# ChatBot Using Python

## Definition :

Chat Bot is a Python library that is designed to deliver automated responses to user inputs. It makes use of a combination of ML algorithms to generate many different types of responses .At the most basic level, a chat bot is a computer program that simulates and processes human conversation (either written or spoken), allowing humans to interact with digital devices as if they were communicating with a real person.

## Prepare the Date :

```
import tensorflow as tf

import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from sklearn.model_selection import train_test_split

import unicodedata
import re
import numpy as np
import os
import io
import time

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

```
file = open('../input/simple-dialogs-for-chatbot/dialogs.txt',
'r').read()

qna_list = [f.split('\t') for f in file.split('\n')]

questions = [x[0] for x in qna_list]answers = [x[1] for x in
qna_list]

print("Question: ", questions[0])print("Answer: ", answers[0])
```

## Preprocess sentences :

```
def unicode_to_ascii(s):

    return ''.join(c for c in unicodedata.normalize('NFD', s)

        if unicodedata.category(c) != 'Mn')
```

```
def preprocess_sentence(w):

    w = unicode_to_ascii(w.lower().strip())

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

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

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

    w = w.strip()

    w = '<start> ' + w + ' <end>'

    return w

print(preprocess_sentence(questions[0]))print
(preprocess_sentence(answers[0]))

pre_questions = [preprocess_sentence(w) for w in
questions]pre_answers = [preprocess_sentence(w) for w in
answers]
```

**Tokenize :**

```
def tokenize(lang):

   lang_tokenizer = tf.keras.preprocessing.text.Tokenizer(

      filters='')

    lang_tokenizer.fit_on_texts(lang)

    tensor = lang_tokenizer.texts_to_sequences(lang)

    tensor = tf.keras.preprocessing.sequence.
pad_sequences(tensor,

                                                    padding=
'post')

    return tensor, lang_tokenizer

def load_dataset(data, num_examples=None):

    # creating cleaned input, output pairs
```

```
    if(num_examples != None):

        targ_lang, inp_lang, = data[:num_examples]

    else:

        targ_lang, inp_lang, = data

    input_tensor, inp_lang_tokenizer = tokenize(inp_lang)

    target_tensor, targ_lang_tokenizer = tokenize(targ_lang)

    return input_tensor, target_tensor, inp_lang_tokenizer,
targ_lang_tokenizer

num_examples = 30000data = pre_answers,
pre_questionsinput_tensor, target_tensor, inp_lang, targ_lang
= load_dataset(data, num_examples)

# Calculate max_length of the target tensorsmax_length_targ,
max_length_inp = target_tensor.shape[1], input_tensor.shape[1
]
```

In [10]:

```
# Creating training and validation sets using an 80-20 split
input_tensor_train, input_tensor_val, target_tensor_train,
target_tensor_val = train_test_split(input_tensor,
target_tensor, test_size=0.2)

# Show lengthprint(len(input_tensor_train), len
(target_tensor_train), len(input_tensor_val), len
(target_tensor_val))
```

**Word to index :**

```
def convert(lang, tensor):

    for t in tensor:

        if t!=0:

            print ("%d ----> %s" % (t, lang.index_word[t]))

print ("Input Language; index to word mapping"
)convert(inp_lang, input_tensor_train[0])print ()print (
"Target Language; index to word mapping")convert(targ_lang,
```

```
target_tensor_train[0])
```

### Create Tensorflow dataset :

```
BUFFER_SIZE = len(input_tensor_train)
BATCH_SIZE = 64
steps_per_epoch = len(input_tensor_train)//BATCH_SIZE
embedding_dim = 256
units = 1024
vocab_inp_size = len(inp_lang.word_index)+1
vocab_tar_size = len(targ_lang.word_index)+1

dataset = tf.data.Dataset.from_tensor_slices((input_tensor_train,
target_tensor_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
```

### Encoder/Decoder with attention equations :

### 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_uniform')
```

```
    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 inputsample_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))
```

**Attention :**

```
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 hidden state shape == (batch_size, hidden
size)

        # query_with_time_axis shape == (batch_size, 1,
hidden size)

        # values shape == (batch_size, max_len, hidden size)
```

```
        # we are doing this to broadcast addition along the
time axis to calculate the score

        query_with_time_axis = tf.expand_dims(query, 1)



        # score shape == (batch_size, max_length, 1)

        # we get 1 at the last axis because we are applying
score to self.V

        # the shape of the tensor before applying self.V is
(batch_size, max_length, units)

        score = self.V(tf.nn.tanh(

            self.W1(query_with_time_axis) + self.W2(values)))

        # attention_weights shape == (batch_size, max_length,
1)

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

        # context_vector shape after sum == (batch_size,
hiddn_size)

        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))
```

**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_uniform')

        self.fc = tf.keras.layers.Dense(vocab_size)

        # used for attention

        self.attention = BahdanauAttention(self.dec_units)

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

        # enc_output shape == (batch_size, max_length,
hidden_size)

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

        # x shape after passing through embedding ==
(batch_size, 1, embedding_dim)

        x = self.embedding(x)

       # x shape after concatenation == (batch_size, 1,
embedding_dim + hidden_size)

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

        # passing the concatenated vector to the GRU
```

```
        output, state = self.gru(x)


        # output shape == (batch_size * 1, hidden_size)
        output = tf.reshape(output, (-1, output.shape[2]))
        # output shape == (batch_size, vocab)
        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))
```

**Training :**

```
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.functiondef 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([targ_lang.word_index[
'<start>']] * BATCH_SIZE, 1)

        # Teacher forcing - feeding the target as the next
input

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

            # passing enc_output to the decoder

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

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

            # using teacher forcing

            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))
```

**Evaluate :**

```
def remove_tags(sentence):

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

def evaluate(sentence):

    sentence = preprocess_sentence(sentence)

    inputs = [inp_lang.word_index[i] for i in sentence.split(
' ')]

    inputs = tf.keras.preprocessing.sequence.
pad_sequences([inputs],

                                  maxlen=max_length_inp,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([targ_lang.word_index[
'<start>']], 0)
```

```
    for t in range(max_length_targ):

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

        # storing the attention weights to plot later on

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

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

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

        if targ_lang.index_word[predicted_id] == '<end>':

            return remove_tags(result), remove_tags(sentence)

        # the predicted ID is fed back into the model

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

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

**Answer question :**

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

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