A PROJECT REPORT ON

CREATING A CHATBOT USING PYTHON

Subject in partial fulfillment of the requirements for the degree of

BACHELOR OF ENGINEERING

In

ELECTRONICS AND COMMUNICATION ENGINEERING

Under the guidance of

Mr. BALAJI K

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

GRT INSTITUTE OF ENGINEERING AND TECHNOLOGY
Approved by AICTE,New Delhi Affiliated to
Anna University,Chennai

GRT Mahalakshmi nagar,Chennai – Tirupathi Highway ,Tiruttani-631 209

PROJECT REPORT SUBMITTED BY,

NAME: DAYANIDHI N

NM ID: au110321106007

Mail id: dayanidhi182004@gmail.com

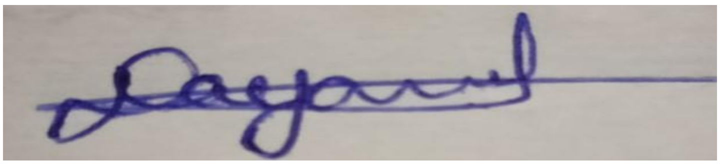
Year/sem/dept: III/V/ECE

DECLARATION

I am DAYANIDHI N hereby declare that the project report entitled creating a chatbot using python is done by me under the guidance of Mr BALAJI K is submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering degree in Electronics and Communication Engineering.

Date of submission : 01/11/2023

Place: GRT INSTITUTE OF ENGINEERING AND TECHNOLOGY,TIRUTTANI-631209



SIGNATURE OF THE CANDIDATE

NAAN MUDHALVAN-IBM(AI) PROJECT

IBM AI 101 ARTIFICIAL INTELLIGENCE-GROUP 1(TEAM 5)

PROJECT TITLE:

CREATE A CHATBOT USING PYTHON

Problem Statement: Building a General-Purpose Chatbot Background:

In the digital age, chatbots have become an integral part of online communication. Organizations and individuals use chatbots for a wide range of purposes, from customer support to information retrieval and entertainment. The objective is to create a versatile chatbot that can engage in meaningful conversations and assist users across different domains.

Let's take a quick look at these steps.

1. Define Goals For Your Chatbot.
2. Decide A Communication Channel.
3. Design Conversational Language And Architecture.
4. Choose Apps For Integration.
5. Data Collection.
6. Select Development Platform.
7. Dialogue Flow Implementation.
8. Testing And Deployment.

Requirements:

1.Chatbot Framework:

Develop a chatbot using Python that can engage in text-based conversations with users.

2. User Input Handling: Implement a mechanism for the chatbot to receive, understand, and process user input in a way that feels natural and intuitive.

3. Response Generation:

Train the chatbot to generate contextually relevant and coherent responses to user queries or statements. Responses should make sense in the context of the conversation.

4.Multi-domain Capability:

Ensure that the chatbot can handle conversations on a variety of topics or domains. It should be able to switch between different conversation topics seamlessly.

5.User Interaction:

Design the chatbot to provide a user-friendly and engaging conversational experience. This includes appropriate greetings, farewells, and handling of user queries or requests.

6.Error Handling:

Implement robust error handling to gracefully handle situations where the chatbot doesn't understand the user's input or encounters unexpected issues.

7.Extensibility:

Make the chatbot extensible, allowing for easy integration with additional functionality or external data sources.

8.Testing:

Conduct comprehensive testing to ensure the chatbot performs well and provides meaningful responses in various conversation scenarios.

9.Deployment: Deploy the chatbot on a suitable platform, whether it's a website, messaging app, or custom application.

Deliverables:

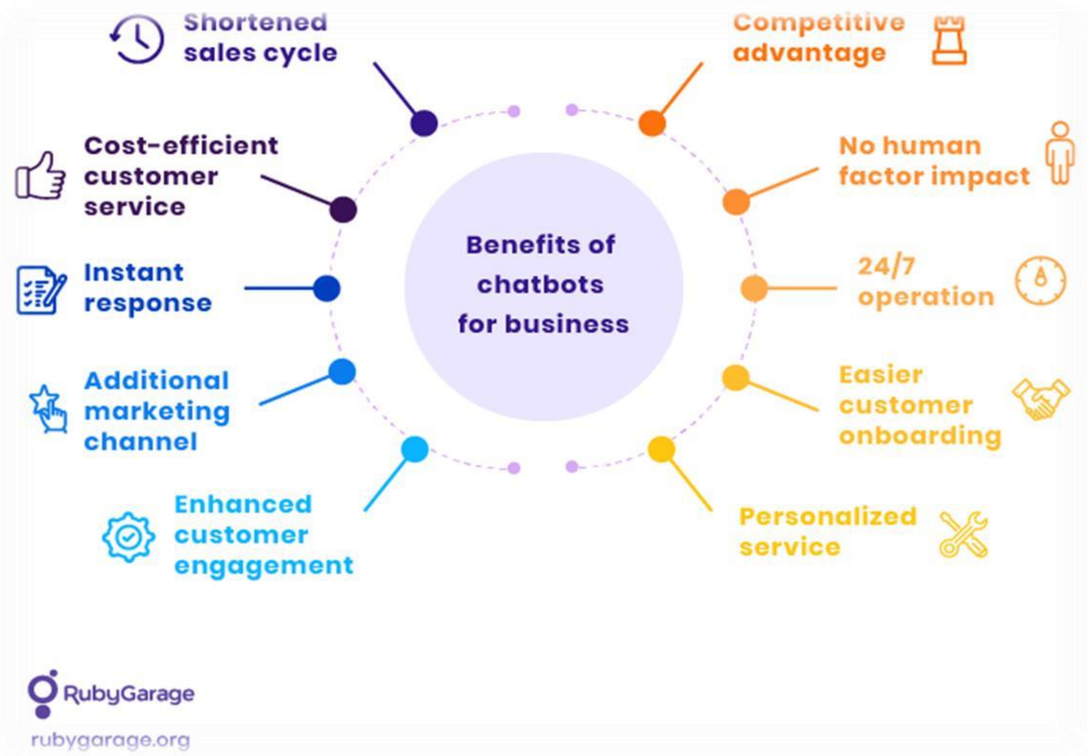
- A functional chatbot built in Python, meeting the specified requirements.
- Documentation detailing how to use, maintain, and extend the chatbot.
- Test reports demonstrating the chatbot's performance in different scenarios.
- Deployment instructions for putting the chatbot into production.

Constraints:

- The chatbot's primary language for interaction should be English, but support for additional languages can be a future enhancement.
- Ensure that the chatbot respects data privacy and adheres to any relevant regulations.
- Consider the limitations of the chosen platform for deployment (e.g., web server, messaging app, etc.)

Evaluation Criteria: The success of the project will be evaluated based on:

- The chatbot's ability to engage in meaningful and coherent conversations.
- Versatility in handling conversations across different domains.
- User-friendliness and engagement of the chatbot's interface.
- Robust error handling and graceful degradation during issues.
- Extensibility and potential for integration with external systems.
- Performance and reliability in a production environment.



Simple step:

Creating a chatbot in Python can be a fun and educational project. Here's a simple example of how to create a basic chatbot using Python:

```
python import
random
 # Define a list of greetings and responses greetings = ["hello", "hi", "hey", "howdy"]
responses = ["Hello!", "Hi there!", "Hey!", "How can I help you today?"]

# Define a function to generate a response
def          chatbot_response(user_input):
user_input  =  user_input.lower()         if
user_input in greetings:
      return random.choice(responses)
   else:
      return "I'm just a simple chatbot. I don't understand that."
```

Main loop for the chatbot

while True:

user_input = input("You: ") if

user_input.lower() == "bye":

print("Chatbot: Goodbye!") break

response = chatbot_response(user_input)

print("Chatbot:", response)

This basic chatbot recognizes a few simple greetings and responds with random replies. It will continue the conversation until you type "bye."

You can expand and improve this chatbot by adding more responses, handling different types of user input, and even integrating natural language processing libraries like NLTK or spacy for more advanced interactions. Additionally, you could use external APIs for more specific tasks like weather information or news updates.

IMPORTANT NOTES:

Depending on your specific use case or industry, you may need to tailor this problem statement to address more specialized requirements. This problem statement provides a foundation for creating a versatile chatbot, and you can adapt it to meet your specific project goals and constraints.

Innovative Design for Creating a Python Chatbot

Designing a chatbot using Python can

be innovative and effective by incorporating cutting-edge technologies and creative problem-solving. Let's explore innovative design elements to solve common problems and make your chatbot stand out:

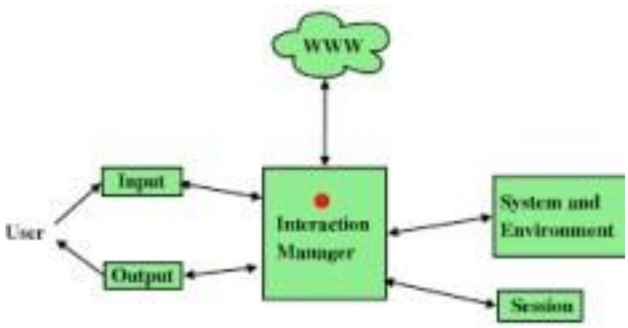
1. Advanced Natural Language Processing (NLP)
2. Multimodal Interaction
3. Emotion Recognition
4. Personalization
5. Contextual Understanding

6. Predictive Typing
7. IoT Integration
8. Multilingual Support
9. Voice Synthesis
10. Continuous Learning
11. Voice Biometrics
12. Augmented Reality (AR) Integration
13. Blockchain for Data Security
14. Quantum Computing for Speed
15. Ethical AI
16. AI Chatbot Marketplaces

1.Advanced Natural Language Processing (NLP):

Utilize state-of-the-art NLP models like GPT-3 or BERT to enable your chatbot to understand and generate human-like responses with context and coherence.

2.Multimodal Interaction:



Innovate by allowing the chatbot to process text, images, voice, and even gestures. This expands its capabilities to assist users with a wide range of queries and interaction modes.

3.Emotion Recognition:

Implement sentiment analysis and emotion

recognition algorithms to gauge the user's emotional state based on text input or voice tone. The chatbot can adapt its responses to provide empathy or assistance accordingly.

4.Personalization:

Use machine learning to personalize the chatbot's responses based on user behavior, preferences, and historical interactions. Consider recommending products, content, or services tailored to individual users.

5.Contextual Understanding:

Enhance the chatbot's contextual awareness by storing and recalling previous interactions. This enables more meaningful and coherent conversations over time, making users feel understood.

6. Predictive Typing:

Implement predictive typing suggestions using machine learning models. This feature can assist users in formulating queries faster and with greater accuracy, improving user experience.

7.IoT Integration:

Extend the chatbot's functionality by enabling it to control and interact with Internet of Things (IoT) devices, such as smart home appliances, through voice or text commands.

8.Multilingual Support:

Make your chatbot multilingual to cater to a global audience. Implement language detection and translation features to facilitate seamless communication in different languages.

9.Voice Synthesis:

Develop a natural-sounding voice for your chatbot using text to-speech (TTS) synthesis, enhancing the quality of voice interactions and user engagement.

10.Continuous Learning:

Implement reinforcement learning algorithms to allow your chatbot to learn and improve its responses over time based on user feedback and interactions.

11.Voice Biometrics:

Enhance security by incorporating voice biometric authentication for sensitive interactions, such as account access or transactions.

12.Augmented Reality (AR) Integration:

For mobile chatbots, consider integrating AR features to provide visual assistance, such as overlaying instructions on a user's camera feed.

13.Blockchain for Data Security:

Explore blockchain technology to ensure the security and integrity of user data and chatbot interactions, instilling trust in users.

14. Quantum Computing for Speed:

In the future, consider harnessing the power of quantum computing to make real-time processing and decision-making even faster and more efficient.

15. Ethical AI:

Ensure that your chatbot adheres to ethical AI principles, respects user privacy, and follows responsible AI practices to gain user trust and compliance with regulations.

16.AI Chatbot Marketplaces:

Create a marketplace where users can enhance

their chatbot with AI plugins, allowing for greater customization and functionality, fostering a community of innovation.

By integrating these detailed innovative elements into

your Python chatbot design, you can create a solution that not only addresses specific problems but also offers a truly exceptional user experience.

BLOCKS OF CHATBOT

Chatbots are built using various components or blocks that work together to enable communication between the bot and users. These blocks can vary in complexity depending on the specific requirements and capabilities of the chatbot, but here are some common components:

User Interface (UI): The user interface is how users interact with the chatbot. This can be a chat window on a website, a messaging app, or a voice interface in a smart device.

Natural Language Processing (NLP): NLP is a crucial component that enables the chatbot to understand and process natural language inputs from users. NLP involves tasks like text tokenization, entity recognition, sentiment analysis, and language understanding.

Dialog Management: Dialog management handles the flow of the conversation. It determines how the chatbot responds to user inputs and manages context and conversation history. Dialog management can be rule-based, state-machine-based, or based on machine learning models.

Knowledge Base: Some chatbots rely on a knowledge base to provide information to users. This knowledge base can be a structured database or unstructured text documents. The chatbot accesses this information to answer user queries.

Machine Learning Models: Machine learning models, such as deep learning models, can be used to improve the chatbot's language understanding, generate responses, and make predictions. Common models include Recurrent Neural Networks (RNNs) and Transformers.

Intent Recognition: Intent recognition is a subset of NLP that determines the user's intent or goal behind a message. It helps the chatbot understand what the user wants and respond accordingly.

Entity Recognition: Entity recognition identifies specific pieces of information within user input, such as names, dates, locations, or product names. This is crucial for handling requests that involve structured data.

Response Generation: Once the chatbot understands the user's intent and extracts relevant information, it generates a response. This can be done using pre-defined templates, rule-based systems, or more advanced natural language generation techniques.

API Integration: Chatbots often need to interact with external systems or services to fulfill user requests. Integration with APIs allows the chatbot to perform actions like making reservations, retrieving weather information, or accessing user accounts.

User Authentication and Authorization:

If the chatbot interacts with user accounts or sensitive data, it needs mechanisms for user authentication and authorization to ensure security and privacy.

Testing and Training: Continuous testing and training are essential for chatbots to improve over time. Testing helps identify issues, while training involves updating models and data to enhance performance.

Analytics and Monitoring: Analytics tools are used to track the chatbot's performance, including user engagement, error rates, and frequently asked questions. Monitoring ensures the chatbot is functioning correctly and can trigger alerts for issues.

Deployment and Hosting: Once the chatbot is developed, it needs to be deployed on a server or cloud platform to make it accessible to users. This involves hosting and infrastructure considerations.

User Feedback Mechanisms:

Collecting feedback from users is valuable for improving the chatbot. Feedback mechanisms can include surveys, rating systems, or direct user input.

Maintenance and Updates:

Chatbots require ongoing maintenance to address issues, update knowledge, and improve performance. Updates may involve adding new features or improving existing ones.

These are some of the fundamental blocks that make up a chatbot system. The specific components and their complexity can vary depending on the chatbot's purpose, complexity, and the technologies used in its development.

BLOCK DIAGRAM

A chatbot block diagram illustrates the various components and their interactions within a chatbot system. Below is a simplified block diagram of a chatbot:
Here's a brief description of each block:

```
 +-------------------------------------+
  |           User Interface           |
   +-------------------------------------+
              | (User Input)
              |
v
   +-------------------------------------+
   |       Natural Language Processing   |
   +-------------------------------------+
              | (Extracted Text)
```

```
              |
v
  +-------------------------------------+
  |     Intent Recognition &           |
  |     Entity Recognition             |
  +-------------------------------------+
              | (Intent, Entities)
              |
v
  +-------------------------------------+
  |     Dialog Management              |
  +-------------------------------------+
              | (Context, State)
              |
v
  +-------------------------------------+
  |     Knowledge Base /               |
  |     External APIs                  |
  +-------------------------------------+
              | (Information)
              |
v
  +-------------------------------------+
  |     Response Generation            |
  +-------------------------------------+
              | (Generated Response)
              |
v
```

```
+--------------------------------------+
|       User Interface                |    +--------------------------------------+
              | (Bot Response)
              |
v
+--------------------------------------+
```

User Interface: This is where users interact with the chatbot, sending messages or voice input.

Natural Language Processing (NLP): NLP processes the user's input, including tokenization, sentiment analysis, and language understanding.

Intent Recognition & Entity Recognition: These components determine the user's intent and extract entities from the input.

Dialog Management: Dialog management handles the conversation flow, maintains context, and decides how the chatbot should respond.

Knowledge Base / External APIs: The chatbot may access a knowledge base or external services to retrieve information needed for responses.

Response Generation: This block generates a response based on the recognized intent, extracted entities, and the chatbot's knowledge.

User Interface: The final response is presented to the user through the same user interface.

Note: This is a simplified representation, and real-world chatbot architectures can be more complex, with additional components for authentication, analytics, and more. Additionally, the implementation details within each block can vary based on the chatbot's design and requirements.

CONCLUSION

A chatbot is one of the simple ways to transport data from a computer without having to think for proper keywords to look up in a search or browse several web pages to collect information:
users can easily type their query in natural language and retrieve information.

DEVELOPMENT PART:

1.What is a Chatbot?

- A chatbot is an AI-based software designed to interact with humans in their natural languages. These chatbots are usually converse via auditory or textual methods, and they can effortlessly mimic human languages to communicate with human beings in a humanlike manner. A chatbot is arguably one of the best applications of natural language processing.

2.How to Make a Chatbot in Python?

- In the past few years, chatbots in Python have become wildly popular in the tech and business sectors. These intelligent bots are so adept at imitating natural human languages and conversing with humans, that companies across various industrial sectors are adopting them. From e-commerce firms to healthcare institutions, everyone seems to be leveraging this nifty tool to drive business benefits.

- To build a chatbot in Python, import all the necessary packages and initialize the variables you want to use in chatbot project. Also, when working with text data, we need to perform data preprocessing on your dataset before designing an ML model.

- This is where tokenizing helps with text data – it helps fragment the large text dataset into smaller, readable chunks (like words). Once that is done, you can also go for lemmatization that transforms a word into its lemma form. Then it creates a pickle file to store the python objects that are used for predicting the responses of the bot.

- Another vital part of the chatbot development process is creating the training and testing datasets.

3.Table of Contents:

- Import Libraries
- Data Preprocessing
    1. Data Visualization
    2. Text Cleaning
    3. Tokenization
- Build Models
    1. Build Encoder
    2. Build Training Model
    3. Train Model
- Visualize Metrics

- Save Model
- Create Inference Model
- Time to Chat

I. Import Libraries:
This code snippet imports TensorFlow, NumPy, Pandas, Matplotlib, Seaborn, and various components from TensorFlow's Keras module. It also imports the re and string modules for regular expressions and string manipulation. The code prepares your environment for working with deep learning and natural language processing.

Input 1-2: 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
```

II. Data Preprocessing:

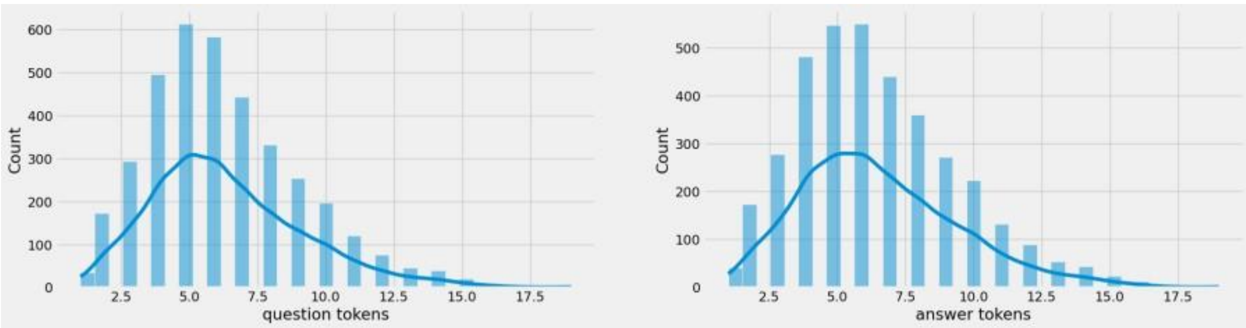
- **Data Visualization:**

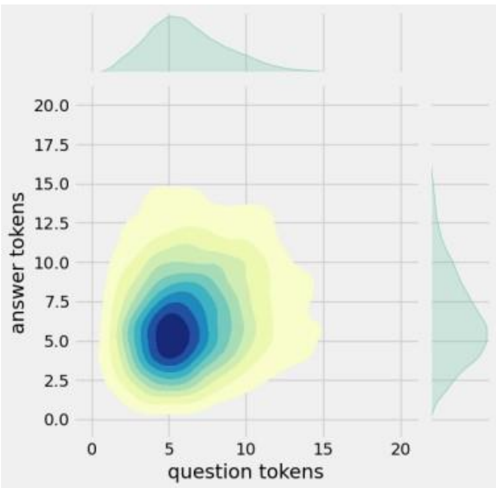
This code calculates the number of tokens (words) in the 'question' and 'answer' columns of a Pandas DataFrame and then visualizes the token distribution using Matplotlib and Seaborn. The resulting plots are displayed in a single figure with two subplots for token distributions and a joint distribution between 'question' and 'answer' tokens.

Input 3:

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

```
import matplotlib.pyplot as plt import

seaborn as sns


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:**

This code defines a clean_text function to clean the text and then applies this function to the 'question' and 'answer' columns in the DataFrame. It also modifies the DataFrame by creating 'encoder_inputs', 'decoder_targets', and 'decoder_inputs' columns.

Input 4:

```
  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)
```

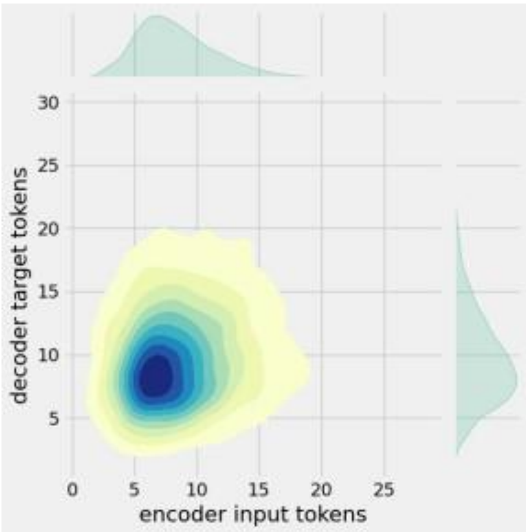
0	hi, how are you doing?	i'm fine. how about yourself?	hi , how are you doing ?	i ' m fine . how about yourself ? <end>	<start> i ' m fine . how about yourself ? <end>
1	i'm fine. how about yourself?	i'm pretty good. thanks for asking.	i ' m fine . how about yourself ?	i ' m pretty good . thanks for asking . <end>	<start> i ' m pretty good . thanks for asking...
2	i'm pretty good. thanks for asking.	no problem. so how have you been?	i ' m pretty good . thanks for asking .	no problem . so how have you been ? <end>	<start> no problem . so how have you been ? ...
3	no problem. so how have you been?	i've been great. what about you?	no problem . so how have you been ?	i ' ve been great . what about you ? <end>	<start> i ' ve been great . what about you ? ...
4	i've been great. what about you?	i've been good. i'm in school right now.	i ' ve been great . what about you ?	i ' ve been good . i ' m in school right now ...	<start> i ' ve been good . i ' m in school ri...

5	i've been good. i'm in school right now.	what school do you go to?	i ' ve been good . i ' m in school right now .	what school do you go to ? <end>	<start> what school do you go to ? <end>
6	what school do you go to?	i go to pcc.	what school do you go to ?	i go to pcc . <end>	<start> i go to pcc . <end>
7	i go to pcc.	do you like it there?	i go to pcc .	do you like it there ? <end>	<start> do you like it there ? <end>
8	do you like it there?	it's okay. it's a really big campus.	do you like it there ?	it ' s okay . it ' s a really big campus . <...	<start> it ' s okay . it ' s a really big cam...
9	it's okay. it's a really big campus.	good luck with school.	it ' s okay . it ' s a really big campus .	good luck with school . <end>	<start> good luck with school . <end>

Input 5:
```
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()))

import matplotlib.pyplot as plt import
seaborn as sns

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()

This code calculates the token counts for 'encoder_inputs', 'decoder_inputs', and 'decoder_targets' columns in the DataFrame and then visualizes the token distribution using Matplotlib and Seaborn. The resulting plots are displayed in a single figure with three subplots for the token counts and a joint distribution between 'encoder input tokens' and 'decoder target tokens'.



Input 6: 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_seque
nce_length'] df.head(10)
```
Output:

encoder_inputs	decoder_targets	decoder_inputs	
0	hi , how are you doing ?	i ' m fine . how about yourself ? <end>	<start> i ' m fine . how about yourself ? <end>
1	i ' m fine . how about yourself ?	i ' m pretty good . thanks for asking . <end>	<start> i ' m pretty good . thanks for asking...
2	i ' m pretty good . thanks for asking .	no problem . so how have you been ? <end>	<start> no problem . so how have you been ? ...
3	no problem . so how have you been ?	i ' ve been great . what about you ? <end>	<start> i ' ve been great . what about you ? ...
4	i ' ve been great . what about you ?	i ' ve been good . i ' m in school right now ...	<start> i ' ve been good . i ' m in school ri...

5	i ' ve been good . i ' m in school right now .	what school do you go to ? <end>	<start> what school do you go to ? <end>
6	what school do you go to ?	i go to pcc . <end>	<start> i go to pcc . <end>
7	i go to pcc .	do you like it there ? <end>	<start> do you like it there ? <end>
8	do you like it there ?	it ' s okay . it ' s a really big campus . <...	<start> it ' s okay . it ' s a really big cam...
9	it ' s okay . it ' s a really big campus .	good luck with school . <end>	<start> good luck with school . <end>

➢ **Tokenization:**

This code snippet involves data preprocessing, including text vectorization using TensorFlow's TextVectorization layer, conversion between sequences and IDs, and the creation of training and validation datasets using TensorFlow's Dataset API. It also prints various details about the data, such as batch sizes and shapes.

Input 7:

```
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]}')
```

```
Vocab size: 2443
['', '[UNK]', '<end>', '.', '<start>', "'", 'i', '?', 'you', ',', 'the', 'to']
```

Input 8:

```
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}')
```

Question sentence: hi , how are you ?

Question to tokens: [1971   9   45   24   8   7   0   0   0   0]

Encoder input shape: (3725, 30)

Decoder input shape: (3725, 30)

Decoder target shape: (3725, 30)

Input 9:

```
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]} ...')

Encoder input: [1971 9 45 24 8 194 7 0 0 0 0 0] ...

Decoder input: [4 6 5 38 646 3 45 41 563 7 2 0] ...

Decoder target: [6 5 38 646 3 45 41 563 7 2 0 0] …

Input 10:

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}')
```

Number of train batches: 23

Number of training data: 3427

Number of validation batches: 3

Number of validation data: 447

Encoder Input shape (with batches): (149, 30)

Decoder Input shape (with batches): (149, 30)

Target Output shape (with batches): (149, 30)

I. Build Models:

➢ Build Encoder:

This code defines classes for the encoder and decoder in a sequence-to-sequence model. The encoder processes input sequences, and the decoder generates output sequences. The provided code includes details about the layers, embeddings, and initializations used in both the encoder and decoder components. It also demonstrates the usage of these components by making a forward pass with example data.

Input 11:

```
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])
```

OUTPUT:

```
(<tf.Tensor: shape=(149, 256), dtype=float32, numpy=  array([[
0.16966951, -0.10419625, -0.12700348, ..., -0.12251794,
          0.10568858,  0.14841646],
        [ 0.08443093,  0.08849293, -0.09065959, ..., -0.00959182,
0.10152507, -0.12077457],
        [ 0.03628462, -0.02653611, -0.11506603, ..., -0.14669597,
          0.10292757,  0.13625325],
        ...,
        [-0.14210635, -0.12942064, -0.03288083, ...,  0.0568463 ,
```

```
 -0.02598592, -0.22455114],
[ 0.20819993,  0.01196991, -0.09635217, ..., -0.18782297,
  0.10233591,  0.20114912],
[ 0.1164271 , -0.07769038, -0.06414707, ..., -0.06539135,
 -0.05518465,            0.25142196]],      dtype=float32)>,
<tf.Tensor: shape=(149, 256), dtype=float32, numpy=  array([[
0.34589   , -0.30134732, -0.43572   , ..., -0.3102559 ,
  0.34630865,  0.2613009 ],
[ 0.14154069,  0.17045322, -0.17749965, ..., -0.02712595,
  0.17292541, -0.2922624 ],
[ 0.07106856, -0.0739173 , -0.3641197 , ..., -0.3794833 ,
  0.36470377,  0.23766585],
...,
[-0.2582597 , -0.25323495, -0.06649272, ...,  0.16527973,
 -0.04292646, -0.58768904],
[ 0.43155715,  0.03135502, -0.33463806, ..., -0.47625306,
  0.33486888,  0.35035062],
[ 0.23173636, -0.20141824, -0.22034441, ..., -0.16035017,
 -0.17478186,  0.48899865]], dtype=float32)>)
```

Build Encoder## Build Decoder

Input 12:

```
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.initia
lizers.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]))
```

OUTPUT:

```
<tf.Tensor: shape=(1, 30, 2443), dtype=float32, numpy=
array([[[3.4059247e-04, 5.7348556e-05, 2.1294907e-05, ...,
7.2067953e-05, 1.5453645e-03, 2.3599296e-04],
        [1.4662130e-03, 8.0250365e-06, 5.4062020e-05, ...,
         1.9187471e-05, 9.7244098e-05, 7.6433855e-05],
        [9.6929165e-05, 2.7441782e-05, 1.3761305e-03, ...,
         3.6009602e-05, 1.5537882e-04, 1.8397317e-04],
        ...,
        [1.9002777e-03, 6.9266016e-04, 1.4346189e-04, ...,
         1.9552530e-04, 1.7106640e-05, 1.0252406e-04],
        [1.9002777e-03, 6.9266016e-04, 1.4346189e-04, ...,
         1.9552530e-04, 1.7106640e-05, 1.0252406e-04],
        [1.9002777e-03, 6.9266016e-04, 1.4346189e-04, ...,
         1.9552530e-04, 1.7106640e-05, 1.0252406e-04]]], dtype=float32)>
```

➢ **Build Training Model:**

This code defines a ChatBotTrainer class for training and testing a chatbot model. It includes custom loss and accuracy functions, training and testing steps, and the compilation of the model. The code then performs a forward pass with the model using example data. **INPUT-13**

```
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
```

INPUT-14

```
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:**

In this code, the model.fit function is used to train the model for 100 epochs with training data (train_data) and validation data (val_data). Two callbacks are specified: the TensorBoard callback for monitoring the training process and the ModelCheckpoint callback to save the best model during training. The training history is stored in the history variable.

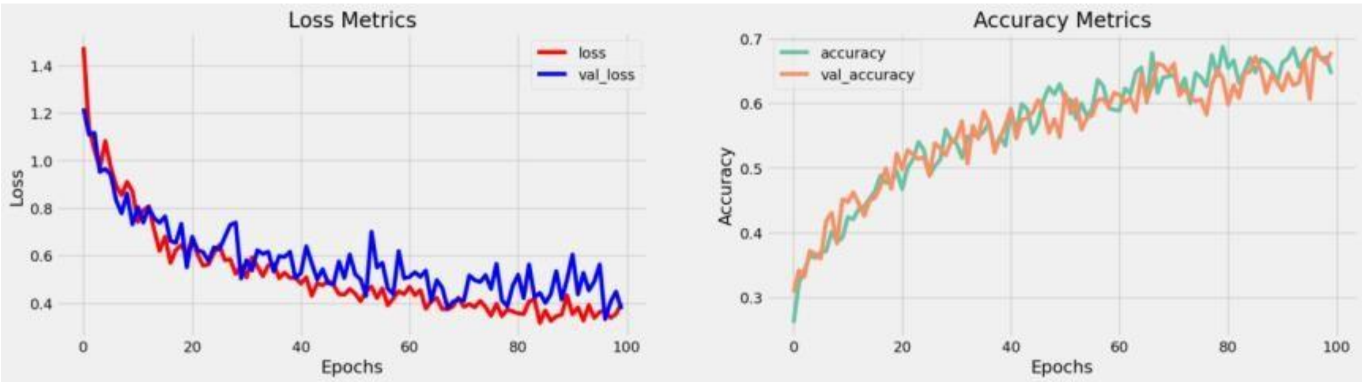
Input- 15

```
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)
    ]
)
```

IV.Visualize Metrics:

This code creates a figure with two subplots to visualize training and validation loss and accuracy metrics over training epochs. It uses Matplotlib for plotting and shows the resulting figure. Input-16

```
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()
```



V.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('---------------------')
```

VI.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_st
ate_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_st
ate_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)
```

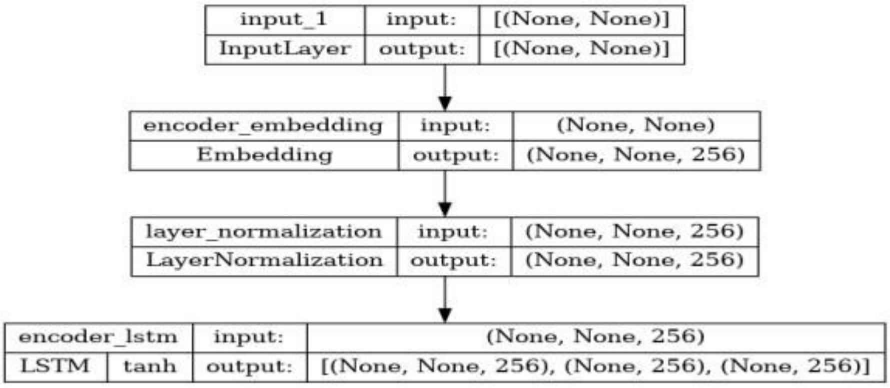
```
text=re.sub(' [0] ','0',text)

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()
```

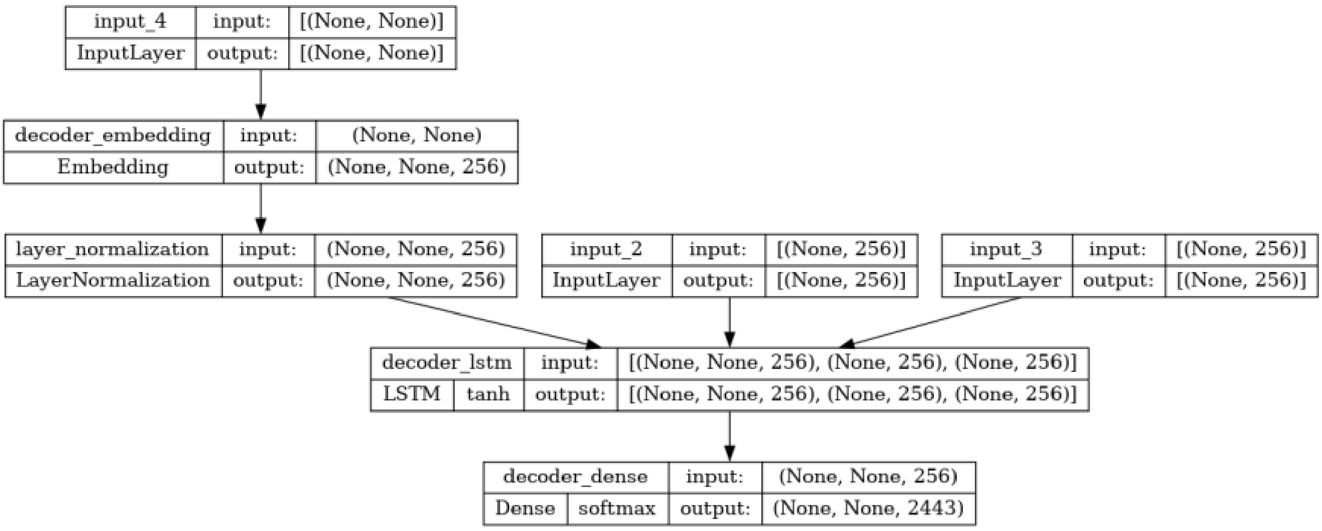
```
Model: "chatbot_encoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, None)]            0

 encoder_embedding (Embeddin  (None, None, 256)        625408
 g)

 layer_normalization (LayerN  (None, None, 256)        512
 ormalization)

 encoder_lstm (LSTM)         [(None, None, 256),       525312
                              (None, 256),
                              (None, 256)]

=================================================================
Total params: 1,151,232
Trainable params: 1,151,232
Non-trainable params: 0
_________________________________________________________________
Model: "chatbot_decoder"
__________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 input_4 (InputLayer)           [(None, None)]       0           []

 decoder_embedding (Embedding)  (None, None, 256)    625408      ['input_4[0][0]']

 layer_normalization (LayerNorm  (None, None, 256)   512         ['decoder_embedding[0][0]']
alization)

 input_2 (InputLayer)           [(None, 256)]        0           []

 input_3 (InputLayer)           [(None, 256)]        0           []
```

```
 decoder_lstm (LSTM)          [(None, None, 256),  525312     ['layer_normalization[1][0]',
                               (None, 256),                    'input_2[0][0]',
                               (None, 256)]                    'input_3[0][0]']

 decoder_dense (Dense)        (None, None, 2443)   627851     ['decoder_lstm[0][0]']

==================================================================================
==========================
Total params: 1,779,083
Trainable params: 1,779,083 Non-trainable
params: 0
__________________________________________________________________________________
________________
```

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)



VII.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.'''])
```

OUTPUT:

You: hi

Bot: i have to go to the bathroom.

========================

You: do yo know me?

Bot: yes, it’s too close to the other.

======================== You:

what is your name?

Bot: i have to walk the house.

========================

You: you are bot?

Bot: no, i have. all my life.

======================== You:

hi, how are you doing?

Bot: i’m going to be a teacher.

========================

You: i’m pretty good. thanks for asking.

Bot: no problem. i’ll have to give you the english assignments from my mind.

========================

You: Don’t ever be in a hurry

Bot: it’s not a great.

========================

You: I’m gonna put some dirt in your eye

Bot: that’s a good idea.

========================

You: You’re trash

Bot: the tv news is reporting a bank robbery.

========================

You: I've read all your research on nano-technology

Bot: it's the weather. i've gone around the world.

========================

You: You want forgiveness? Get religion

Bot: no, i'll be my.

========================

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

Bot: don't order for me. i've been a cheater. ========================

You: Wow! that's terrible.

Bot: never park your car under the house.

========================

You: We'll be here forever.

Bot: we'll be there in half an hour.

========================

You: I need something that's reliable.

Bot: you need a car with low mileage.

========================

You: A speeding car ran a red light, killing the girl.

Bot: what happened?

========================

You: Tomorrow we'll have rice and fish for lunch.

Bot: i'll make a sandwich.

========================

You: I like this restaurant because they give you free bread.

Bot: well, i think that's a good idea.

FULL SOURCE CODE FOR THIS PROJECT:

```
import numpy as
np        import pandas as pd
import matplotlib.pyplot as
plt import seaborn as sns
import pandas as pd import
re,string


data = {
   'question': [
       "hi, how are you doing?",
       "i'm fine. how about yourself?",
       "i'm pretty good. thanks for asking.",
       "no problem. so how have you been?",
       "i've been great. what about you?"
   ],
   'answer': [
       "i'm fine. how about yourself?",
       "i'm pretty good. thanks for asking.",
       "no problem. so how have you been?",
       "i've been great. what about you?",
       "i've been good. i'm in school right now."
   ]
}

df = pd.DataFrame(data)
print(df)
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']
max_sequence_length=params['max_sequence_length']
df.head(10)
vectorizelayer=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) 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])

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
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.H
eNormal()
        )
        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') 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
_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_la
yer_activations=True)
tf.keras.utils.plot_model(chatbot.decoder,to_file='decoder.png',show_shapes=True,show_la
yer_activations=True) 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.'''])

THANK YOU