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

```
True
```

```
question[:5]
```

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

```
answer[:5]
```

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

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

```
answer[:5]
```

```
["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."]
```

```
data = pd.DataFrame({"question" : question ,"answer":answer})
data.head()
```

	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.

```
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]

     'hi, how are you doing?'


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


data["question"][0]

     '<sos> hi how are you doing <eos>'
```

```
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)


 #len(inp_question) ==  len(inp_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,


#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

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

     (TensorShape([64, 22]), TensorShape([64, 22]))


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))

         Encoder output shape: (batch size, sequence length, units) (64, 22, 1024)
         Encoder Hidden state shape: (batch size, units) (64, 1024)


    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))
```

```
Attention result shape: (batch size, units) (64, 1024)
Attention weights shape: (batch_size, sequence_length, 1) (64, 22, 1)
```

```
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))
```

```
Decoder output shape: (batch_size, vocab size) (64, 2347)
```

```
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))


 Epochs:   0%|          | 0/40 [00:00<?, ?epoch/s]
```

```
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)

    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])
```

Executing (48s) <cell line: 3> › error_handler() › __call__() › _call() › __call__() › _call_flat() › call() › quick_execute()