```
from google.colab import drive
drive.mount('/content/drive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3pfee649

Enter your authorization code:
··········
Mounted at /content/drive
```

```
# for google colab only
%tensorflow_version 2.x

import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing import sequence

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

### Reference article ###
### https://www.tensorflow.org/tutorials/text/nmt_with_attention
```

```
TensorFlow 2.x selected.
```

```
# this path is for google drive mounted in google colab. If you want to use file from local directory
# I request you to modify this path

file_path = "/content/drive/My Drive/Stevens/Fall 2019/NLP/fra.txt"
```

```
# Converts the unicode file to ascii
```

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

# data preprocessing
def data_preprocessing(w):
    w = unicode_to_ascii(w.lower().strip())

    # adding extra space before punctuation
    w = re.sub(r"([?.!,¿])", r" \1 ", w)
    w = re.sub(r'[" "]+', " ", w)

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

    w = w.rstrip().strip()

    # adding a start and end token
    w = '<start> ' + w + ' <end>'
    return w

# dataset creation
def getPreprocessedLanguagePairs(path, sample_size):
    with open(path, encoding='UTF-8') as f:
      lines = f.read().strip().split('\n')
      lang_pairs = [[data_preprocessing(w) for w in l.split('\t')]  for l in lines[:sample_size]]

    return zip(*lang_pairs)

def tokenize_corpus(corpus):
  tokenizer = Tokenizer(filters='')
  tokenizer.fit_on_texts(corpus)
  tokenized_text = tokenizer.texts_to_sequences(corpus)
  tokenized_text = sequence.pad_sequences(tokenized_text, padding='post')

  return tokenized_text, tokenizer

def getTokenizedPairs(path, sample_size=None):
  eng_set, fr_set,   = getPreprocessedLanguagePairs(path, sample_size)
```

```
  eng_tokenized_text, eng_set_tokenizer = tokenize_corpus(eng_set)
  fr_tokenized_text, fr_set_tokenizer = tokenize_corpus(fr_set)

  return eng_tokenized_text, fr_tokenized_text, eng_set_tokenizer, fr_set_tokenizer
```

```
# Google colab is crashing when using whole dataset, so taking small sample out of it
sample_size = 50000
eng_tokenized_text, fr_tokenized_text, eng_set_tokenizer, fr_set_tokenizer = getTokenizedPairs(file_path,
                                                                                               sample_size)

max_length_fr = max(len(text) for text in fr_tokenized_text)
max_length_eng = max(len(text) for text in eng_tokenized_text)
```

```
# Splitting set into train and test
X_train, X_test, y_train, y_test = train_test_split(eng_tokenized_text, fr_tokenized_text, test_size=0.2, random_st

# Show length
print(len(X_train), len(y_train), len(X_test), len(y_test))
```

```
40000 40000 10000 10000
```

```
# Hyperparameters
BUFFER_SIZE = len(X_train)
BATCH_SIZE = 64
steps_per_epoch = len(X_train)//BATCH_SIZE
embedding_dim = 100
units = 512
vocab_en_size = len(eng_set_tokenizer.word_index)+1
vocab_fr_size = len(fr_set_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)

input_batch, target_batch = next(iter(dataset))
```

```
# Encoder RNN Class
class Encoder(tf.keras.Model):
```

```
  def __init__(self, vocab_size, embedding_dim, encoder_units, batch_size):
    super(Encoder, self).__init__()
    self.batch_size = batch_size
    self.encoder_units = encoder_units
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(self.encoder_units,
                                   return_sequences=True,
                                   return_state=True)

  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_size, self.encoder_units))

# Attention Class
class AttentionLayer(tf.keras.layers.Layer):
  def __init__(self, units):
    super(AttentionLayer, 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):
    hidden_with_time_axis = tf.expand_dims(query, 1)

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

    # attention_weights.shape = (batch_size, max_length, 1)
    attention_weights = tf.nn.softmax(score, axis=1)

    # context_vector.shape = (batch_size, hidden_size)
    context_vector = attention_weights * values
    context_vector = tf.reduce_sum(context_vector, axis=1)

    return context_vector, attention_weights
```

```
# Decoder RNN class
class Decoder(tf.keras.Model):
  def __init__(self, vocab_size, embedding_dim, decoder_units, batch_size):
    super(Decoder, self).__init__()
    self.batch_size = batch_size
    self.decoder_units = decoder_units
    self.embedding = tf.keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = tf.keras.layers.GRU(self.decoder_units,
                                   return_sequences=True,
                                   return_state=True)
    self.fc = tf.keras.layers.Dense(vocab_size)

    # attention
    self.attention = AttentionLayer(self.decoder_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)

    # passing the concatenated vector to the LSTM
    output, state = self.gru(x)
    output = tf.reshape(output, (-1, output.shape[2]))
    x = self.fc(output)

    return x, state, attention_weights
```

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

# sample input
encoder_hidden = encoder.initialize_hidden_state()
encoder_output, encoder_hidden = encoder(input_batch, encoder_hidden)

attention_layer = AttentionLayer(10)
attention_result, attention_weights = attention_layer(encoder_hidden, encoder_output)
```

```
decoder = Decoder(vocab_fr_size, embedding_dim, units, BATCH_SIZE)

sample_decoder_output, _, _ = decoder(tf.random.uniform((64, 1)),
                                      encoder_hidden, encoder_output)
```

```
# Loss
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_)
```

```
# Model training
@tf.function
def training(en, fr, enc_hidden):
  loss = 0

  with tf.GradientTape() as tape:
    enc_output, enc_hidden = encoder(en, enc_hidden)
    dec_hidden = enc_hidden
    dec_input = tf.expand_dims([fr_set_tokenizer.word_index['<start>']] * BATCH_SIZE, 1)

    # taking target as input to next
    for t in range(1, fr.shape[1]):
      # passing enc_output to the decoder
      predictions, dec_hidden, _ = decoder(dec_input, dec_hidden, enc_output)
      loss += loss_function(fr[:, t], predictions)
      dec_input = tf.expand_dims(fr[:, t], 1)

  # calculate batch loss
  batch_loss = (loss / int(fr.shape[1]))
```

```
  variables = encoder.trainable_variables + decoder.trainable_variables
  gradients = tape.gradient(loss, variables)
  optimizer.apply_gradients(zip(gradients, variables))

  return batch_loss
```

```
EPOCHS = 201

for epoch in range(1, EPOCHS):
  # print("Epoch #%d"%epoch)
  enc_hidden = encoder.initialize_hidden_state()
  total_loss = 0

  for (batch, (en, fr)) in enumerate(dataset.take(steps_per_epoch)):
    batch_loss = training(en, fr, enc_hidden)
    total_loss += batch_loss

  # printing loss after every 20 epochs
  if(epoch%20==0):
    print('Epoch {} Loss {:.4f}'.format(epoch,
                                        total_loss / steps_per_epoch))
```

```
Epoch 20 Loss 0.0957
Epoch 40 Loss 0.0663
Epoch 60 Loss 0.0590
Epoch 80 Loss 0.0551
Epoch 100 Loss 0.0534
Epoch 120 Loss 0.0511
Epoch 140 Loss 0.0491
Epoch 160 Loss 0.0471
Epoch 180 Loss 0.0458
Epoch 200 Loss 0.0456
```

```
def translate(sentence):
    inputs = [eng_set_tokenizer.word_index[i] for i in sentence.split(' ')]
    inputs = tf.keras.preprocessing.sequence.pad_sequences([inputs],
                                                           maxlen=max_length_eng,
                                                           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([fr_set_tokenizer.word_index['<start>']], 0)

    for t in range(max_length_fr):
        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 += fr_set_tokenizer.index_word[predicted_id] + ' '

        if fr_set_tokenizer.index_word[predicted_id] == '<end>':
            return result

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

    return result
```

```
# train-test set split in raw sentence format
full_en_, full_fr_, _ = getPreprocessedLanguagePairs(file_path, sample_size)

X_train_en, X_test_en, y_train_fr, y_test_fr = train_test_split(list(full_en_), list(full_fr_), test_size=0.2, rand

len(y_test_fr), len(X_test_en)
```

```
(10000, 10000)
```

Translation Demo

```
for english_sent in X_test_en[:10]:
  french_translation = translate(english_sent)
  print('English Sentence: %s' % (english_sent))
  print('Predicted French translation: {}'.format(french_translation))
```

```
English Sentence: <start> add a little milk . <end>
Predicted French translation: ajoute un peu de lait . <end>
English Sentence: <start> i think it s funny . <end>
Predicted French translation: je pense que c est drole . <end>
English Sentence: <start> i liked the movie . <end>
Predicted French translation: j ai aime le film . <end>
English Sentence: <start> don t judge me . <end>
Predicted French translation: ne me jugez pas . <end>
English Sentence: <start> please come this way . <end>
Predicted French translation: par ici , je vous prie . <end>
English Sentence: <start> can you do it faster ? <end>
Predicted French translation: arrives tu a le faire plus vite ? <end>
English Sentence: <start> the airplane is ready . <end>
Predicted French translation: le vent est pret . <end>
English Sentence: <start> no one does that . <end>
Predicted French translation: personne ne le sait . <end>
English Sentence: <start> this is tasteless . <end>
Predicted French translation: c est absurde . <end>
English Sentence: <start> may i join you ? <end>
Predicted French translation: puis je me joindre a vous ? <end>
```

BLEU Calculation for test set

```
from nltk.translate.bleu_score import corpus_bleu
from nltk.translate import bleu_score
import nltk
nltk.download('punkt')

def remove_unnecessary_tokens(sentence):

    # remove <start> and <end> tokens
    w = re.sub(r'<start>', "", sentence)
```

```
    w = re.sub(r'<end>', "", w)

    # removing unnecessary characters
    w = re.sub(r"[^a-zA-Z]", " ", w)

    return w.strip()
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
```

```
references = []
hypotheses = []

for i in range(len(X_test_en)):
  french_translation = translate(X_test_en[i])

  #remove <start> and <end> tokens and other unnecessary characters
  fr_ref = nltk.word_tokenize(remove_unnecessary_tokens(y_test_fr[i]))
  fr_trans = nltk.word_tokenize(remove_unnecessary_tokens(french_translation))

  references.append([fr_ref])
  hypotheses.append(fr_trans)
```

```
references[:10]
```

```
[[['ajoutez', 'un', 'peu', 'de', 'lait']],
 [['je', 'pense', 'que', 'c', 'est', 'drole']],
 [['j', 'ai', 'aime', 'le', 'film']],
 [['ne', 'me', 'juge', 'pas']],
 [['par', 'ici', 's', 'il', 'vous', 'plait']],
 [['arrivez', 'vous', 'a', 'le', 'faire', 'plus', 'vite']],
 [['l', 'avion', 'est', 'pret']],
 [['personne', 'ne', 'fait', 'cela']],
 [['ca', 'n', 'a', 'aucun', 'gout']],
 [['puis', 'je', 'me', 'joindre', 'a', 'vous']]]
```

```
hypotheses[:10]
```

```
    [['ajoute', 'un', 'peu', 'de', 'lait'],
     ['je', 'pense', 'que', 'c', 'est', 'drole'],
     ['j', 'ai', 'aime', 'le', 'film'],
     ['ne', 'me', 'jugez', 'pas'],
     ['par', 'ici', 'je', 'vous', 'prie'],
     ['arrives', 'tu', 'a', 'le', 'faire', 'plus', 'vite'],
     ['le', 'vent', 'est', 'pret'],
     ['personne', 'ne', 'le', 'sait'],
     ['c', 'est', 'absurde'],
     ['puis', 'je', 'me', 'joindre', 'a', 'vous']]
```

```
1 corpus_bleu(references, hypotheses)
```

```
0.3106569217107635
```

```
1
```