Experiment No. 7
Design and implement LSTM model for handwriting recognition
Date of Performance: 25/09/23
Date of Submission: 09/10/23

Aim: Design and implement LSTM model for handwriting recognition.

Objective: Ability to design a LSTM network to solve the given problem.

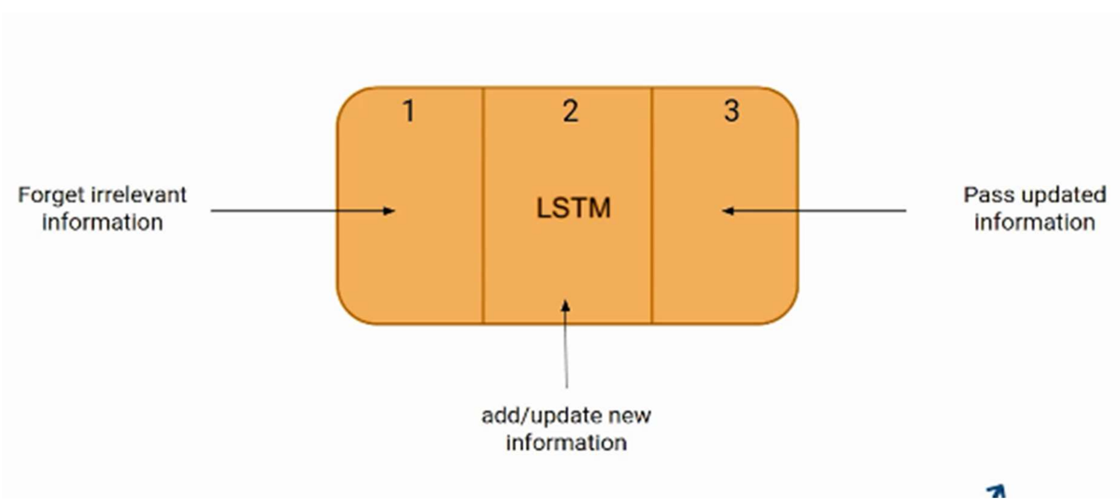
Theory:

LSTM (Long Short-Term Memory) is a recurrent neural network (RNN) architecture widely used in Deep Learning. It excels at capturing long-term dependencies, making it ideal for sequence prediction tasks.

Unlike traditional neural networks, LSTM incorporates feedback connections, allowing it to process entire sequences of data, not just individual data points. This makes it highly effective in understanding and predicting patterns in sequential data like time series, text, and speech.

LSTM Architecture

In the introduction to long short-term memory, we learned that it resolves the vanishing gradient problem faced by RNN, so now, in this section, we will see how it resolves this problem by learning the architecture of the LSTM. At a high level, LSTM works very much like an RNN cell. Here is the internal functioning of the LSTM network. The LSTM network architecture consists of three parts, as shown in the image below, and each part performs an individual function.



The Logic Behind LSTM

The first part chooses whether the information coming from the previous timestamp is to be remembered or is irrelevant and can be forgotten. In the second part, the cell tries to learn new information from the input to this cell. At last, in the third part, the cell passes the updated information from the current timestamp to the next timestamp. This one cycle of LSTM is considered a single-time step.

These three parts of an LSTM unit are known as gates. They control the flow of information in and out of the memory cell or lstm cell. The first gate is called Forget gate, the second gate is known as the Input gate, and the last one is the Output gate. An LSTM unit that consists of these three gates and a memory cell or lstm cell can be considered as a layer of neurons in traditional feedforward neural network, with each neuron having a hidden layer and a current state.

Code:

```
import os
import cv2
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import tensorflow as tf
from keras import backend as K
from keras.models import Model
from tensorflow.keras.callbacks import ModelCheckpoint
from keras.layers import Input, Conv2D, MaxPooling2D, Reshape,
Bidirectional, LSTM, Dense, Lambda, Activation, BatchNormalization,
Dropout
from keras.optimizers import Adam

train = pd.read_csv('/kaggle/input/handwriting-
recognition/written_name_train_v2.csv')
valid = pd.read_csv('/kaggle/input/handwriting-
recognition/written_name_validation_v2.csv')
train

plt.figure(figsize=(15, 10))

for i in range(9):
```

```
    ax = plt.subplot(3,3,i+1)
    img_dir = '/kaggle/input/handwriting-
recognition/train_v2/train/'+train.loc[i, 'FILENAME']
    image = cv2.imread(img_dir, cv2.IMREAD_GRAYSCALE)
    plt.imshow(image, cmap = 'gray')
    plt.title(train.loc[i, 'IDENTITY'], fontsize=12)
    plt.axis('off')

plt.subplots_adjust(wspace=0.2, hspace=-0.8)
print("Number of NaNs in train set      : ",
train['IDENTITY'].isnull().sum())
print("Number of NaNs in validation set : ",
valid['IDENTITY'].isnull().sum())

print("Number of NaNs in train set      : ",
train['IDENTITY'].isnull().sum())
print("Number of NaNs in validation set : ",
valid['IDENTITY'].isnull().sum())

train.dropna(axis=0, inplace=True)#axis =0, removing rows otherwisw
axis =1. removing columns
valid.dropna(axis=0, inplace=True) #true means dropping

unreadable = train[train['IDENTITY'] == 'UNREADABLE']
unreadable.reset_index(inplace = True, drop=True)

plt.figure(figsize=(15, 10))

for i in range(9):
    ax = plt.subplot(3, 3, i+1)
    img_dir = '/kaggle/input/handwriting-
recognition/train_v2/train/'+unreadable.loc[i, 'FILENAME']
    image = cv2.imread(img_dir, cv2.IMREAD_GRAYSCALE)
    plt.imshow(image, cmap = 'gray')
    plt.title(unreadable.loc[i, 'IDENTITY'], fontsize=12)
    plt.axis('off')

plt.subplots_adjust(wspace=0.2, hspace=-0.8)

train = train[train['IDENTITY'] != 'UNREADABLE']
valid = valid[valid['IDENTITY'] != 'UNREADABLE']
valid

train['IDENTITY'] = train['IDENTITY'].str.upper()
```

```
valid['IDENTITY'] = valid['IDENTITY'].str.upper()

train.reset_index(inplace = True, drop=True)
valid.reset_index(inplace = True, drop=True)

def preprocess(img):
    (h, w) = img.shape

    final_img = np.ones([64, 256])*255 # black white image

    # crop
    if w > 256:
        img = img[:, :256]

    if h > 64:
        img = img[:64, :]


    final_img[:h, :w] = img
    return cv2.rotate(final_img, cv2.ROTATE_90_CLOCKWISE)

train_size = 30000
valid_size= 3000

train_x = []

for i in range(train_size):
    img_dir = '/kaggle/input/handwriting-
recognition/train_v2/train/'+train.loc[i, 'FILENAME']
    image = cv2.imread(img_dir, cv2.IMREAD_GRAYSCALE)
    image = preprocess(image)
    image = image/255
    train_x.append(image)

valid_x = []

for i in range(valid_size):
    img_dir = '/kaggle/input/handwriting-
recognition/validation_v2/validation/'+valid.loc[i, 'FILENAME']
    image = cv2.imread(img_dir, cv2.IMREAD_GRAYSCALE)
    image = preprocess(image)
    image = image/255
    valid_x.append(image)
```

```
train_x = np.array(train_x).reshape(-1, 256, 64, 1)#array will get
reshaped in such a way that the resulting array has only 1 column
valid_x = np.array(valid_x).reshape(-1, 256, 64, 1) #(16384,1)
```

```
alphabets = u"!\"#&'()*+,-
./0123456789:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz "
max_str_len = 24 # max length of input labels
num_of_characters = len(alphabets) + 1 # +1 for ctc pseudo
blank(epsilon)
num_of_timestamps = 64 # max length of predicted labels


def label_to_num(label):
    label_num = []
    for ch in label:
        label_num.append(alphabets.find(ch))
        #find() method returns the lowest index of the substring if it
is found in given string otherwise -1

    return np.array(label_num)

def num_to_label(num):
    ret = ""
    for ch in num:
        if ch == -1:  # CTC Blank
            break
        else:
            ret+=alphabets[ch]
    return ret
```

```
name = 'JEBASTIN'
print(name, '\n',label_to_num(name))
```

```
train_y = np.ones([train_size, max_str_len]) * -1
train_label_len = np.zeros([train_size, 1])
train_input_len = np.ones([train_size, 1]) * (num_of_timestamps-2)
train_output = np.zeros([train_size])

for i in range(train_size):
    train_label_len[i] = len(train.loc[i, 'IDENTITY'])
```

```
    train_y[i, 0:len(train.loc[i, 'IDENTITY'])]=
label_to_num(train.loc[i, 'IDENTITY'])
```

```
valid_y = np.ones([valid_size, max_str_len]) * -1
valid_label_len = np.zeros([valid_size, 1])
valid_input_len = np.ones([valid_size, 1]) * (num_of_timestamps-2)
valid_output = np.zeros([valid_size])

for i in range(valid_size):
    valid_label_len[i] = len(valid.loc[i, 'IDENTITY'])
    valid_y[i, 0:len(valid.loc[i, 'IDENTITY'])]=
label_to_num(valid.loc[i, 'IDENTITY'])
```

```
print('True label : ',train.loc[100, 'IDENTITY'] , '\ntrain_y :
',train_y[100],'\ntrain_label_len : ',train_label_len[100],
      '\ntrain_input_len : ', train_input_len[100])
```

```
input_data = Input(shape=(256, 64, 1), name='input')

inner = Conv2D(32, (3, 3), padding='same', name='conv1',
kernel_initializer='he_normal')(input_data)
inner = BatchNormalization()(inner)
inner = Activation('relu')(inner)
inner = MaxPooling2D(pool_size=(2, 2), name='max1')(inner)

inner = Conv2D(64, (3, 3), padding='same', name='conv2',
kernel_initializer='he_normal')(inner)
inner = BatchNormalization()(inner)
inner = Activation('relu')(inner)
inner = MaxPooling2D(pool_size=(2, 2), name='max2')(inner)
inner = Dropout(0.3)(inner)

inner = Conv2D(128, (3, 3), padding='same', name='conv3',
kernel_initializer='he_normal')(inner)
inner = BatchNormalization()(inner)
inner = Activation('relu')(inner)
inner = MaxPooling2D(pool_size=(1, 2), name='max3')(inner)
inner = Dropout(0.3)(inner)

# CNN to RNN
inner = Reshape(target_shape=((64, 1024)), name='reshape')(inner)
```

```
inner = Dense(64, activation='relu', kernel_initializer='he_normal',
name='dense1')(inner)

## RNN
inner = Bidirectional(LSTM(256, return_sequences=True), name =
'lstm1')(inner)
inner = Bidirectional(LSTM(256, return_sequences=True), name =
'lstm2')(inner)

## OUTPUT
inner = Dense(num_of_characters,
kernel_initializer='he_normal',name='dense2')(inner)
y_pred = Activation('softmax', name='softmax')(inner)

model = Model(inputs=input_data, outputs=y_pred)
model.summary()
```

```
# the ctc loss function
def ctc_lambda_func(args):
    y_pred, labels, input_length, label_length = args
    # the 2 is critical here since the first couple outputs of the RNN
    # tend to be garbage
    y_pred = y_pred[:, 2:, :]
    return K.ctc_batch_cost(labels, y_pred, input_length, label_length)
```

```
labels = Input(name='gtruth_labels', shape=[max_str_len],
dtype='float32')
input_length = Input(name='input_length', shape=[1], dtype='int64')
label_length = Input(name='label_length', shape=[1], dtype='int64')

ctc_loss = Lambda(ctc_lambda_func, output_shape=(1,),
name='ctc')([y_pred, labels, input_length, label_length])
model_final = Model(inputs=[input_data, labels, input_length,
label_length], outputs=ctc_loss)
```

```
# the loss calculation occurs elsewhere, so we use a dummy lambda
function for the loss
file_path_best = "C_LSTM_best.hdf5"

model_final.compile(loss={'ctc': lambda y_true, y_pred: y_pred},
optimizer=Adam(lr = 0.0001))
```

```
checkpoint = ModelCheckpoint(filepath=file_path_best,
                             monitor='val_loss',
                             verbose=1,
                             save_best_only=True,
                             mode='min')

callbacks_list = [checkpoint]
```

```
history = model_final.fit(x=[train_x, train_y, train_input_len,
train_label_len], y=train_output,validation_data=([valid_x, valid_y,
valid_input_len, valid_label_len],
valid_output),callbacks=callbacks_list,verbose=1,epochs=60,
batch_size=128,shuffle=True)
```

```
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()
```

```
model.load_weights('/kaggle/working/C_LSTM_best.hdf5')
```

```
preds = model.predict(valid_x)
decoded = K.get_value(K.ctc_decode(preds,
input_length=np.ones(preds.shape[0])*preds.shape[1],
                                   greedy=True)[0][0])

prediction = []
for i in range(valid_size):
    prediction.append(num_to_label(decoded[i]))
```

```
y_true = valid.loc[0:valid_size, 'IDENTITY']
correct_char = 0
total_char = 0
correct = 0
```

```
for i in range(valid_size):
    pr = prediction[i]
    tr = y_true[i]
    total_char += len(tr)

    for j in range(min(len(tr), len(pr))):
        if tr[j] == pr[j]:
            correct_char += 1

    if pr == tr :
        correct += 1

print('Correct characters predicted : %.2f%%'
%(correct_char*100/total_char))
print('Correct words predicted      : %.2f%%'
%(correct*100/valid_size))
```

```
test = pd.read_csv('/kaggle/input/handwriting-
recognition/written_name_validation_v2.csv')

plt.figure(figsize=(15, 10))
for i in range(16):
    ax = plt.subplot(4, 4, i+1)
    img_dir = '/kaggle/input/handwriting-
recognition/validation_v2/validation/'+test.loc[i, 'FILENAME']
    image = cv2.imread(img_dir, cv2.IMREAD_GRAYSCALE)
    plt.imshow(image, cmap='gray')

    image = preprocess(image)
    image = image/255.
    pred = model.predict(image.reshape(1, 256, 64, 1))
    decoded = K.get_value(K.ctc_decode(pred,
input_length=np.ones(pred.shape[0])*pred.shape[1],
                                       greedy=True)[0][0])
    plt.title(num_to_label(decoded[0]), fontsize=12)
    plt.axis('off')

plt.subplots_adjust(wspace=0.2, hspace=-0.8)
```

```
plt.figure(figsize=(1, 1))
for i in range(1):
```

```
    ax = plt.subplot(1, 1, i+1)
    img_dir = "/kaggle/input/test123/tr.PNG"
    image = cv2.imread(img_dir, cv2.IMREAD_GRAYSCALE)
    plt.imshow(image, cmap='gray')

    image = preprocess(image)
    image = image/255
    pred = model.predict(image.reshape(1, 256, 64, 1))
    decoded = K.get_value(K.ctc_decode(pred,
input_length=np.ones(pred.shape[0])*pred.shape[1],
                                       greedy=True)[0][0])
    plt.title(num_to_label(decoded[0]), fontsize=12)
    plt.axis('off')
```

Output:


```
plt.figure(figsize=(3, 1))
for i in range(1):
    ax = plt.subplot(1, 1, i+1)
    img_dir = "/kaggle/input/test234567575/test2.PNG"
    image = cv2.imread(img_dir, cv2.IMREAD_GRAYSCALE)
    plt.imshow(image, cmap='gray')

    image = preprocess(image)
    image = image/255
    pred = model.predict(image.reshape(1, 256, 64, 1))
    decoded = K.get_value(K.ctc_decode(pred,
input_length=np.ones(pred.shape[0])*pred.shape[1],
                                       greedy=True)[0][0])
    plt.title(num_to_label(decoded[0]), fontsize=12)
    plt.axis('off')
```

Output:

Conclusion:

Designing and implementing an LSTM (Long Short-Term Memory) model for handwriting recognition involves creating a recurrent neural network that can effectively capture sequential information from handwritten data. The architecture typically consists of input layers to process the image data, LSTM layers to analyze the sequential information in the handwriting, and output layers for character or word recognition. The LSTM cells are crucial for maintaining context and handling variable-length sequences. The model is trained on a dataset of handwritten samples, often preprocessed using techniques like image normalization and feature extraction. The results of this approach often yield impressive accuracy in recognizing handwritten characters or words, making it a valuable tool for applications such as digit recognition, signature verification, and text transcription. The performance may vary based on the dataset and the complexity of the handwriting styles, but LSTM models have proven to be effective in this domain.