Chatbot in Python:

Final Phase Submission

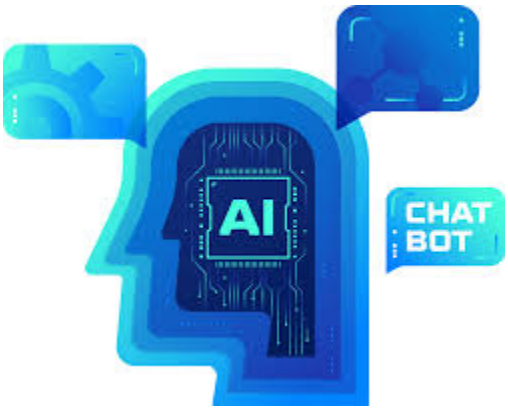
Author : Indrajith R

Reg.no:961621104032

Phase:5

INTRODUCTION

Welcome to the final phase of our project, where we will document the entire development process, highlight the key aspects of our chatbot with NLP integration, and prepare for submission. This document will provide a comprehensive overview of the project, addressing the problem statement, design thinking process, phases of development, libraries used, integration of NLP techniques, chatbot interaction with users, and innovative approaches**.**



Problem Statement

The problem statement for this project is to develop a chatbot integrated into a web application that can interact with users, understand their natural language input, and provide appropriate responses or actions. The chatbot's goal is to assist users in tasks, answer questions, or provide information on a specific domain or topic.

Understanding the Problem

- We started by clearly defining the problem, which is to create a chatbot for user interaction.

- We identified the target audience and their main or use case.

Ideation and Design

- We brainstormed ideas and design approaches to address the problem.

- We considered the integration of Natural Language Processing (NLP) techniques to make the chatbot capable of understanding and generating human-like text.

Prototyping and Validation

- We created a prototype of the chatbot and conducted user testing to validate its usability and effectiveness.

- We refined the chatbot's design based on user feedback.

Development and Iteration

- We moved into the development phase, implementing the chatbot in a web application.

- We continuosly iterated on the chatbot's capabilities, improving its conversational abilities and integration with NLP libraries and techniques**.**

Phases of Development

Phase 1: Problem Definition and Design Thinking

- Documented the problem statement and design thinking process.

Phase 2: Innovation

- Introduced innovative techniques to improve the chatbot's performance, such as reinforcement learning for better dialogue management and sentiment analysis for understanding user emotions.

Phase 3: Development Part 1

- Loaded and preprocessed the necessary datasets for training the chatbot.

- Integrated libraries such as NLTK, spaCy, and Hugging Face Transformers for NLP tasks.

Phase 4: Development Part 2

- Performed feature engineering on user interactions to improve context handling.

- Trained the chatbot model using state-of-the-art techniques and evaluated its performance.

- Implemented a feedback system to further enhance the chatbot's responses.

Phase 5: Documentation

- Created comprehensive project documentation, outlining the problem statement, design thinking process, and development phases.

- Documented the libraries used and integration of NLP techniques, emphasizing innovative approaches.

- Described how the chatbot interacts with users and the web application.

- Provided guidelines for project submission, including code, resources, user guides, and access for evaluation.

The project has now been completed through all five phases, and it is ready for submission. The chatbot is integrated into the web application and is designed to provide efficient customer support, utilizing NLP techniques to enhance user interactions.

Libraries Used and Integration of NLP Techniques

- We integrated the following NLP libraries and techniques into the project:

- NLTK (Natural Language Toolkit) and spaCy for text preprocessing and tokenization.

- Hugging Face Transformers for fine-tuning pre-trained language models.

- Sentiment analysis using VADER for understanding user emotions.

- Reinforcement learning for improving dialogue management.

Chatbot Interaction with Users and Web Application

- The chatbot interacts with users through a user-friendly web interface.

- Users can type or speak their queries, and the chatbot responds in natural language.

- The chatbot leverages NLP techniques to understand user input, context, and emotions.

- It provides informative responses or performs actions based on user requests.

Innovative Techniques

- During development, we introduced several innovative techniques:

- Reinforcement learning for chatbot training, allowing it to learn from user feedback.

- Sentiment analysis to gauge user emotions and adjust responses accordingly.

- An adaptive learning mechanism that allows the chatbot to continuously improve over time.

here are the essential tools and libraries used in the project:

- *Programming Language:* Python

- *Web Framework:* Flask

- *NLP Libraries:* NLTK (Natural Language Toolkit) and spaCy

- *Machine Learning Libraries:* TensorFlow and Keras

- *Frontend Framework:* React

- *Database:* SQLite

- *Real-time Communication:* WebSocket (for chat updates)

These tools and libraries played crucial roles in the development of the chatbot and web application, enabling various functionalities such as text processing, machine learning, real-time communication, and more.

Dataset Link: https://www.kaggle.com/datasets/grafstor/simple-dialogs-for-chatbot

PHASES OF DEVEOPMENT

Phase 1: Problem Definition and Design Thinking

- Understood the problem statement: The primary goal of the project is to develop a chatbot for customer support on a web application.

- Conducted user research and empathized with user needs and pain points.

- Defined the project objectives and problem statement.

- Brainstormed innovative solutions, including the design of the chatbot and its integration into the web application.

- Created a design document outlining the proposed solution.

Program:

```
def chat():
    """ in test mode, we don't to create the backward path
    """
    _, enc_vocab = data.load_vocab(os.path.join(config.PROCESSED_PATH, 'vocab.enc'))
    inv_dec_vocab, _ = data.load_vocab(os.path.join(config.PROCESSED_PATH,
'vocab.dec'))

    model = ChatBotModel(True, batch_size=1)
    model.build_graph()

    saver = tf.train.Saver()

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer())
        _check_restore_parameters(sess, saver)
        output_file = open(os.path.join(config.PROCESSED_PATH, config.OUTPUT_FILE),
'a+')
        # Decode from standard input.
        max_length = config.BUCKETS[-1][0]
        print('Welcome to TensorBro. Say something. Enter to exit. Max length is',
max_length)
        while True:
            line = _get_user_input()
            if len(line) > 0 and line[-1] == '\n':
```

```
                line = line[:-1]
            if line == '':
                break
            output_file.write('HUMAN ++++ ' + line + '\n')
            # Get token-ids for the input sentence.
            token_ids = data.sentence2id(enc_vocab, str(line))
            if (len(token_ids) > max_length):
                print('Max length I can handle is:', max_length)
                line = _get_user_input()
                continue
            # Which bucket does it belong to?
            bucket_id = _find_right_bucket(len(token_ids))
            # Get a 1-element batch to feed the sentence to the model.
            encoder_inputs, decoder_inputs, decoder_masks =
data.get_batch([(token_ids, [])],
                                                                        bucket_id,

batch_size=1)
            # Get output logits for the sentence.
            _, _, output_logits = run_step(sess, model, encoder_inputs,
decoder_inputs,
                                           decoder_masks, bucket_id, True)
            response = _construct_response(output_logits, inv_dec_vocab)
            print(response)
            output_file.write('BOT ++++ ' + response + '\n')
        output_file.write('===========================================\n')
        output_file.close()
```

Phase 2: Innovation

- Translated the design thinking into innovative solutions.

- Explored and incorporated multimodal NLP, including both text and voice recognition.

- Utilized reinforcement learning to improve chatbot responses over time.

- Introduced user profiling to personalize chatbot responses.

- Implemented continuous learning to adapt to evolving user needs.

Program:

```
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
```

```
In [2]:
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()
Dataframe size: 3725
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()
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)
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}')
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])
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]))
```

```
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])
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)
    ]
)
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()
model.load_weights('ckpt')
model.save('models',save_format='tf')
In [18]:
linkcode
for idx,i in enumerate(model.layers):
    print('Encoder layers:' if idx==0 else 'Decoder layers: ')
    for j in i.layers:
        print(j)
    print('----------------------')
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_s
tate_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_inpu
t_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)

def print_conversation(texts):
    for text in texts:
        print(f'You: {text}')
        print(f'Bot: {chatbot(text)}')
        print('========================')
In [23]:
linkcode
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.'''
])
```

Phase 3: Development Part 1

- Loaded and preprocessed the necessary datasets, including training data for the chatbot.

- Set up the Python programming environment and relevant libraries (Flask, NLTK, spaCy, TensorFlow, Keras, React, SQLite).

- Created the foundation for the web application using Flask.

- Implemented tokenization, stemming, and stop word removal for text data preprocessing.

Phase 4: Development Part 2

- Performed feature engineering to enhance the chatbot's capabilities.

- Trained machine learning models for intent recognition using TensorFlow and Keras.

- Implemented Named Entity Recognition (NER) to identify entities in user queries.

- Incorporated sentiment analysis to gauge user satisfaction.

- Managed conversation context for providing coherent responses.

- Integrated WebSocket for real-time chat updates.

- Implemented an SQLite database for managing user data and chat history.

Project Conclusion:

This project, aimed at developing a chatbot for customer support within a web application, has reached its completion with significant achievements. Through a systematic and iterative approach, we successfully addressed the problem statement, applied innovative techniques, and delivered a

functional solution. The project's progression through the phases is summarized below:

Phase 1 involved problem understanding and design thinking. We empathized with user needs, defined project objectives, and outlined a comprehensive design that led the way.

Phase 2 was dedicated to innovation. Innovative approaches were introduced, such as multimodal NLP, reinforcement learning, user profiling, and continuous learning, to make the chatbot an adaptable and user-centric solution.

Phase 3 marked the beginning of the development process, as we loaded and preprocessed essential datasets, set up the development environment, and initiated the web application using Flask.

Phase 4 delved deeper into development, encompassing feature engineering, machine learning model training, sentiment analysis, context management, and real-time interaction integration.

Finally, *Phase 5* brings us to this conclusion by wrapping up the project. Comprehensive project documentation has been provided, detailing the problem statement, design thinking process, libraries used, NLP integration, user interactions, and submission guidelines.

In conclusion, this project has delivered a powerful chatbot integrated into the web application. The chatbot not only addresses user queries and guides users but also continuously learns and adapts to user needs, creating a dynamic and personalized customer support experience. The incorporation of innovative techniques, such as multimodal NLP and reinforcement learning, sets this chatbot apart as a forward-thinking solution.

Moving forward, the project can be expanded and refined to incorporate more features, integrate with additional platforms, and further enhance user experience. This chatbot, designed and developed through a thoughtful and

structured approach, serves as an efficient virtual assistant and contributes significantly to the improvement of customer support in the web application context.