from \_\_future\_\_ import absolute\_import, division,

print\_function, unicode\_literals

import numpy as np

import tensorflow as tf

from keras.models import Sequential

from keras.layers import Dense, Activation

from keras.layers import LSTM

from keras.optimizers import RMSprop

from keras.callbacks import LambdaCallback

from keras.callbacks import ModelCheckpoint

from keras.callbacks import ReduceLROnPlateau

import random

import sys

# Changing the working location to the location of the text file

cd C:\Users\Dev\Desktop\Kaggle\Poems

# Reading the text file into a string

with open('poems.txt', 'r') as file:

text = file.read()

# A preview of the text file

print(text)

# Storing all the unique characters present in the text

vocabulary = sorted(list(set(text)))

# Creating dictionaries to map each character to an index

char\_to\_indices = dict((c, i) for i, c in enumerate(vocabulary))

indices\_to\_char = dict((i, c) for i, c in enumerate(vocabulary))

print(vocabulary)

# Dividing the text into subsequences of length max\_length

# So that at each time step the next max\_length characters

# are fed into the network

max\_length = 100

steps = 5

sentences = []

next\_chars = []

for i in range(0, len(text) - max\_length, steps):

sentences.append(text[i: i + max\_length])

next\_chars.append(text[i + max\_length])

# Hot encoding each character into a boolean vector

# Initializing a matrix of boolean vectors with each column representing

# the hot encoded representation of the character

X = np.zeros((len(sentences), max\_length, len(vocabulary)), dtype = np.bool)

y = np.zeros((len(sentences), len(vocabulary)), dtype = np.bool)

# Placing the value 1 at the appropriate position for each vector

# to complete the hot-encoding process

for i, sentence in enumerate(sentences):

for t, char in enumerate(sentence):

X[i, t, char\_to\_indices[char]] = 1

y[i, char\_to\_indices[next\_chars[i]]] = 1

# Initializing the LSTM network

model = Sequential()

# Defining the cell type

model.add(GRU(128, input\_shape =(max\_length, len(vocabulary))))

# Defining the densely connected Neural Network layer

model.add(Dense(len(vocabulary)))

# Defining the activation function for the cell

model.add(Activation('softmax'))

# Defining the optimizing function

optimizer = RMSprop(lr = 0.01)

# Configuring the model for training

model.compile(loss ='categorical\_crossentropy', optimizer = optimizer)

# Helper function to sample an index from a probability array

def sample\_index(preds, temperature = 1.0):

# temperature determines the freedom the function has when generating text

# Converting the predictions vector into a numpy array

preds = np.asarray(preds).astype('float64')

# Normalizing the predictions array

preds = np.log(preds) / temperature

exp\_preds = np.exp(preds)

preds = exp\_preds / np.sum(exp\_preds)

# The main sampling step. Creates an array of probabilities signifying

# the probability of each character to be the next character in the

# generated text

probas = np.random.multinomial(1, preds, 1)

# Returning the character with maximum probability to be the next character

# in the generated text

return np.argmax(probas)

# Helper function to generate text after the end of each epoch

def on\_epoch\_end(epoch, logs):

print()

print('----- Generating text after Epoch: % d' % epoch)

# Choosing a random starting index for the text generation

start\_index = random.randint(0, len(text) - max\_length - 1)

# Sampling for different values of diversity

for diversity in [0.2, 0.5, 1.0, 1.2]:

print('----- diversity:', diversity)

generated = ''

# Seed sentence

sentence = text[start\_index: start\_index + max\_length]

generated += sentence

print('----- Generating with seed: "' + sentence + '"')

sys.stdout.write(generated)

for i in range(400):

# Initializing the predictions vector

x\_pred = np.zeros((1, max\_length, len(vocabulary)))

for t, char in enumerate(sentence):

x\_pred[0, t, char\_to\_indices[char]] = 1.

# Making the predictions for the next character

preds = model.predict(x\_pred, verbose = 0)[0]

# Getting the index of the most probable next character

next\_index = sample\_index(preds, diversity)

# Getting the most probable next character using the mapping built

next\_char = indices\_to\_char[next\_index]

# Building the generated text

generated += next\_char

sentence = sentence[1:] + next\_char

sys.stdout.write(next\_char)

sys.stdout.flush()

print()

# Defining a custom callback function to

# describe the internal states of the network

print\_callback = LambdaCallback(on\_epoch\_end = on\_epoch\_end)

# Defining a helper function to save the model after each epoch

# in which the loss decreases

filepath = "weights.hdf5"

checkpoint = ModelCheckpoint(filepath, monitor ='loss',

verbose = 1, save\_best\_only = True,

mode ='min')

# Defining a helper function to reduce the learning rate each time

# the learning plateaus

reduce\_alpha = ReduceLROnPlateau(monitor ='loss', factor = 0.2,

patience = 1, min\_lr = 0.001)

callbacks = [print\_callback, checkpoint, reduce\_alpha]

# Training the GRU model

model.fit(X, y, batch\_size = 128, epochs = 30, callbacks = callbacks)

def generate\_text(length, diversity):

# Get random starting text

start\_index = random.randint(0, len(text) - max\_length - 1)

# Defining the generated text

generated = ''

sentence = text[start\_index: start\_index + max\_length]

generated += sentence

# Generating new text of given length

for i in range(length):

# Initializing the prediction vector

x\_pred = np.zeros((1, max\_length, len(vocabulary)))

for t, char in enumerate(sentence):

x\_pred[0, t, char\_to\_indices[char]] = 1.

# Making the predictions

preds = model.predict(x\_pred, verbose = 0)[0]

# Getting the index of the next most probable index

next\_index = sample\_index(preds, diversity)

# Getting the most probable next character using the mapping built

next\_char = indices\_to\_char[next\_index]

# Generating new text

generated += next\_char

sentence = sentence[1:] + next\_char

return generated

print(generate\_text(500, 0.2))