Transformer model for language understanding

Modified from:

- https://www.tensorflow.org/tutorials/tensorflow_text/subwords_tokenizer
- https://www.tensorflow.org/tutorials/text/transformer

which are:

- Copyright 2019 The TensorFlow Authors.
- Licensed under the Apache License, Version 2.0 (the "License");

Introduction

This tutorial trains a Transformer model to translate a Portuguese to English dataset. This is an advanced example that assumes knowledge of text generation and attention.

The core idea behind the Transformer model is *self-attention*—the ability to attend to different positions of the input sequence to compute a representation of that sequence. Transformer creates stacks of self-attention layers and is explained below in the sections *Scaled dot product attention* and *Multi-head attention*.

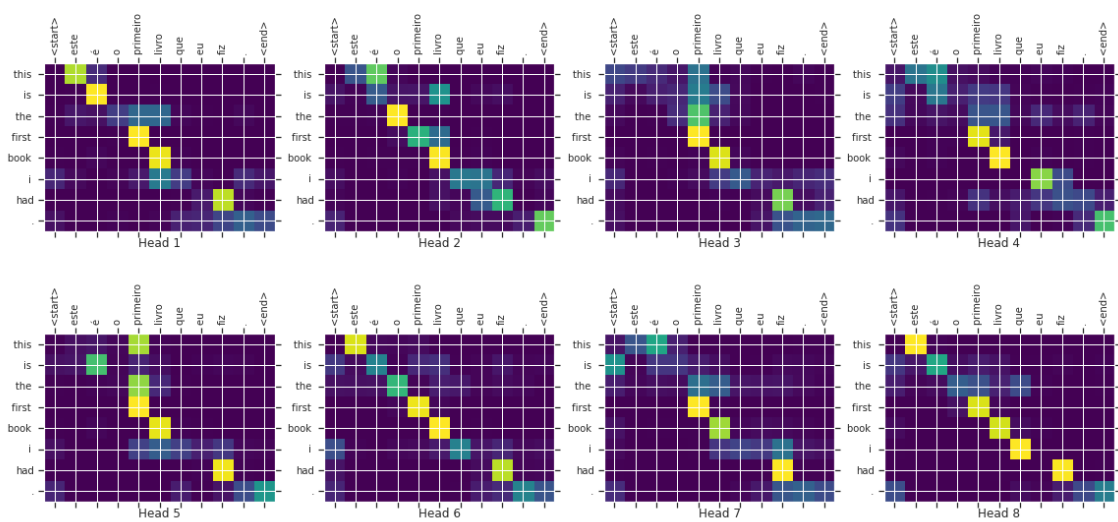
A transformer model handles variable-sized input using stacks of self-attention layers instead of RNNs or CNNs. This general architecture has a number of advantages:

- It makes no assumptions about the temporal/spatial relationships across the data. This is ideal for processing a set of objects (for example, StarCraft units).
- Layer outputs can be calculated in parallel, instead of a series like an RNN.
- Distant items can affect each other's output without passing through many RNN-steps, or convolution layers (see Scene Memory Transformer for example).
- It can learn long-range dependencies. This is a challenge in many sequence tasks.

The downsides of this architecture are:

- For a time-series, the output for a time-step is calculated from the *entire history* instead of only the inputs and current hidden-state. This *may* be less efficient.
- If the input *does* have a temporal/spatial relationship, like text, some positional encoding must be added or the model will effectively see a bag of words.

After training the model in this notebook, you will be able to input a Portuguese sentence and return the English translation.



Setup

```
from google.colab import drive # Load the Drive helper and mount
drive.mount('/content/drive') # This will prompt for authorization
```

```
    Mounted at /content/drive
```

```
BASE_MODEL_PATH = 'drive/MyDrive/models/transformer_port_eng_v2'
BASE_DATA_PATH = 'drive/MyDrive/data/transformer_port_eng_v2'
CONVERTER_MODEL_NAME = 'ted_hrlr_translate_pt_en_converter'
build_tokenizer = False
```

```
#Cleanup for a fresh start
#!rm -rf drive/MyDrive/data/downloads/*
#!rm -Rf $BASE_MODEL_PATH/*
#!rm -Rf $BASE_DATA_PATH/*
```

```
import os.path
from os import path
```

```
if not path.exists(BASE_MODEL_PATH):
  os.mkdir(BASE_MODEL_PATH)
  print(f'Created model directory {base_model_dir}')

if not path.exists(BASE_DATA_PATH):
  os.mkdir(BASE_DATA_PATH)
  print(f'Created model directory {base_data_dir}')
```

```
print(BASE_MODEL_PATH)
!ls $BASE_MODEL_PATH
print(BASE_DATA_PATH)
!ls $BASE_DATA_PATH
```

```
    drive/MyDrive/models/transformer_port_eng_v2
    ted_hrlr_translate_pt_en_converter
    drive/MyDrive/data/transformer_port_eng_v2
    downloads  en_vocab.txt  pt_vocab.txt  ted_hrlr_translate
```

```
import collections
import logging
import os
import pathlib
import re
import string
import sys
import time

import numpy as np
import matplotlib.pyplot as plt

from nltk.translate.bleu_score import sentence_bleu
```

```
if not path.exists(BASE_MODEL_PATH + '/' + CONVERTER_MODEL_NAME):
  build_tokenizer = True
  print(f"Building tokenizer since not found at {BASE_MODEL_PATH}/{CONVERTER_MODEL_NAME}")

  # `BertTokenizer.detokenize` is not in `tf-text` stable yet (currently 2.4.3).
  os.system("pip install -q tensorflow_text_nightly > /tmp/out.txt")

  # tf-text-nightly resquires tf-nightly
  os.system("pip install -q tf-nightly >> /tmp/out.txt")

  # Generate the vocabulary
  os.system("cat /tmp/out.txt")
else:
  build_tokenizer = False
  print(f"Tokenizer already exists at {BASE_MODEL_PATH}/{CONVERTER_MODEL_NAME}")
  os.system("pip install -q tensorflow_text > /tmp/out.txt")
  os.system("cat /tmp/out.txt")
```

```
    Tokenizer already exists at drive/MyDrive/models/transformer_port_eng_v2/ted_hrlr_translate_pt_en_converter
```

```
import tensorflow as tf
```

```
import tensorflow_datasets as tfds

import tensorflow_text as text

if build_tokenizer:
  # Generate the vocabulary
  from tensorflow_text.tools.wordpiece_vocab import bert_vocab_from_dataset as bert_vocab


logging.getLogger('tensorflow').setLevel(logging.ERROR)  # suppress warnings
```

Download the Dataset

Use [TensorFlow datasets](#) to load the [Portuguese-English translation dataset](#) from the [TED Talks Open Translation Project](#).

This dataset contains approximately 50000 training examples, 1100 validation examples, and 2000 test examples.

```
examples, metadata = tfds.load('ted_hrlr_translate/pt_to_en', with_info=True,
                               as_supervised=True, data_dir=BASE_DATA_PATH)
train_examples, val_examples, test_examples = examples['train'], examples['validation'], examples['test']
```

The `tf.data.Dataset` object returned by TensorFlow datasets yields pairs of text examples:

```
for pt_examples, en_examples in train_examples.batch(3).take(1):
  for pt in pt_examples.numpy():
    print(pt.decode('utf-8'))

  print()

  for en in en_examples.numpy():
    print(en.decode('utf-8'))
```

```
e quando melhoramos a procura , tiramos a única vantagem da impressão , que é a serendipidade .
mas e se estes fatores fossem ativos ?
mas eles não tinham a curiosidade de me testar .

and when you improve searchability , you actually take away the one advantage of print , which is serendipity .
but what if it were active ?
but they did n't test for curiosity .
```

Text tokenization & detokenization

For more detail on how to build the tokenizer and detokenizer see: https://www.tensorflow.org/tutorials/tensorflow_text/subwords_tokenizer

You can't train a model directly on text. The text needs to be converted some numeric representation first. Typically you convert the text to sequences of token IDs, which are as indexes into an embedding.

START = tf.argmax(tf.constant(reserved_tokens) == "[START]") END = tf.argmax(tf.constant(reserved_tokens) == "[END]")

def add_start_end(ragged): count = ragged.bounding_shape()[0] starts = tf.fill([count,1], START) ends = tf.fill([count,1], END) return tf.concat([starts, ragged, ends], axis=1)One popular implementation is demonstrated in the [Subword tokenizer tutorial](#) builds subword tokenizers (`text.BertTokenizer`) optimized for this dataset and exports them in a [saved_model](#).

Download and unzip and import the `saved_model`:

```
reserved_tokens=["[PAD]", "[UNK]", "[START]", "[END]"]

START = tf.argmax(tf.constant(reserved_tokens) == "[START]")
END = tf.argmax(tf.constant(reserved_tokens) == "[END]")

def add_start_end(ragged):
  global START, END
  count = ragged.bounding_shape()[0]
  starts = tf.fill([count,1], START)
  ends = tf.fill([count,1], END)
  return tf.concat([starts, ragged, ends], axis=1)
```

```
  return tf.concat([starts, ragged, ends], axis=1)

def write_vocab_file(filepath, vocab):
  global base_path
  with open(BASE_DATA_PATH + '/' + filepath, 'w') as f:
    for token in vocab:
      print(token, file=f)


def write_vocab_files():
  global reserved_tokens
  bert_tokenizer_params=dict(lower_case=True)

  bert_vocab_args = dict(
      # The target vocabulary size
      vocab_size = 8300,
      # Reserved tokens that must be included in the vocabulary
      reserved_tokens=reserved_tokens,
      # Arguments for `text.BertTokenizer`
      bert_tokenizer_params=bert_tokenizer_params,
      # Arguments for `wordpiece_vocab.wordpiece_tokenizer_learner_lib.learn`
      learn_params={},
  )

  train_en = train_examples.map(lambda pt, en: en)
  train_pt = train_examples.map(lambda pt, en: pt)

  %%time
  pt_vocab = bert_vocab.bert_vocab_from_dataset(
      train_pt.batch(1000).prefetch(2),
      **bert_vocab_args
  )

  write_vocab_file('pt_vocab.txt', pt_vocab)

  %%time
  en_vocab = bert_vocab.bert_vocab_from_dataset(
      train_en.batch(1000).prefetch(2),
      **bert_vocab_args
  )

  write_vocab_file('en_vocab.txt', en_vocab)

def cleanup_text(reserved_tokens, token_txt):
  # Generate clean text output, so drop reserved tokens like [START], [END] and [PAD].
  # apply a string join along the words axis of the result.

  # Drop the reserved tokens, except for "[UNK]".
  bad_tokens = [re.escape(tok) for tok in reserved_tokens if tok != "[UNK]"]
  bad_token_re = "|".join(bad_tokens)

  bad_cells = tf.strings.regex_full_match(token_txt, bad_token_re)
  result = tf.ragged.boolean_mask(token_txt, ~bad_cells)

  # Join them into strings.
  result = tf.strings.reduce_join(result, separator=' ', axis=-1)

  return result

def build_bert_tokenizer():
  pt_tokenizer = text.BertTokenizer(BASE_DATA_PATH + '/' + 'pt_vocab.txt', **bert_tokenizer_params)
  en_tokenizer = text.BertTokenizer(BASE_DATA_PATH + '/' + 'en_vocab.txt', **bert_tokenizer_params)

  # Run it through the BertTokenizer.tokenize method. Initially, this returns a
  # tf.RaggedTensor with axes (batch, word, word-piece):*italicized text*

  # Custom detokenization
  # Tokenize the examples -> (batch, word, word-piece)
  # Merge the word and word-piece axes -> (batch, tokens)
  token_batch = en_tokenizer.tokenize(en_examples).merge_dims(-2,-1)
  words = en_tokenizer.detokenize(token_batch)
```

Export Custom BERT Tokenizer

The following code block builds a CustomTokenizer class to contain the text.BertTokenizer instances, the custom logic, and the @tf.function wrappers required for export.

```
class CustomTokenizer(tf.Module):
  def __init__(self, reserved_tokens, vocab_path):
    self.tokenizer = text.BertTokenizer(vocab_path, lower_case=True)
    self._reserved_tokens = reserved_tokens
    self._vocab_path = tf.saved_model.Asset(vocab_path)

    vocab = pathlib.Path(vocab_path).read_text().splitlines()
    self.vocab = tf.Variable(vocab)

    ## Create the signatures for export:

    # Include a tokenize signature for a batch of strings.
    self.tokenize.get_concrete_function(
        tf.TensorSpec(shape=[None], dtype=tf.string))

    # Include `detokenize` and `lookup` signatures for:
    #   * `Tensors` with shapes [tokens] and [batch, tokens]
    #   * `RaggedTensors` with shape [batch, tokens]
    self.detokenize.get_concrete_function(
        tf.TensorSpec(shape=[None, None], dtype=tf.int64))
    self.detokenize.get_concrete_function(
          tf.RaggedTensorSpec(shape=[None, None], dtype=tf.int64))

    self.lookup.get_concrete_function(
        tf.TensorSpec(shape=[None, None], dtype=tf.int64))
    self.lookup.get_concrete_function(
          tf.RaggedTensorSpec(shape=[None, None], dtype=tf.int64))

    # These `get_*` methods take no arguments
    self.get_vocab_size.get_concrete_function()
    self.get_vocab_path.get_concrete_function()
    self.get_reserved_tokens.get_concrete_function()

  @tf.function
  def tokenize(self, strings):
    enc = self.tokenizer.tokenize(strings)
    # Merge the `word` and `word-piece` axes.
    enc = enc.merge_dims(-2,-1)
    enc = add_start_end(enc)
    return enc

  @tf.function
  def detokenize(self, tokenized):
    words = self.tokenizer.detokenize(tokenized)
    return cleanup_text(self._reserved_tokens, words)

  @tf.function
  def lookup(self, token_ids):
    return tf.gather(self.vocab, token_ids)

  @tf.function
  def get_vocab_size(self):
    return tf.shape(self.vocab)[0]

  @tf.function
  def get_vocab_path(self):
    return self._vocab_path

  @tf.function
  def get_reserved_tokens(self):
    return tf.constant(self._reserved_tokens)
```

Build a CustomTokenizer for each language:

```
def build_custom_tokenizers():
  tokenizers = tf.Module()
  tokenizers.pt = CustomTokenizer(reserved_tokens, BASE_DATA_PATH + '/' + 'pt_vocab.txt')
```

```
  tokenizers.en = CustomTokenizer(reserved_tokens, BASE_DATA_PATH + '/' + 'en_vocab.txt')

  # Export the tokenizers as a saved_model:
  tf.saved_model.save(tokenizers, BASE_MODEL_PATH + '/' + CONVERTER_MODEL_NAME)

  # Generate the vocabulary
  os.system("zip -r $BASE_MODEL_PATH/{BASE_MODEL_PATH}.zip $BASE_MODEL_PATH/{CONVERTER_MODEL_NAME} > /tmp/out.txt")
  os.system("cat /tmp/out.txt")

  return tokenizers


if build_tokenizer:
  global BASE_MODEL_PATH, CONVERTER_MODEL_NAME
  print("writing vocab files")
  write_vocab_files()
  print("building custom BERT tokenizer")
  tokenizers = build_custom_tokenizers()
else:
  tokenizers = reloaded_tokenizers = tf.saved_model.load(BASE_MODEL_PATH + '/' + CONVERTER_MODEL_NAME)
  reloaded_tokenizers.en.get_vocab_size().numpy()
  print(f"Loaded custom BERT tokenizer from {BASE_MODEL_PATH}/{CONVERTER_MODEL_NAME}")
```

```
Loaded custom BERT tokenizer from drive/MyDrive/models/transformer_port_eng_v2/ted_hrlr_translate_pt_en_converter
```

```
!ls -l $BASE_MODEL_PATH
```

```
total 4
drwx------ 4 root root 4096 Mar 26 21:36 ted_hrlr_translate_pt_en_converter
```

The `tf.saved_model` contains two text tokenizers, one for English and one for Portugese. Both have the same methods:

```
[item for item in dir(tokenizers.en) if not item.startswith('_')]
```

```
['detokenize',
 'get_reserved_tokens',
 'get_vocab_path',
 'get_vocab_size',
 'lookup',
 'tokenize',
 'tokenizer',
 'vocab']
```

The `tokenize` method converts a batch of strings to a padded-batch of token IDs. This method splits punctuation, lowercases and unicode-normalizes the input before tokenizing. That standardization is not visible here because the input data is already standardized.

```
for en in en_examples.numpy():
  print(en.decode('utf-8'))
```

```
and when you improve searchability , you actually take away the one advantage of print , which is serendipity .
but what if it were active ?
but they did n't test for curiosity .
```

```
encoded = tokenizers.en.tokenize(en_examples)

for row in encoded.to_list():
  print(row)
```

```
[2, 72, 117, 79, 1259, 1491, 2362, 13, 79, 150, 184, 311, 71, 103, 2308, 74, 2679, 13, 148, 80, 55, 4840, 1434, 2423,
[2, 87, 90, 107, 76, 129, 1852, 30, 3]
[2, 87, 83, 149, 50, 9, 56, 664, 85, 2512, 15, 3]
```

The `detokenize` method attempts to convert these token IDs back to human readable text:

```
round_trip = tokenizers.en.detokenize(encoded)
for line in round_trip.numpy():
  print(line.decode('utf-8'))
```

```
and when you improve searchability , you actually take away the one advantage of print , which is serendipity .
but what if it were active ?
```

```
but they did n ' t test for curiosity .
```

The lower level `lookup` method converts from token-IDs to token text:

```
tokens = tokenizers.en.lookup(encoded)
tokens
```

```
<tf.RaggedTensor [[b'[START]', b'and', b'when', b'you', b'improve', b'search', b'##ability', b',', b'you', b'actually'
```

Here you can see the "subword" aspect of the tokenizers. The word "searchability" is decomposed into "search ##ability" and the word "serindipity" into "s ##ere ##nd ##ip ##ity"

```
if build_tokenizer:
  print("Recommend factory reset notebook and rerunning since training runs much slower with nightly build code")
```

Setup input pipeline

To build an input pipeline suitable for training you'll apply some transformations to the dataset.

This function will be used to encode the batches of raw text:

```
def tokenize_pairs(pt, en):
    pt = tokenizers.pt.tokenize(pt)
    # Convert from ragged to dense, padding with zeros.
    pt = pt.to_tensor()

    en = tokenizers.en.tokenize(en)
    # Convert from ragged to dense, padding with zeros.
    en = en.to_tensor()
    return pt, en
```

Here's a simple input pipeline that processes, shuffles and batches the data:

```
BUFFER_SIZE = 20000
BATCH_SIZE = 64
```

```
def make_batches(ds):
  return (
      ds
      .cache()
      .shuffle(BUFFER_SIZE)
      .batch(BATCH_SIZE, drop_remainder=True)
      .map(tokenize_pairs, num_parallel_calls=tf.data.AUTOTUNE)
      .prefetch(tf.data.AUTOTUNE))

train_batches = make_batches(train_examples)
val_batches = make_batches(val_examples)
```

Positional encoding

Since this model doesn't contain any recurrence or convolution, positional encoding is added to give the model some information about the relative position of the words in the sentence.

The positional encoding vector is added to the embedding vector. Embeddings represent a token in a d-dimensional space where tokens with similar meaning will be closer to each other. But the embeddings do not encode the relative position of words in a sentence. So after adding the positional encoding, words will be closer to each other based on the *similarity of their meaning and their position in the sentence*, in the d-dimensional space.

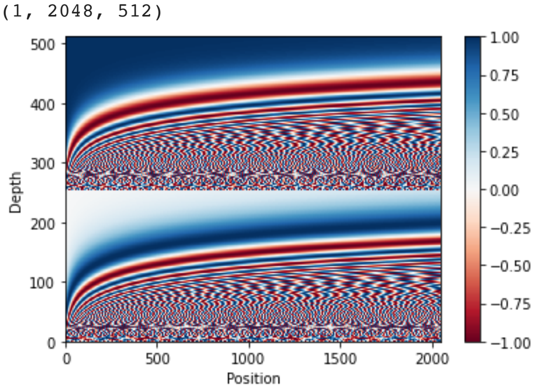
See the notebook on [positional encoding] to learn more about it. The formula for calculating the positional encoding is as follows:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

```
def get_angles(pos, i, d_model):
  angle_rates = 1 / np.power(10000, (2 * (i//2)) / np.float32(d_model))
  return pos * angle_rates


def positional_encoding(position, d_model):
  angle_rads = get_angles(np.arange(position)[:, np.newaxis],
                          np.arange(d_model)[np.newaxis, :],
                          d_model)

  # apply sin to even indices in the array; 2i
  angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])

  # apply cos to odd indices in the array; 2i+1
  angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])

  pos_encoding = angle_rads[np.newaxis, ...]

  return tf.cast(pos_encoding, dtype=tf.float32)


n, d = 2048, 512
pos_encoding = positional_encoding(n, d)
print(pos_encoding.shape)
pos_encoding = pos_encoding[0]

# Juggle the dimensions for the plot
pos_encoding = tf.reshape(pos_encoding, (n, d//2, 2))
pos_encoding = tf.transpose(pos_encoding, (2,1,0))
pos_encoding = tf.reshape(pos_encoding, (d, n))

plt.pcolormesh(pos_encoding, cmap='RdBu')
plt.ylabel('Depth')
plt.xlabel('Position')
plt.colorbar()
plt.show()
```

```
(1, 2048, 512)
```




```
pos_encoding.shape
```

```
TensorShape([512, 2048])
```

Masking

Mask all the pad tokens in the batch of sequence. It ensures that the model does not treat padding as the input. The mask indicates where pad value `0` is present: it outputs a `1` at those locations, and a `0` otherwise.

```
def create_padding_mask(seq):
  seq = tf.cast(tf.math.equal(seq, 0), tf.float32)

  # add extra dimensions to add the padding
  # to the attention logits.
  return seq[:, tf.newaxis, tf.newaxis, :]  # (batch_size, 1, 1, seq_len)
```

```
x = tf.constant([[7, 6, 0, 0, 1], [1, 2, 3, 0, 0], [0, 0, 0, 4, 5]])
create_padding_mask(x)
```

```
<tf.Tensor: shape=(3, 1, 1, 5), dtype=float32, numpy=
array([[[[0., 0., 1., 1., 0.]]],


       [[[0., 0., 0., 1., 1.]]],


       [[[1., 1., 1., 0., 0.]]]], dtype=float32)>
```

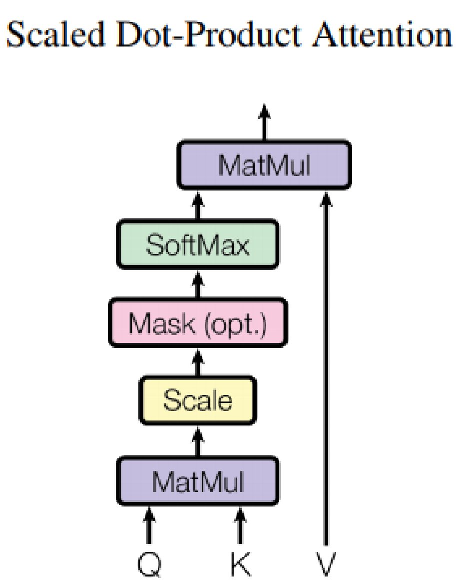
The look-ahead mask is used to mask the future tokens in a sequence. In other words, the mask indicates which entries should not be used.

This means that to predict the third word, only the first and second word will be used. Similarly to predict the fourth word, only the first, second and the third word will be used and so on.

```
def create_look_ahead_mask(size):
  mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)
  return mask  # (seq_len, seq_len)


x = tf.random.uniform((1, 3))
temp = create_look_ahead_mask(x.shape[1])
temp
```

```
<tf.Tensor: shape=(3, 3), dtype=float32, numpy=
array([[0., 1., 1.],
       [0., 0., 1.],
       [0., 0., 0.]], dtype=float32)>
```

Scaled dot product attention



The attention function used by the transformer takes three inputs: Q (query), K (key), V (value). The equation used to calculate the attention weights is:

$$Attention(Q, K, V) = softmax_k(\frac{QK^T}{\sqrt{d_k}})V$$

The dot-product attention is scaled by a factor of square root of the depth. This is done because for large values of depth, the dot product grows large in magnitude pushing the softmax function where it has small gradients resulting in a very hard softmax.

For example, consider that `Q` and `K` have a mean of 0 and variance of 1. Their matrix multiplication will have a mean of 0 and variance of `dk`. So the *square root of `dk`* is used for scaling so you get a consistent variance regardless of the value of `dk`. If the variance is too low the output may be too flat to optimize effectively. If the variance is too high the softmax may saturate at initilization making it dificult to learn.

The mask is multiplied with -1e9 (close to negative infinity). This is done because the mask is summed with the scaled matrix multiplication of Q and K and is applied immediately before a softmax. The goal is to zero out these cells, and large negative inputs to softmax are near zero in

```
def scaled_dot_product_attention(q, k, v, mask):
  """Calculate the attention weights.
  q, k, v must have matching leading dimensions.
  k, v must have matching penultimate dimension, i.e.: seq_len_k = seq_len_v.
  The mask has different shapes depending on its type(padding or look ahead)
  but it must be broadcastable for addition.

  Args:
    q: query shape == (..., seq_len_q, depth)
    k: key shape == (..., seq_len_k, depth)
    v: value shape == (..., seq_len_v, depth_v)
    mask: Float tensor with shape broadcastable
          to (..., seq_len_q, seq_len_k). Defaults to None.

  Returns:
    output, attention_weights
  """

  matmul_qk = tf.matmul(q, k, transpose_b=True)  # (..., seq_len_q, seq_len_k)

  # scale matmul_qk
  dk = tf.cast(tf.shape(k)[-1], tf.float32)
  scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)

  # add the mask to the scaled tensor.
  if mask is not None:
    scaled_attention_logits += (mask * -1e9)

  # softmax is normalized on the last axis (seq_len_k) so that the scores
  # add up to 1.
  attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)  # (..., seq_len_q, seq_len_k)

  output = tf.matmul(attention_weights, v)  # (..., seq_len_q, depth_v)

  return output, attention_weights
```

As the softmax normalization is done on K, its values decide the amount of importance given to Q.

The output represents the multiplication of the attention weights and the V (value) vector. This ensures that the words you want to focus on are kept as-is and the irrelevant words are flushed out.

```
def print_out(q, k, v):
  temp_out, temp_attn = scaled_dot_product_attention(
      q, k, v, None)
  print ('Attention weights are:')
  print (temp_attn)
  print ('Output is:')
  print (temp_out)
```

```
np.set_printoptions(suppress=True)

temp_k = tf.constant([[10,0,0],
                      [0,10,0],
                      [0,0,10],
                      [0,0,10]], dtype=tf.float32)  # (4, 3)

temp_v = tf.constant([[   1,0],
                      [  10,0],
                      [ 100,5],
                      [1000,6]], dtype=tf.float32)  # (4, 2)
```

```
# This `query` aligns with the second `key`,
# so the second `value` is returned.
temp_q = tf.constant([[0, 10, 0]], dtype=tf.float32)  # (1, 3)
print_out(temp_q, temp_k, temp_v)
```

```
    Attention weights are:
    tf.Tensor([[0. 1. 0. 0.]], shape=(1, 4), dtype=float32)
    Output is:
    tf.Tensor([[10.  0.]], shape=(1, 2), dtype=float32)
```

```
# This query aligns with a repeated key (third and fourth),
# so all associated values get averaged.
temp_q = tf.constant([[0, 0, 10]], dtype=tf.float32)  # (1, 3)
print_out(temp_q, temp_k, temp_v)
```

```
    Attention weights are:
    tf.Tensor([[0.  0.  0.5 0.5]], shape=(1, 4), dtype=float32)
    Output is:
    tf.Tensor([[550.    5.5]], shape=(1, 2), dtype=float32)
```

```
# This query aligns equally with the first and second key,
# so their values get averaged.
temp_q = tf.constant([[10, 10, 0]], dtype=tf.float32)  # (1, 3)
print_out(temp_q, temp_k, temp_v)
```

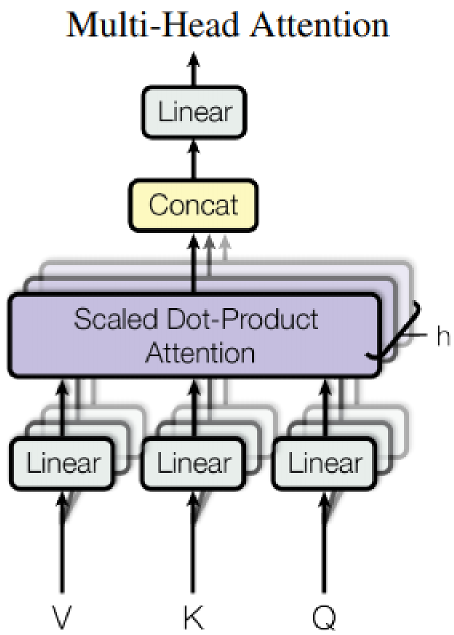
```
    Attention weights are:
    tf.Tensor([[0.5 0.5 0.  0. ]], shape=(1, 4), dtype=float32)
    Output is:
    tf.Tensor([[5.5 0. ]], shape=(1, 2), dtype=float32)
```

Pass all the queries together.

```
temp_q = tf.constant([[0, 0, 10], [0, 10, 0], [10, 10, 0]], dtype=tf.float32)  # (3, 3)
print_out(temp_q, temp_k, temp_v)
```

```
    Attention weights are:
    tf.Tensor(
    [[0.  0.  0.5 0.5]
     [0.  1.  0.  0. ]
     [0.5 0.5 0.  0. ]], shape=(3, 4), dtype=float32)
    Output is:
    tf.Tensor(
    [[550.    5.5]
     [ 10.    0. ]
     [  5.5   0. ]], shape=(3, 2), dtype=float32)
```

Multi-head attention



Multi-head attention consists of four parts:

- Linear layers and split into heads.
- Scaled dot-product attention.
- Concatenation of heads.
- Final linear layer.

Each multi-head attention block gets three inputs; Q (query), K (key), V (value). These are put through linear (Dense) layers and split up into multiple heads.

The `scaled_dot_product_attention` defined above is applied to each head (broadcasted for efficiency). An appropriate mask must be used in the attention step. The attention output for each head is then concatenated (using `tf.transpose`, and `tf.reshape`) and put through a final `Dense` layer.

Instead of one single attention head, Q, K, and V are split into multiple heads because it allows the model to jointly attend to information at different positions from different representational spaces. After the split each head has a reduced dimensionality, so the total computation cost is the same as a single head attention with full dimensionality.

```
class MultiHeadAttention(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads):
    super(MultiHeadAttention, self).__init__()
    self.num_heads = num_heads
    self.d_model = d_model

    assert d_model % self.num_heads == 0

    self.depth = d_model // self.num_heads

    self.wq = tf.keras.layers.Dense(d_model)
    self.wk = tf.keras.layers.Dense(d_model)
    self.wv = tf.keras.layers.Dense(d_model)

    self.dense = tf.keras.layers.Dense(d_model)

  def split_heads(self, x, batch_size):
    """Split the last dimension into (num_heads, depth).
    Transpose the result such that the shape is (batch_size, num_heads, seq_len, depth)
    """
    x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
    return tf.transpose(x, perm=[0, 2, 1, 3])

  def call(self, v, k, q, mask):
```

```
  def call(self, v, k, q, mask):
    batch_size = tf.shape(q)[0]

    q = self.wq(q)  # (batch_size, seq_len, d_model)
    k = self.wk(k)  # (batch_size, seq_len, d_model)
    v = self.wv(v)  # (batch_size, seq_len, d_model)

    q = self.split_heads(q, batch_size)  # (batch_size, num_heads, seq_len_q, depth)
    k = self.split_heads(k, batch_size)  # (batch_size, num_heads, seq_len_k, depth)
    v = self.split_heads(v, batch_size)  # (batch_size, num_heads, seq_len_v, depth)

    # scaled_attention.shape == (batch_size, num_heads, seq_len_q, depth)
    # attention_weights.shape == (batch_size, num_heads, seq_len_q, seq_len_k)
    scaled_attention, attention_weights = scaled_dot_product_attention(
        q, k, v, mask)

    scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])  # (batch_size, seq_len_q, num_heads, depth)

    concat_attention = tf.reshape(scaled_attention,
                                  (batch_size, -1, self.d_model))  # (batch_size, seq_len_q, d_model)

    output = self.dense(concat_attention)  # (batch_size, seq_len_q, d_model)

    return output, attention_weights
```

Create a `MultiHeadAttention` layer to try out. At each location in the sequence, `y`, the `MultiHeadAttention` runs all 8 attention heads across all other locations in the sequence, returning a new vector of the same length at each location.

```
temp_mha = MultiHeadAttention(d_model=512, num_heads=8)
y = tf.random.uniform((1, 60, 512))  # (batch_size, encoder_sequence, d_model)
out, attn = temp_mha(y, k=y, q=y, mask=None)
out.shape, attn.shape
```

```
(TensorShape([1, 60, 512]), TensorShape([1, 8, 60, 60]))
```

Point wise feed forward network

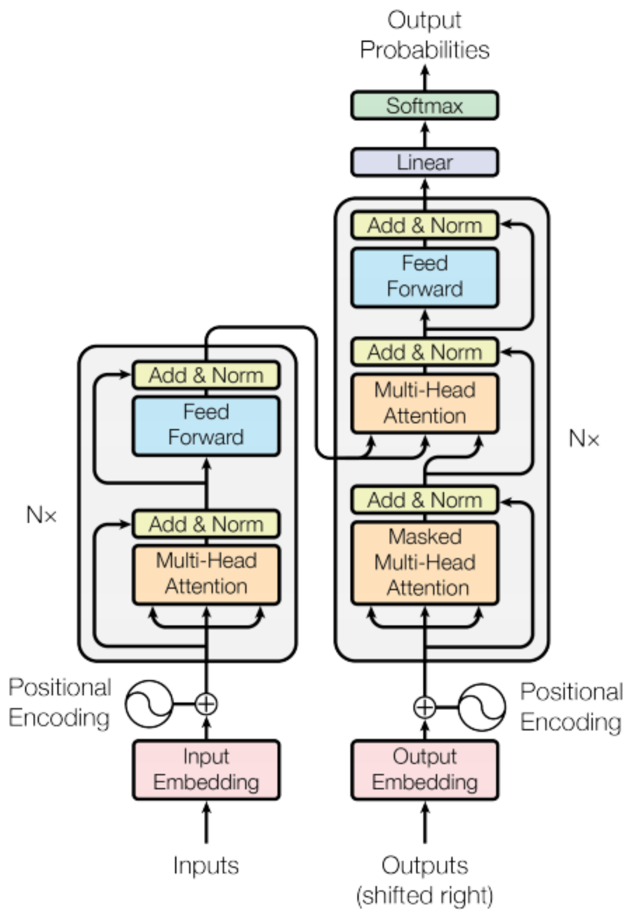
Point wise feed forward network consists of two fully-connected layers with a ReLU activation in between.

```
def point_wise_feed_forward_network(d_model, dff):
  return tf.keras.Sequential([
      tf.keras.layers.Dense(dff, activation='relu'),  # (batch_size, seq_len, dff)
      tf.keras.layers.Dense(d_model)  # (batch_size, seq_len, d_model)
  ])
```

```
sample_ffn = point_wise_feed_forward_network(512, 2048)
sample_ffn(tf.random.uniform((64, 50, 512))).shape
```

```
TensorShape([64, 50, 512])
```

Encoder and decoder



The transformer model follows the same general pattern as a standard sequence to sequence with attention model.

- The input sentence is passed through `N` encoder layers that generates an output for each word/token in the sequence.
- The decoder attends on the encoder's output and its own input (self-attention) to predict the next word.

Encoder layer

Each encoder layer consists of sublayers:

1. Multi-head attention (with padding mask)
2. Point wise feed forward networks.

Each of these sublayers has a residual connection around it followed by a layer normalization. Residual connections help in avoiding the vanishing gradient problem in deep networks.

The output of each sublayer is `LayerNorm(x + Sublayer(x))`. The normalization is done on the `d_model` (last) axis. There are N encoder layers in the transformer.

```
class EncoderLayer(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads, dff, rate=0.1):
    super(EncoderLayer, self).__init__()

    self.mha = MultiHeadAttention(d_model, num_heads)
    self.ffn = point_wise_feed_forward_network(d_model, dff)

    self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

    self.dropout1 = tf.keras.layers.Dropout(rate)
    self.dropout2 = tf.keras.layers.Dropout(rate)
```

```
  def call(self, x, training, mask):

    attn_output, _ = self.mha(x, x, x, mask)  # (batch_size, input_seq_len, d_model)
    attn_output = self.dropout1(attn_output, training=training)
    out1 = self.layernorm1(x + attn_output)  # (batch_size, input_seq_len, d_model)

    ffn_output = self.ffn(out1)  # (batch_size, input_seq_len, d_model)
    ffn_output = self.dropout2(ffn_output, training=training)
    out2 = self.layernorm2(out1 + ffn_output)  # (batch_size, input_seq_len, d_model)

    return out2


sample_encoder_layer = EncoderLayer(512, 8, 2048)

sample_encoder_layer_output = sample_encoder_layer(
    tf.random.uniform((64, 43, 512)), False, None)

sample_encoder_layer_output.shape  # (batch_size, input_seq_len, d_model)
```

```
TensorShape([64, 43, 512])
```

Decoder layer

Each decoder layer consists of sublayers:

1. Masked multi-head attention (with look ahead mask and padding mask)
2. Multi-head attention (with padding mask). V (value) and K (key) receive the *encoder output* as inputs. Q (query) receives the *output from the masked multi-head attention sublayer.*
3. Point wise feed forward networks

Each of these sublayers has a residual connection around it followed by a layer normalization. The output of each sublayer is `LayerNorm(x + Sublayer(x))`. The normalization is done on the `d_model` (last) axis.

There are N decoder layers in the transformer.

As Q receives the output from decoder's first attention block, and K receives the encoder output, the attention weights represent the importance given to the decoder's input based on the encoder's output. In other words, the decoder predicts the next word by looking at the encoder output and self-attending to its own output. See the demonstration above in the scaled dot product attention section.

```
class DecoderLayer(tf.keras.layers.Layer):
  def __init__(self, d_model, num_heads, dff, rate=0.1):
    super(DecoderLayer, self).__init__()

    self.mha1 = MultiHeadAttention(d_model, num_heads)
    self.mha2 = MultiHeadAttention(d_model, num_heads)

    self.ffn = point_wise_feed_forward_network(d_model, dff)

    self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
    self.layernorm3 = tf.keras.layers.LayerNormalization(epsilon=1e-6)

    self.dropout1 = tf.keras.layers.Dropout(rate)
    self.dropout2 = tf.keras.layers.Dropout(rate)
    self.dropout3 = tf.keras.layers.Dropout(rate)


  def call(self, x, enc_output, training,
           look_ahead_mask, padding_mask):
    # enc_output.shape == (batch_size, input_seq_len, d_model)

    attn1, attn_weights_block1 = self.mha1(x, x, x, look_ahead_mask)  # (batch_size, target_seq_len, d_model)
    attn1 = self.dropout1(attn1, training=training)
    out1 = self.layernorm1(attn1 + x)

    attn2, attn_weights_block2 = self.mha2(
        enc_output, enc_output, out1, padding_mask)  # (batch_size, target_seq_len, d_model)
    attn2 = self.dropout2(attn2, training=training)
    out2 = self.layernorm2(attn2 + out1)  # (batch_size, target_seq_len, d_model)
```

```
    ffn_output = self.ffn(out2)  # (batch_size, target_seq_len, d_model)
    ffn_output = self.dropout3(ffn_output, training=training)
    out3 = self.layernorm3(ffn_output + out2)  # (batch_size, target_seq_len, d_model)

    return out3, attn_weights_block1, attn_weights_block2


sample_decoder_layer = DecoderLayer(512, 8, 2048)

sample_decoder_layer_output, _, _ = sample_decoder_layer(
    tf.random.uniform((64, 50, 512)), sample_encoder_layer_output,
    False, None, None)

sample_decoder_layer_output.shape  # (batch_size, target_seq_len, d_model)
```

```
TensorShape([64, 50, 512])
```

Encoder

The `Encoder` consists of:

1. Input Embedding
2. Positional Encoding
3. N encoder layers

The input is put through an embedding which is summed with the positional encoding. The output of this summation is the input to the encoder layers. The output of the encoder is the input to the decoder.

```
class Encoder(tf.keras.layers.Layer):
  def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
               maximum_position_encoding, rate=0.1):
    super(Encoder, self).__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    print(f'Port Encoder::init input_vocab_size={input_vocab_size}, d_model={d_model}')
    self.embedding = tf.keras.layers.Embedding(input_vocab_size, d_model)
    self.pos_encoding = positional_encoding(maximum_position_encoding,
                                            self.d_model)


    self.enc_layers = [EncoderLayer(d_model, num_heads, dff, rate)
                       for _ in range(num_layers)]

    self.dropout = tf.keras.layers.Dropout(rate)

  def call(self, x, training, mask):

    seq_len = tf.shape(x)[1]

    # adding embedding and position encoding.
    x = self.embedding(x)  # (batch_size, input_seq_len, d_model)

    x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
    x += self.pos_encoding[:, :seq_len, :]

    x = self.dropout(x, training=training)

    for i in range(self.num_layers):
      x = self.enc_layers[i](x, training, mask)

    return x  # (batch_size, input_seq_len, d_model)


sample_encoder = Encoder(num_layers=2, d_model=512, num_heads=8,
                         dff=2048, input_vocab_size=8500,
                         maximum_position_encoding=10000)
temp_input = tf.random.uniform((64, 62), dtype=tf.int64, minval=0, maxval=200)

sample_encoder_output = sample_encoder(temp_input, training=False, mask=None)
```

```
print (sample_encoder_output.shape)  # (batch_size, input_seq_len, d_model)
```

```
Port Encoder::init input_vocab_size=8500, d_model=512
(64, 62, 512)
```

Decoder

The `Decoder` consists of:

1. Output Embedding
2. Positional Encoding
3. N decoder layers

The target is put through an embedding which is summed with the positional encoding. The output of this summation is the input to the decoder layers. The output of the decoder is the input to the final linear layer.

```
class Decoder(tf.keras.layers.Layer):
  def __init__(self, num_layers, d_model, num_heads, dff, target_vocab_size,
               maximum_position_encoding, rate=0.1):
    super(Decoder, self).__init__()

    self.d_model = d_model
    self.num_layers = num_layers

    self.embedding = tf.keras.layers.Embedding(target_vocab_size, d_model)
    self.pos_encoding = positional_encoding(maximum_position_encoding, d_model)

    self.dec_layers = [DecoderLayer(d_model, num_heads, dff, rate)
                       for _ in range(num_layers)]
    self.dropout = tf.keras.layers.Dropout(rate)

  def call(self, x, enc_output, training,
           look_ahead_mask, padding_mask):

    seq_len = tf.shape(x)[1]
    attention_weights = {}

    x = self.embedding(x)  # (batch_size, target_seq_len, d_model)
    x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
    x += self.pos_encoding[:, :seq_len, :]

    x = self.dropout(x, training=training)

    for i in range(self.num_layers):
      x, block1, block2 = self.dec_layers[i](x, enc_output, training,
                                             look_ahead_mask, padding_mask)

      attention_weights['decoder_layer{}_block1'.format(i+1)] = block1
      attention_weights['decoder_layer{}_block2'.format(i+1)] = block2

    # x.shape == (batch_size, target_seq_len, d_model)
    return x, attention_weights


sample_decoder = Decoder(num_layers=2, d_model=512, num_heads=8,
                         dff=2048, target_vocab_size=8000,
                         maximum_position_encoding=5000)
temp_input = tf.random.uniform((64, 26), dtype=tf.int64, minval=0, maxval=200)

output, attn = sample_decoder(temp_input,
                              enc_output=sample_encoder_output,
                              training=False,
                              look_ahead_mask=None,
                              padding_mask=None)

output.shape, attn['decoder_layer2_block2'].shape
```

```
(TensorShape([64, 26, 512]), TensorShape([64, 8, 26, 62]))
```

Create the Transformer

Transformer consists of the encoder, decoder and a final linear layer. The output of the decoder is the input to the linear layer and its output is returned.

```
class Transformer(tf.keras.Model):
  def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
               target_vocab_size, pe_input, pe_target, rate=0.1):
    super(Transformer, self).__init__()

    self.tokenizer = Encoder(num_layers, d_model, num_heads, dff,
                           input_vocab_size, pe_input, rate)

    self.decoder = Decoder(num_layers, d_model, num_heads, dff,
                           target_vocab_size, pe_target, rate)

    self.final_layer = tf.keras.layers.Dense(target_vocab_size)

  def call(self, inp, tar, training, enc_padding_mask,
           look_ahead_mask, dec_padding_mask):

    enc_output = self.tokenizer(inp, training, enc_padding_mask)  # (batch_size, inp_seq_len, d_model)

    # dec_output.shape == (batch_size, tar_seq_len, d_model)
    dec_output, attention_weights = self.decoder(
        tar, enc_output, training, look_ahead_mask, dec_padding_mask)

    final_output = self.final_layer(dec_output)  # (batch_size, tar_seq_len, target_vocab_size)

    return final_output, attention_weights


sample_transformer = Transformer(
    num_layers=2, d_model=512, num_heads=8, dff=2048,
    input_vocab_size=8500, target_vocab_size=8000,
    pe_input=10000, pe_target=6000)

temp_input = tf.random.uniform((64, 38), dtype=tf.int64, minval=0, maxval=200)
temp_target = tf.random.uniform((64, 36), dtype=tf.int64, minval=0, maxval=200)

fn_out, _ = sample_transformer(temp_input, temp_target, training=False,
                               enc_padding_mask=None,
                               look_ahead_mask=None,
                               dec_padding_mask=None)

fn_out.shape  # (batch_size, tar_seq_len, target_vocab_size)
```

```
Port Encoder::init input_vocab_size=8500, d_model=512
TensorShape([64, 36, 8000])
```

Set hyperparameters

To keep this example small and relatively fast, the values for *num_layers, d_model, and dff* have been reduced.

The values used in the base model of transformer were; *num_layers=6*, *d_model = 512*, *dff = 2048*. See the [paper](#) for all the other versions of the transformer.

Note: By changing the values below, you can get the model that achieved state of the art on many tasks.

```
num_layers = 4
d_model = 128
dff = 512
num_heads = 8
dropout_rate = 0.1
```

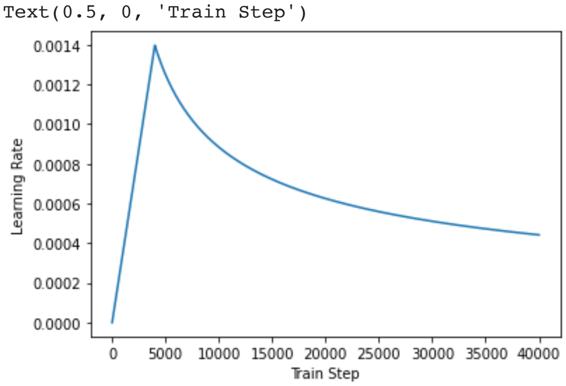
Optimizer

Use the Adam optimizer with a custom learning rate scheduler according to the formula in the paper.

$$lrate = d_{model}^{-0.5} * min(step\_num^{-0.5}, step\_num * warmup\_steps^{-1.5})$$

```
class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
  def __init__(self, d_model, warmup_steps=4000):
    super(CustomSchedule, self).__init__()

    self.d_model = d_model
    self.d_model = tf.cast(self.d_model, tf.float32)

    self.warmup_steps = warmup_steps

  def __call__(self, step):
    arg1 = tf.math.rsqrt(step)
    arg2 = step * (self.warmup_steps ** -1.5)

    return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)


learning_rate = CustomSchedule(d_model)

optimizer = tf.keras.optimizers.Adam(learning_rate, beta_1=0.9, beta_2=0.98,
                                     epsilon=1e-9)


temp_learning_rate_schedule = CustomSchedule(d_model)

plt.plot(temp_learning_rate_schedule(tf.range(40000, dtype=tf.float32)))
plt.ylabel("Learning Rate")
plt.xlabel("Train Step")
```

```
Text(0.5, 0, 'Train Step')
```



Loss and metrics

Since the target sequences are padded, it is important to apply a padding mask when calculating the loss.

```
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_sum(loss_)/tf.reduce_sum(mask)
```

```
def accuracy_function(real, pred):
  accuracies = tf.equal(real, tf.argmax(pred, axis=2))

  mask = tf.math.logical_not(tf.math.equal(real, 0))
  accuracies = tf.math.logical_and(mask, accuracies)

  accuracies = tf.cast(accuracies, dtype=tf.float32)
  mask = tf.cast(mask, dtype=tf.float32)
  return tf.reduce_sum(accuracies)/tf.reduce_sum(mask)
```

```
train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.Mean(name='train_accuracy')
```

Training and checkpointing

```
transformer = Transformer(
    num_layers=num_layers,
    d_model=d_model,
    num_heads=num_heads,
    dff=dff,
    input_vocab_size=tokenizers.pt.get_vocab_size(),
    target_vocab_size=tokenizers.en.get_vocab_size(),
    pe_input=1000,
    pe_target=1000,
    rate=dropout_rate)
```

```
Port Encoder::init input_vocab_size=8318, d_model=128
```

```
def create_masks(inp, tar):
  # Encoder padding mask
  enc_padding_mask = create_padding_mask(inp)

  # Used in the 2nd attention block in the decoder.
  # This padding mask is used to mask the encoder outputs.
  dec_padding_mask = create_padding_mask(inp)

  # Used in the 1st attention block in the decoder.
  # It is used to pad and mask future tokens in the input received by
  # the decoder.
  look_ahead_mask = create_look_ahead_mask(tf.shape(tar)[1])
  dec_target_padding_mask = create_padding_mask(tar)
  combined_mask = tf.maximum(dec_target_padding_mask, look_ahead_mask)

  return enc_padding_mask, combined_mask, dec_padding_mask
```

Create the checkpoint path and the checkpoint manager. This will be used to save checkpoints every `n` epochs.

```
checkpoint_path = BASE_MODEL_PATH

ckpt = tf.train.Checkpoint(transformer=transformer,
                           optimizer=optimizer)

ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path, max_to_keep=3)

# if a checkpoint exists, restore the latest checkpoint.
#if ckpt_manager.latest_checkpoint:
#  ckpt.restore(ckpt_manager.latest_checkpoint)
#  print ('Latest checkpoint restored!!')
```

```
!ls drive/MyDrive/models/transformer_port_eng_v2
```

```
ted_hrlr_translate_pt_en_converter
```

The target is divided into tar_inp and tar_real. tar_inp is passed as an input to the decoder. `tar_real` is that same input shifted by 1: At each location in `tar_input`, `tar_real` contains the next token that should be predicted.

For example, `sentence` = "SOS A lion in the jungle is sleeping EOS"

`tar_inp` = "SOS A lion in the jungle is sleeping"

`tar_real` = "A lion in the jungle is sleeping EOS"

The transformer is an auto-regressive model: it makes predictions one part at a time, and uses its output so far to decide what to do next.

During training this example uses teacher-forcing (like in the [text generation tutorial]). Teacher forcing is passing the true output to the next time step regardless of what the model predicts at the current time step.

As the transformer predicts each word, *self-attention* allows it to look at the previous words in the input sequence to better predict the next word.

To prevent the model from peeking at the expected output the model uses a look-ahead mask.

```
EPOCHS = 20
```

```
# The @tf.function trace-compiles train_step into a TF graph for faster
# execution. The function specializes to the precise shape of the argument
# tensors. To avoid re-tracing due to the variable sequence lengths or variable
# batch sizes (the last batch is smaller), use input_signature to specify
# more generic shapes.

train_step_signature = [
    tf.TensorSpec(shape=(None, None), dtype=tf.int64),
    tf.TensorSpec(shape=(None, None), dtype=tf.int64),
]

@tf.function(input_signature=train_step_signature)
def train_step(inp, tar):
  tar_inp = tar[:, :-1]
  tar_real = tar[:, 1:]

  enc_padding_mask, combined_mask, dec_padding_mask = create_masks(inp, tar_inp)

  with tf.GradientTape() as tape:
    predictions, _ = transformer(inp, tar_inp,
                                 True,
                                 enc_padding_mask,
                                 combined_mask,
                                 dec_padding_mask)
    loss = loss_function(tar_real, predictions)

  gradients = tape.gradient(loss, transformer.trainable_variables)
  optimizer.apply_gradients(zip(gradients, transformer.trainable_variables))

  train_loss(loss)
  train_accuracy(accuracy_function(tar_real, predictions))
```

Portuguese is used as the input language and English is the target language.

```
for (batch, (inp, tar)) in enumerate(train_batches):
  print(f'inp.shape={inp.shape}')
  print(f'tar.shape={tar.shape}')
  print(inp)
  break
```

```
inp.shape=(64, 119)
tar.shape=(64, 113)
tf.Tensor(
[[   2   39   39 ...    0    0    0]
 [   2  493   28 ...    0    0    0]
 [   2  107   88 ...    0    0    0]
 ...
 [   2  240 4764 ...    0    0    0]
 [   2  191  559 ...    0    0    0]
 [   2  461  131 ...    0    0    0]], shape=(64, 119), dtype=int64)
```

```
for epoch in range(EPOCHS):
  start = time.time()

  train_loss.reset_states()
```

```
train_accuracy.reset_states()

# inp -> portuguese, tar -> english
batch_start = time.time()
for (batch, (inp, tar)) in enumerate(train_batches):

  train_step(inp, tar)

  if batch % 50 == 0:
    print(f'Epoch {epoch + 1} Batch {batch} Loss {train_loss.result():.4f} Accuracy {train_accuracy.result():.4f} with tim
    batch_start = time.time()

#if (epoch + 1) % 5 == 0:
ckpt_save_path = ckpt_manager.save()
print (f'Saving checkpoint for epoch {epoch+1} at {ckpt_save_path}')

print(f'Epoch {epoch + 1} Loss {train_loss.result():.4f} Accuracy {train_accuracy.result():.4f}')

print(f'Time taken for 1 epoch: {time.time() - start:.2f} secs\n')
```

```
Epoch 18 Batch 200 Loss 1.4738 Accuracy 0.6757 with time taken 3.12 secs
Epoch 18 Batch 250 Loss 1.4736 Accuracy 0.6756 with time taken 3.05 secs
Epoch 18 Batch 300 Loss 1.4740 Accuracy 0.6752 with time taken 3.20 secs
Epoch 18 Batch 350 Loss 1.4769 Accuracy 0.6747 with time taken 2.95 secs
Epoch 18 Batch 400 Loss 1.4788 Accuracy 0.6746 with time taken 3.11 secs
Epoch 18 Batch 450 Loss 1.4825 Accuracy 0.6742 with time taken 3.05 secs
Epoch 18 Batch 500 Loss 1.4844 Accuracy 0.6739 with time taken 3.03 secs
Epoch 18 Batch 550 Loss 1.4872 Accuracy 0.6736 with time taken 3.13 secs
Epoch 18 Batch 600 Loss 1.4906 Accuracy 0.6731 with time taken 3.19 secs
Epoch 18 Batch 650 Loss 1.4952 Accuracy 0.6724 with time taken 3.12 secs
Epoch 18 Batch 700 Loss 1.4969 Accuracy 0.6722 with time taken 3.08 secs
Epoch 18 Batch 750 Loss 1.4999 Accuracy 0.6718 with time taken 3.24 secs
Epoch 18 Batch 800 Loss 1.5040 Accuracy 0.6712 with time taken 3.09 secs
Saving checkpoint for epoch 18 at drive/MyDrive/models/transformer_port_eng_v2/ckpt-18
Epoch 18 Loss 1.5043 Accuracy 0.6712
Time taken for 1 epoch: 50.76 secs

Epoch 19 Batch 0 Loss 1.3641 Accuracy 0.6945 with time taken 0.49 secs
Epoch 19 Batch 50 Loss 1.4034 Accuracy 0.6875 with time taken 3.05 secs
Epoch 19 Batch 100 Loss 1.4086 Accuracy 0.6869 with time taken 3.08 secs
Epoch 19 Batch 150 Loss 1.4202 Accuracy 0.6850 with time taken 3.01 secs
Epoch 19 Batch 200 Loss 1.4291 Accuracy 0.6835 with time taken 3.31 secs
Epoch 19 Batch 250 Loss 1.4388 Accuracy 0.6819 with time taken 3.38 secs
Epoch 19 Batch 300 Loss 1.4416 Accuracy 0.6812 with time taken 3.19 secs
Epoch 19 Batch 350 Loss 1.4413 Accuracy 0.6817 with time taken 2.96 secs
Epoch 19 Batch 400 Loss 1.4450 Accuracy 0.6811 with time taken 3.12 secs
Epoch 19 Batch 450 Loss 1.4509 Accuracy 0.6799 with time taken 3.25 secs
Epoch 19 Batch 500 Loss 1.4508 Accuracy 0.6799 with time taken 3.09 secs
Epoch 19 Batch 550 Loss 1.4523 Accuracy 0.6797 with time taken 3.16 secs
Epoch 19 Batch 600 Loss 1.4547 Accuracy 0.6792 with time taken 3.12 secs
Epoch 19 Batch 650 Loss 1.4570 Accuracy 0.6787 with time taken 3.31 secs
Epoch 19 Batch 700 Loss 1.4592 Accuracy 0.6785 with time taken 3.15 secs
Epoch 19 Batch 750 Loss 1.4620 Accuracy 0.6780 with time taken 3.20 secs
Epoch 19 Batch 800 Loss 1.4650 Accuracy 0.6776 with time taken 3.02 secs
Saving checkpoint for epoch 19 at drive/MyDrive/models/transformer_port_eng_v2/ckpt-19
Epoch 19 Loss 1.4656 Accuracy 0.6776
Time taken for 1 epoch: 51.75 secs

Epoch 20 Batch 0 Loss 1.2796 Accuracy 0.7099 with time taken 0.51 secs
Epoch 20 Batch 50 Loss 1.4007 Accuracy 0.6884 with time taken 3.24 secs
Epoch 20 Batch 100 Loss 1.3968 Accuracy 0.6884 with time taken 3.21 secs
Epoch 20 Batch 150 Loss 1.3995 Accuracy 0.6876 with time taken 3.03 secs
Epoch 20 Batch 200 Loss 1.4018 Accuracy 0.6877 with time taken 3.10 secs
Epoch 20 Batch 250 Loss 1.4018 Accuracy 0.6877 with time taken 3.02 secs
Epoch 20 Batch 300 Loss 1.4042 Accuracy 0.6872 with time taken 3.04 secs
Epoch 20 Batch 350 Loss 1.4081 Accuracy 0.6863 with time taken 3.05 secs
Epoch 20 Batch 400 Loss 1.4111 Accuracy 0.6859 with time taken 3.17 secs
Epoch 20 Batch 450 Loss 1.4143 Accuracy 0.6855 with time taken 3.19 secs
Epoch 20 Batch 500 Loss 1.4195 Accuracy 0.6845 with time taken 3.24 secs
Epoch 20 Batch 550 Loss 1.4188 Accuracy 0.6846 with time taken 3.02 secs
Epoch 20 Batch 600 Loss 1.4200 Accuracy 0.6844 with time taken 3.05 secs
Epoch 20 Batch 650 Loss 1.4223 Accuracy 0.6842 with time taken 2.97 secs
Epoch 20 Batch 700 Loss 1.4246 Accuracy 0.6839 with time taken 3.11 secs
Epoch 20 Batch 750 Loss 1.4281 Accuracy 0.6832 with time taken 3.12 secs
Epoch 20 Batch 800 Loss 1.4311 Accuracy 0.6827 with time taken 3.20 secs
Saving checkpoint for epoch 20 at drive/MyDrive/models/transformer_port_eng_v2/ckpt-20
Epoch 20 Loss 1.4316 Accuracy 0.6826
Time taken for 1 epoch: 51.12 secs
```

```
# Portugese to English
# Epoch 20 Batch 800 Loss 1.1240 Accuracy 0.7345
# Saving checkpoint for epoch 20 at drive/MyDrive/models/transformer_port_v2/ckpt-7
# Epoch 20 Loss 1.1249 Accuracy 0.7343
# Time taken for 1 epoch: 59.58 secs
```

```
checkpoint = tf.keras.callbacks.ModelCheckpoint(BASE_MODEL_PATH + '/' + 'final_model.h5', save_weights_only=False)
```

Evaluate

The following steps are used for evaluation:

- Encode the input sentence using the Portuguese tokenizer (`tokenizers.pt`). This is the encoder input.
- The decoder input is initialized to the `[START]` token.
- Calculate the padding masks and the look ahead masks.
- The `decoder` then outputs the predictions by looking at the `encoder output` and its own output (self-attention).
- The model makes predictions of the next word for each word in the output. Most of these are redundant. Use the predictrions from the last word.
- Concatentate the predicted word to the decoder input and pass it to the decoder.
- In this approach, the decoder predicts the next word based on the previous words it predicted.

Note: The model used here has less capacity to keep the example relatively faster so the predictions maybe less right. To reproduce the results in the paper, use the entire dataset and base transformer model or transformer XL, by changing the hyperparameters above.

```
def evaluate(sentence, max_length=40):
  # inp sentence is portuguese, hence adding the start and end token
  sentence = tf.convert_to_tensor([sentence])
  sentence = tokenizers.pt.tokenize(sentence).to_tensor()

  encoder_input = sentence

  # as the target is english, the first word to the transformer should be the
  # english start token.
  start, end = tokenizers.en.tokenize([''])[0]
  output = tf.convert_to_tensor([start])
  output = tf.expand_dims(output, 0)

  for i in range(max_length):
    enc_padding_mask, combined_mask, dec_padding_mask = create_masks(
        encoder_input, output)

    # predictions.shape == (batch_size, seq_len, vocab_size)
    predictions, attention_weights = transformer(encoder_input,
                                                 output,
                                                 False,
                                                 enc_padding_mask,
                                                 combined_mask,
                                                 dec_padding_mask)

    # select the last word from the seq_len dimension
    predictions = predictions[: ,-1:, :]  # (batch_size, 1, vocab_size)

    predicted_id = tf.argmax(predictions, axis=-1)

    # concatentate the predicted_id to the output which is given to the decoder
    # as its input.
    output = tf.concat([output, predicted_id], axis=-1)

    # return the result if the predicted_id is equal to the end token
    if predicted_id == end:
      break

  # output.shape (1, tokens)
  text = tokenizers.en.detokenize(output)[0] # shape: ()

  tokens = tokenizers.en.lookup(output)[0]
```

```
  return text, tokens, attention_weights


def print_translation(sentence, tokens, ground_truth):
  print(f'{"Input:":15s}: {sentence}')
  print(f'{"Prediction":15s}: {tokens.numpy().decode("utf-8")}')
  print(f'{"Ground truth":15s}: {ground_truth}')


sentence = "este é um problema que temos que resolver."
ground_truth = "this is a problem we have to solve ."

translated_text, translated_tokens, attention_weights = evaluate(sentence)
print_translation(sentence, translated_text, ground_truth)
```

```
Input:         : este é um problema que temos que resolver.
Prediction     : this is a problem we have to solve .
Ground truth   : this is a problem we have to solve .
```

```
sentence = "os meus vizinhos ouviram sobre esta ideia."
ground_truth = "and my neighboring homes heard about this idea ."

translated_text, translated_tokens, attention_weights = evaluate(sentence)
print_translation(sentence, translated_text, ground_truth)
```

```
Input:         : os meus vizinhos ouviram sobre esta ideia.
Prediction     : my neighbors heard about this idea .
Ground truth   : and my neighboring homes heard about this idea .
```

```
sentence = "vou então muito rapidamente partilhar convosco algumas histórias de algumas coisas mágicas que aconteceram."
ground_truth = "so i \'ll just share with you some stories very quickly of some magical things that have happened ."

translated_text, translated_tokens, attention_weights = evaluate(sentence)
print_translation(sentence, translated_text, ground_truth)
```

```
Input:         : vou então muito rapidamente partilhar convosco algumas histórias de algumas coisas mágicas que aconte
Prediction     : so i ' m going to be very quickly to share with you some stories of some magic things that happened .
Ground truth   : so i 'll just share with you some stories very quickly of some magical things that have happened .
```

You can pass different layers and attention blocks of the decoder to the `plot` parameter.

Attention plots

The `evaluate` function also returns a dictionary of attention maps you can use to visualize the internal working of the model:

```
sentence = "este é o primeiro livro que eu fiz."
ground_truth = "this is the first book i've ever done."

translated_text, translated_tokens, attention_weights = evaluate(sentence)
print_translation(sentence, translated_text, ground_truth)
```

```
Input:         : este é o primeiro livro que eu fiz.
Prediction     : this is the first book i did .
Ground truth   : this is the first book i've ever done.
```

```
def plot_attention_head(in_tokens, translated_tokens, attention):
  # The plot is of the attention when a token was generated.
  # The model didn't generate `<START>` in the output. Skip it.
  translated_tokens = translated_tokens[1:]

  ax = plt.gca()
  ax.matshow(attention)
  ax.set_xticks(range(len(in_tokens)))
  ax.set_yticks(range(len(translated_tokens)))


  labels = [label.decode('utf-8') for label in in_tokens.numpy()]
  ax.set_xticklabels(
      labels, rotation=90)
```

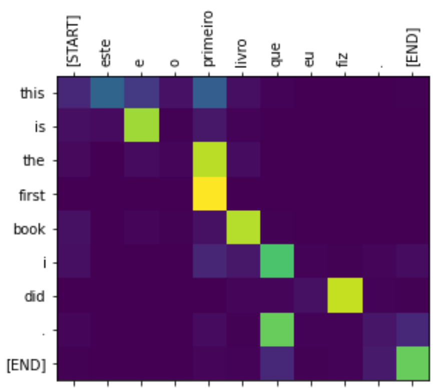
```
  labels = [label.decode('utf-8') for label in translated_tokens.numpy()]
  ax.set_yticklabels(labels)


head = 0
# shape: (batch=1, num_heads, seq_len_q, seq_len_k)
attention_heads = tf.squeeze(
  attention_weights['decoder_layer4_block2'], 0)
attention = attention_heads[head]
attention.shape
```

```
    TensorShape([9, 11])
```

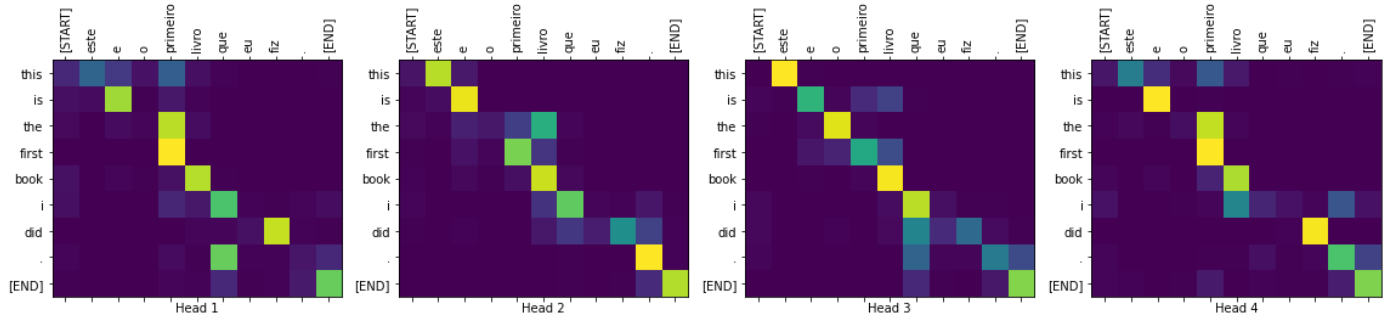
```
in_tokens = tf.convert_to_tensor([sentence])
in_tokens = tokenizers.pt.tokenize(in_tokens).to_tensor()
in_tokens = tokenizers.pt.lookup(in_tokens)[0]
in_tokens
```

```
    <tf.Tensor: shape=(11,), dtype=string, numpy=
    array([b'[START]', b'este', b'e', b'o', b'primeiro', b'livro', b'que',
           b'eu', b'fiz', b'.', b'[END]'], dtype=object)>
```

```
plot_attention_head(in_tokens, translated_tokens, attention)
```




```
def plot_attention_weights(sentence, translated_tokens, attention_heads):
  in_tokens = tf.convert_to_tensor([sentence])
  in_tokens = tokenizers.pt.tokenize(in_tokens).to_tensor()
  in_tokens = tokenizers.pt.lookup(in_tokens)[0]
  in_tokens

  fig = plt.figure(figsize=(16, 8))

  for h, head in enumerate(attention_heads):
    ax = fig.add_subplot(2, 4, h+1)

    plot_attention_head(in_tokens, translated_tokens, head)

    ax.set_xlabel('Head {}'.format(h+1))

  plt.tight_layout()
  plt.show()


plot_attention_weights(sentence, translated_tokens,
                       attention_weights['decoder_layer4_block2'][0])
```



The model does okay on unfamiliar words. Neither "triceratops" or "encyclopedia" are in the input dataset and the model almost learns to transliterare them, even withoput a shared vocabulary:


```
def translate(ground_truth, input_sentence):

  ground_truth = ground_truth.lower()

  translated_text, translated_tokens, attention_weights = evaluate(input_sentence)
  print_translation(input_sentence, translated_text, ground_truth)

  translated_text = translated_text.numpy()
  translated_text = translated_text.decode("utf-8")

  score = sentence_bleu(ground_truth, translated_text, weights=(1.0,0,0,0))
  print(f"BLEU-1 score: {score*100:.2f}")
  score = sentence_bleu(ground_truth, translated_text, weights=(0.5,0.5,0,0))
  print(f"BLEU-2 score: {score*100:.2f}")
  score = sentence_bleu(ground_truth, translated_text, weights=(0.3,0.3,0.3,0))
  print(f"BLEU-3 score: {score*100:.2f}")
  score = sentence_bleu(ground_truth, translated_text, weights=(0.25,0.25,0.25,0.25))
  print(f"BLEU-4 score: {score*100:.2f}")


input_sentence = "Eu li sobre triceratops na enciclopédia."
prediction = "I read about triceratops in the encyclopedia."

translate(prediction, input_sentence)

plot_attention_weights(sentence, translated_tokens,
                       attention_weights['decoder_layer4_block2'][0])
```

```
    Input:         : Eu li sobre triceratops na enciclopédia.
    Prediction     : i read about triumphops in the enclopedia .
    Ground truth   : i read about triceratops in the encyclopedia.
    BLEU-1 score: 39.53
    BLEU-2 score: 62.88
    BLEU-3 score: 75.70
    BLEU-4 score: 79.29
    /usr/local/lib/python3.7/dist-packages/nltk/translate/bleu_score.py:490: UserWarning:
    Corpus/Sentence contains 0 counts of 2-gram overlaps.
    BLEU scores might be undesirable; use SmoothingFunction().
```

```
input_sentence = "Fui à loja comprar mantimentos."
ground_truth = "I went to the store to buy some groceries."

translate(ground_truth, input_sentence)
```

```
    Input:         : Fui à loja comprar mantimentos.
    Prediction     : i went to the store supplies .
    Ground truth   : i went to the store to buy some groceries.
    BLEU-1 score: 40.00
    BLEU-2 score: 63.25
    BLEU-3 score: 75.97
    BLEU-4 score: 79.53
    /usr/local/lib/python3.7/dist-packages/nltk/translate/bleu_score.py:490: UserWarning:
    Corpus/Sentence contains 0 counts of 2-gram overlaps.
    BLEU scores might be undesirable; use SmoothingFunction().
      warnings.warn(_msg)
```

```
input_sentence = "Qual é o seu filme favorito?"
ground_truth = "What is your favorite movie?"

translate(ground_truth, input_sentence)
```

```
    Input:         : Qual é o seu filme favorito?
    Prediction     : what is their favorite film ?
    Ground truth   : what is your favorite movie?
    BLEU-1 score: 48.28
    BLEU-2 score: 69.48
    BLEU-3 score: 80.37
    BLEU-4 score: 83.36
    /usr/local/lib/python3.7/dist-packages/nltk/translate/bleu_score.py:490: UserWarning:
    Corpus/Sentence contains 0 counts of 2-gram overlaps.
    BLEU scores might be undesirable; use SmoothingFunction().
      warnings.warn(_msg)
```

```
input_sentence = "Você viu minha carteira que perdi quando estava no trabalho?"
ground_truth = "Have you seen my wallet that I lost when I was at work?"

translate(ground_truth, input_sentence)
```

```
    Input:         : Você viu minha carteira que perdi quando estava no trabalho?
    Prediction     : you saw my wallet when i was lost when i was in work ?
    Ground truth   : have you seen my wallet that i lost when i was at work?
    BLEU-1 score: 31.48
    BLEU-2 score: 56.11
    BLEU-3 score: 70.70
    BLEU-4 score: 74.91
    /usr/local/lib/python3.7/dist-packages/nltk/translate/bleu_score.py:490: UserWarning:
    Corpus/Sentence contains 0 counts of 2-gram overlaps.
    BLEU scores might be undesirable; use SmoothingFunction().
      warnings.warn(_msg)
```

0s completed at 3:38 PM