PROJECT TITLE : CREATE CHATBOT USING PYTHON

INTRODUCTION: A chatbot is a computer program that simulates human conversation with an end user. Though not all chatbots are equipped with artificial intelligence (AI), modern chatbots increasingly use conversational AI techniques like natural language processing (NLP) to understand the user's questions and automate responses to them.

ENSEMBLE METHOD :

Ensemble methods can also be applied to chatbot development to enhance their performance and robustness.

Here are some ways to use ensemble methods for chatbots:

(●)Model Stacking:

(●)Intent Recognition Ensemble:

(●)Response Generation Ensemble:

(●)Fallback Mechanisms:

(●)Multi-Modal Chatbots:

(●)User Feedback Integration:

(●)Context Management:

Remember that implementing ensemble methods for chatbots can be complex and resource-intensive. It's essential to carefully design and evaluate your ensemble approach to ensure that it genuinely enhances the chatbot's performance and doesn't introduce unnecessary complexity.

DEEP LEARNING ARCHITECTURE

Deep learning architectures can significantly enhance the accuracy and robustness of chatbot prediction systems.

Here are some deep learning techniques and architectures commonly used for chatbot development:

(✿)Recurrent Neural Networks (RNNs):

RNNs are suitable for chatbots that need to process sequences of data, such as natural language. Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are popular RNN variants that address the vanishing gradient problem and can capture long-range dependencies in text.

(✿)Sequence-to-Sequence (Seq2Seq) Models:

Seq2Seq models, often based on RNNs or Transformers, are used for tasks like language translation and conversation generation. They consist of an encoder-decoder architecture and can be extended to handle multi-turn conversations.

(✿)Attention Mechanisms:

(✿)Attention Mechanisms:

Attention mechanisms, as seen in Transformer models, allow the model to focus on different parts of the input sequence when generating the output. Transformers have revolutionized natural language processing and are the foundation of state-of-the-art chatbots.

(✿)BERT (Bidirectional Encoder Representations from Transformers):

BERT is a pre-trained transformer-based model that excels in understanding the context of words in a sentence. Fine-tuning BERT for specific chatbot tasks can improve comprehension and generation capabilities.

(✿)GPT (Generative Pre-trained Transformer):

Models like GPT-3 and its variants have shown remarkable performance in natural language generation. GPT-based chatbots can generate human-like responses and adapt to different conversation contexts.

(✿)Transfer Learning:

Pre-training deep learning models on large language corpora and fine-tuning them for specific chatbot tasks can save training time and lead to better results.

(✿)Memory Networks:

Memory-augmented neural networks, like the Neural Turing Machine (NTM) or the Memory Network (MemNN), are suitable for chatbots that need to recall and store information across multiple turns in a conversation.

(✿)Hybrid Models:

Combining deep learning models with rule-based systems or traditional machine learning algorithms can leverage the strengths of both approaches. For example, using deep learning for natural language understanding and rule-based systems for generating structured responses.

(✿)Multi-Modal Architectures:

For chatbots that interact with users through text, voice, and images, multi-modal deep learning architectures can process and generate responses across different modalities.

(✿)Adversarial Training:

Adversarial training can be used to improve the robustness of chatbots by exposing them to adversarial inputs and training them to handle such scenarios more effectively.

(✿)Reinforcement Learning (RL):

RL can be used to fine-tune chatbot responses through user feedback. Chatbots can learn to optimize their responses over time based on user interactions.

When selecting a deep learning architecture for a chatbot, it's crucial to consider the specific requirements of the task, the available data, and the desired level of performance. Additionally, proper data preprocessing, hyperparameter tuning, and evaluation are essential for building accurate and robust chatbot systems.

DATASET :

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

```
# Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kadata:text/mce-internal,editor,Thisggle/docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
for filename in filenames:
print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved outside of the current session
#ignore warning
import warnings
warnings.filterwarnings('ignore')
/kaggle/input/simple-dialogs-for-chatbot/dialogs.txt
df = pd.read_csv('/kaggle/input/simple-dialogs-for-chatbot/dialogs.txt' , sep='\t' , names=['Question' , 'Answer'])

df
Question Answer
0 hi, how are you doing? i'm fine. how about yourself?
1 i'm fine. how about yourself? i'm pretty good. thanks for asking.
2 i'm pretty good. thanks for asking. no problem. so how have you been?
3 no problem. so how have you been? 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.
... ... ...
3720 that's a good question. maybe it's not old age. are you right-handed?
3721 are you right-handed? yes. all my life.
3722 yes. all my life. you're wearing out your right hand. stop using...
3723 you're wearing out your right hand. stop using... but i do all my writing with my right hand.
3724 but i do all my writing with my right hand. start typing instead. that way your left hand ...
3725 rows × 2 columns

# Check for null values
null_question = df['Question'].isnull().sum()
null_answer = df['Answer'].isnull().sum()

if null_question > 0:
print("There are", null_question, "null values in the 'Question' column.")
else:
print("There are no null values in the 'Question' column.")

if null_answer > 0:
print("There are", null_answer, "null values in the 'Answer' column.")
else:
print("There are no null values in the 'Answer' column.")
```

```
print("There are no null values in the 'Answer' column.")

# Check for whitespace values
whitespace_question = df['Question'].apply(lambda x: x.isspace()).sum()
whitespace_answer = df['Answer'].apply(lambda x: x.isspace()).sum()

if whitespace_question > 0:
print("There are", whitespace_question, "whitespace values in the 'Question' column.")
else:
print("There are no whitespace values in the 'Question' column.")

if whitespace_answer > 0:
print("There are", whitespace_answer, "whitespace values in the 'Answer' column.")
else:
print("There are no whitespace values in the 'Answer' column.")
```

```
There are no null values in the 'Question' column.
There are no null values in the 'Answer' column.
There are no whitespace values in the 'Question' column.
There are no whitespace values in the 'Answer' column.
```

```
import re

def clean_text(text):
text = text.lower()
text = re.sub(r'\d+', ' ', text) # Replace all digits with spaces
text = re.sub(r'([^\w\s])', r' \1 ', text) # Add a space before and after each punctuation character
text = re.sub(r'\s+', ' ', text) # Replace all consecutive spaces with a single space
text = text.strip() # Remove leading and trailing spaces
return text

df['Encoder Inputs']=df['Question'].apply(clean_text)
df['Decoder Inputs']="<sos> " + df['Answer'].apply(clean_text) + ' <eos>'
df["Decoder Targets"] = df['Answer'].apply(clean_text) + ' <eos>'

df.head()
```

	Question	Answer	Encoder Inputs	Decoder Inputs	Decoder Targets
0	hi, how are you doing?	i'm fine. how about yourself?	hi , how are you doing ?	<sos> i ' m fine . how about yourself ?	i ' m fine . how about yourself ? <eos>
1	i'm fine. how about yourself?	i'm pretty good. thanks for asking.	i ' m fine . how about yourself ?	<sos> i ' m pretty good . thanks for asking	i ' m pretty good . thanks for asking . <eos>
2	i'm pretty good. thanks for asking.	no problem. so how have you been?	i ' m pretty good . thanks for asking .	<sos> no problem . so how have you been ?	<eos> no problem . so how have you been ? <eos>
3	no problem. so how have you been?	i've been great. what about you?	no problem . so how have you been ?	<sos> i ' ve been great . what about you ?	<eos> i ' ve been great . what about you ? <eos>
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 ?	<sos> i ' ve been good . i ' m in school right...	i ' ve been good . i ' m in school right now ....

```
df['Question Length'] = df['Encoder Inputs'].apply(lambda x: len(x))
df['Answer Length'] = df['Decoder Inputs'].apply(lambda x: len(x))

df.head()
```

	Question	Answer	Encoder Inputs	Decoder Inputs	Decoder Targets	Question Length	Answer Length
0	hi, how are you doing?	i'm fine. how about yourself?	hi , how are you doing ?	<sos> i ' m fine . how about yourself ?	i ' m fine . how about yourself ? <eos>	24	45
1	i'm fine. how about yourself?	i'm pretty good. thanks for asking.	i ' m fine . how about yourself ?	<sos> i ' m pretty good . thanks for asking	i ' m pretty good . thanks for asking . <eos>	33	51
2	i'm pretty good. thanks for asking.	no problem. so how have you been?	i ' m pretty good . thanks for asking .	<sos> no problem . so how have you been ?	<eos> no problem . so how have you been ? <eos>	39	47
3	no problem. so how have you been?	i've been great. what about you?	no problem . so how have you been ?	<sos> i ' ve been great . what about you ?	<eos> i ' ve been great . what about you ? <eos>	35	48
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 ?	<sos> i ' ve been good . i ' m in school right...	i ' ve been good . i ' m in school right now	48	52

I've been great. What about you? I've been good. I'm in school right now. I've been great. What about you ?
<sos> i ' ve been good . i ' m in school right... i ' ve been good . i ' m in school right now 36 58
```
import plotly.express as px

fig1 = px.histogram(df, x='Question Length', nbins=50, opacity=0.7)
fig2 = px.histogram(df, x='Answer Length', nbins=50, opacity=0.7)

print("Maximum Question Length:", df['Question Length'].max())
print("Maximum Answer Length:", df['Answer Length'].max())

fig1.show()
fig2.show()
```
Maximum Question Length: 101
Maximum Answer Length: 113
```
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# Define the maximum number of words to keep based on word frequency
num_words = 10000

# Define the maximum sequence length
max_seq_length = 10

# Create a tokenizer and fit it on the 'Encoder Inputs' and 'Decoder Inputs' columns of the DataFrame
tokenizer = Tokenizer(num_words=num_words, oov_token='<unk>')
tokenizer.fit_on_texts(df['Encoder Inputs'].tolist() + df['Decoder Inputs'].tolist())

# Convert the text data to sequences of integers using the tokenizer
encoder_inputs = tokenizer.texts_to_sequences(df['Encoder Inputs'].tolist())
decoder_inputs = tokenizer.texts_to_sequences(df['Decoder Inputs'].tolist())
decoder_targets = tokenizer.texts_to_sequences(df['Decoder Targets'].tolist())

# Pad the sequences to ensure they all have the same length
encoder_inputs = pad_sequences(encoder_inputs, maxlen=max_seq_length, padding='post',
truncating='post')
decoder_inputs = pad_sequences(decoder_inputs, maxlen=max_seq_length, padding='post',
truncating='post')
decoder_targets = pad_sequences(decoder_targets, maxlen=max_seq_length, padding='post',
truncating='post')
decoder_targets[1:3]
```
array([[5, 4, 35, 161, 49, 245, 30, 481, 3, 0],
[34, 173, 26, 42, 19, 6, 102, 3, 0, 0]], dtype=int32)
```
df['Decoder Targets'][1:3]
```
1 i ' m pretty good . thanks for asking . <eos>
2 no problem . so how have you been ? <eos>
Name: Decoder Targets, dtype: object
```
# Get the vocabulary size of the tokenizer
vocab_size = len(tokenizer.word_index)
print('Vocabulary Size: %d' % vocab_size)
```
Vocabulary Size: 2410
```
print(encoder_inputs.shape , "\n" , decoder_inputs.shape , "\n" , decoder_targets.shape)
```
(3725, 10)
(3725, 10)
(3725, 10)
```
from sklearn.model_selection import train_test_split

# Split the data into train and test sets
encoder_inputs_train, encoder_inputs_test, decoder_inputs_train, decoder_inputs_test,
decoder_targets_train, decoder_targets_test = train_test_split(encoder_inputs, decoder_inputs,
```

```
decoder_targets, test_size=0.2, random_state=42)

# Print the shapes of the train and test sets
print("Train set shapes:", encoder_inputs_train.shape, decoder_inputs_train.shape,
decoder_targets_train.shape)
print("Test set shapes:", encoder_inputs_test.shape, decoder_inputs_test.shape,
decoder_targets_test.shape)
Train set shapes: (2980, 10) (2980, 10) (2980, 10)
Test set shapes: (745, 10) (745, 10) (745, 10)
from tensorflow.keras.layers import Input, LSTM, Dense, Embedding
from tensorflow.keras.models import Model

num_encoder_tokens = len(tokenizer.word_index) + 1
num_decoder_tokens = len(tokenizer.word_index) + 1
latent_dim = 32
embedding_dim = 50

# Define the input sequence
encoder_inputs = Input(shape=(max_seq_length,))

#_____Embedding_____

encoder_embedding = Embedding(num_encoder_tokens, embedding_dim , mask_zero=True)
encoder_inputs_embedded = encoder_embedding(encoder_inputs)


#_____Encoder_____

# Encoder - LSTM1
encoder_lstm1 =
LSTM(latent_dim,return_sequences=True,return_state=True,dropout=0.4,recurrent_dropout=0.4)
encoder_output1, state_h1, state_c1 = encoder_lstm1(encoder_inputs_embedded)

# Encoder - LSTM2
encoder_lstm2 =
LSTM(latent_dim,return_sequences=True,return_state=True,dropout=0.4,recurrent_dropout=0.4)
encoder_output2, state_h2, state_c2 = encoder_lstm2(encoder_output1)

# Encoder - LSTM2
encoder_lstm3=LSTM(latent_dim, return_state=True,
return_sequences=True,dropout=0.4,recurrent_dropout=0.4)
encoder_outputs, state_h, state_c= encoder_lstm3(encoder_output2)
#_____

# Discard the encoder outputs and only keep the states
encoder_states = [state_h, state_c]

# Define the decoder input sequence
decoder_inputs = Input(shape=(max_seq_length,))

# Add an embedding layer
decoder_embedding = Embedding(num_decoder_tokens, embedding_dim , mask_zero=True)
decoder_inputs_embedded = decoder_embedding(decoder_inputs)

#_____Decoder_____

decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
```

```
# Get the decoder outputs and states
decoder_outputs, _, _ = decoder_lstm(decoder_inputs_embedded, initial_state=encoder_states)

# Define the decoder output layer
decoder_dense = Dense(num_decoder_tokens, activation='softmax')

# Get the decoder outputs
decoder_outputs = decoder_dense(decoder_outputs)

# Define the Seq2Seq model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

model.summary()
```

```
Model: "model"
__________________________________________________________________________________________________
Layer (type) Output Shape Param # Connected to
==================================================================================================
input_1 (InputLayer) [(None, 10)] 0 []

embedding (Embedding) (None, 10, 50) 120550 ['input_1[0][0]']

lstm (LSTM) [(None, 10, 32), 10624 ['embedding[0][0]']
(None, 32),
(None, 32)]

input_2 (InputLayer) [(None, 10)] 0 []

lstm_1 (LSTM) [(None, 10, 32), 8320 ['lstm[0][0]']
(None, 32),
(None, 32)]

embedding_1 (Embedding) (None, 10, 50) 120550 ['input_2[0][0]']

lstm_2 (LSTM) [(None, 10, 32), 8320 ['lstm_1[0][0]']
(None, 32),
(None, 32)]

lstm_3 (LSTM) [(None, 10, 32), 10624 ['embedding_1[0][0]',
(None, 32), 'lstm_2[0][1]',
(None, 32)] 'lstm_2[0][2]']

dense (Dense) (None, 10, 2411) 79563 ['lstm_3[0][0]']

==================================================================================================
Total params: 358,551
Trainable params: 358,551
Non-trainable params: 0
__________________________________________________________________________________________________
```

```
from tensorflow.keras.utils import to_categorical
import tensorflow as tf

early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3)


batch_size = 32
epochs = 30
```

```
# One-hot encode the decoder targets
decoder_targets_train = to_categorical(decoder_targets_train, num_decoder_tokens)
decoder_targets_test = to_categorical(decoder_targets_test, num_decoder_tokens)

# Define the Seq2Seq model
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'] ,
sample_weight_mode='temporal')

# Train the model
model.fit([encoder_inputs_train, decoder_inputs_train], decoder_targets_train,
validation_data=([encoder_inputs_test, decoder_inputs_test], decoder_targets_test),
batch_size=batch_size, epochs=epochs , callbacks=[early_stopping])
Epoch 1/30
94/94 [==============================] - 47s 312ms/step - loss: 6.7108 - accuracy: 0.0869 -
val_loss: 5.3961 - val_accuracy: 0.0928
Epoch 2/30
94/94 [==============================] - 20s 215ms/step - loss: 5.3310 - accuracy: 0.0943 -
val_loss: 5.3034 - val_accuracy: 0.0930
Epoch 3/30
94/94 [==============================] - 20s 212ms/step - loss: 5.2223 - accuracy: 0.1047 -
val_loss: 5.1698 - val_accuracy: 0.1280
Epoch 4/30
94/94 [==============================] - 18s 187ms/step - loss: 5.0835 - accuracy: 0.1332 -
val_loss: 5.0813 - val_accuracy: 0.1463
Epoch 5/30
94/94 [==============================] - 18s 197ms/step - loss: 4.9987 - accuracy: 0.1468 -
val_loss: 5.0264 - val_accuracy: 0.1688
Epoch 6/30
94/94 [==============================] - 18s 193ms/step - loss: 4.9282 - accuracy: 0.1704 -
val_loss: 4.9702 - val_accuracy: 0.1929
Epoch 7/30
94/94 [==============================] - 19s 200ms/step - loss: 4.8579 - accuracy: 0.1966 -
val_loss: 4.9100 - val_accuracy: 0.2087
Epoch 8/30
94/94 [==============================] - 17s 182ms/step - loss: 4.7839 - accuracy: 0.2178 -
val_loss: 4.8454 - val_accuracy: 0.2234
Epoch 9/30
94/94 [==============================] - 18s 194ms/step - loss: 4.7069 - accuracy: 0.2323 -
val_loss: 4.7812 - val_accuracy: 0.2352
Epoch 10/30
94/94 [==============================] - 18s 193ms/step - loss: 4.6269 - accuracy: 0.2407 -
val_loss: 4.7146 - val_accuracy: 0.2376
Epoch 11/30
94/94 [==============================] - 17s 183ms/step - loss: 4.5527 - accuracy: 0.2438 -
val_loss: 4.6623 - val_accuracy: 0.2394
Epoch 12/30
94/94 [==============================] - 18s 189ms/step - loss: 4.4903 - accuracy: 0.2458 -
val_loss: 4.6196 - val_accuracy: 0.2398
Epoch 13/30
94/94 [==============================] - 18s 189ms/step - loss: 4.4336 - accuracy: 0.2544 -
val_loss: 4.5823 - val_accuracy: 0.2535
Epoch 14/30
94/94 [==============================] - 19s 200ms/step - loss: 4.3840 - accuracy: 0.2640 -
val_loss: 4.5514 - val_accuracy: 0.2584
```

```
Epoch 15/30
94/94 [==============================] - 17s 183ms/step - loss: 4.3383 - accuracy: 0.2670 -
val_loss: 4.5223 - val_accuracy: 0.2597
Epoch 16/30
94/94 [==============================] - 18s 193ms/step - loss: 4.2965 - accuracy: 0.2682 -
val_loss: 4.4970 - val_accuracy: 0.2631
Epoch 17/30
94/94 [==============================] - 18s 196ms/step - loss: 4.2580 - accuracy: 0.2707 -
val_loss: 4.4717 - val_accuracy: 0.2650
Epoch 18/30
94/94 [==============================] - 18s 187ms/step - loss: 4.2189 - accuracy: 0.2741 -
val_loss: 4.4494 - val_accuracy: 0.2665
Epoch 19/30
94/94 [==============================] - 18s 194ms/step - loss: 4.1817 - accuracy: 0.2757 -
val_loss: 4.4274 - val_accuracy: 0.2689
Epoch 20/30
94/94 [==============================] - 18s 187ms/step - loss: 4.1472 - accuracy: 0.2781 -
val_loss: 4.4103 - val_accuracy: 0.2681
Epoch 21/30
94/94 [==============================] - 18s 191ms/step - loss: 4.1131 - accuracy: 0.2809 -
val_loss: 4.3946 - val_accuracy: 0.2725
Epoch 22/30
94/94 [==============================] - 17s 184ms/step - loss: 4.0805 - accuracy: 0.2855 -
val_loss: 4.3810 - val_accuracy: 0.2747
Epoch 23/30
94/94 [==============================] - 18s 193ms/step - loss: 4.0487 - accuracy: 0.2907 -
val_loss: 4.3662 - val_accuracy: 0.2791
Epoch 24/30
94/94 [==============================] - 17s 183ms/step - loss: 4.0189 - accuracy: 0.2961 -
val_loss: 4.3577 - val_accuracy: 0.2832
Epoch 25/30
94/94 [==============================] - 18s 186ms/step - loss: 3.9895 - accuracy: 0.3002 -
val_loss: 4.3448 - val_accuracy: 0.2856
Epoch 26/30
94/94 [==============================] - 18s 187ms/step - loss: 3.9640 - accuracy: 0.3042 -
val_loss: 4.3356 - val_accuracy: 0.2875
Epoch 27/30
94/94 [==============================] - 17s 181ms/step - loss: 3.9346 - accuracy: 0.3067 -
val_loss: 4.3241 - val_accuracy: 0.2887
Epoch 28/30
94/94 [==============================] - 18s 190ms/step - loss: 3.9099 - accuracy: 0.3096 -
val_loss: 4.3183 - val_accuracy: 0.2903
Epoch 29/30
94/94 [==============================] - 18s 187ms/step - loss: 3.8859 - accuracy: 0.3105 -
val_loss: 4.3081 - val_accuracy: 0.2914
Epoch 30/30
94/94 [==============================] - 18s 189ms/step - loss: 3.8610 - accuracy: 0.3127 -
val_loss: 4.3015 - val_accuracy: 0.2919
<keras.callbacks.History at 0x78cda42eacb0>
from tensorflow.keras.models import Model

# Define encoder model to get encoder states
encoder_model = Model(encoder_inputs, encoder_states)

# Define decoder model with encoder states as initial state
decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
```

```
decoder_inputs_single = Input(shape=(1,))
decoder_inputs_single_embedded = decoder_embedding(decoder_inputs_single)

decoder_outputs, state_h, state_c = decoder_lstm(decoder_inputs_single_embedded,
initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]
decoder_outputs = decoder_dense(decoder_outputs)

decoder_model = Model([decoder_inputs_single] + decoder_states_inputs, [decoder_outputs] +
decoder_states)

# Helper function to generate a response given an input sequence
def generate_response(input_seq):
# Encode the input sequence to get the initial decoder states
states_value = encoder_model.predict(input_seq)

# Initialize the target sequence with a start token
target_seq = np.array([[tokenizer.word_index['sos']]])

stop_condition = False
response = []

while not stop_condition:
output_tokens, h, c = decoder_model.predict([target_seq] + states_value)

# Sample a token from the output distribution
sampled_token_index = np.argmax(output_tokens[0, -1, :])

# If the predicted word index is 0, use a period instead
if sampled_token_index == 0:
sampled_token = '.'
else:
sampled_token = tokenizer.index_word[sampled_token_index]

response.append(sampled_token)

# Exit condition: either hit max length or find stop token
if sampled_token == 'eos' or len(response) > max_seq_length:
stop_condition = True

# Update the target sequence with the sampled token
target_seq = np.array([[sampled_token_index]])

# Update the decoder states
states_value = [h, c]

return ' '.join(response)
# Test the response generation
input_sequence = tokenizer.texts_to_sequences(["say something!"])
input_sequence = pad_sequences(input_sequence, maxlen=max_seq_length, padding='post',
truncating='post')
response = generate_response(input_sequence)
print("Input:", f'{input_sequence}')
print("Response:", response)
1/1 [==============================] - 0s 27ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 38ms/step
```

1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 22ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 21ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 20ms/step
1/1 [==============================] - 0s 20ms/step
Input: [[117 110 0 0 0 0 0 0 0 0]]
Response: i ' s a lot of the lot of the lot