IBM NAN MUTHALVAN

CHAT-BOT IN PYTHON

PHASE III

NAME- T.SRINATH

REGISTER NO - 721921243302

INTRODUCTION

Natural Language Processing or NLP is a prerequisite for our project. NLP allows computers and algorithms to understand human interactions via various languages. In order to process a large amount of natural language data, an AI will definitely need NLP or Natural Language Processing. Currently, we have a number of NLP research ongoing in order to improve the AI chatbots and help them understand the complicated nuances and undertones of human conversations.

Chatbots are nothing but applications that are used by businesses or other entities to conduct an automatic conversation between a human and an AI. These conversations may be via text or speech. Chatbots are required to understand and mimic human conversation while interacting with humans from all over the world. From the first chatbot to be created ELIZA to Amazon’s ALEXA today, chatbots have come a long way. In this tutorial, we are going to

cover all the basics you need to follow along and create a basic chatbot that can understand human interaction and also respond accordingly. We will be using speech recognition APIs and also pre-trained Transformer models.

NLP

NLP stands for Natural Language Processing. Using NLP technology, you can help a machine understand human speech and spoken words. NLP combines computational linguistics that is the rule-based modelling of the human spoken language with intelligent algorithms such as statistical, machine, and deep learning algorithms. These technologies together create the smart voice assistants and chatbots that you may be used in everyday life.

There are a number of human errors, differences, and special intonations that humans use every day in their speech. NLP technology allows the machine to understand, process, and respond to large volumes of text rapidly in real-time. In everyday life, you have encountered NLP tech in voice-guided GPS apps, virtual assistants, speech-to-text note creation apps, and other app support chatbots. This tech has found immense use cases in the business sphere where it’s used to streamline processes, monitor employee productivity, and increase sales and after-sales efficiency.

TYPES OF CHATBOTS

Chatbots are a relatively recent concept and despite having a huge number of programs and NLP tools, we basically have just two different categories of chatbots based on the NLP technology that they utilize. These two types of chatbots are as follows:

Scripted chatbots: Scripted chatbots are classified as chatbots that work on pre-determined scripts that are created and stored in their library. Whenever a user types a query or speaks a query (in the case of chatbots equipped with speech to text conversion modules), the chatbot responds to this query according to the pre-determined script that is stored within its library.One of the cons of such a chatbot is the fact that user needs to provide their que3ry in a very structured manner with comma-separated commands or other forms of a regular expression that makes it easier for the bot to perform string analysis and understand the query.

Artificially Intelligent Chatbots: Artificially intelligent chatbots, as the name suggests, are created to mimic human-like traits and responses. NLP or Natural Language Processing is hugely responsible for enabling such chatbots to understand the dialects and undertones of human conversation. NLP combined with artificial intelligence creates a truly intelligent chatbot that can respond to nuanced questions and learn from every interaction to create better-suited responses the next time.

DEVELOPMENT

Importing libraries

```
import tensorflow as tf
from sklearn.model_selection import train_test_split


import unicodedata
import re
import numpy as np


import warnings
warnings.filterwarnings('ignore')
```

Data preprocessing

The basic text processing in NLP are:

1. Sentence Segmentation
2. Normalization
3. Tokenization

Segmentation

```
data=open('/content/dialogs.txt','r').read()
```

```
QA_list=[QA.split('\t') for QA in data.split('\n')]
print(QA_list[:5])
```

Output:

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

```
questions=[row[0] for row in QA_list]
answers=[row[1] for row in QA_list]
```

```
print(questions[0:5])
print(answers[0:5])
```

Output:

```
['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?"]
["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."]
```

Normalization

```
def remove_diacritic(text):
    return ''.join(char for char in unicodedata.normalize('NFD',text)
if unicodedata.category(char) !='Mn')
```

```
def preprocessing(text):
```

```
    text=remove_diacritic(text.lower().strip())

text=re.sub(r"([?.!,¿])", r" \1 ", text)

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

    text=re.sub(r"[^a-zA-Z?.!,¿]+", " ", text)

    text=text.strip()

    text='<start> ' + text + ' <end>'

    return text

preprocessed_questions=[preprocessing(sen) for sen in questions]
preprocessed_answers=[preprocessing(sen) for sen in answers]

print(preprocessed_questions[0])
print(preprocessed_answers[0])
```

Output:

```
<start> hi , how are you doing ? <end> <start>
i m fine . how about yourself ? <end>
```

Tokenization

```
def tokenize(lang):
    lang_tokenizer = tf.keras.preprocessing.text.Tokenizer(
filters='')

    lang_tokenizer.fit_on_texts(lang)

    return lang_tokenizer
```

Word Embedding

```
def vectorization(lang_tokenizer,lang):


    tensor = lang_tokenizer.texts_to_sequences(lang)


```

```
    tensor = tf.keras.preprocessing.sequence.pad_sequences(tensor,
                                                           padding='post')

    return tensor
```

Creating Dataset

```
def load_Dataset(data,size=None):

    if(size!=None):
        y,X=data[:size]
else:
        y,X=data

    X_tokenizer=tokenize(X)
    y_tokenizer=tokenize(y)

    X_tensor=vectorization(X_tokenizer,X)
    y_tensor=vectorization(y_tokenizer,y)

    return  X_tensor,X_tokenizer, y_tensor, y_tokenizer
```

```
size=30000
data=preprocessed_answers,preprocessed_questions\


X_tensor,X_tokenizer, y_tensor, y_tokenizer=load_Dataset(data,size)
```

```

max_length_y, max_length_X = y_tensor.shape[1], X_tensor.shape[1]
```

Splitting Data

```
X_train, X_val, y_train, y_val = train_test_split(X_tensor, y_tensor,
test_size=0.2)
```

```
print(len(X_train), len(y_train), len(X_val), len(y_val))
```

Output:

```
2980 2980 745 745
```

Tensorflow Dataset

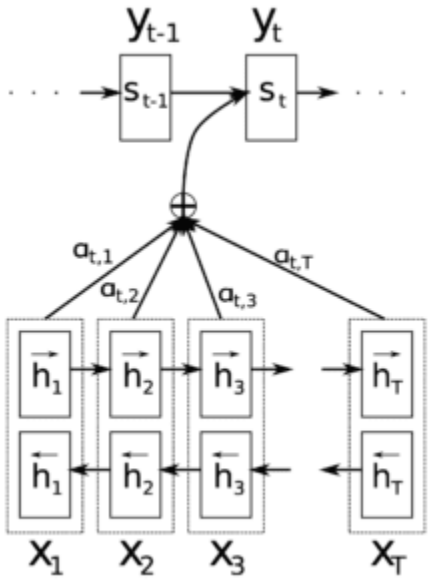
```
BUFFER_SIZE = len(X_train)
BATCH_SIZE = 64
steps_per_epoch = len(X_train)//BATCH_SIZE
embedding_dim = 256
units = 1024
```

```
vocab_inp_size  =  len(X_tokenizer.word_index)+1
vocab_tar_size = len(y_tokenizer.word_index)+1

dataset = tf.data.Dataset.from_tensor_slices((X_train,
y_train)).shuffle(BUFFER_SIZE)
dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)

example_input_batch, example_target_batch = next(iter(dataset))
example_input_batch.shape, example_target_batch.shape
```

Output:

```
(TensorShape([64, 24]), TensorShape([64, 24]))
```

Model



Buliding Model Architecture

Encoder

```
class Encoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, enc_units, batch_sz):
super(Encoder, self).__init__()
        self.batch_sz = batch_sz
        self.enc_units = enc_units
```

```
        self.embedding = tf.keras.layers.Embedding(vocab_size,
embedding_dim)
        self.gru = tf.keras.layers.GRU(self.enc_units,
                                       return_sequences=True,
                                       return_state=True,
                                       recurrent_initializer='glorot_unifo
rm')

    def call(self, x, hidden):
        x = self.embedding(x)
        output, state = self.gru(x, initial_state = hidden)
return output, state

    def initialize_hidden_state(self):
        return tf.zeros((self.batch_sz, self.enc_units))
```

```
encoder = Encoder(vocab_inp_size, embedding_dim, units, BATCH_SIZE)


sample_hidden = encoder.initialize_hidden_state()
sample_output, sample_hidden = encoder(example_input_batch, sample_hidden)
print ('Encoder output shape: (batch size, sequence length, units)
{}'.format(sample_output.shape))
print ('Encoder Hidden state shape: (batch size, units)
{}'.format(sample_hidden.shape))
```

Output:

```
Encoder output shape: (batch size, sequence length, units) (64, 24, 1024)
Encoder Hidden state shape: (batch size, units) (64, 1024)
```

Attention Mechanism

```
class BahdanauAttention(tf.keras.layers.Layer):
def __init__(self, units):
        super(BahdanauAttention, self).__init__()
self.W1 = tf.keras.layers.Dense(units)
        self.W2 = tf.keras.layers.Dense(units)
        self.V = tf.keras.layers.Dense(1)

    def call(self, query, values):

        query_with_time_axis = tf.expand_dims(query, 1)
```

```
        score = self.V(tf.nn.tanh(
            self.W1(query_with_time_axis) + self.W2(values)))



        attention_weights = tf.nn.softmax(score, axis=1)

        context_vector = attention_weights * values
        context_vector = tf.reduce_sum(context_vector, axis=1)

        return context_vector, attention_weights
```

```
attention_layer = BahdanauAttention(10)
attention_result, attention_weights = attention_layer(sample_hidden,
sample_output)

print("Attention result shape: (batch size, units)
{}".format(attention_result.shape))
print("Attention weights shape: (batch_size, sequence_length, 1)
{}".format(attention_weights.shape))
```

Output:

```
Attention result shape: (batch size, units) (64, 1024)
Attention weights shape: (batch_size, sequence_length, 1) (64, 24, 1)
```

Decoder

```
class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, dec_units, batch_sz):
super(Decoder, self).__init__()
        self.batch_sz = batch_sz
        self.dec_units = dec_units
        self.embedding = tf.keras.layers.Embedding(vocab_size,
embedding_dim)
        self.gru = tf.keras.layers.GRU(self.dec_units,
                                       return_sequences=True,
                                       return_state=True,
                                       recurrent_initializer='glorot_unifo
rm')
        self.fc = tf.keras.layers.Dense(vocab_size)

        self.attention = BahdanauAttention(self.dec_units)

def call(self, x, hidden, enc_output):

        context_vector, attention_weights = self.attention(hidden,
enc_output)
```

```
        x = self.embedding(x)

        x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

        output, state = self.gru(x)

        output = tf.reshape(output, (-1, output.shape[2]))

        x = self.fc(output)

        return x, state, attention_weights
decoder = Decoder(vocab_tar_size, embedding_dim, units, BATCH_SIZE)
```

```
sample_decoder_output, _, _ = decoder(tf.random.uniform((BATCH_SIZE, 1)),
sample_hidden, sample_output)

print ('Decoder output shape: (batch_size, vocab size)
{}'.format(sample_decoder_output.shape))
```

Output:

```
Decoder output shape: (batch_size, vocab size) (64, 2349)
```

Training Model

```
optimizer = tf.keras.optimizers.Adam()
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
from_logits=True, reduction='none')

def loss_function(real, pred):
    mask = tf.math.logical_not(tf.math.equal(real, 0))
loss_ = loss_object(real, pred)

    mask = tf.cast(mask, dtype=loss_.dtype)
    loss_ *= mask

    return tf.reduce_mean(loss_)
```

```
@tf.function
def train_step(inp, targ, enc_hidden):
loss = 0
```

```
    with tf.GradientTape() as tape:
        enc_output, enc_hidden = encoder(inp, enc_hidden)

        dec_hidden = enc_hidden

        dec_input = tf.expand_dims([y_tokenizer.word_index['<start>']] *
BATCH_SIZE, 1)


        for t in range(1, targ.shape[1]):

            predictions, dec_hidden, _ = decoder(dec_input, dec_hidden,
enc_output)

            loss += loss_function(targ[:, t], predictions)

            dec_input = tf.expand_dims(targ[:, t], 1)

    batch_loss = (loss / int(targ.shape[1]))

    variables = encoder.trainable_variables + decoder.trainable_variables

gradients = tape.gradient(loss, variables)

    optimizer.apply_gradients(zip(gradients, variables))

    return batch_loss
```

```
EPOCHS = 40

for epoch in range(1, EPOCHS + 1):
    enc_hidden = encoder.initialize_hidden_state()
total_loss = 0

    for (batch, (inp, targ)) in enumerate(dataset.take(steps_per_epoch)):

        batch_loss = train_step(inp, targ, enc_hidden)
        total_loss += batch_loss

    if(epoch % 4 == 0):
        print('Epoch:{:3d} Loss:{:.4f}'.format(epoch,
                                               total_loss / steps_per_epoch))
```

Output:

```
 Epoch:   4  Loss:1.5338
Epoch:   8  Loss:1.2803
```

```
Epoch:   12   Loss:1.0975
Epoch:   16   Loss:0.9404
Epoch:   20   Loss:0.7773
Epoch:   24   Loss:0.6040
Epoch:   28   Loss:0.4042
Epoch:   32   Loss:0.2233
Epoch:   36   Loss:0.0989
Epoch: 40 Loss:0.0470
```

Model Evaluation

```
def remove_tags(sentence):
    return sentence.split("<start>")[-1].split("<end>")[0]
```

```
def evaluate(sentence):
    sentence = preprocessing(sentence)

    inputs = [X_tokenizer.word_index[i] for i in sentence.split(' ')]
    inputs = tf.keras.preprocessing.sequence.pad_sequences([inputs],
                                                           maxlen=max_length
_X,
                                                           padding='post')
inputs = tf.convert_to_tensor(inputs)

    result = ''

    hidden = [tf.zeros((1, units))]
    enc_out, enc_hidden = encoder(inputs, hidden)

    dec_hidden = enc_hidden
    dec_input = tf.expand_dims([y_tokenizer.word_index['<start>']], 0)

    for t in range(max_length_y):
```

```
        predictions, dec_hidden, attention_weights = decoder(dec_input,
dec_hidden,
enc_out)

        attention_weights = tf.reshape(attention_weights, (-1, ))

        predicted_id = tf.argmax(predictions[0]).numpy()

        result += y_tokenizer.index_word[predicted_id] + ' '

        if y_tokenizer.index_word[predicted_id] == '<end>':
return remove_tags(result), remove_tags(sentence)
```

```
        dec_input  =  tf.expand_dims([predicted_id],  0)

return remove_tags(result), remove_tags(sentence)
```

```
def ask(sentence):
    result, sentence = evaluate(sentence)

    print('Question: %s' % (sentence))
    print('Predicted answer: {}'.format(result))
```

```
ask(questions[1])
```

Output:

```
Question:  i m fine . how about yourself ?  Predicted
answer: I m pretty good . thanks for asking
```

Conclusion

A chatbot in Python is a powerful and versatile tool that can be used for a wide range of applications. In conclusion, here are some key points to consider:

1. **Accessibility**: Python is a popular and accessible programming language, making it a great choice for developing chatbots. It offers a wide range of libraries and frameworks that simplify the development process.
2. **Natural Language Processing (NLP):** Python boasts robust NLP libraries such as NLTK, spaCy, and the Hugging Face Transformers library, which enable chatbots to understand and generate human-like text. These libraries have made it easier than ever to create sophisticated conversational agents.
3. **Versatility**: Python chatbots can be implemented in various domains, from customer service and e-commerce to healthcare and education. Their versatility allows them to adapt to a wide range of industries and tasks.

4. **Integration**: Python chatbots can easily integrate with existing systems and platforms, allowing for seamless communication with users through popular messaging services like WhatsApp, Facebook Messenger, or Slack.
5. **Machine Learning**: Python's ecosystem is rich with machine learning tools, such as scikit-learn and TensorFlow, which can be used to enhance chatbot functionality through supervised or reinforcement learning, improving its ability to understand user intent and generate relevant responses.

6. **Continuous Improvement:** Chatbots can learn and adapt over time through user interactions and feedback. By implementing data collection and analysis, Python chatbots can be continuously improved to provide better user experiences.
7. **Scalability:** Python chatbots can be deployed on a variety of platforms, from web applications to mobile apps, and they can scale to handle a large number of users simultaneously, making them suitable for businesses of all sizes.
8. **Ethical Considerations**: It's important to consider ethical and privacy aspects when developing chatbots. Developers should be mindful of user data and privacy concerns, and ensure that chatbots provide value while respecting ethical guidelines.

In conclusion, Python is an excellent choice for building chatbots due to its accessibility, powerful NLP libraries, versatility, and robust ecosystem. With the right design, implementation, and ongoing improvements, Python chatbots can provide efficient and engaging interactions with users in a wide range of domains.