Transformer

MokkeMeguru

<2019-08-24 土>

Contents

1 TensorFlow ライブラリのインポート

```
1 from __future__ import division, absolute_import
2 from __future__ import print_function, unicode_literals
```

```
from functools import reduce, partial

import tensorflow as tf
# import tensorflow_hub as hub
import tensorflow_datasets as tfds
import tensorflow_probability as tfp
# from tensorflow_examples.models.pix2pix import pix2pix

from tensorflow import keras
from tensorflow.keras import layers, datasets, models
from tensorflow.keras.models import Sequential
# import tensorflow_text as text

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

import pandas as pd
from sklearn.model_selection import train_test_split
# import seaborn as sns
import os
import time
# import yaml
# import h5py
# import pathlib
# import random
# import IPython.display as display
# from IPython.display import clear_output
# import PIL.Image as Image
# import urllib3
import io
import tempfile
from pprint import pprint

import unicodedata
import re
import time

def print_infos(infolist: list):
    for info in infolist:
        print(info)


def pprint_infos(infolist: list):
    for info in infolist:
        pprint(info)


print_infos([
```

```
    '{:25}: {}'.format("tensorflow\'s version", tf.__version__),
    # '{:25}: {}'.format("tensorflow\'s version", hub.__version__),
])

AUTOTUNE = tf.data.experimental.AUTOTUNE
# urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
```

tensorflow’s version : 2.0.0-rc0

2 データセットのインポート

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

for pt_ex, en_ex in train_examples.take(1):
    print(pt_ex.numpy())
    print(en_ex.numpy())
```

WARNING: Logging before flag parsing goes to stderr.
W0904 14:16:00.669344 139700101117568 dataset$_{\text{builder.py}}$:439] Warning: Setting shuffle$_{\text{files}}$=True because split=TRAIN and shuffle$_{\text{files}}$=None. This behavior will be deprecated on 2019-08-06, at which point shuffle$_{\text{files}}$=False will be the default for all splits.

b’os astr33nomos acreditam que cada estrela da gal31xia tem um planeta , e especulam que at39 um quinto deles tem um planeta do tipo da terra que poder31 ter vida , mas ainda n33o vimos nenhum deles .’
b"astronomers now believe that every star in the galaxy has a planet , and they speculate that up to one fifth of them have an earth-like planet that might be able to harbor life , but we have n’t seen any of them ."

```
tokenizer_en = tfds.features.text.SubwordTextEncoder.build_from_corpus(
    (en.numpy() for pt, en in train_examples), target_vocab_size=2**13)

tokenizer_pt = tfds.features.text.SubwordTextEncoder.build_from_corpus(
    (pt.numpy() for pt, en in train_examples), target_vocab_size=2**13)
sample_string = 'Transformer is awesome.'
tokenized_string = tokenizer_en.encode(sample_string)
original_string = tokenizer_en.decode(tokenized_string)
assert original_string == sample_string

print_infos([
    'Tokenized string is {}'.format(tokenized_string),
    'The original string: {}'.format(original_string)
])
```

Tokenized string is [7915, 1248, 7946, 7194, 13, 2799, 7877]
The original string: Transformer is awesome.

```
for ts in tokenized_string:
    print('{} -> {}'.format(ts, tokenizer_en.decode([ts])))
```

7915 -> T
1248 -> ran
7946 -> s
7194 -> former
13 -> is
2799 -> awesome
7877 -> .

```
BUFFER_SIZE = 20000
BATCH_SIZE = 64
def encode(lang1, lang2):
    # <start> = (tokenizer_pt / tokenizer_en) .vocab_size
    # <end> = (tokenizer_pt / tokenizer_en) .vocab_size + 1
    lang1 = [tokenizer_pt.vocab_size] + tokenizer_pt.encode(
        lang1.numpy()) + [tokenizer_pt.vocab_size + 1]
    lang2 = [tokenizer_en.vocab_size] + tokenizer_en.encode(
        lang2.numpy()) + [tokenizer_en.vocab_size + 1]
    return lang1, lang2
```

```
MAX_LENGTH = 40


def filter_max_length(x, y, max_length=MAX_LENGTH):
    return tf.logical_and(tf.size(x) <= max_length, tf.size(y) <=
    ↪  max_length)


def tf_encode(pt, en):
    return tf.py_function(encode, [pt, en], [tf.int64, tf.int64])
```

```
train_dataset = train_examples.map(tf_encode).filter(
    filter_max_length).cache().shuffle(BUFFER_SIZE).padded_batch(
        BATCH_SIZE, padded_shapes=([-1], [-1])).prefetch(AUTOTUNE)

```

```
val_dataset = val_examples.map(tf_encode).filter(
    filter_max_length).padded_batch(BATCH_SIZE, padded_shapes=([-1], [-1]))
```

```
pt_batch, en_batch = next(iter(val_dataset))
pt_batch, en_batch
```

```
(<tf.Tensor: id=546958, shape=(64, 40), dtype=int64, numpy=
 array([[8214, 1259,    5, ...,    0,    0,    0],
        [8214,  299,   13, ...,    0,    0,    0],
        [8214,   59,    8, ...,    0,    0,    0],
        ...,
        [8214,   95,    3, ...,    0,    0,    0],
        [8214, 5157,    1, ...,    0,    0,    0],
        [8214, 4479, 7990, ...,    0,    0,    0]])>,
 <tf.Tensor: id=546959, shape=(64, 40), dtype=int64, numpy=
 array([[8087,   18,   12, ...,    0,    0,    0],
        [8087,  634,   30, ...,    0,    0,    0],
        [8087,   16,   13, ...,    0,    0,    0],
        ...,
        [8087,   12,   20, ...,    0,    0,    0],
        [8087,   17, 4981, ...,    0,    0,    0],
        [8087,   12, 5453, ...,    0,    0,    0]])>)
```

3 Positional Encoding

ref. https://github.com/tensorflow/examples/blob/master/community/en/position_encoding.ipynb

The formula is here.

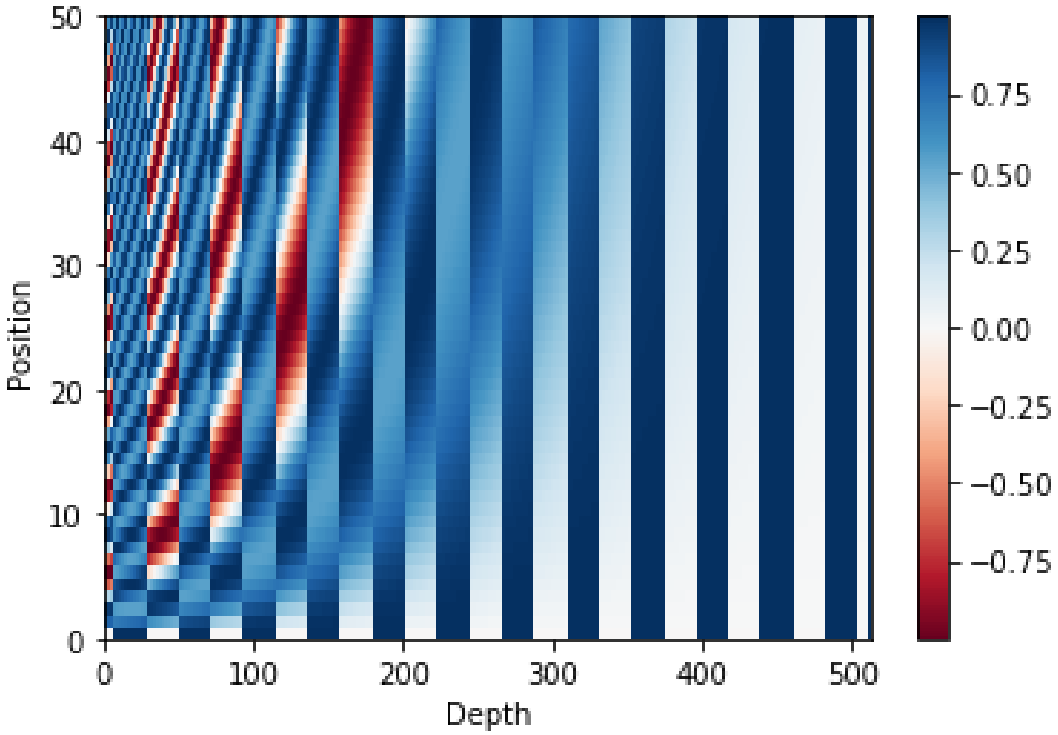
$$
\begin{aligned}
PE_{(pos,2i)} &= \sin(pos/10000^{2i/d_{model}}) \\
PE_{(pos,2i+1)} &= \cos(pos/10000^{2i/d_{model}})
\end{aligned}
$$

```
def get_angles(pos, i, d_model):
    angle_rates = 1 / np.power(1000, (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)

    angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
```

```
    angle_rads[:, 1::2] = np.cos(angle_rads[:, 0::2])
    pos_encoding = angle_rads[np.newaxis, ...]
    return tf.cast(pos_encoding, dtype=tf.float32)
```

```
pos_encoding = positional_encoding(50, 512)
print(pos_encoding.shape)
plt.pcolormesh(pos_encoding[0], cmap='RdBu')
plt.xlabel('Depth')
plt.xlim((0, 512))
plt.ylabel('Position')
plt.colorbar()
plt.show()
```

(1, 50, 512)



appendix

```
print(np.arange(10)[:, np.newaxis])
print(np.arange(10)[np.newaxis ,:])
```

```
[[0]
 [1]
 [2]
 [3]
 [4]
 [5]
 [6]
 [7]
 [8]
 [9]]
[[0 1 2 3 4 5 6 7 8 9]]
```

4 Masking

for ignoreing padding in calculation

```python
def create_padding_mask(seq):
    seq = tf.cast(tf.math.equal(seq, 0), tf.float32)
    # (batch_size, 1, 1,  seq_len)
    return seq[:, tf.newaxis, tf.newaxis, :]


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

```
tf.Tensor(
[[7 6 0 0 1]
 [1 2 3 0 0]
 [0 0 0 4 5]], shape=(3, 5), dtype=int32)
tf.Tensor(
[[[[0. 0. 1. 1. 0.]]]


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


 [[[1. 1. 1. 0. 0.]]]], shape=(3, 1, 1, 5), dtype=float32)
```

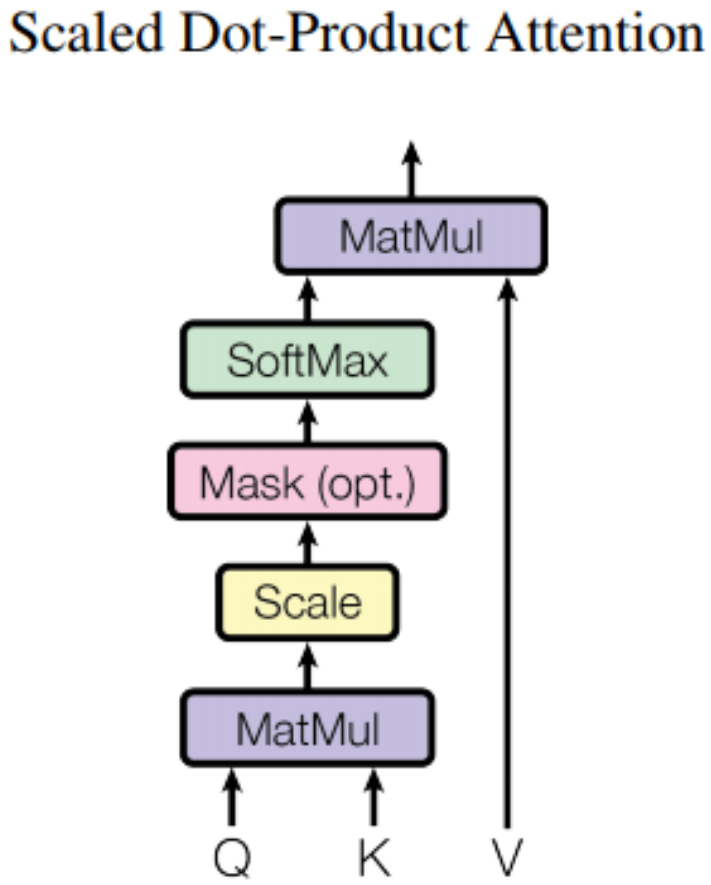
for ignoreing prediction word (in decoding model)

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


```

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

```
tf.Tensor([[0.22582567 0.58736205 0.92260003]], shape=(1, 3), dtype=float32)
tf.Tensor(
[[0. 1. 1.]
 [0. 0. 1.]
 [0. 0. 0.]], shape=(3, 3), dtype=float32)
```

5 Scaled dot product attention



Attention formula is here (Q is query, K is key, V is value)

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

```python
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.

  q, k, v は次に示される次元である必要があります
  k, v は 第 2 次元のサイズを統一されている必要があります (つまり seq_len_k == seq_len_v)
  マスクは、そのタイプ (padding / look ahead) に応じてサイズが変わります。
  しかし (... seq_len_q, seq_len_k) へブロードキャストできるようになっていなければなりません

  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
```

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

# この qruery は 2 番目の key と一致するので、2番めの value が返されます。
# => v[k.search(query)]
temp_q = tf.constant([[0, 10, 0]], dtype=tf.float32)  # (1, 3)
print_out(temp_q, temp_k, temp_v)
print()

# この query は 3, 4 番目の key と一致するので、3,  4 番目の value の平均値が返されま
↪ す。
temp_q = tf.constant([[0, 0, 10]], dtype=tf.float32)  # (1, 3)
print_out(temp_q, temp_k, temp_v)

# この query は 1, 2 番目の key と一致するので、1, 2 番目の value の平均値が返されます。
↪
temp_q = tf.constant([[10, 10, 0]], dtype=tf.float32)  # (1, 3)
print_out(temp_q, temp_k, temp_v)
```

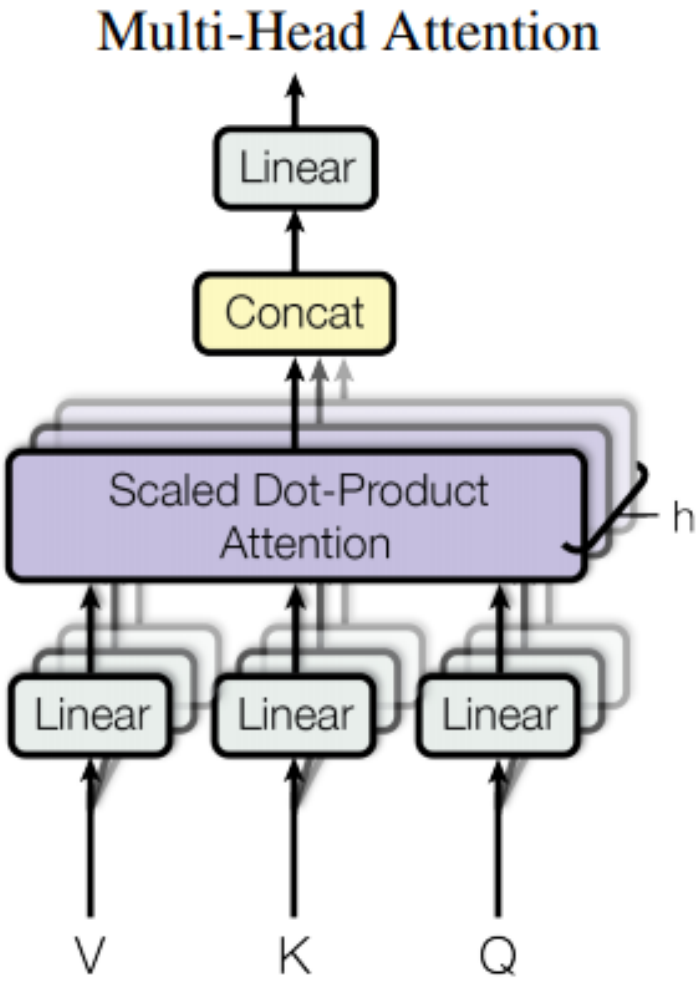
```
Attention weights are:
tf.Tensor([[0. 1. 0. 0.]], shape=(1, 4), dtype=float32)
Output is:
tf.Tensor([[10.  0.]], shape=(1, 2), dtype=float32)

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)
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)
```

```
# 上の query をすべて行列にまとめて実行すると次のようになります。
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)
```

6 Multi-head attention



Multi head Attention はの 4 つのパートから構成されます。

- Linear layer と 複数の head への分割
- Scaled dot-product attention
- heads の集約
- final linear layer

```python
class MultiHeadAttention(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 = layers.Dense(d_model)
        self.wk = layers.Dense(d_model)
        self.wv = layers.Dense(d_model)

        self.dense = 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)

        最後の次元である d_model を (num_heads, depth) へ分割します。
        また出力時には
        (batch_size, seq_len, num_heads, depth) -> (batch_size, num_heads, seq_len, depth) します。

        つまり
        (batch_size, seq_len, d_model)
        -> (batch_size, seq_len, num_heads, depth) (次元分割)
        -> (batch_size, num_heads, seq_len, depth) (軸入れ替え)
        """
        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):
        batch_size = tf.shape(q)[0]

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

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

        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
```

```
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
```

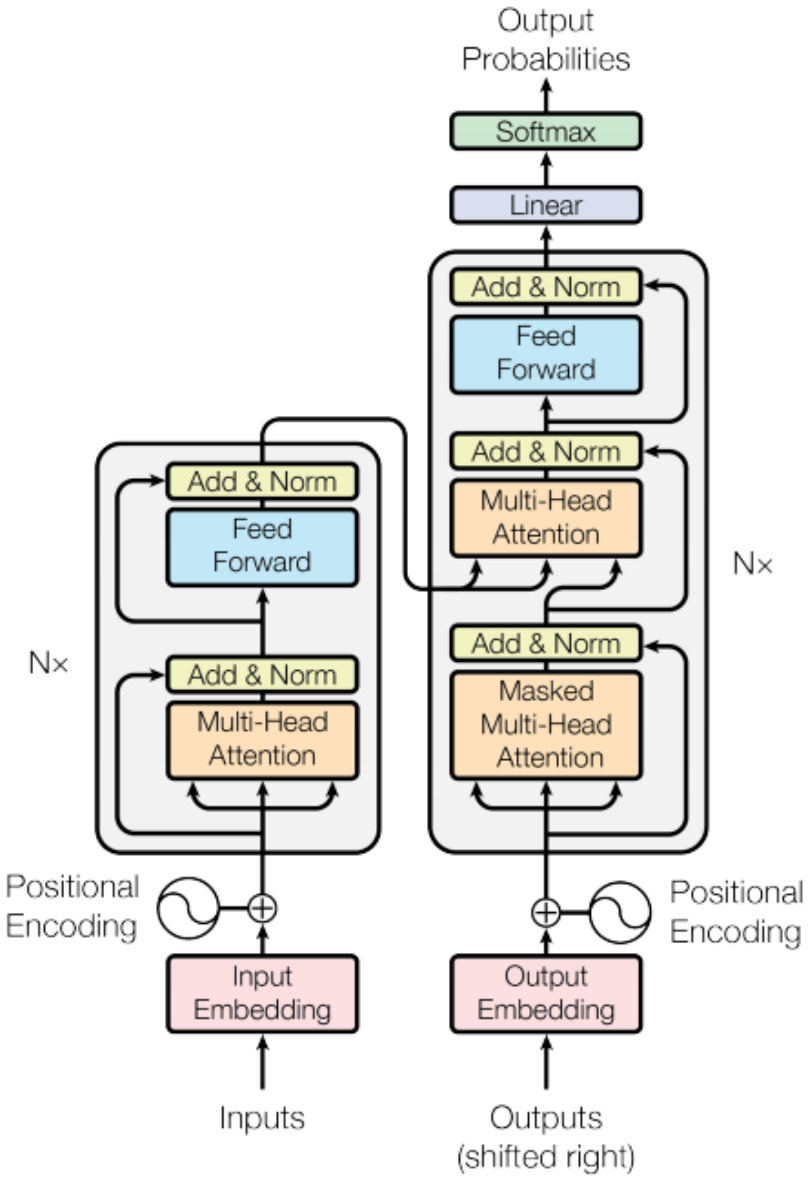
```
(TensorShape([1, 60, 512]), TensorShape([1, 8, 60, 60]))
```

7 Point wise feed forward network

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


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

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

8 Encoder and Decoder



8.1 Encoder Layer

```
class EncoderLayer(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.layer_norm1 = layers.LayerNormalization(epsilon=1e-6)

        self.ffn = point_wise_feed_forward_network(d_model, dff)
        self.layer_norm2 = layers.LayerNormalization(epsilon=1e-6)

        self.dropout1 = layers.Dropout(rate)
        self.dropout2 = layers.Dropout(rate)

    def call(self, x, training, padding_mask):
        # (..., input_seq_len, d_model)
        attn_output, _ = self.mha(x, x, x, padding_mask)
        attn_output = self.dropout1(attn_output, training=training)
        # (..., input_seq_len, d_model)
        out1 = self.layer_norm1(x + attn_output)

        # (..., input_seq_len, d_model)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        # (..., input_seq_len, d_model)
        out2 = self.layer_norm2(out1 + ffn_output)

        return out2
```

```
sample_encoder_layer = EncoderLayer(d_model=512, num_heads=8, dff=2048)
sample_encoder_layer_output = sample_encoder_layer(
    tf.random.uniform((64, 42, 512)), False, None)
sample_encoder_layer_output.shape
```

```
TensorShape([64, 42, 512])
```

8.2 Decoder Layer

```
class DecoderLayer(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.layernorm1 = layers.LayerNormalization(epsilon=1e-6)

        self.mha2 = MultiHeadAttention(d_model, num_heads)
        self.layernorm2 = layers.LayerNormalization(epsilon=1e-6)

        self.ffn = point_wise_feed_forward_network(d_model, dff)
        self.layernorm3 = layers.LayerNormalization(epsilon=1e-6)

```

```
        self.dropout1 = layers.Dropout(rate)
        self.dropout2 = layers.Dropout(rate)
        self.dropout3 = layers.Dropout(rate)

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

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

            #(..., target_seq_len, d_model)
            # これは 'self attention algorithm'
            #       v = enc_output, k=enc_output,  q=x
            #       => enc_output でできた v, k に query out1 でアクセスする
            attn2, attn_weights_block2 = self.mha2(enc_output, enc_output,
            ↪ out1,
                                                   padding_mask)
            attn2 = self.dropout1(attn2, training=training)
            out2 = self.layernorm1(attn2 + out1)

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

            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
```

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

8.3 Encoder

```
class Encoder(layers.Layer):
    def __init__(self,
```

```
                 num_layers,
                 d_model,
                 num_heads,
                 dff,
                 input_vocab_size,
                 rate=0.1):
        super(Encoder, self).__init__()
        self.d_model = d_model
        self.num_layers = num_layers
        self.embedng = layers.Embedding(input_vocab_size, self.d_model)

        # assumption: input_vocab_size > seq_len
        self.pos_encoding = positional_encoding(input_vocab_size,
        ↪  self.d_model)
        self.enc_layers = [
            EncoderLayer(d_model, num_heads, dff, rate)
            for _ in range(num_layers)
        ]
        self.dropout = layers.Dropout(rate)

    def call(self, x, training, mask):
        # x.shape == (..., seq_len)
        seq_len = tf.shape(x)[1]

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

        self.dropout(x, training=training)

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

        # (batch_size,  input_seq_len, d_model)
        return x
```

```
sample_encoder = Encoder(num_layers=2,
                         d_model=512,
                         num_heads=8,
                         dff=2048,
                         input_vocab_size=8500)
sample_encoder_output = sample_encoder(tf.random.uniform((64, 62)),
                                       training=False,
                                       mask=None)

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

```
(64, 62, 512)
```

8.4 Decoder

```python
class Decoder(layers.Layer):
    def __init__(self,
                 num_layers,
                 d_model,
                 num_heads,
                 dff,
                 target_vocab_size,
                 rate=0.1):
        super(Decoder, self).__init__()
        self.d_model = d_model
        self.num_layers = num_layers
        self.embedding = layers.Embedding(target_vocab_size, d_model)

        # assumption: target_vocab_size > seq_len
        self.pos_encoding = positional_encoding(target_vocab_size, d_model)
        self.dec_layers = [
            DecoderLayer(d_model, num_heads, dff, rate)
            for _ in range(num_layers)
        ]
        self.dropout = layers.Dropout(rate)

    def call(self, x, enc_output, training, look_ahead_mask, padding_mask):
        seq_len = tf.shape(x)[1]
        attention_weights = {}
        # (batch_size, target_seq_len, d_model)
        x = self.embedding(x)
        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)

output, attn = sample_decoder(tf.random.uniform((64, 26)),
                              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]))
```

9 Create the Transformer

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

        self.decoder = Decoder(num_layers, d_model, num_heads, dff,
                               target_vocab_size, rate)
        self.final_layer = layers.Dense(target_vocab_size)

    def call(self, inp, tar, training, enc_padding_mask, look_ahead_mask,
             dec_padding_mask):
        # inp.shape ==  (batch_size, tar_seq_len, target_vocab_size)

        # (batch_size, tar_seq_len, target_vocab_size)
        enc_output = self.encoder(inp, training, enc_padding_mask)

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

temp_input = tf.random.uniform((64, 62))
temp_target = tf.random.uniform((64, 26))

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

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

```
TensorShape([64, 26, 8000])
```

10 Set hyperparameters

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

input_vocab_size = tokenizer_pt.vocab_size + 2
target_vocab_size = tokenizer_en.vocab_size + 2
dropout_rate = 0.1
```

11 Optimizer

custom Adam optimizer ref. `https://arxiv.org/abs/1706.03762` (Attention is All You Need) This formula is here.

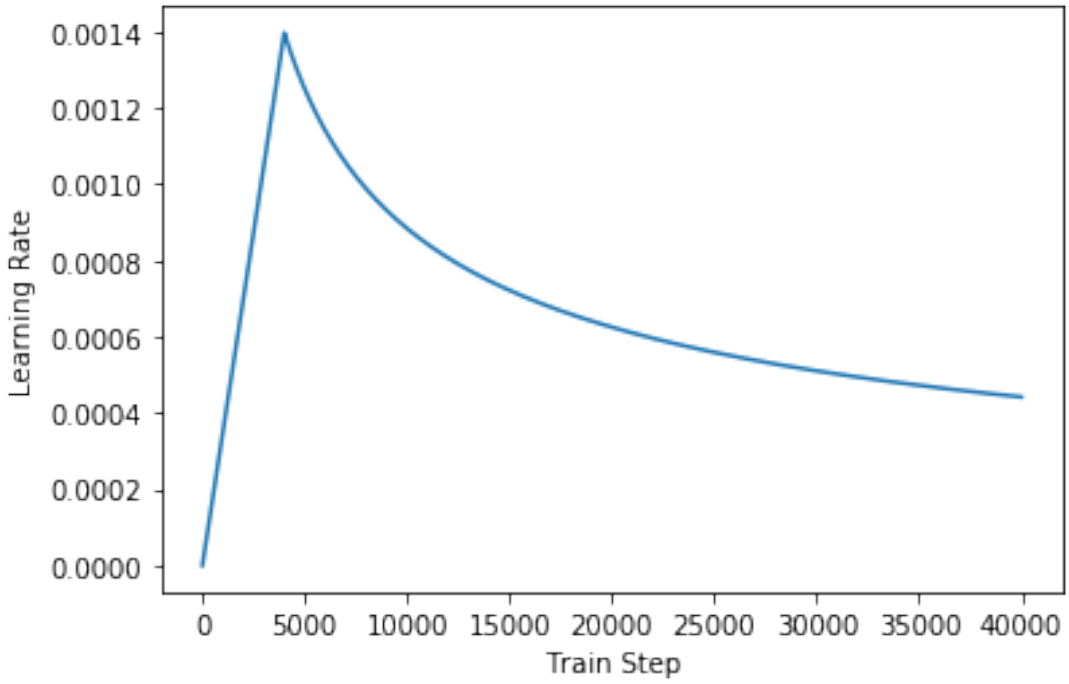
$$lrate = d_{model}^{-0.5} * min(step\_num^{-0.5}, step\_num * warmup\_steps^{-1.5})$$

```
class CustomSchedule(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 = step**-0.5
        arg2 = step * (self.warmup_steps**-1.5)
        return (self.d_model**-0.5) * tf.math.minimum(arg1, arg2)
```

```
learning_rate = CustomSchedule(d_model)
optimizer = 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')



12 Loss and metrics

```
loss_object = keras.losses.SparseCategoricalCrossentropy(from_logits=True,
                                                          reduction='none')


def loss_function(real, pred):
    # 0 == padding
    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_)


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

13 Training and checkpointing

```
transformer = Transformer(num_layers, d_model, num_heads, dff,
                          input_vocab_size, target_vocab_size,
                          dropout_rate)
```

```
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
```

```
checkpoint_path = "./checkpoints/train"

ckpt = tf.train.Checkpoint(transformer=transformer, optimizer=optimizer)
ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path,
                                          max_to_keep=5)

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

```
EPOCHS = 20
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_acc(tar_real, predictions)
```

```
# trainining
for epoch in range(EPOCHS):
    start = time.time()
    train_loss.reset_states()
    train_acc.reset_states()
    # inp -> portuguess, tar -> english
    for (batch, (inp, tar)) in enumerate(train_dataset):
        train_step(inp, tar)
        if batch % 50 == 0:
            print('Epoch {} Batch {} Loss {:.4f} Acc {:.4f}'.format(
                epoch + 1, batch, train_loss.result(), train_acc.result()))
    if (epoch + 1) % 5 == 0:
        ckpt_save_path = ckpt_manager.save()
        print('Saving checkpoint for epoch {} at {}'.format(
            epoch + 1, ckpt_save_path))
        print('Epoch {} Loss {:.4f} Accuracy {:.4f}'.format(
            epoch + 1, train_loss.result(), train_acc.result()))
        print('Time taken for 1 epoch: {} secs\n'.format(time.time() -
                                                         start))
```

Epoch 20 Batch 700 Loss 0.4377 Acc 0.3641
Saving checkpoint for epoch 20 at ./checkpoints/train/ckpt-4
Epoch 20 Loss 0.4379 Accuracy 0.3641
Time taken for 1 epoch: 229.03856348991394 secs

14 Evaluation

```
def evaluate(inp_sentence):
    start_token = [tokenizer_pt.vocab_size]
```

```
    end_token = [tokenizer_pt.vocab_size + 1]

    # inp sentence is portuguese, hence adding the start and end token
    inp_sentence = start_token + tokenizer_pt.encode(inp_sentence) +
    ↪ end_token
    encoder_input = tf.expand_dims(inp_sentence, 0)

    # as the target is english, the first word to the transformer should be
    ↪ the
    # english start token.
    decoder_input = [tokenizer_en.vocab_size]
    output = tf.expand_dims(decoder_input, 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.cast(tf.argmax(predictions, axis=-1), tf.int32)

        # return the result if the predicted_id is equal to the end token
        if predicted_id == tokenizer_en.vocab_size + 1:
            return tf.squeeze(output, axis=0), attention_weights

        # 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 tf.squeeze(output, axis=0), attention_weights
```

```
def plot_attention_weights(attention, sentence, result, layer):
    fig = plt.figure(figsize=(16, 8))

    sentence = tokenizer_pt.encode(sentence)

    attention = tf.squeeze(attention[layer], axis=0)

    for head in range(attention.shape[0]):
```

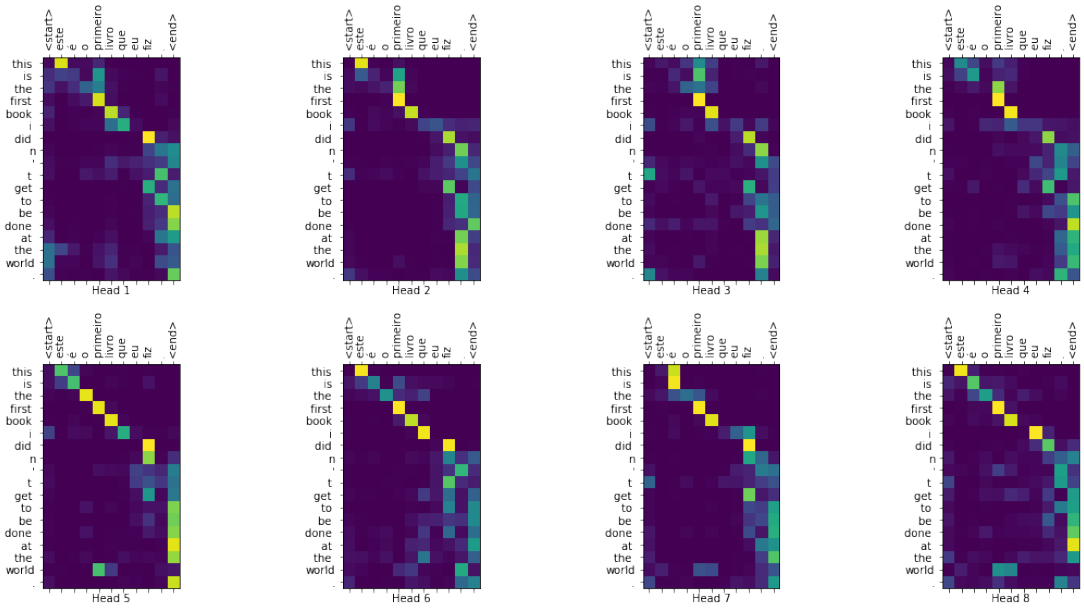
```
        ax = fig.add_subplot(2, 4, head + 1)

        # plot the attention weights
        ax.matshow(attention[head][:-1, :], cmap='viridis')

        fontdict = {'fontsize': 10}

        ax.set_xticks(range(len(sentence) + 2))
        ax.set_yticks(range(len(result)))

        ax.set_ylim(len(result) - 1.5, -0.5)

        ax.set_xticklabels(['<start>'] +
                           [tokenizer_pt.decode([i])
                            for i in sentence] + ['<end>'],
                           fontdict=fontdict,
                           rotation=90)

        ax.set_yticklabels([
            tokenizer_en.decode([i])
            for i in result if i < tokenizer_en.vocab_size
        ],
                           fontdict=fontdict)

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

    plt.tight_layout()
    plt.show()
```

```
def translate(sentence, plot=''):
    result, attention_weights = evaluate(sentence)

    predicted_sentence = tokenizer_en.decode(
        [i for i in result if i < tokenizer_en.vocab_size])

    print('Input: {}'.format(sentence))
    print('Predicted translation: {}'.format(predicted_sentence))

    if plot:
        plot_attention_weights(attention_weights, sentence, result, plot)
```

decoder layer's multi-head attention (num _ heads == 8)

```
translate("este ^^c3^^a9 o primeiro livro que eu fiz.",
 ↪  plot='decoder_layer4_block2')
print("Real translation: this is the first book i've ever done.")
```

Input: este é o primeiro livro que eu fiz.
Predicted translation: this is the first book i did n’t get to be done at the world .



Real translation: this is the first book i’ve ever done.