CREATE A CHATBOT USING PYTHON

TEAM MEMBERS:

1. S.Logeswaran
2. C.P.Arun Kumar
3. M.Jagan
4. S.Aakash

CREATE A CHABOT USING PYTHON

What is Text Preprocessing?

Natural Language Processing (NLP) is a branch of Data Science which deals with Text data. Apart from numerical data, Text data is available to a great extent which is used to analyze and solve business problems. But before using the data for analysis or prediction, processing the data is important.

To prepare the text data for the model building we perform text preprocessing. It is the very first step of NLP projects. Some of the preprocessing steps are:

- Removing punctuations like . , ! $( ) * % @
- Removing URLs
- Removing Stop words
- Lower casing
- Tokenization
- Stemming
- Lemmatization

Prepocessing and coding :

```
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
```

CREATE A CHABOT USING PYTHON

```
from tensorflow.keras.layers import
LSTM,Dense,Embedding,Dropout,LayerNormalization


df=pd.read_csv('/kaggle/input/simple-dialogs-for-
chatbot/dialogs.txt',sep='\t',names=['question','answer'])

print(f'Dataframe size: {len(df)}')

df.head()
```

output:

```
Dataframe size: 3725
```

[4]:

	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.

Data preprocessing:

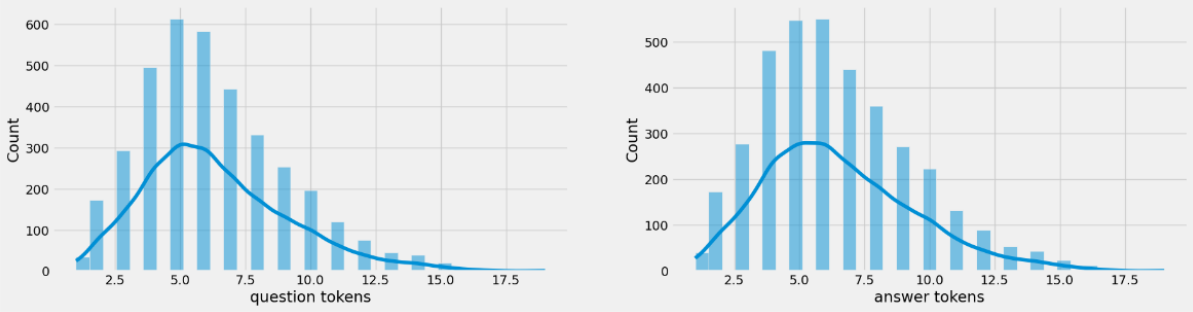
Data visualization:

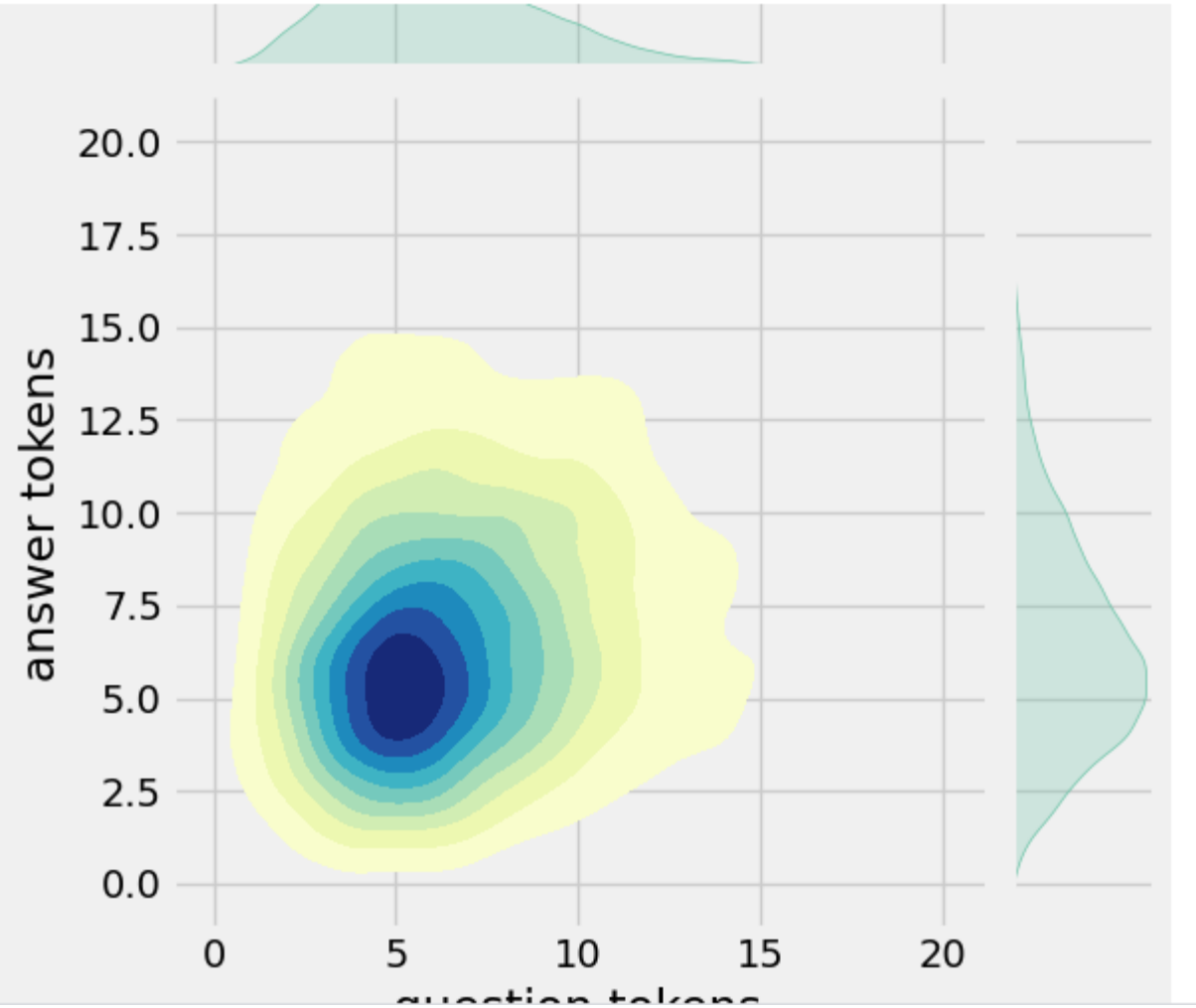
```
df['question tokens']=df['question'].apply(lambda x:len(x.split()))

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])
```

CREATE A CHABOT USING PYTHON

```
sns.jointplot(x='question tokens',y='answer
tokens',data=df,kind='kde',fill=True,cmap='YlGnBu')

plt.show()
```

output:





CREATE A CHABOT USING PYTHON

Text cleaning:

```
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)
```

CREATE A CHABOT USING PYTHON

```
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)
```

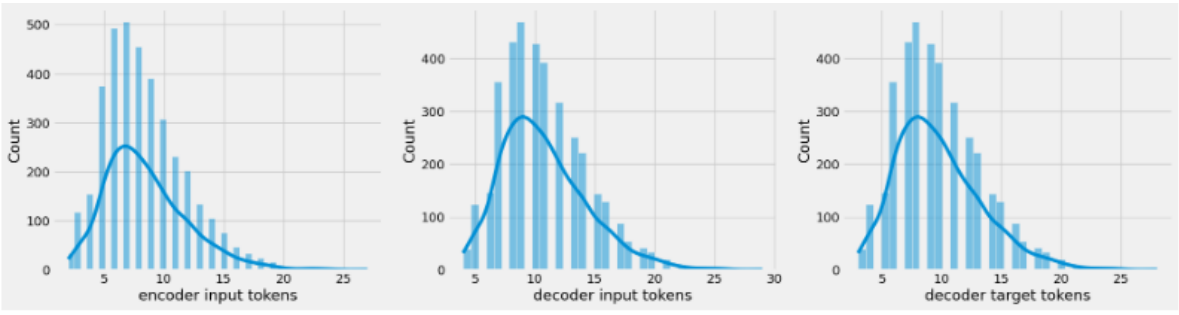
output:

	question	answer	encoder_inputs	decoder_targets	decoder_inputs
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>

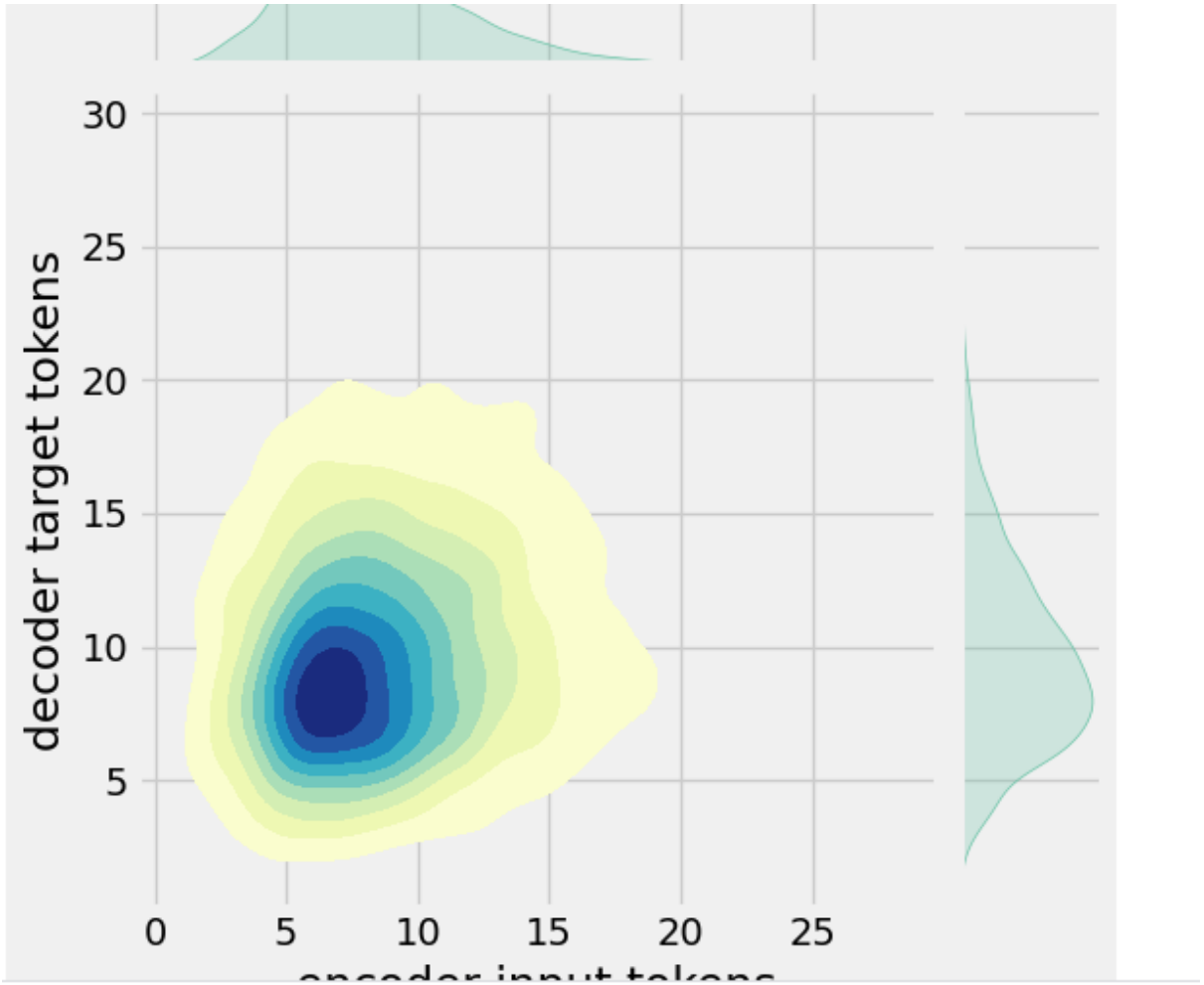
CREATE A CHABOT USING PYTHON

```
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()
```

output:



CREATE A CHABOT USING PYTHON



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()}")

CREATE A CHABOT USING PYTHON

```
df.drop(columns=['question','answer','encoder input
tokens','decoder input tokens','decoder target
tokens'],axis=1,inplace=True)
params={
   "vocab_size":2500,
   "max_sequence_length":30,
   "learning_rate":0.008,
   "batch_size":149,
   "lstm_cells":256,
   "embedding_dim":256,
   "buffer_size":10000
}
learning_rate=params['learning_rate']
batch_size=params['batch_size']
embedding_dim=params['embedding_dim']
lstm_cells=params['lstm_cells']
vocab_size=params['vocab_size']
buffer_size=params['buffer_size']
max_sequence_length=params['max_sequence_length']
df.head(10)
```

output:

CREATE A CHABOT USING PYTHON

```
After preprocessing: for example ,  if your birth date is january  1  2  ,  1  9  8  7  ,  write  0  1  /  1
2  /  8  7  .
Max encoder input length: 27
Max decoder input length: 29
Max decoder target length: 28
```

	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:

```
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]}')
```

CREATE A CHABOT USING PYTHON

output:

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

```
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}')

output:

```
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)
```

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]} ...')

output:

```
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] ...
```

data=tf.data.Dataset.from_tensor_slices((x,yd,y))

data=data.shuffle(buffer_size)

CREATE A CHABOT USING PYTHON

```
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}')
```

output:

```
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)
```

Build models:

Build encounter:

```
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='enc
oder')
encoder.call(_[0])
```

Output:

```
(<tf.Tensor: shape=(149, 256), dtype=float32, numpy=
 array([[ 0.03649763, -0.02320833, -0.00988133, ...,  0.04249467,
          0.00426203, -0.09942936],
        [ 0.13327979,  0.2136438 , -0.04917734, ...,  0.08147815,
          0.01274394, -0.00944547],
        [ 0.12572058,  0.04214957, -0.05221859, ...,  0.02721892,
          0.28633878,  0.17121683],
        ...,
        [ 0.13974643,  0.10875662, -0.09624026, ...,  0.09803431,
          0.22875303, -0.25971597],
        [ 0.20825417,  0.07539225, -0.07739356, ...,  0.21375725,
          0.22656499, -0.04885658],
        [ 0.15003441,  0.04473909, -0.07097201, ...,  0.08152565,
          0.08286292, -0.06150594]], dtype=float32)>,
 <tf.Tensor: shape=(149, 256), dtype=float32, numpy=
 array([[ 0.09377024, -0.07678603, -0.01484064, ...,  0.08086851,
          0.01450279, -0.24342684],
        [ 0.25469437,  0.63573027, -0.21134387, ...,  0.12172738,
          0.01771331, -0.02017355],
        [ 0.25336558,  0.10952759, -0.21970585, ...,  0.04286795,
          0.41379124,  0.37899172],
        ...,
        [ 0.31207675,  0.28705305, -0.44277436, ...,  0.14716351,
          0.32020888, -0.631289  ],
        [ 0.4436314 ,  0.18735178, -0.3490557 , ...,  0.33589238,
          0.32183608, -0.10789847],
        [ 0.31745085,  0.1167425 , -0.3114372 , ...,  0.12000009,
          0.11160254, -0.13327222]], dtype=float32)>)
```

Build decoder:

```
class Decoder(tf.keras.models.Model):
   def _init_(self,units,embedding_dim,vocab_size,*args,**kwargs) ->
None:
      super()._init_(*args,**kwargs)
      self.units=units
      self.embedding_dim=embedding_dim
      self.vocab_size=vocab_size
      self.embedding=Embedding(
         vocab_size,
```

```
        embedding_dim,
        name='decoder_embedding',
        mask_zero=True,
        embeddings_initializer=tf.keras.initializers.HeNormal()
    )
    self.normalize=LayerNormalization()
    self.lstm=LSTM(
        units,
        dropout=.4,
        return_state=True,
        return_sequences=True,
        name='decoder_lstm',
        kernel_initializer=tf.keras.initializers.HeNormal()
    )
    self.fc=Dense(
        vocab_size,
        activation='softmax',
        name='decoder_dense',
        kernel_initializer=tf.keras.initializers.HeNormal()
    )

def call(self,decoder_inputs,encoder_states):
    x=self.embedding(decoder_inputs)
```

```
    x=self.normalize(x)
    x=Dropout(.4)(x)

x,decoder_state_h,decoder_state_c=self.lstm(x,initial_state=encoder
_states)
    x=self.normalize(x)
    x=Dropout(.4)(x)
    return self.fc(x)

decoder=Decoder(lstm_cells,embedding_dim,vocab_size,name='dec
oder')
decoder(_[1][:1],encoder(_[0][:1]))
```

Output:

```
<tf.Tensor: shape=(1, 30, 2443), dtype=float32, numpy=
array([[[1.52762041e-05, 1.11079324e-04, 3.70255118e-04, ...,
         1.15379211e-04, 4.04916616e-04, 4.50501539e-04],
        [1.33365666e-05, 8.83720859e-05, 4.40978183e-04, ...,
         2.40944559e-04, 2.39710018e-04, 1.74776942e-04],
        [1.60832915e-05, 3.40561557e-04, 5.67587966e-04, ...,
         3.26075155e-04, 7.79727852e-05, 2.11102015e-05],
        ...,
        [1.98478865e-05, 4.85003577e-04, 3.92090267e-04, ...,
         6.06390997e-04, 7.15351343e-05, 1.29978798e-04],
        [1.98478865e-05, 4.85003577e-04, 3.92090267e-04, ...,
         6.06390997e-04, 7.15351343e-05, 1.29978798e-04],
        [1.98478865e-05, 4.85003577e-04, 3.92090267e-04, ...,
         6.06390997e-04, 7.15351343e-05, 1.29978798e-04]]], dtype=float32)>
```

Build training model:

```
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)
```

CREATE A CHABOT USING PYTHON

```
    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_va
riables
        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])

output:

```
<tf.Tensor: shape=(149, 30, 2443), dtype=float32, numpy=
array([[[1.52761968e-05, 1.11079273e-04, 3.70255235e-04, ...,
         1.15379211e-04, 4.04916791e-04, 4.50501771e-04],
        [1.33365484e-05, 8.83720422e-05, 4.40978096e-04, ...,
         2.40944573e-04, 2.39709902e-04, 1.74776869e-04],
        [1.60832988e-05, 3.40561412e-04, 5.67587966e-04, ...,
         3.26075475e-04, 7.79727852e-05, 2.11101706e-05],
        ...,
        [1.98479011e-05, 4.85003693e-04, 3.92090733e-04, ...,
         6.06391754e-04, 7.15351125e-05, 1.29978769e-04],
        [1.98479011e-05, 4.85003693e-04, 3.92090733e-04, ...,
         6.06391754e-04, 7.15351125e-05, 1.29978769e-04],
        [1.98479011e-05, 4.85003693e-04, 3.92090733e-04, ...,
         6.06391754e-04, 7.15351125e-05, 1.29978769e-04]],

       [[5.09164856e-05, 2.98033527e-04, 1.40225969e-03, ...,
         6.57317214e-05, 5.60949440e-04, 4.89029044e-04],
        [3.02779681e-05, 3.16660735e-04, 1.92077452e-04, ...,
         1.50211374e-04, 5.16688306e-05, 1.53324881e-03],
        [8.80527568e-06, 3.04767309e-04, 1.11181806e-04, ...,
         4.28329186e-05, 7.90638223e-05, 2.69777112e-04],
        ...,
        [1.44784535e-05, 3.74377822e-04, 9.32166367e-05, ...,
         9.89597756e-05, 1.54792851e-05, 5.54081598e-05],
        [1.44784535e-05, 3.74377822e-04, 9.32166367e-05, ...,
         9.89597756e-05, 1.54792851e-05, 5.54081598e-05],
        [1.44784535e-05, 3.74377822e-04, 9.32166367e-05, ...,
         9.89597756e-05, 1.54792851e-05, 5.54081598e-05]],
```

Train model:

```
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)
    ]
)
```

Output:

```
Epoch 1/100
23/23 [==============================] - ETA: 0s - loss: 1.6556 - accuracy: 0.2180
Epoch 1: val_loss improved from inf to 1.35436, saving model to ckpt
23/23 [==============================] - 59s 2s/step - loss: 1.6469 - accuracy: 0.2194 - val_loss: 1.3544 - v
al_accuracy: 0.2770
Epoch 2/100
23/23 [==============================] - ETA: 0s - loss: 1.2266 - accuracy: 0.3095
Epoch 2: val_loss improved from 1.35436 to 1.07659, saving model to ckpt
23/23 [==============================] - 48s 2s/step - loss: 1.2259 - accuracy: 0.3096 - val_loss: 1.0766 - v
al_accuracy: 0.3371
Epoch 3/100
23/23 [==============================] - ETA: 0s - loss: 1.0943 - accuracy: 0.3413
Epoch 3: val_loss did not improve from 1.07659
23/23 [==============================] - 19s 828ms/step - loss: 1.0934 - accuracy: 0.3414 - val_loss: 1.1560
- val_accuracy: 0.3299
Epoch 4/100
23/23 [==============================] - ETA: 0s - loss: 1.0168 - accuracy: 0.3560
Epoch 4: val_loss improved from 1.07659 to 0.89152, saving model to ckpt
23/23 [==============================] - 48s 2s/step - loss: 1.0172 - accuracy: 0.3569 - val_loss: 0.8915 - v
al_accuracy: 0.3661
Epoch 5/100
23/23 [==============================] - ETA: 0s - loss: 0.9597 - accuracy: 0.3694
Epoch 5: val_loss did not improve from 0.89152
23/23 [==============================] - 19s 837ms/step - loss: 0.9592 - accuracy: 0.3689 - val_loss: 0.9583
- val_accuracy: 0.3723
Epoch 6/100
23/23 [==============================] - ETA: 0s - loss: 0.9148 - accuracy: 0.3792
Epoch 6: val_loss did not improve from 0.89152
23/23 [==============================] - 20s 851ms/step - loss: 0.9199 - accuracy: 0.3786 - val_loss: 0.9103
- val_accuracy: 0.3866
Epoch 7/100
```

Visualize Metrics:

```
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()
```

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('--------------------')
```

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=[en
coder_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_stat
e_c]],

outputs=[decoder_outputs,[decoder_state_h,decoder_state_c]],nam
e='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))
```

CREATE A CHABOT USING PYTHON

```
    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)
```

CREATE A CHABOT USING PYTHON

```
    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],traini
ng=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',sho
w_shapes=True,show_layer_activations=True)
```

```
tf.keras.utils.plot_model(chatbot.decoder,to_file='decoder.png',sho
w_shapes=True,show_layer_activations=True)
```

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.'''
])
```

CODING:

```
import warnings
warnings.filterwarnings('ignore')
import numpy as np
```

CREATE A CHABOT USING PYTHON

```
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
import keras
from tqdm import tqdm
from keras.layers import Dense
import json
import re
import string
from sklearn.feature_extraction.text import TfidfVectorizer
import unicodedata
from sklearn.model_selection import train_test_split


question  =[]
answer = []
with open("../input/simple-dialogs-for-chatbot/dialogs.txt",'r') as f :
   for line in f :
      line  =  line.split('\t')
      question.append(line[0])
      answer.append(line[1])
print(len(question) == len(answer))


question[:5]
```

CREATE A CHABOT USING PYTHON

```
answer[:5]
answer = [ i.replace("\n","") for i in answer]

answer[:5]
data = pd.DataFrame({"question" : question ,"answer":answer})
data.head()

def unicode_to_ascii(s):
   return ''.join(c for c in unicodedata.normalize('NFD', s)
     if unicodedata.category(c) != 'Mn')

def clean_text(text):
   text = unicode_to_ascii(text.lower().strip())
   text = re.sub(r"i'm", "i am", text)
   text = re.sub(r"\r", "", text)
   text = re.sub(r"he's", "he is", text)
   text = re.sub(r"she's", "she is", text)
   text = re.sub(r"it's", "it is", text)
   text = re.sub(r"that's", "that is", text)
   text = re.sub(r"what's", "that is", text)
   text = re.sub(r"where's", "where is", text)
   text = re.sub(r"how's", "how is", text)
```

```
    text = re.sub(r"\'ll", " will", text)
    text = re.sub(r"\'ve", " have", text)
    text = re.sub(r"\'re", " are", text)
    text = re.sub(r"\'d", " would", text)
    text = re.sub(r"\'re", " are", text)
    text = re.sub(r"won't", "will not", text)
    text = re.sub(r"can't", "cannot", text)
    text = re.sub(r"n't", " not", text)
    text = re.sub(r"n'", "ng", text)
    text = re.sub(r"'bout", "about", text)
    text = re.sub(r"'til", "until", text)
    text = re.sub(r"[-()\"#/@;:<>{}`+=~|.!?,]", "", text)
    text = text.translate(str.maketrans('', '', string.punctuation))
    text = re.sub("(\\W)"," ",text)
    text = re.sub('\S*\d\S*\s*','', text)
    text =  "<sos> " +  text + " <eos>"
    return text


data["question"][0]
data["question"] = data.question.apply(clean_text)


data["question"][0]
```

```
data["answer"] = data.answer.apply(clean_text)

question  = data.question.values.tolist()
answer =  data.answer.values.tolist()

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

input_tensor , inp_lang  =  tokenize(question)


target_tensor , targ_lang  =  tokenize(answer)

def remove_tags(sentence):
```

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


max_length_targ, max_length_inp = target_tensor.shape[1],
input_tensor.shape[1]



# 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)



#print(len(train_inp) , len(val_inp) , len(train_target) ,
len(val_target))


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
```

CREATE A CHABOT USING PYTHON

```
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


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 input
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))


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, hidden_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))


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))

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([targ_lang.word_index['<sos>']] *
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 tqdm(range(1, EPOCHS + 1), desc='Epochs',
unit='epoch'):
    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))



def evaluate(sentence):
    sentence = clean_text(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['<sos>']], 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] == '<eos>':
            return remove_tags(result), remove_tags(sentence)

        # the predicted ID is fed back into the model
        dec_input = tf.expand_dims([predicted_id], 0)
```

CREATE A CHABOT USING PYTHON

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

questions  =[]
answers = []
with open("../input/simple-dialogs-for-chatbot/dialogs.txt",'r') as f :
    for line in f :
        line  =  line.split('\t')
        questions.append(line[0])
        answers.append(line[1])
print(len(question) == len(answer))

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

    print('Question: %s' % (sentence))
    print('Predicted answer: {}'.format(result))
ask(questions[100])

ask(questions[20])

print(answers[20])
```

```
ask(questions[10])

print(answers[10])
```