**IBM** **Naan Mudhalvan**

**Artificial Intelligence**

**Phase - 4**

Create a chatbot in python

**Importing required libraries :**

import tensorflow as tf

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from tensorflow.keras.layers import TextVectorization

import re,string

from tensorflow.keras.layers import LSTM,Dense,Embedding,Dropout,LayerNormalization

from flask import Flask, render\_template, request, jsonify

**Reading Dataset :**

df=pd.read\_csv('/kaggle/input/simple-dialogs-for chatbot/dialogs.txt',sep='\t', names=['question','answer'])

print(f'Dataframe size: {len(df)}')

df.head()

**Data Preprocessing :**

* **Data Visualization :**

df['question tokens']=df['question'].apply(lambda x:len(x.split()))

df['answer tokens']=df['answer'].apply(lambda x:len(x.split()))

plt.style.use('fivethirtyeight')

fig,ax=plt.subplots(nrows=1,ncols=2,figsize=(20,5))

sns.set\_palette('Set2')

sns.histplot(x=df['question tokens'],data=df,kde=True,ax=ax[0])

sns.histplot(x=df['answer tokens'],data=df,kde=True,ax=ax[1])

sns.jointplot(x='question tokens',y='answer tokens',data=df,kind='kde',fill=True,cmap='YlGnBu')

plt.show()

* **Text Cleaning :**

def clean\_text(text):

text=re.sub('-',' ',text.lower())

text=re.sub('[.]',' . ',text)

text=re.sub('[1]',' 1 ',text)

text=re.sub('[2]',' 2 ',text)

text=re.sub('[3]',' 3 ',text)

text=re.sub('[4]',' 4 ',text)

text=re.sub('[5]',' 5 ',text)

text=re.sub('[6]',' 6 ',text)

text=re.sub('[7]',' 7 ',text)

text=re.sub('[8]',' 8 ',text)

text=re.sub('[9]',' 9 ',text)

text=re.sub('[0]',' 0 ',text)

text=re.sub('[,]',' , ',text)

text=re.sub('[?]',' ? ',text)

text=re.sub('[!]',' ! ',text)

text=re.sub('[$]',' $ ',text)

text=re.sub('[&]',' & ',text)

text=re.sub('[/]',' / ',text)

text=re.sub('[:]',' : ',text)

text=re.sub('[;]',' ; ',text)

text=re.sub('[\*]',' \* ',text)

text=re.sub('[\']',' \' ',text)

text=re.sub('[\"]',' \" ',text)

text=re.sub('\t',' ',text)

return text

df.drop(columns=['answer tokens','question tokens'],axis=1,inplace=True)

df['encoder\_inputs']=df['question'].apply(clean\_text)

df['decoder\_targets']=df['answer'].apply(clean\_text)+' <end>'

df['decoder\_inputs']='<start> '+df['answer'].apply(clean\_text)+' <end>'

df.head(10)

df['encoder input tokens']=df['encoder\_inputs'].apply(lambda x:len(x.split()))

df['decoder input tokens']=df['decoder\_inputs'].apply(lambda x:len(x.split()))

df['decoder target tokens']=df['decoder\_targets'].apply(lambda x:len(x.split()))

plt.style.use('fivethirtyeight')

fig,ax=plt.subplots(nrows=1,ncols=3,figsize=(20,5))

sns.set\_palette('Set2')

sns.histplot(x=df['encoder input tokens'],data=df,kde=True,ax=ax[0])

sns.histplot(x=df['decoder input tokens'],data=df,kde=True,ax=ax[1])

sns.histplot(x=df['decoder target tokens'],data=df,kde=True,ax=ax[2])

sns.jointplot(x='encoder input tokens',y='decoder target tokens',data=df,kind='kde',fill=True,cmap='YlGnBu')

plt.show()

print(f"After preprocessing: {' '.join(df[df['encoder input tokens'].max()==df['encoder input tokens']]['encoder\_inputs'].values.tolist())}")

print(f"Max encoder input length: {df['encoder input tokens'].max()}")

print(f"Max decoder input length: {df['decoder input tokens'].max()}")

print(f"Max decoder target length: {df['decoder target tokens'].max()}")

df.drop(columns=['question','answer','encoder input tokens','decoder input tokens','decoder target tokens'],axis=1,inplace=True)

params={

"vocab\_size":2500,

"max\_sequence\_length":30,

"learning\_rate":0.008,

"batch\_size":149,

"lstm\_cells":256,

"embedding\_dim":256,

"buffer\_size":10000

}

learning\_rate=params['learning\_rate']

batch\_size=params['batch\_size']

embedding\_dim=params['embedding\_dim']

lstm\_cells=params['lstm\_cells']

vocab\_size=params['vocab\_size']

buffer\_size=params['buffer\_size']

max\_sequence\_length=params['max\_sequence\_length']

df.head(10)

* **Tokenization :**

vectorize\_layer=TextVectorization(

max\_tokens=vocab\_size,

standardize=None,

output\_mode='int',

output\_sequence\_length=max\_sequence\_length

)

vectorize\_layer.adapt(df['encoder\_inputs']+' '+df['decoder\_targets']+' <start> <end>')

vocab\_size=len(vectorize\_layer.get\_vocabulary())

print(f'Vocab size: {len(vectorize\_layer.get\_vocabulary())}')

print(f'{vectorize\_layer.get\_vocabulary()[:12]}')

def sequences2ids(sequence):

return vectorize\_layer(sequence)

def ids2sequences(ids):

decode=''

if type(ids)==int:

ids=[ids]

for id in ids:

decode+=vectorize\_layer.get\_vocabulary()[id]+' '

return decode

x=sequences2ids(df['encoder\_inputs'])

yd=sequences2ids(df['decoder\_inputs'])

y=sequences2ids(df['decoder\_targets'])

print(f'Question sentence: hi , how are you ?')

print(f'Question to tokens: {sequences2ids("hi , how are you ?")[:10]}')

print(f'Encoder input shape: {x.shape}')

print(f'Decoder input shape: {yd.shape}')

print(f'Decoder target shape: {y.shape}')

print(f'Encoder input: {x[0][:12]} ...')

print(f'Decoder input: {yd[0][:12]} ...') # shifted by one time step of the target as input to decoder is the output of the previous timestep

print(f'Decoder target: {y[0][:12]} ...')

data=tf.data.Dataset.from\_tensor\_slices((x,yd,y))

data=data.shuffle(buffer\_size)

train\_data=data.take(int(.9\*len(data)))

train\_data=train\_data.cache()

train\_data=train\_data.shuffle(buffer\_size)

train\_data=train\_data.batch(batch\_size)

train\_data=train\_data.prefetch(tf.data.AUTOTUNE)

train\_data\_iterator=train\_data.as\_numpy\_iterator()

val\_data=data.skip(int(.9\*len(data))).take(int(.1\*len(data)))

val\_data=val\_data.batch(batch\_size)

val\_data=val\_data.prefetch(tf.data.AUTOTUNE)

\_=train\_data\_iterator.next()

print(f'Number of train batches: {len(train\_data)}')

print(f'Number of training data: {len(train\_data)\*batch\_size}')

print(f'Number of validation batches: {len(val\_data)}')

print(f'Number of validation data: {len(val\_data)\*batch\_size}')

print(f'Encoder Input shape (with batches): {\_[0].shape}')

print(f'Decoder Input shape (with batches): {\_[1].shape}')

print(f'Target Output shape (with batches): {\_[2].shape}')

**Building Models :**

* **Build Encoder :**

class Encoder(tf.keras.models.Model):

def \_\_init\_\_(self,units,embedding\_dim,vocab\_size,\*args,\*\*kwargs) -> None:

super().\_\_init\_\_(\*args,\*\*kwargs)

self.units=units

self.vocab\_size=vocab\_size

self.embedding\_dim=embedding\_dim

self.embedding=Embedding(

vocab\_size,

embedding\_dim,

name='encoder\_embedding',

mask\_zero=True,

embeddings\_initializer=tf.keras.initializers.GlorotNormal()

)

self.normalize=LayerNormalization()

self.lstm=LSTM(

units,

dropout=.4,

return\_state=True,

return\_sequences=True,

name='encoder\_lstm',

kernel\_initializer=tf.keras.initializers.GlorotNormal()

)

def call(self,encoder\_inputs):

self.inputs=encoder\_inputs

x=self.embedding(encoder\_inputs)

x=self.normalize(x)

x=Dropout(.4)(x)

encoder\_outputs,encoder\_state\_h,encoder\_state\_c=self.lstm(x)

self.outputs=[encoder\_state\_h,encoder\_state\_c]

return encoder\_state\_h,encoder\_state\_c

encoder=Encoder(lstm\_cells,embedding\_dim,vocab\_size,name='encoder')

encoder.call(\_[0])

* **Build Decoder :**

class Decoder(tf.keras.models.Model):

def \_\_init\_\_(self,units,embedding\_dim,vocab\_size,\*args,\*\*kwargs) -> None:

super().\_\_init\_\_(\*args,\*\*kwargs)

self.units=units

self.embedding\_dim=embedding\_dim

self.vocab\_size=vocab\_size

self.embedding=Embedding(

vocab\_size,

embedding\_dim,

name='decoder\_embedding',

mask\_zero=True,

embeddings\_initializer=tf.keras.initializers.HeNormal()

)

self.normalize=LayerNormalization()

self.lstm=LSTM(

units,

dropout=.4,

return\_state=True,

return\_sequences=True,

name='decoder\_lstm',

kernel\_initializer=tf.keras.initializers.HeNormal()

)

self.fc=Dense(

vocab\_size,

activation='softmax',

name='decoder\_dense',

kernel\_initializer=tf.keras.initializers.HeNormal()

)

def call(self,decoder\_inputs,encoder\_states):

x=self.embedding(decoder\_inputs)

x=self.normalize(x)

x=Dropout(.4)(x)

x,decoder\_state\_h,decoder\_state\_c=self.lstm(x,initial\_state=encoder\_states)

x=self.normalize(x)

x=Dropout(.4)(x)

return self.fc(x)

decoder=Decoder(lstm\_cells,embedding\_dim,vocab\_size,name='decoder')

decoder(\_[1][:1],encoder(\_[0][:1]))

* **Building Train Model :**

class ChatBotTrainer(tf.keras.models.Model):

def \_\_init\_\_(self,encoder,decoder,\*args,\*\*kwargs):

super().\_\_init\_\_(\*args,\*\*kwargs)

self.encoder=encoder

self.decoder=decoder

def loss\_fn(self,y\_true,y\_pred):

loss=self.loss(y\_true,y\_pred)

mask=tf.math.logical\_not(tf.math.equal(y\_true,0))

mask=tf.cast(mask,dtype=loss.dtype)

loss\*=mask

return tf.reduce\_mean(loss)

def accuracy\_fn(self,y\_true,y\_pred):

pred\_values = tf.cast(tf.argmax(y\_pred, axis=-1), dtype='int64')

correct = tf.cast(tf.equal(y\_true, pred\_values), dtype='float64')

mask = tf.cast(tf.greater(y\_true, 0), dtype='float64')

n\_correct = tf.keras.backend.sum(mask \* correct)

n\_total = tf.keras.backend.sum(mask)

return n\_correct / n\_total

def call(self,inputs):

encoder\_inputs,decoder\_inputs=inputs

encoder\_states=self.encoder(encoder\_inputs)

return self.decoder(decoder\_inputs,encoder\_states)

def train\_step(self,batch):

encoder\_inputs,decoder\_inputs,y=batch

with tf.GradientTape() as tape:

encoder\_states=self.encoder(encoder\_inputs,training=True)

y\_pred=self.decoder(decoder\_inputs,encoder\_states,training=True)

loss=self.loss\_fn(y,y\_pred)

acc=self.accuracy\_fn(y,y\_pred)

variables=self.encoder.trainable\_variables+self.decoder.trainable\_variables

grads=tape.gradient(loss,variables)

self.optimizer.apply\_gradients(zip(grads,variables))

metrics={'loss':loss,'accuracy':acc}

return metrics

def test\_step(self,batch):

encoder\_inputs,decoder\_inputs,y=batch

encoder\_states=self.encoder(encoder\_inputs,training=True)

y\_pred=self.decoder(decoder\_inputs,encoder\_states,training=True)

loss=self.loss\_fn(y,y\_pred)

acc=self.accuracy\_fn(y,y\_pred)

metrics={'loss':loss,'accuracy':acc}

return metrics

model=ChatBotTrainer(encoder,decoder,name='chatbot\_trainer')

model.compile(

loss=tf.keras.losses.SparseCategoricalCrossentropy(),

optimizer=tf.keras.optimizers.Adam(learning\_rate=learning\_rate),

weighted\_metrics=['loss','accuracy']

)

model(\_[:2])

* **Train Model :**

history=model.fit(

train\_data,

epochs=100,

validation\_data=val\_data,

callbacks=[

tf.keras.callbacks.TensorBoard(log\_dir='logs'),

tf.keras.callbacks.ModelCheckpoint('ckpt',verbose=1,save\_best\_only=True)

]

)

# Visualize Metrics :

# fig,ax=plt.subplots(nrows=1,ncols=2,figsize=(20,5))

# ax[0].plot(history.history['loss'],label='loss',c='red')

# ax[0].plot(history.history['val\_loss'],label='val\_loss',c = 'blue')

# ax[0].set\_xlabel('Epochs')

# ax[1].set\_xlabel('Epochs')

# ax[0].set\_ylabel('Loss')

# ax[1].set\_ylabel('Accuracy')

# ax[0].set\_title('Loss Metrics')

# ax[1].set\_title('Accuracy Metrics')

# ax[1].plot(history.history['accuracy'],label='accuracy')

# ax[1].plot(history.history['val\_accuracy'],label='val\_accuracy')

# ax[0].legend()

# ax[1].legend()

# plt.show()

# Save Model :

# model.load\_weights('ckpt')

# model.save('models',save\_format='tf')

# for idx,i in enumerate(model.layers):

# print('Encoder layers:' if idx==0 else 'Decoder layers: ')

# for j in i.layers:

# print(j)

# print('---------------------')

# Create Inference Model :

# class ChatBot(tf.keras.models.Model):

# def \_\_init\_\_(self,base\_encoder,base\_decoder,\*args,\*\*kwargs):

# super().\_\_init\_\_(\*args,\*\*kwargs)

# self.encoder,self.decoder=self.build\_inference\_model(base\_encoder,base\_decoder)

# def build\_inference\_model(self,base\_encoder,base\_decoder):

# encoder\_inputs=tf.keras.Input(shape=(None,))

# x=base\_encoder.layers[0](encoder\_inputs)

# x=base\_encoder.layers[1](x)

# x,encoder\_state\_h,encoder\_state\_c=base\_encoder.layers[2](x)

# encoder=tf.keras.models.Model(inputs=encoder\_inputs,outputs=[encoder\_state\_h,encoder\_state\_c],name='chatbot\_encoder')

# decoder\_input\_state\_h=tf.keras.Input(shape=(lstm\_cells,))

# decoder\_input\_state\_c=tf.keras.Input(shape=(lstm\_cells,))

# decoder\_inputs=tf.keras.Input(shape=(None,))

# x=base\_decoder.layers[0](decoder\_inputs)

# x=base\_encoder.layers[1](x)

# x,decoder\_state\_h,decoder\_state\_c=base\_decoder.layers[2](x,initial\_state=[decoder\_input\_state\_h,decoder\_input\_state\_c])

# decoder\_outputs=base\_decoder.layers[-1](x)

# decoder=tf.keras.models.Model(

# inputs=[decoder\_inputs,[decoder\_input\_state\_h,decoder\_input\_state\_c]],

# outputs=[decoder\_outputs,[decoder\_state\_h,decoder\_state\_c]],name='chatbot\_decoder'

# )

# return encoder,decoder

# def summary(self):

# self.encoder.summary()

# self.decoder.summary()

# def softmax(self,z):

# return np.exp(z)/sum(np.exp(z))

# def sample(self,conditional\_probability,temperature=0.5):

# conditional\_probability = np.asarray(conditional\_probability).astype("float64")

# conditional\_probability = np.log(conditional\_probability) / temperature

# reweighted\_conditional\_probability = self.softmax(conditional\_probability)

# probas = np.random.multinomial(1, reweighted\_conditional\_probability, 1)

# return np.argmax(probas)

# def preprocess(self,text):

# text=clean\_text(text)

# seq=np.zeros((1,max\_sequence\_length),dtype=np.int32)

# for i,word in enumerate(text.split()):

# seq[:,i]=sequences2ids(word).numpy()[0]

# return seq

# 

# def postprocess(self,text):

# text=re.sub(' - ','-',text.lower())

# text=re.sub(' [.] ','. ',text)

# text=re.sub(' [1] ','1',text)

# text=re.sub(' [2] ','2',text)

# text=re.sub(' [3] ','3',text)

# text=re.sub(' [4] ','4',text)

# text=re.sub(' [5] ','5',text)

# text=re.sub(' [6] ','6',text)

# text=re.sub(' [7] ','7',text)

# text=re.sub(' [8] ','8',text)

# text=re.sub(' [9] ','9',text)

# text=re.sub(' [0] ','0',text)

# text=re.sub(' [,] ',', ',text)

# text=re.sub(' [?] ','? ',text)

# text=re.sub(' [!] ','! ',text)

# text=re.sub(' [$] ','$ ',text)

# text=re.sub(' [&] ','& ',text)

# text=re.sub(' [/] ','/ ',text)

# text=re.sub(' [:] ',': ',text)

# text=re.sub(' [;] ','; ',text)

# text=re.sub(' [\*] ','\* ',text)

# text=re.sub(' [\'] ','\'',text)

# text=re.sub(' [\"] ','\"',text)

# return text

# def call(self,text,config=None):

# input\_seq=self.preprocess(text)

# states=self.encoder(input\_seq,training=False)

# target\_seq=np.zeros((1,1))

# target\_seq[:,:]=sequences2ids(['<start>']).numpy()[0][0]

# stop\_condition=False

# decoded=[]

# while not stop\_condition:

# decoder\_outputs,new\_states=self.decoder([target\_seq,states],training=False)

# # index=tf.argmax(decoder\_outputs[:,-1,:],axis=-1).numpy().item()

# index=self.sample(decoder\_outputs[0,0,:]).item()

# word=ids2sequences([index])

# if word=='<end> ' or len(decoded)>=max\_sequence\_length:

# stop\_condition=True

# else:

# decoded.append(index)

# target\_seq=np.zeros((1,1))

# target\_seq[:,:]=index

# states=new\_states

# return self.postprocess(ids2sequences(decoded))

# chatbot=ChatBot(model.encoder,model.decoder,name='chatbot')

# chatbot.summary()

# tf.keras.utils.plot\_model(chatbot.encoder,to\_file='encoder.png',show\_shapes=True,show\_layer\_activations=True)

# tf.keras.utils.plot\_model(chatbot.decoder,to\_file='decoder.png',show\_shapes=True,show\_layer\_activations=True)

# app = Flask(\_\_name\_\_)

# def home():

# return render\_template('index.html')

# @app.route('/chat', methods=['POST'])

# def chat():

# user\_message = request.form['user\_message']

# bot\_response = chatbot(user\_message)

# return jsonify({'bot\_response': bot\_response})

# Template:

# <!DOCTYPE html>

# <html>

# <head>

# <title>Chatbot</title>

# </head>

# <body>

# <h1>Chatbot</h1>

# <div id="chat-history"></div>

# <form id="chat-form">

# <input type="text" id="user-input" placeholder="Type your message">

# <button type="submit">Send</button>

# </form>

# <script>

# const chatForm = document.getElementById('chat-form');

# const userInput = document.getElementById('user-input');

# const chatHistory = document.getElementById('chat-history');

# chatForm.addEventListener('submit', async (e) => {

# e.preventDefault();

# const userMessage = userInput.value;

# chatHistory.innerHTML += `<p>You: ${userMessage}</p>`;

# userInput.value = '';

# const response = await fetch('/chat', {

# method: 'POST',

# body: new URLSearchParams({ user\_message: userMessage }),

# headers: { 'Content-Type': 'application/x-www-form-urlencoded' },

# });

# const data = await response.json();

# const botResponse = data.bot\_response;

# chatHistory.innerHTML += `<p>Bot: ${botResponse}</p>`;

# });

# </script>

# </body>

# </html>

# if \_\_name\_\_ == '\_\_main\_\_':

# app.run(debug=True)

# Time to Chat :

# def print\_conversation(texts):

# for text in texts:

# print(f'You: {text}')

# print(f'Bot: {chatbot(text)}')

# print('========================')

# print\_conversation([

# 'hi',

# 'do yo know me?',

# 'what is your name?',

# 'you are bot?',

# 'hi, how are you doing?',

# "i'm pretty good. thanks for asking.",

# "Don't ever be in a hurry",

# '''I'm gonna put some dirt in your eye ''',

# '''You're trash ''',

# '''I've read all your research on nano-technology ''',

# '''You want forgiveness? Get religion''',

# '''While you're using the bathroom, i'll order some food.''',

# '''Wow! that's terrible.''',

# '''We'll be here forever.''',

# '''I need something that's reliable.''',

# '''A speeding car ran a red light, killing the girl.''',

# '''Tomorrow we'll have rice and fish for lunch.''',

# '''I like this restaurant because they give you free bread.'''

# ])

# The provided code is a combination of a chatbot system and a web interface created using Flask, a Python web framework.

# Import Required Libraries: The code starts by importing the necessary libraries and modules. This includes TensorFlow for building the chatbot model, Flask for creating a web application, and other libraries for data processing and visualization.

# Initialize Flask App: The Flask app is initialized using Flask(\_\_name\_\_). This creates an instance of a Flask web application.

# Define Routes:

# /: This is the home route, and it typically corresponds to the main page of the web app where users interact with the chatbot. The route is associated with a function that renders an HTML template for the user interface.

# /chat: This is an API route responsible for handling chatbot interactions. When a user submits a message, the server processes it and returns a response. This route is associated with a function that extracts the user's message from the request and returns the chatbot's response.

# Create HTML Templates: In this step, HTML templates for the web interface are defined. These templates are used for rendering the user interface of the chatbot. The templates usually include an input field for users to type their messages and a section for displaying the chat history

# Run the Flask App: The code includes a block that checks if the script is being run as the main program. If it is, the Flask app is started using app.run(). The debug=True argument is included for development purposes, allowing the app to automatically restart when code changes are detected.

# Integration: The chatbot logic should be integrated into the code where indicated. This typically includes loading the chatbot model and implementing the logic for generating responses based on user input.

# This setup allows users to interact with the chatbot through a web interface. When a user enters a message, the server processes it, and the chatbot generates a response that is displayed on the web page. Users can continue the conversation by typing more messages in the input field.