence in the reverse order

    # the highest probability
    prob_max = max(seq[len(sentence)-1].values())
    # the index of the highest probability
    index_max = list(seq[len(sentence)-1].values()).index(prob_max)
    # the pos of the highest probability
    pos_max = list(seq[len(sentence)-1].keys())[index_max]
    seq_predict.append(pos_max)

    # iterate through pre_pos
    for i in range(len(sentence)-1, 0, -1):
        # for some rare ss or sx pairs, there is no corresponding key in the
        # transition or emission dictionary. In this case, we need to set manually
        # the pos to 'UNK' at those positions
        try:
            pos_max = pre_pos[i][pos_max]
            seq_predict.append(pos_max)
        except:
            seq_predict.append('UNK')

    # The final seq_predict should be the reverse of the original
    seq_predict = [seq_predict[i] for i in range(len(seq_predict)-1, -1, -1)]
    return seq_predict
# use viterbi to predict pos for dev
pos_viterbi = [viterbi(s) for s in word_dev2]

# merge the list of sublists to a single list
pos_viterbi = fc.reduce(lambda a, b: a + b, pos_viterbi)
pos_dev = fc.reduce(lambda a, b: a + b, pos_dev2)

acc = accuracy_score(pos_dev, pos_viterbi)
print('The prediction accuracy on the dev data is {:.2f}%'.format(acc * 100))
```

Output:

	index	word	POS
0	1	Pierre	NNP
1	2	Vinken	NNP
2	3	,	,
3	4	61	CD
4	5	years	NNS

	index	word	POS
0	1	The	DT
1	2	Arizona	NNP
2	3	Corporations	NNP
3	4	Commission	NNP
4	5	authorized	VBD

```
      The total size of my vocabulary is 43193
      The total occurrences of <unk> is 32537

      There are 7 transition parameters in my HMM
      There are 6 emission parameters in my HMM
['Pierre',
 '',

 '61', 'years', 'old', 'will', 'join', 'the', 'board', 'as', 'a', 'nonexecutive',
 'director', 'Nov.', '29', '.', 'Mr.', 'is', 'chairman', 'of', 'N.V.', 'Dutch',
 'publishing', 'group', 'Rudolph', 'Agnew', '55', 'and', 'former', 'Consolidated',
 'Gold', 'Fields', 'PLC', 'was', 'named', 'this', 'British', 'industrial', 'conglomerate', 'A', 'form', 'asbesto
s', 'once', 'used', 'to', 'make', 'Kent', 'cigarette',
 'filters', 'has', 'caused', 'high', 'percentage', 'cancer', 'deaths', 'among',
 'workers', 'exposed', 'it', 'more',]


  The prediction accuracy on the dev data is 0.00%

          index        word           POS
        0       1      The            DT
        1       2      Arizona        NNP
        2       3      Corporations   NNP
        3       4      Commission     NNP
        4       5      authorized     VBD

  The prediction accuracy on the dev data is 0.01%
```

Result:

Thus written a python program to implemented successfully for the parts of speech tagging using Sequence to Sequence architecture.

Ex. No: 7	MACHINE TRANSLATION USING ENCODER-DECODER MODEL
Date :	

Aim:

To write a python program to implement the Machine Translation using Encoder-Decoder model

Algorithm:

Start the program
Input data files are available in the read-only
Decoder_target_data is ahead of decoder_input_data by one timestep
Set up the decoder, using `encoder_states` as initial state
Run training
Reverse-lookup token index to decode sequences back to something readable.
Generate empty target sequence of length 1.
Sampling loop for a batch of sequences (to simplify, here we assume a batch of size 1).
Sample a token
Exit condition: either hit max length or find stop character.
Update the target sequence (of length 1).
Update states
Stop the program

Program:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under
the
  input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
   for filename in filenames:
      print(os.path.join(dirname, filename))

# You can write up to 5GB to the current directory (/kaggle/working/) that gets preserved
as output
  when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved outside of
the
   current session
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input,LSTM,Dense

batch_size=64
epochs=100
latent_dim=256 # here latent dim represent hidden state or cell state
num_samples=10000
```

```
data_path='fra.txt'
# Vectorize the data.
input_texts = []
target_texts = []
input_characters = set()
target_characters = set()
with open(data_path, 'r', encoding='utf-8') as f:
    lines = f.read().split('\n')
for line in lines[: min(num_samples, len(lines) - 1)]:
    input_text, target_text, _ = line.split('\t')
    # We use "tab" as the "start sequence" character
    # for the targets, and "\n" as "end sequence" character.
    target_text = '\t' + target_text + '\n'
    input_texts.append(input_text)
    target_texts.append(target_text)
    for char in input_text:
        if char not in input_characters:
            input_characters.add(char)
    for char in target_text:
        if char not in target_characters:
            target_characters.add(char)
input_characters=sorted(list(input_characters))
target_characters=sorted(list(target_characters))

num_encoder_tokens=len(input_characters)
num_decoder_tokens=len(target_characters)

max_encoder_seq_length=max([len(txt) for txt in input_texts])
max_decoder_seq_length=max([len(txt) for txt in target_texts])
print('Number of samples:', len(input_texts))
print('Number of unique input tokens:', num_encoder_tokens)
print('Number of unique output tokens:', num_decoder_tokens)
print('Max sequence length for inputs:', max_encoder_seq_length)
print('Max sequence length for outputs:', max_decoder_seq_length)

input_token_index=dict(
    [(char,i) for i, char in enumerate(input_characters)])
target_token_index=dict(
[(char,i) for i, char in enumerate(target_characters)])

encoder_input_data = np.zeros(
    (len(input_texts), max_encoder_seq_length, num_encoder_tokens),
    dtype='float32')
decoder_input_data = np.zeros(
    (len(input_texts), max_decoder_seq_length, num_decoder_tokens),
    dtype='float32')
decoder_target_data = np.zeros(
    (len(input_texts), max_decoder_seq_length, num_decoder_tokens),
    dtype='float32')

for i, (input_text, target_text) in enumerate(zip(input_texts, target_texts)):
    for t, char in enumerate(input_text):
        encoder_input_data[i, t, input_token_index[char]] = 1.
    encoder_input_data[i, t + 1:, input_token_index[' ']] = 1.
```

```
    for t, char in enumerate(target_text):
        # decoder_target_data is ahead of decoder_input_data by one timestep
        decoder_input_data[i, t, target_token_index[char]] = 1.
        if t > 0:
            # decoder_target_data will be ahead by one timestep
            # and will not include the start character.
            decoder_target_data[i, t - 1, target_token_index[char]] = 1.
    decoder_input_data[i, t + 1:, target_token_index[' ']] = 1.
    decoder_target_data[i, t:, target_token_index[' ']] = 1.
encoder_inputs = Input(shape=(None, num_encoder_tokens))
encoder = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
# We discard `encoder_outputs` and only keep the states.
encoder_states = [state_h, state_c]

# Set up the decoder, using `encoder_states` as initial state.
decoder_inputs = Input(shape=(None, num_decoder_tokens))
# We set up our decoder to return full output sequences,
# and to return internal states as well. We don't use the
# return states in the training model, but we will use them in inference.
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(decoder_inputs,
                                     initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_outputs)

model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

# Run training
model.compile(optimizer='rmsprop', loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit([encoder_input_data, decoder_input_data], decoder_target_data,
          batch_size=batch_size,
          epochs=epochs,
          validation_split=0.2)

model.save('eng2french.h5')

encoder_model = Model(encoder_inputs, encoder_states)

decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
decoder_outputs, state_h, state_c = decoder_lstm(
    decoder_inputs, initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)
decoder_model = Model(
    [decoder_inputs] + decoder_states_inputs,
    [decoder_outputs] + decoder_states)

# Reverse-lookup token index to decode sequences back to
# something readable.
reverse_input_char_index = dict(
```

```
    (i, char) for char, i in input_token_index.items())
reverse_target_char_index = dict(
    (i, char) for char, i in target_token_index.items())
def decode_sequence(input_seq):
    # Encode the input as state vectors.
    states_value = encoder_model.predict(input_seq)

    # Generate empty target sequence of length 1.
    target_seq = np.zeros((1, 1, num_decoder_tokens))
    # Populate the first character of target sequence with the start character.
    target_seq[0, 0, target_token_index['\t']] = 1.

    # Sampling loop for a batch of sequences
    # (to simplify, here we assume a batch of size 1).
    stop_condition = False
    decoded_sentence = ''
    while not stop_condition:
        output_tokens, h, c = decoder_model.predict(
            [target_seq] + states_value)

        # Sample a token
        sampled_token_index = np.argmax(output_tokens[0, -1, :])
        sampled_char = reverse_target_char_index[sampled_token_index]
        decoded_sentence += sampled_char

        # Exit condition: either hit max length
        # or find stop character.
        if (sampled_char == '\n' or
           len(decoded_sentence) > max_decoder_seq_length):
            stop_condition = True

        # Update the target sequence (of length 1).
        target_seq = np.zeros((1, 1, num_decoder_tokens))
        target_seq[0, 0, sampled_token_index] = 1.

        # Update states
        states_value = [h, c]

    return decoded_sentence

for seq_index in range(100):
    # Take one sequence (part of the training set)
    # for trying out decoding.
    input_seq = encoder_input_data[seq_index: seq_index + 1]
    decoded_sentence = decode_sequence(input_seq)
    print('-')
    print('Input sentence:', input_texts[seq_index])
    print('Decoded sentence:', decoded_sentence)
```

Output:

```
    Number of samples: 10000
    Number of unique input tokens: 71
    Number of unique output tokens: 93
    Max sequence length for inputs: 15
    Max sequence length for outputs: 59
```

Epoch 1/100
125/125 [==============================] - 15s 105ms/step - loss: 1.2150 - accuracy: 0.7315 - val_loss: 1.0873 - val_accuracy: 0.7068
Epoch 2/100
125/125 [==============================] - 13s 106ms/step - loss: 0.9334 - accuracy: 0.7490 - val_loss: 0.9959 - val_accuracy: 0.7128
Epoch 3/100
125/125 [==============================] - 13s 105ms/step - loss: 0.8396 - accuracy: 0.7679 - val_loss: 0.9039 - val_accuracy: 0.7500
…
Epoch 98/100
125/125 [==============================] - 13s 107ms/step - loss: 0.1532 - accuracy: 0.9531 - val_loss: 0.5529 - val_accuracy: 0.8705
Epoch 99/100
125/125 [==============================] - 13s 108ms/step - loss: 0.1517 - accuracy: 0.9533 - val_loss: 0.5561 - val_accuracy: 0.8697
Epoch 100/100
125/125 [==============================] - 13s 108ms/step - loss: 0.1497 - accuracy: 0.9543 - val_loss: 0.5522 - val_accuracy: 0.8706

Input sentence: Smile.
Decoded sentence: Pours pres votr.

1/1 [==============================] - 0s 14ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 18ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 15ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 16ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 18ms/step
-
Input sentence: Sorry?
Decoded sentence: Pardon ?

1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 17ms/step

```
1/1 [==============================] - 0s 13ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 19ms/step
1/1 [==============================] - 0s 23ms/step
1/1 [==============================] - 0s 17ms/step
1/1 [==============================] - 0s 13ms/step
-
```

Result:

Thus written a python program to implemented successfully for the Machine Translation using Encoder-Decoder model.

Ex. No: 8	IMAGE AUGMENTATION USING GANS
Date :	

Aim:

To write a python program to implement the Image augmentation using GANs

Algorithm:

Start the program
Load the images from the directory.
Split the data into train and test images
Training the CNN using the two different sized datasets with no augmented data.
Train a network based on ResNet-50V2 with data augmentation
Preprocess the image and submit it to the network for classification.
Now try it with a walrus image that the network hasn't seen before. Start by loading the image.
Finally, Preprocess the image and make a prediction.
Stop the program.

Program:

```
import os
import numpy as np
import keras.utils as image
import matplotlib.pyplot as plt
%matplotlib inline

def load_images_from_path(path, label):
    images = []
    labels = []

    for file in os.listdir(path):
        img = image.load_img(os.path.join(path, file), target_size=(224, 224, 3))
        images.append(image.img_to_array(img))
        labels.append((label))

    return images, labels

def show_images(images):
    fig, axes = plt.subplots(1, 8, figsize=(20, 20), subplot_kw={'xticks': [], 'yticks':
[]})

    for i, ax in enumerate(axes.flat):
        ax.imshow(images[i] / 255)

x_train = []
y_train = []
x_test = []
```

```
y_test = []

images, labels = load_images_from_path('arctic-wildlife/train/arctic_fox', 0)
show_images(images)
x_train += images
y_train += labels

images, labels = load_images_from_path('arctic-wildlife/train/walrus', 2)
show_images(images)
x_train += images
y_train += labels

images, labels = load_images_from_path('arctic-wildlife/test/polar_bear', 1)
show_images(images)
x_test += images
y_test += labels


from tensorflow.keras.utils import to_categorical

from tensorflow.keras.applications.resnet50 import preprocess_input

x_train = preprocess_input(np.array(x_train))

x_test = preprocess_input(np.array(x_test))

y_train_encoded  =  to_categorical(y_train)

y_test_encoded = to_categorical(y_test)

from tensorflow.keras.applications import  ResNet50V2

base_model = ResNet50V2(weights='imagenet', include_top=False)

for layer in base_model.layers:

    layer.trainable = False

from keras.models import Sequential

from keras.layers import Flatten, Dense, Dropout

from keras.layers import Rescaling, RandomFlip, RandomRotation,
RandomTranslation, RandomZoom

model = Sequential()

model.add(Rescaling(1./255))

model.add(RandomFlip(mode='horizontal'))

model.add(RandomTranslation(0.2, 0.2))

model.add(RandomRotation(0.2))

model.add(RandomZoom(0.2))
```

```
model.add(base_model)
model.add(Flatten())
model.add(Dense(1024, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(3, activation='softmax'))
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
hist = model.fit(x_train, y_train_encoded, validation_data=(x_test,
y_test_encoded), batch_size=10, epochs=25)
acc = hist.history['accuracy']
val_acc = hist.history['val_accuracy']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, acc, '-', label='Training Accuracy')
plt.plot(epochs, val_acc, ':', label='Validation Accuracy')

plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.plot()

from sklearn.metrics import confusion_matrix
import seaborn as sns
sns.set()
y_predicted = model.predict(x_test)
mat = confusion_matrix(y_test_encoded.argmax(axis=1),
y_predicted.argmax(axis=1))
class_labels = ['arctic fox', 'polar bear', 'walrus']
sns.heatmap(mat, square=True, annot=True, fmt='d', cbar=False, cmap='Blues',
        xticklabels=class_labels,
        yticklabels=class_labels)
plt.xlabel('Predicted label')
```

```
plt.ylabel('Actual label')
x = image.load_img('arctic-wildlife/samples/arctic_fox/arctic_fox_140.jpeg',
target_size=(224, 224))
plt.xticks([])
plt.yticks([])
plt.imshow(x)
x = image.img_to_array(x)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)
predictions = model.predict(x)
for i, label in enumerate(class_labels):
    print(f'{label}: {predictions[0][i]}')
x = image.load_img('arctic-wildlife/samples/walrus/walrus_143.png',
target_size=(224, 224))
plt.xticks([])
plt.yticks([])
plt.imshow(x)
x = image.img_to_array(x)
x = np.expand_dims(x, axis=0)
x = preprocess_input(x)
predictions = model.predict(x)

for i, label in enumerate(class_labels):
    print(f'{label}: {predictions[0][i]}')
```

Output:

Train :

Test :

Train a network based on ResNet-50V2 with data augmentation

Epoch 1/25

30/30 [==============================] - 27s 848ms/step - loss: 31.6208 - accuracy: 0.7400 - val_loss: 8.3153 - val_accuracy: 0.8667

Epoch 2/25

30/30 [==============================] - 25s 848ms/step - loss: 6.1519 - accuracy: 0.8900 - val_loss: 2.5947 - val_accuracy: 0.9583

Epoch 3/25

30/30 [==============================] - 26s 870ms/step - loss: 3.0037 - accuracy: 0.9467 - val_loss: 2.7972 - val_accuracy: 0.9750

.

.

.

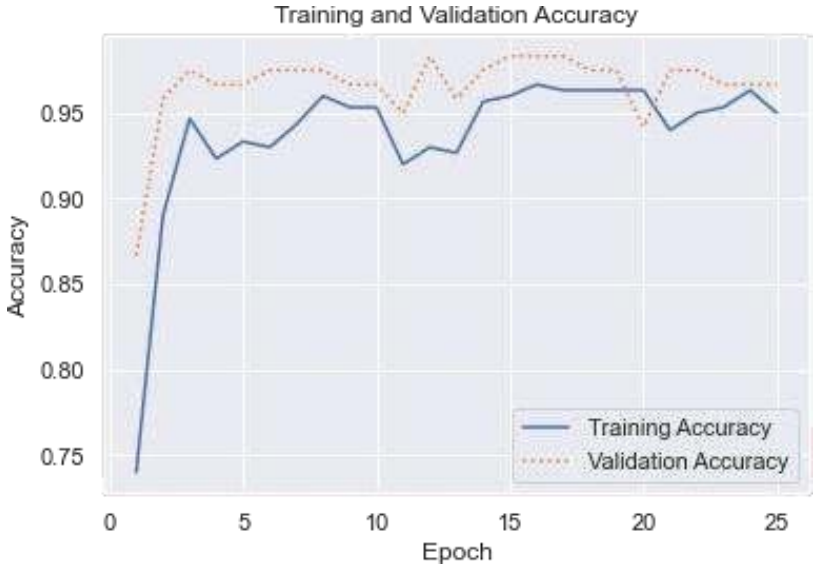
.

Epoch 24/25

30/30 [==============================] - 27s 896ms/step - loss: 0.5841 - accuracy: 0.9633 - val_loss: 0.5701 - val_accuracy: 0.9667

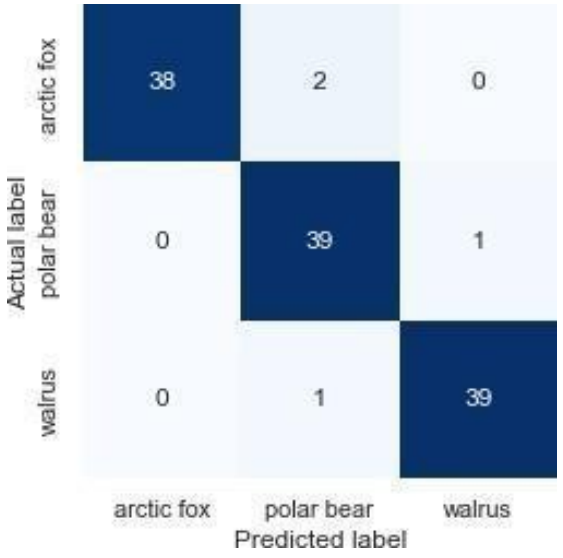
Epoch 25/25

30/30 [==============================] - 25s 844ms/step - loss: 0.7861 - accuracy: 0.9500 - val_loss: 0.5762 - val_accuracy: 0.9667

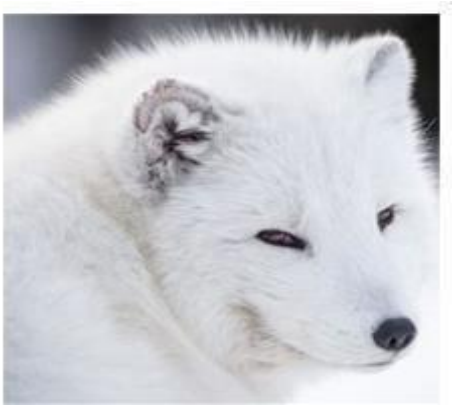


4/4 [==============================] - 3s 635ms/step

Text(89.18, 0.5, 'Actual label')



<matplotlib.image.AxesImage at 0x2c5496dfa00>



Preprocess the image and submit it to the network for classification.

1/1 [==============================] - 0s 57ms/step

arctic fox: 1.0

polar bear: 1.1824497264458205e-28

walrus: 0.0

Now try it with a walrus image that the network hasn't seen before. Start by loading the image.

<matplotlib.image.AxesImage at 0x2c546cb7790>

Preprocess the image and make a prediction.

1/1 [==============================] - 0s 51ms/step

arctic fox: 0.0

polar bear: 0.0

walrus: 1.0

Result:

Thus written a python program to implemented successfully for the Image augmentation using GANs.

Ex.No:9	MINI-PROJECT ON REAL WORLD APPLICATIONS
Date:	

AIM:

To implement a Mini-Project on real world applications [Hand Written Recognition] using deep learning.

INTRODUCTION:

A "AD3511 mini project" in real-world applications would typically refer to a small-scale project utilizing deep learning concepts (likely covered in a course with the code "AD3511") to solve a practical problem in a real-world scenario, such as: image recognition for product classification in e- commerce, sentiment analysis on customer reviews, facial recognition for security systems, anomaly detection in industrial processes, or medical diagnosis using image analysis; essentially applying deep learning techniques to address tangible issues across various industries

Specific examples of AD3511 mini projects with real-world applications:

- **Visual Inspection in Manufacturing:**
- Detecting defects on a production line using convolutional neural networks (CNNs) to analyze images of manufactured products for quality control
- **Healthcare Diagnostics:**
- Identifying abnormalities in medical scans (X-rays, MRIs) using CNNs to assist doctors in disease diagnosis.
- Analyzing retinal images to detect signs of diabetic retinopathy.

Autonomous Vehicle Navigation:

- Object detection in real-time traffic scenes using CNNs to identify pedestrians, vehicles, and traffic signs for self-driving car applications.

Natural Language Processing (NLP):

- Sentiment analysis of customer reviews to gauge customer satisfaction levels on social media or online platforms.
- Chatbot development for customer service using recurrent neural networks (RNNs) to understand natural language queries.
- **Retail Analytics:**
- Customer behavior analysis by tracking customer movement patterns in stores using video footage and object tracking algorithms.

Key points to consider for an AD3511 mini project:

- **Data Availability:**

Ensure access to a relevant and sufficient dataset for training the deep learning model.

- **Problem Definition:**

Clearly state the specific problem you want to address and the desired outcome of your project.

- **Model Selection:**

Choose an appropriate deep learning architecture (CNN, RNN, etc.) based on the nature of your data and task.

- **Evaluation Metrics:**

Define relevant metrics to measure the performance of your model (accuracy, precision, recall, F1 - score).

HANDWRITTEN DIGIT RECOGNITION USING NEURAL NETWORK

INTRODUCTION:

Handwritten digit recognition using MNIST dataset is a major project made with the help of Neural Network. It basically detects the scanned images of handwritten digits.
We have taken this a step further where our handwritten digit recognition system not only detects scanned images of handwritten digits but also allows writing digits on the screen with the help of an integrated GUI for recognition.

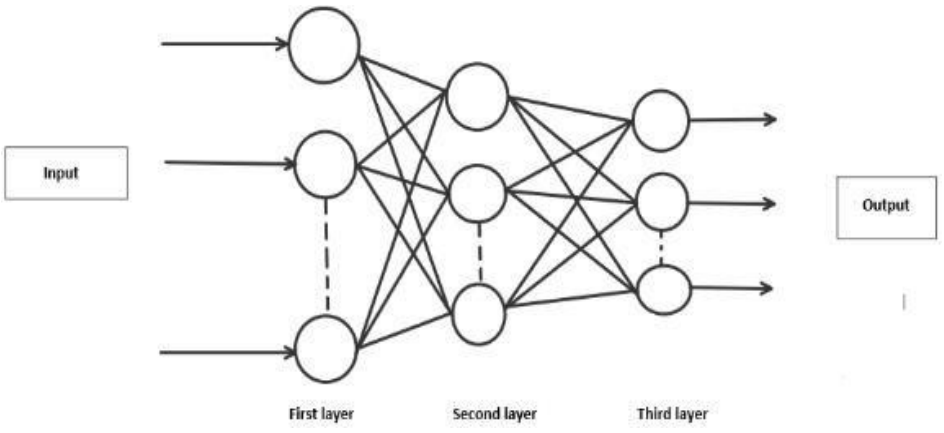
APPROACH:
We will approach this project by using a three-layered Neural Network.

- **The input layer:** It distributes the features of our examples to the next layer for calculation of activations of the next layer.
- **The hidden layer:** They are made of hidden units called activations providing nonlinear ties for the network. A number of hidden layers can vary according to our requirements.
- **The output layer:** The nodes here are called output units. It provides us with the final prediction of the Neural Network on the basis of which final predictions can be made.

A neural network is a model inspired by how the brain works. It consists of multiple layers having many activations, this activation resembles neurons of our brain. A neural network tries to learn a set of parameters in a set of data which could help to recognize the underlying relationships. Neural networks can adapt to changing input; so the network generates the best possible result without needing to redesign the output criteria.

METHODOLOGY:

We have implemented a Neural Network with 1 hidden layer having *100* activation units (excluding bias units). The data is loaded from a *.mat* file, features(X) and labels(y) were extracted. Then features are divided by *255* to rescale them into a range of *[0,1]* to avoid overflow during computation. Data is split up into *60,000* training and *10,000* testing examples. Feedforward is performed with the training set for calculating the hypothesis and then backpropagation is done in order to reduce the error between the layers. The regularization parameter lambda is set to 0.1 to address the problem of overfitting. Optimizer is run for 70 iterations to find the best fit model.



Layers of Neural Network

Note:

- Save all *.py* files in the same directory.
- Download dataset from Kaggle.

Main.py

Importing all the required libraries, extract the data from *mnist-original.mat* file. Then features and labels will be separated from extracted data. After that data will be split into training (60,000) and testing (10,000) examples. Randomly initialize Thetas in the range of [-0.15, +0.15] to break symmetry and get better results. Further, the optimizer is called for the training of weights, to minimize the cost function for appropriate predictions. We have used the "*minimize*" optimizer from "*scipy.optimize*" library with "*L-BFGS-B*" method. We have calculated the test, the "training set accuracy and precision using "predict" function.

```
from scipy.io import loadmat

import numpy as np

from Model import neural_network

from RandInitialize import initialise

from Prediction import predict

from scipy.optimize import minimize

# Loading mat file

data = loadmat('mnist-original.mat')

# Extracting features from mat file

X = data['data']
```

```
X = X.transpose()
# Normalizing the data
X = X / 255
# Extracting labels from mat file
y = data['label']
y = y.flatten()
# Splitting data into training set with 60,000 examples
X_train = X[:60000, :]
y_train = y[:60000]
# Splitting data into testing set with 10,000 examples
X_test = X[60000:, :]
y_test = y[60000:]
m = X.shape[0]
input_layer_size = 784  # Images are of (28 X 28) px so there will be 784 features
hidden_layer_size = 100
num_labels = 10  # There are 10 classes [0, 9]
# Randomly initialising Thetas
initial_Theta1 = initialise(hidden_layer_size, input_layer_size)
initial_Theta2 = initialise(num_labels, hidden_layer_size)
# Unrolling parameters into a single column vector
initial_nn_params = np.concatenate((initial_Theta1.flatten(), initial_Theta2.flatten()))
maxiter = 100
lambda_reg = 0.1  # To avoid overfitting
myargs = (input_layer_size, hidden_layer_size, num_labels, X_train, y_train, lambda_reg)
```

```
# Calling minimize function to minimize cost function and to train weights
results = minimize(neural_network, x0=initial_nn_params, args=myargs,
        options={'disp': True, 'maxiter': maxiter}, method="L-BFGS-B", jac=True)
nn_params = results["x"]  # Trained Theta is extracted
# Weights are split back to Theta1, Theta2
Theta1 = np.reshape(nn_params[:hidden_layer_size * (input_layer_size + 1)], (
                    hidden_layer_size, input_layer_size + 1))  # shape = (100, 785)
Theta2 = np.reshape(nn_params[hidden_layer_size * (input_layer_size + 1):],
                (num_labels, hidden_layer_size + 1))  # shape = (10, 101)
# Checking test set accuracy of our model
pred = predict(Theta1, Theta2, X_test)
print('Test Set Accuracy: {:f}'.format((np.mean(pred == y_test) * 100)))
# Checking train set accuracy of our model
pred = predict(Theta1, Theta2, X_train)
print('Training Set Accuracy: {:f}'.format((np.mean(pred == y_train) * 100)))
# Evaluating precision of our model
true_positive = 0
for i in range(len(pred)):
    if pred[i] == y_train[i]:
        true_positive += 1
false_positive = len(y_train) - true_positive
print('Precision =', true_positive/(true_positive + false_positive))
# Saving Thetas in .txt file
np.savetxt('Theta1.txt', Theta1, delimiter=' ')
np.savetxt('Theta2.txt', Theta2, delimiter=' ')
```

RandInitialise.py

It randomly initializes theta between a range of [-epsilon, +epsilon].

```
import numpy as np

def initialise(a, b):

    epsilon = 0.15

    c = np.random.rand(a, b + 1) * (

      # Randomly initialises values of thetas between [-epsilon, +epsilon]

      2 * epsilon) - epsilon

    return c
```

Model.py

The function performs feed-forward and backpropagation.

- Forward propagation: Input data is fed in the forward direction through the network. Each hidden layer accepts the input data, processes it as per the activation function and passes it to the successive layer. We will use the sigmoid function as our "activation function".
- Backward propagation: It is the practice of fine-tuning the weights of a neural net based on the error rate obtained in the previous iteration.

It also calculates cross-entropy costs for checking the errors between the prediction and original values. In the end, the gradient is calculated for the optimization objective.

```
import numpy as np

def neural_network(nn_params, input_layer_size, hidden_layer_size, num_labels, X, y, lamb):

    Theta1 = np.reshape(nn_params[:hidden_layer_size * (input_layer_size + 1)],

                        (hidden_layer_size, input_layer_size + 1))

    Theta2 = np.reshape(nn_params[hidden_layer_size * (input_layer_size + 1):],

                        (num_labels, hidden_layer_size + 1))

    m = X.shape[0]

    one_matrix = np.ones((m, 1))

    X = np.append(one_matrix, X, axis=1)  # Adding bias unit to first layer

    a1 = X

    z2 = np.dot(X, Theta1.transpose())
```

```
    a2 = 1 / (1 + np.exp(-z2))  # Activation for second layer
    one_matrix = np.ones((m, 1))
    a2 = np.append(one_matrix, a2, axis=1)  # Adding bias unit to hidden layer
    z3 = np.dot(a2, Theta2.transpose())
    a3 = 1 / (1 + np.exp(-z3))  # Activation for third layer
    # Changing the y labels into vectors of boolean values.
    # For each label between 0 and 9, there will be a vector of length 10
    # where the ith element will be 1 if the label equals i
    y_vect = np.zeros((m, 10))
    for i in range(m):
        y_vect[i, int(y[i])] = 1
    # Calculating cost function
    J = (1 / m) * (np.sum(np.sum(-y_vect * np.log(a3) - (1 - y_vect) * np.log(1 - a3)))) + (lamb / (2 *
m)) * (
            sum(sum(pow(Theta1[:, 1:], 2))) + sum(sum(pow(Theta2[:, 1:], 2))))
    # backprop
    Delta3 = a3 - y_vect
    Delta2 = np.dot(Delta3, Theta2) * a2 * (1 - a2)
    Delta2 = Delta2[:, 1:]
    # gradient
    Theta1[:, 0] = 0
    Theta1_grad = (1 / m) * np.dot(Delta2.transpose(), a1) + (lamb / m) * Theta1
    Theta2[:, 0] = 0
    Theta2_grad = (1 / m) * np.dot(Delta3.transpose(), a2) + (lamb / m) * Theta2
    grad = np.concatenate((Theta1_grad.flatten(), Theta2_grad.flatten()))
    return J, grad
```

Prediction.py

It performs forward propagation to predict the digit.

```
import numpy as np

def predict(Theta1, Theta2, X):

    m = X.shape[0]

    one_matrix = np.ones((m, 1))

    X = np.append(one_matrix, X, axis=1)  # Adding bias unit to first layer

    z2 = np.dot(X, Theta1.transpose())

    a2 = 1 / (1 + np.exp(-z2))  # Activation for second layer

    one_matrix = np.ones((m, 1))

    a2 = np.append(one_matrix, a2, axis=1)  # Adding bias unit to hidden layer

    z3 = np.dot(a2, Theta2.transpose())

    a3 = 1 / (1 + np.exp(-z3))  # Activation for third layer

    p = (np.argmax(a3, axis=1))  # Predicting the class on the basis of max value of hypothesis

    return p
```

GUI.py

It launches a GUI for writing digits. The image of the digit is stored in the same directory after converting it to grayscale and reducing the size to *(28 X 28)* pixels.

```
from tkinter import *

import numpy as np

from PIL import ImageGrab

from Prediction import predict

window = Tk()

window.title("Handwritten digit recognition")

l1 = Label()
```

```
    def MyProject():
global l1
widget = cv
# Setting co-ordinates of canvas
x = window.winfo_rootx() + widget.winfo_x()
y = window.winfo_rooty() + widget.winfo_y()
x1 = x + widget.winfo_width()
y1 = y + widget.winfo_height()
# Image is captured from canvas and is resized to (28 X 28) px
img = ImageGrab.grab().crop((x, y, x1, y1)).resize((28, 28))
# Converting rgb to grayscale image
img = img.convert('L')
# Extracting pixel matrix of image and converting it to a vector of (1, 784)
x = np.asarray(img)
vec = np.zeros((1, 784))
k = 0
for i in range(28):
    for j in range(28):
        vec[0][k] = x[i][j]
        k += 1
# Loading Thetas
Theta1  =  np.loadtxt('Theta1.txt')
Theta2  =  np.loadtxt('Theta2.txt')
# Calling function for prediction
pred = predict(Theta1, Theta2, vec / 255)
```

```
    # Displaying the result
    l1 = Label(window, text="Digit = " + str(pred[0]), font=('Algerian', 20))
    l1.place(x=230, y=420)
lastx, lasty = None, None
# Clears the canvas
def clear_widget():
    global cv, l1
    cv.delete("all")
    l1.destroy()
# Activate canvas
def event_activation(event):
    global lastx, lasty
    cv.bind('<B1-Motion>', draw_lines)
    lastx, lasty = event.x, event.y
# To draw on canvas
def draw_lines(event):
    global lastx, lasty
    x, y = event.x, event.y
    cv.create_line((lastx, lasty, x, y), width=30, fill='white', capstyle=ROUND, smooth=TRUE,
splinesteps=12)
    lastx, lasty = x, y
# Label
L1 = Label(window, text="Handwritten Digit Recoginition", font=('Algerian', 25), fg="blue")
L1.place(x=35, y=10)
# Button to clear canvas
b1 = Button(window, text="1. Clear Canvas", font=('Algerian', 15), bg="orange", fg="black",
command=clear_widget)
```

```
b1.place(x=120, y=370)

# Button to predict digit drawn on canvas

b2 = Button(window, text="2. Prediction", font=('Algerian', 15), bg="white", fg="red",
command=MyProject)

b2.place(x=320, y=370)

# Setting properties of canvas

cv = Canvas(window, width=350, height=290, bg='black')

cv.place(x=120, y=70)

cv.bind('<Button-1>', event_activation)

window.geometry("600x500")

window.mainloop()
```

OUTPUT:
Training set accuracy of 99.440000%
Test set accuracy of 97.320000%
Precision of 0.9944

RESULT:

Thus the Mini-Project on real world applications has been completed successfully.