Rapport NLP TP5

Lasheb Mohamed Amine - iasd2

Modèle de transformateur pour la compréhension du langage

---

un modèle Transformer pour traduire un jeu de données portugais vers anglais .
Il s'agit d'un exemple avancé qui suppose une connaissance de la génération de texte et de l'attention .

L'idée centrale derrière le modèle Transformer est *l'auto-attention* - la capacité de s'occuper de différentes positions de la séquence d'entrée pour calculer une représentation de cette séquence. Transformer crée des piles de couches d'auto-attention et est expliqué ci-dessous dans les sections Attention au *produit scalaire mis à l'échelle et Attention* à *plusieurs têtes* .

Un modèle de transformateur gère les entrées de taille variable en utilisant des piles de couches d'auto-attention au lieu de RNN ou de CNN . Cette architecture générale présente plusieurs avantages :

- Il ne fait aucune hypothèse sur les relations temporelles/spatiales entre les données. C'est idéal pour traiter un ensemble d'objets (par exemple, des unités StarCraft).
- Les sorties de couche peuvent être calculées en parallèle, au lieu d'une série comme un RNN.
- Les éléments distants peuvent affecter la sortie les uns des autres sans passer par de nombreuses étapes RNN ou couches de convolution (voir Scene Memory Transformer par exemple).
- Il peut apprendre des dépendances à longue portée. C'est un défi dans de nombreuses tâches de séquence.

Les inconvénients de cette architecture sont :

- Pour une série chronologique, la sortie d'un pas de temps est calculée à partir de l' *historique complet* au lieu des seules entrées et de l'état caché actuel. Cela *peut* être moins efficace.
- Si l'entrée a une relation temporelle/spatiale , comme le texte, un encodage positionnel doit être ajouté ou le modèle verra effectivement un sac de mots.

Installer

```
!pip install tensorflow_datasets

!pip install -U tensorflow-text

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

import tensorflow_datasets as tfds

import tensorflow_text as text

import tensorflow as tf
```

Télécharger le jeu de données

Cet ensemble de données contient environ 50 000 exemples de formation, 1 100 exemples de validation et 2 000 exemples de test.

```
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_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'))
```

console :

mas e se estes fatores fossem ativos ?

but what if it were active ?

Tokénisation et détokénisation de texte

Nous pouvons pas entraîner un modèle directement sur du texte. Le texte doit d'abord être converti en une représentation numérique. En règle générale, nousconvertissons le texte en séquences d'ID de jeton, qui sont utilisées comme indices dans une incorporation.

Une implémentation populaire est démontrée dans le [didacticiel Subword tokenizer] construit des tokenizers de sous-mots (`text.BertTokenizer`) optimisés pour cet ensemble de données et les exporte dans un save_model

```
model_name = "ted_hrlr_translate_pt_en_converter"
tf.keras.utils.get_file(
    f"{model_name}.zip",
f"https://storage.googleapis.com/download.tensorflow.org/models/{model_name}.zip",
    cache_dir='.', cache_subdir='', extract=True
)
```

```
tokenizers = tf.saved_model.load(model_name)
```

Le `tf.saved_model` contient deux tokenizers de texte, un pour l'anglais et un pour le portugais. Les deux ont les mêmes méthodes :

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

-La méthode `tokenize` convertit un lot de chaînes en un lot rembourré d'ID de jeton. Cette méthode divise la ponctuation, les minuscules et normalise l'entrée en unicode avant la segmentation. Cette standardisation n'est pas visible ici car les données d'entrée sont déjà standardisées.

** 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 .

** [2, 72, 117, 79, 1259, 1491, 2362, 13, 79, 150, 184, 311, 71, 103, 2308, 74, 2679, 13, 148, 80, 55, 4840, 1434, 2423, 540, 15, 3] [2, 87, 90, 107, 76, 129, 1852, 30, 3] [2, 87, 83, 149, 50, 9, 56, 664, 85, 2512, 15, 3]

-La méthode `detokenize` tente de reconvertir ces ID de jeton en texte lisible

La méthode de `lookup` de niveau inférieur convertit les ID de jeton en texte de jeton

<tf.RaggedTensor [[b'[START]', b'and', b'when', b'you', b'improve', b'search', b'##ability', b',', b'you', b'actually', b'take', b'away', b'the', b'one', b'advantage', b'of', b'print', b',', b'which', b'is', b's', b'##ere', b'##nd', b'##ip', b'##ity', b'.', b'[END]'] , [b'[START]', b'but', b'what', b'if', b'it', b'were', b'active', b'?', b'[END]'] , [b'[START]', b'but', b'they', b'did', b'n', b"'", b't', b'test', b'for',]>

Ici pouvons voir l'aspect "sous-mot" des tokenizers. Le mot "searchability" est décomposé en "search ##ability" et le mot "serendipity" en "s ##ere ##nd ##ip ##ity" .

Configurer le pipeline d'entrée

Pour créer un pipeline d'entrée adapté à la formation, nous appliquerons certaines transformations à l'ensemble de données. Cette fonction servira à encoder les lots de texte brut :

```
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
```

Voici un pipeline d'entrée simple qui traite, mélange et regroupe les données :

```
BUFFER_SIZE = 20000

BATCH_SIZE = 64
```

```
def make_batches(ds):

  return (

      ds

      .cache()

      .shuffle(BUFFER_SIZE)

      .batch(BATCH_SIZE)

      .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)
```

Encodage positionnel

Les couches d'attention voient leur entrée comme un ensemble de vecteurs, sans ordre séquentiel. Ce modèle ne contient pas non plus de couches récurrentes ou

convolutionnelles. Pour cette raison, un "codage positionnel" est ajouté pour donner au modèle des informations sur la position relative des jetons dans la phrase.

Le vecteur de codage positionnel est ajouté au vecteur d'intégration. Les incorporations représentent un jeton dans un espace de dimension d où les jetons ayant une signification similaire seront plus proches les uns des autres. Mais les plongements n'encodent pas la position relative des jetons dans une phrase. Ainsi, après avoir ajouté le codage positionnel, les jetons seront plus proches les uns des autres en fonction de la *similitude de leur sens et de leur position dans la phrase* , dans l'espace d-dimensionnel.

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

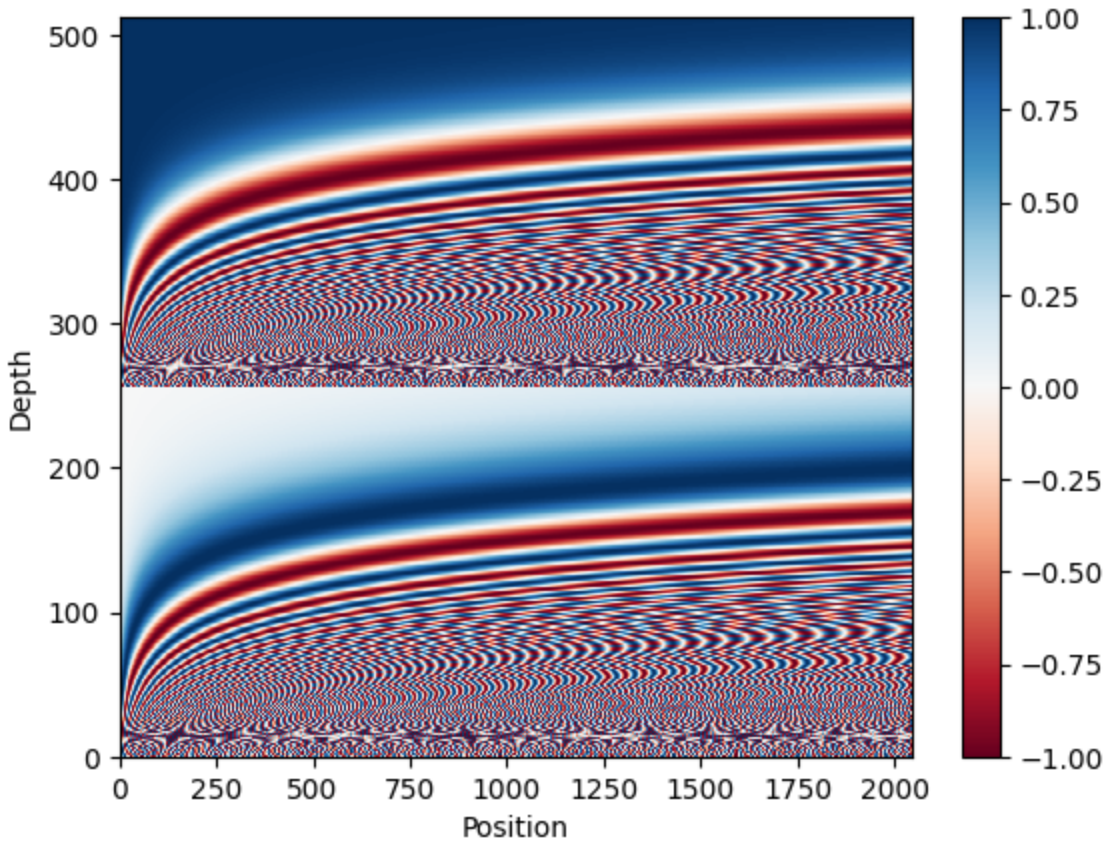
```
def positional_encoding(position, d_model):
  angle_rads = get_angles(np.arange(position)[:, np.newaxis],
                          np.arange(d_model)[np.newaxis, :],
                          d_model)

  # 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)



Masquage

Masquons tous les jetons de pad dans le lot de séquence. Cela garantit que le modèle ne traite pas le rembourrage comme entrée. Le masque indique où la valeur de pad `0` est présente : il sort un `1` à ces emplacements, et un `0` sinon.

```
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)>

Le masque d'anticipation est utilisé pour masquer les futurs jetons dans une séquence. En d'autres termes, le masque indique quelles entrées ne doivent pas être utilisées.

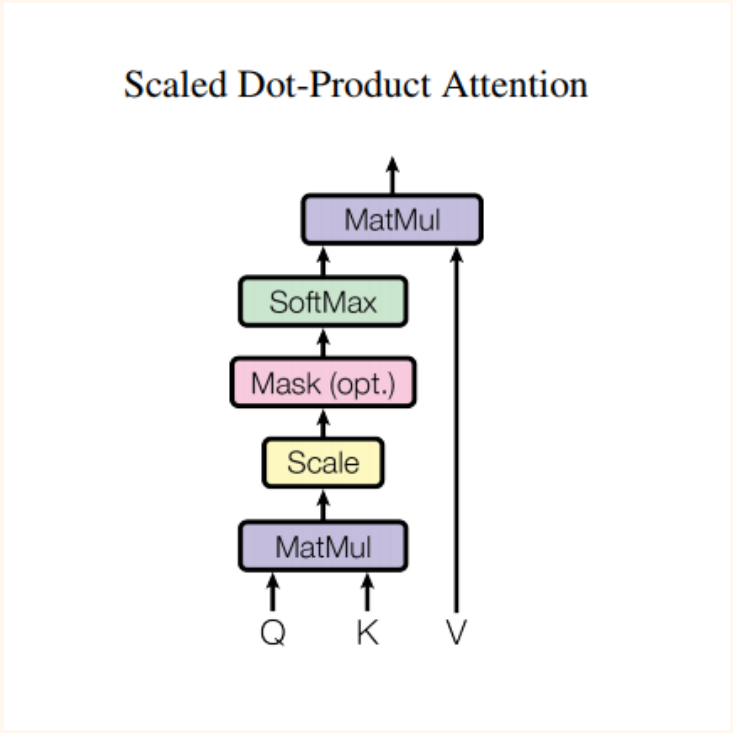
Cela signifie que pour prédire le troisième jeton, seuls le premier et le deuxième jeton seront utilisés. De même pour prédire le quatrième jeton, seuls les premier, deuxième et troisième jetons seront utilisés et ainsi de suite.

```
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)>

Attention au produit scalaire mis à l'échelle



La fonction d'attention utilisée par le transformateur prend trois entrées : Q (requête), K (clé), V (valeur).

L'attention du produit scalaire est mise à l'échelle par un facteur de racine carrée de la profondeur. Ceci est fait parce que pour de grandes valeurs de profondeur, le produit scalaire devient important en poussant la fonction softmax où il a de petits gradients résultant en un softmax très dur.

Par exemple, considérons que Q et K ont une moyenne de 0 et une variance de 1. Leur multiplication matricielle aura une moyenne de 0 et une variance de dk . Ainsi, la *racine carrée de dk* est utilisée pour la mise à l'échelle, de sorte que obtenons une variance cohérente quelle que soit la valeur de dk . Si la variance est trop faible, la sortie peut être trop plate pour être optimisée efficacement. Si la variance est trop élevée, le softmax peut saturer à l'initialisation, ce qui rend l'apprentissage difficile.

Le masque est multiplié par -1e9 (proche de moins l'infini). Ceci est fait parce que le masque est additionné avec la multiplication matricielle mise à l'échelle de Q et K et est appliqué immédiatement avant un softmax. L'objectif est de mettre à zéro ces cellules, et les grandes entrées négatives de softmax sont proches de zéro dans la sortie.

```
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
```

Comme la normalisation softmax est effectuée sur K, ses valeurs décident de l'importance accordée à Q.

La sortie représente la multiplication des poids d'attention et du vecteur V (valeur). Cela garantit que les jetons sur lesquels souhaitons nous concentrer sont conservés tels quels et que les jetons non pertinents sont éliminés.

```
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)
```

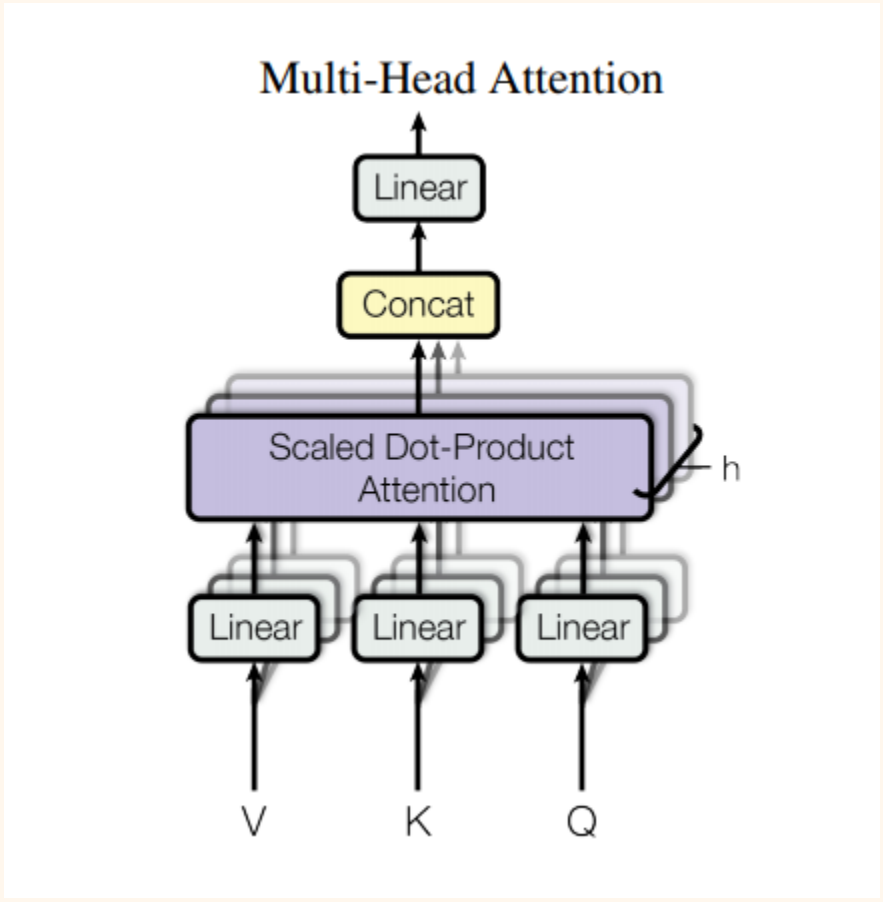
Attention weights are:

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)

Attention multi-tête



L'attention multi-tête se compose de quatre parties :

- Couches linéaires.
- Attention mise à l'échelle du produit scalaire.
- Couche linéaire finale.

Chaque bloc d'attention multi-tête reçoit trois entrées ; Q (requête), K (clé), V (valeur). Ceux-ci sont placés à travers des couches linéaires (Dense) avant la fonction d'attention multi-tête.

Dans le diagramme ci-dessus `(K,Q,V)` sont passés à travers des couches linéaires séparées (`Dense`) pour chaque tête d'attention. Pour plus de simplicité/d'efficacité, le code ci-dessous implémente cela en utilisant une seule couche dense avec `num_heads` fois autant de sorties. La sortie est réorganisée en une forme de `(batch, num_heads, ...)` avant d'appliquer la fonction d'attention.

La fonction `scaled_dot_product_attention` définie ci-dessus est appliquée en un seul appel, diffusé pour plus d'efficacité. Un masque approprié doit être utilisé dans l'étape d'attention. La sortie d'attention pour chaque tête est ensuite concaténée (à l'aide `tf.transpose` et `tf.reshape`) et soumise à une couche `Dense` finale.

Au lieu d'une seule tête d'attention, Q, K et V sont divisés en plusieurs têtes car cela permet au modèle de s'occuper conjointement des informations de différents sous-espaces de représentation à différentes positions. Après la scission, chaque tête a une dimensionnalité réduite, de sorte que le coût de calcul total est le même que celui d'une seule tête avec une dimensionnalité complète.

```
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):

    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
```

Créons une couche `MultiHeadAttention` pour l'essayer. À chaque emplacement de la séquence, `y` , le `MultiHeadAttention` exécute les 8 têtes d'attention sur tous les autres emplacements de la séquence, renvoyant un nouveau vecteur de la même longueur à chaque emplacement.

```
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]))

Réseau d'alimentation en aval point par point

Le réseau d'alimentation ponctuelle se compose de deux couches entièrement connectées avec une activation ReLU entre les deux.

```
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)
  ])
```

exemple : dimension d'entrée (d_model) de 512 et une taille de couche cachée (dff) de 2048

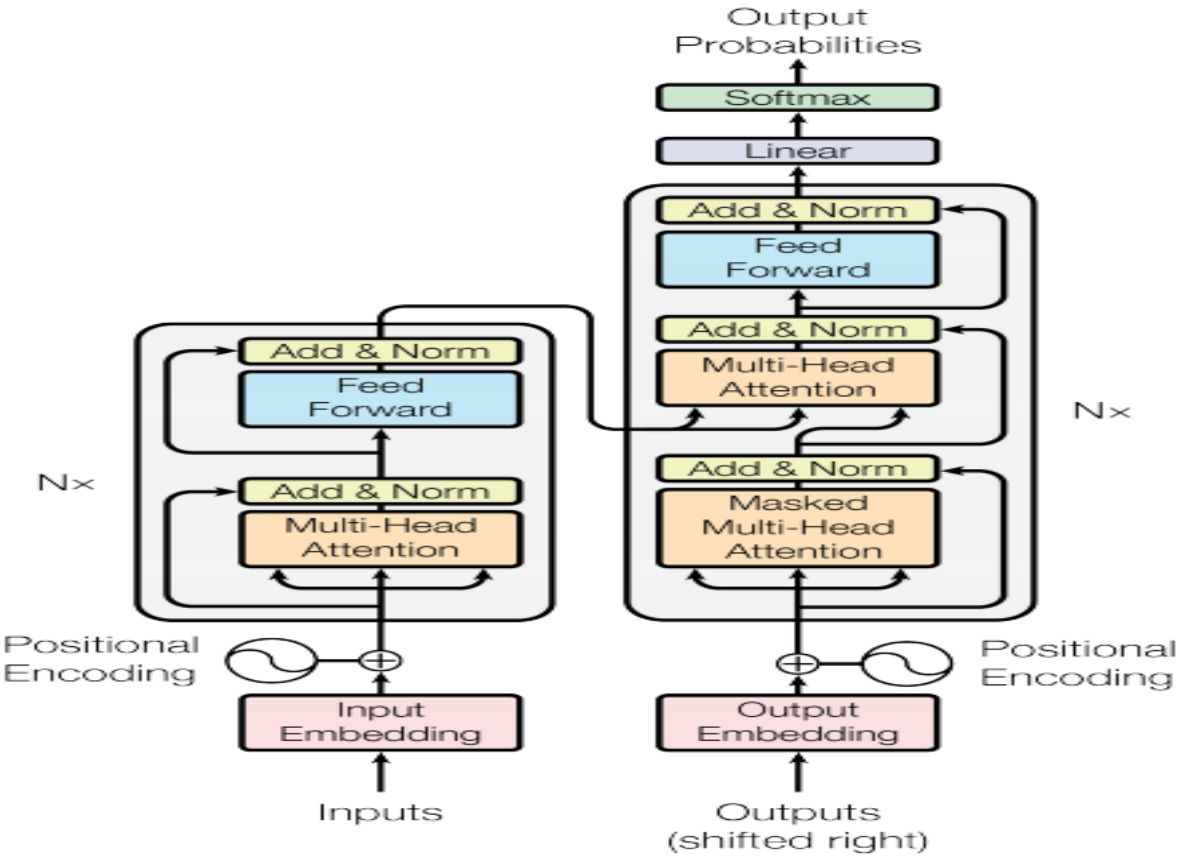
```
sample_ffn = point_wise_feed_forward_network(512, 2048)
sample_ffn(tf.random.uniform((64, 50, 512))).shape
```

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

Encodeur et décodeur

Le modèle de transformateur suit le même schéma général qu'un modèle standard de séquence à séquence avec attention .

- La phrase d'entrée est transmise à travers `N` couches d'encodeur qui génèrent une sortie pour chaque jeton de la séquence.
- Le décodeur s'occupe de la sortie de l'encodeur et de sa propre entrée (auto-attention) pour prédire le mot suivant.



Couche d'encodeur

Chaque couche d'encodeur se compose de sous-couches :

1. Attention multi-tête (avec masque de rembourrage)
2. Réseaux d'alimentation point par point.

Chacune de ces sous-couches est entourée d'une connexion résiduelle suivie d'une normalisation de couche. Les connexions résiduelles aident à éviter le problème du gradient de fuite dans les réseaux profonds.

La sortie de chaque sous-couche est `LayerNorm(x + Sublayer(x))`. La normalisation se fait sur l' `d_model` (dernier). Il y a N couches d'encodeur dans le transformateur.

```
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])

Couche décodeur

Chaque couche de décodeur se compose de sous-couches :

1. Attention masquée à plusieurs têtes (avec masque d'anticipation et masque de rembourrage)
2. Attention multi-tête (avec masque de rembourrage). V (valeur) et K (clé) reçoivent la *sortie du codeur* comme entrées. Q (requête) reçoit la *sortie de la sous-couche d'attention masquée à plusieurs têtes.*
3. Réseaux d'alimentation en aval point par point

Chacune de ces sous-couches est entourée d'une connexion résiduelle suivie d'une normalisation de couche. La sortie de chaque sous-couche est `LayerNorm(x + Sublayer(x))` . La normalisation se fait sur l' `d_model` (dernier).

Il y a N couches de décodeur dans le transformateur.

Lorsque Q reçoit la sortie du premier bloc d'attention du décodeur et que K reçoit la sortie de l'encodeur, les poids d'attention représentent l'importance accordée à l'entrée du décodeur sur la base de la sortie de l'encodeur. En d'autres termes, le décodeur prédit le jeton suivant en regardant la sortie de l'encodeur et en s'occupant lui-même de sa propre sortie. Voir la démonstration ci-dessus dans la section Attention aux produits scalaires mis à l'échelle.

```
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])

Encodeur

L'`Encoder` se compose de :

1. Incorporation d'entrée
2. Encodage positionnel
3. N couches d'encodeur

L'entrée est soumise à une intégration qui est additionnée au codage positionnel. La sortie de cette sommation est l'entrée des couches d'encodeur. La sortie du codeur est l'entrée du décodeur.

```
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



    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)
```

(64, 62, 512)

Décodeur

Le `Decoder` se compose de :

1. Incorporation de sortie

2. Encodage positionnel
3. N couches de décodeur

La cible est soumise à une imbrication qui est sommée avec le codage positionnel. La sortie de cette sommation est l'entrée des couches du décodeur. La sortie du décodeur est l'entrée de la couche linéaire finale.

```
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[f'decoder_layer{i+1}_block1'] = block1
      attention_weights[f'decoder_layer{i+1}_block2'] = block2
```

```
    # x.shape == (batch_size, target_seq_len, d_model)

    return x, attention_weights
```

Créer le transformateur

Le transformateur se compose de l'encodeur, du décodeur et d'une couche linéaire finale. La sortie du décodeur est l'entrée de la couche linéaire et sa sortie est renvoyée.

```
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().__init__()

    self.encoder = 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, inputs, training):

    # Keras models prefer if you pass all your inputs in the first
argument

    inp, tar = inputs



    enc_padding_mask, look_ahead_mask, dec_padding_mask =
self.create_masks(inp, tar)



    enc_output = self.encoder(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



  def create_masks(self, 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)
    look_ahead_mask = tf.maximum(dec_target_padding_mask, look_ahead_mask)

    return enc_padding_mask, look_ahead_mask, dec_padding_mask
```

```
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)



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

TensorShape([64, 36, 8000])

Définir des hyperparamètres

Pour que cet exemple reste petit et relativement rapide, les valeurs de `num_layers, d_model, dff` ont été réduites.

Le modèle de base utilisé : `num_layers=6, d_model=512, dff=2048` .

```
num_layers = 4

d_model = 128

dff = 512

num_heads = 8

dropout_rate = 0.1
```

num_layers : Il s'agit du nombre de couches dans le modèle de transformer. Chaque couche est composée de plusieurs sous-couches qui effectuent des opérations de transformation sur les données d'entrée.

d_model : C'est la dimension des vecteurs d'entrée et de sortie dans le modèle de transformer. Il s'agit également de la dimension des vecteurs d'attention. Par exemple, si d_model est fixé à 128, cela signifie que chaque vecteur d'entrée et de sortie aura une dimension de 128.

dff : Il s'agit de la taille de la couche cachée (ou couche intermédiaire) dans le réseau d'alimentation en aval point par point. Cette couche joue un rôle important dans la transformation non linéaire des caractéristiques extraites par le modèle. num_heads : Le mécanisme d'attention multi-têtes permet au modèle de transformer de capturer des relations à différentes échelles en parallèle.

num_heads représente le nombre de têtes d'attention utilisées dans chaque couche du modèle. Plus le nombre de têtes est élevé, plus le modèle peut capturer des motifs complexes et fins dans les données.

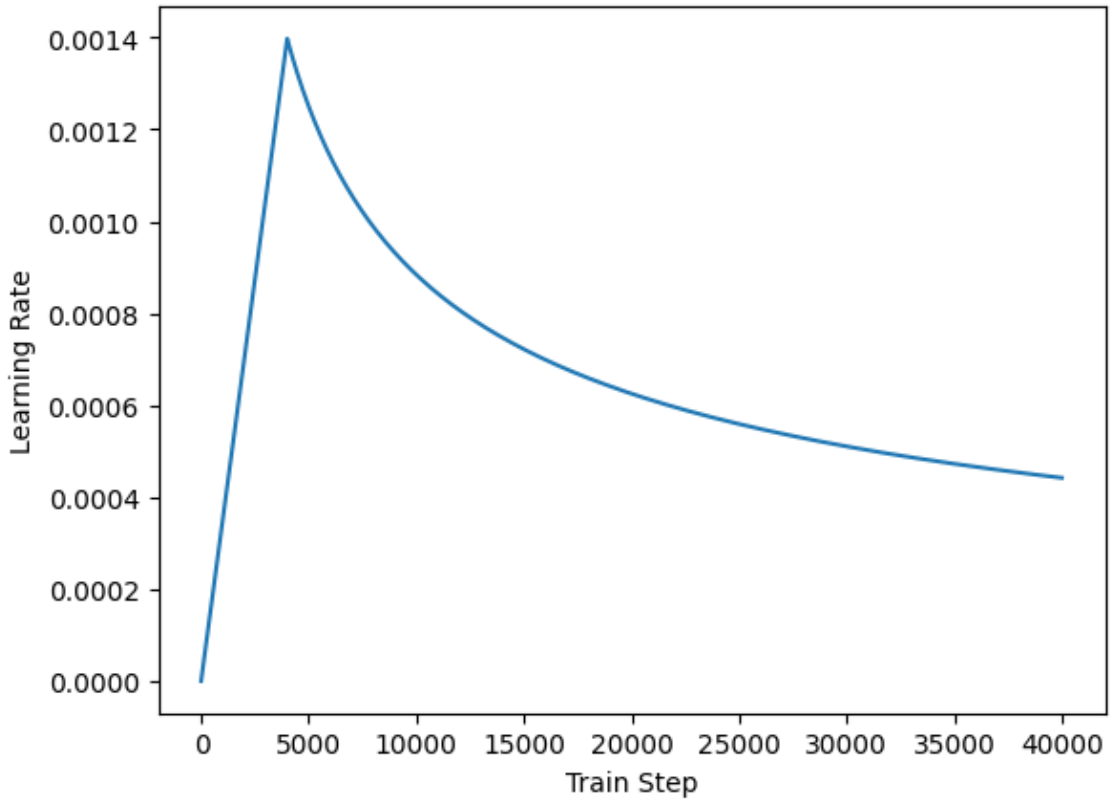
dropout_rate : Le dropout est une technique de régularisation qui consiste à aléatoirement "ignorer" (en les mettant à zéro) certains neurones pendant l'entraînement, ce qui aide à prévenir le surapprentissage. Le dropout_rate spécifie la probabilité de mise à zéro d'un neurone lors de l'entraînement.

Optimiseur

Utilisons l'optimiseur Adam avec un planificateur de taux d'apprentissage personnalisé

```
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(0.0008, beta_1=0.9,
beta_2=0.98,epsilon=1e-9)

# 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")
```



Perte et mesures

Étant donné que les séquences cibles sont rembourrées, il est important d'appliquer un masque de remplissage lors du calcul de la perte.

```
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')
```

Formation et contrôle

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

Créons le chemin du point de contrôle et le gestionnaire de points de contrôle. Cela sera utilisé pour enregistrer des points de contrôle toutes les n époques.

```
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!!')
```

La cible est divisée en tar_inp et tar_real. tar_inp est passé en entrée au décodeur. `tar_real` est la même entrée décalée de 1 : à chaque emplacement dans `tar_input` , `tar_real` contient le jeton suivant qui doit être prédit.

Par exemple, `sentence` = "SOS Un lion dans la jungle dort EOS"

`tar_inp` = "SOS Un lion dans la jungle dort"

`tar_real` = "Un lion dans la jungle dort EOS"

Le transformateur est un modèle auto-régressif : il fait des prédictions une partie à la fois et utilise sa sortie jusqu'à présent pour décider quoi faire ensuite.

Pendant la formation, cet exemple utilise le forçage de l'enseignant (comme dans le didacticiel de génération de texte). Le forçage de l'enseignant transmet la vraie sortie au pas de temps suivant, indépendamment de ce que le modèle prédit au pas de temps actuel.

Au fur et à mesure que le transformateur prédit chaque jeton, *l'auto-attention* lui permet de regarder les jetons précédents dans la séquence d'entrée pour mieux prédire le jeton suivant.

Pour empêcher le modèle de jeter un coup d'œil à la sortie attendue, le modèle utilise un masque d'anticipation.

```
EPOCHS = 100
```

```
# 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:]

  with tf.GradientTape() as tape:
    predictions, _ = transformer([inp, tar_inp],
                                 training = True)
    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))
```

Le portugais est utilisé comme langue d'entrée et l'anglais est la langue cible.

```
import time
start_time = time.time()
```

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

  train_loss.reset_states()
  train_accuracy.reset_states()

  # inp -> portuguese, tar -> english
  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}')

  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 100 Batch 800 Loss 1.0375 Accuracy 0.7454
Saving checkpoint for epoch 100 at ./checkpoints/train/ckpt-20
Epoch 100 Loss 1.0379 Accuracy 0.7453
Time taken for 1 epoch: 95.38 secs
```

```
end_time = time.time()
training_time = end_time - start_time
```

```
minutes = training_time // 60
seconds = training_time % 60

print("Training time: {} minutes {:.2f} seconds".format(int(minutes),
      seconds))
```

```
Training time: 160 minutes 8.84 seconds
```

Exécuter l'inférence

Les étapes suivantes sont utilisées pour l'inférence :

- Encodons la phrase d'entrée à l'aide du tokenizer portugais (`tokenizers.pt`). C'est l'entrée de l'encodeur.
- L'entrée du décodeur est initialisée au jeton `[START]` .
- Calculons les masques de remplissage et les masques d'anticipation.
- Le `decoder` produit ensuite les prédictions en regardant la `encoder output` et sa propre sortie (auto-attention).
- Concaténons le jeton prédit à l'entrée du décodeur et transmettons-le au décodeur.
- Dans cette approche, le décodeur prédit le jeton suivant sur la base des jetons précédents qu'il a prédits.

```
class Translator(tf.Module):
  def __init__(self, tokenizers, transformer):
    self.tokenizers = tokenizers
    self.transformer = transformer

  def __call__(self, sentence, max_length=20):
    # input sentence is portuguese, hence adding the start and end token
    assert isinstance(sentence, tf.Tensor)
    if len(sentence.shape) == 0:
      sentence = sentence[tf.newaxis]

    sentence = self.tokenizers.pt.tokenize(sentence).to_tensor()

    encoder_input = sentence

    # as the target is english, the first token to the transformer should
      be the
    # english start token.
    start_end = self.tokenizers.en.tokenize([''])[0]
    start = start_end[0][tf.newaxis]
    end = start_end[1][tf.newaxis]

    # `tf.TensorArray` is required here (instead of a python list) so that
      the
    # dynamic-loop can be traced by `tf.function`.
    output_array = tf.TensorArray(dtype=tf.int64, size=0,
      dynamic_size=True)
    output_array = output_array.write(0, start)

    for i in tf.range(max_length):
      output = tf.transpose(output_array.stack())
      predictions, _ = self.transformer([encoder_input, output],
      training=False)

      # select the last token 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_array = output_array.write(i+1, predicted_id[0])

      if predicted_id == end:
        break

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

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

    # `tf.function` prevents us from using the attention_weights that were
    # calculated on the last iteration of the loop. So recalculate them
      outside
    # the loop.
    _, attention_weights = self.transformer([encoder_input,
      output[:,:-1]], training=False)

    return text, tokens, attention_weights
```

Créons une instance de cette classe `Translator` et essayons-la plusieurs fois :

```
translator = Translator(tokenizers, transformer)
```

```
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 = translator(
    tf.constant(sentence))
print_translation(sentence, translated_text, ground_truth)
```

Input: : este é um problema que temos que resolver.

Prediction : this is a problem that 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 = translator(
    tf.constant(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 = translator(
    tf.constant(sentence))
print_translation(sentence, translated_text, ground_truth)
```

Input: : vou então muito rapidamente partilhar convosco algumas histórias de algumas coisas mágicas que aconteceram.

Prediction : so i ' m very quickly sharing with you some stories of some magical things that happened .

Ground truth : so i 'll just share with you some stories very quickly of some magical things that have happened .

Tracés d'attention

La classe `Translator` renvoie un dictionnaire de cartes d'attention que nous pouvons utiliser pour visualiser le fonctionnement interne du modèle :

```
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 = translator(
    tf.constant(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([20, 70])

```
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
```

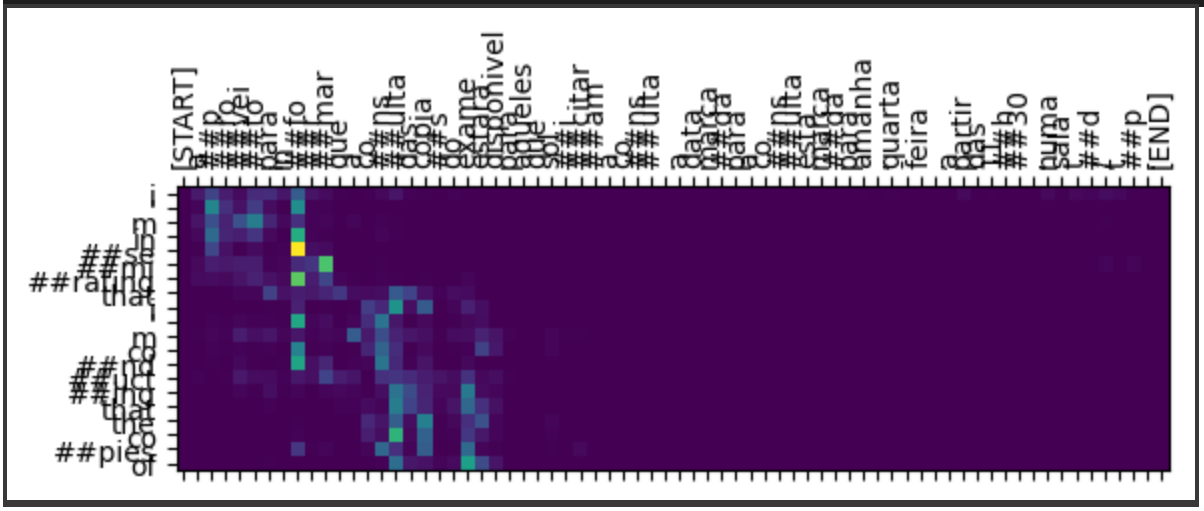
```
<tf.Tensor: shape=(70,), dtype=string, numpy=
array([b'[START]', b'a', b'##p', b'##ro', b'##vei', b'##to', b'para',
       b'in', b'##fo', b'##r', b'##mar', b'que', b'a', b'co', b'##ns',
       b'##ulta', b'das', b'copia', b'##s', b'do', b'exame', b'estara',
       b'disponivel', b'para', b'aqueles', b'que', b'sol', b'##i',
       b'##citar', b'##am', b'a', b'co', b'##ns', b'##ulta', b'.', b'a',
       b'data', b'marca', b'##da', b'para', b'a', b'co', b'##ns',
       b'##ulta', b'esta', b'marca', b'##da', b'para', b'amanha', b',',
       b'quarta', b'-', b'feira', b',', b'a', b'partir', b'das', b'11',
       b'##h', b'##30', b',', b'numa', b'sala', b't', b'##d', b'/', b't',
       b'##p', b'.', b'[END]'], dtype=object)>
```

```
translated_tokens
```

```
translated_tokens
```

```
<tf.Tensor: shape=(21,), dtype=string, numpy=
array([b'[START]', b'i', b"'", b'm', b'in', b'##se', b'##mi', b'##rating',
       b'that', b'i', b"'", b'm', b'co', b'##nd', b'##uct', b'##ing',
       b'that', b'the', b'co', b'##pies', b'of'], dtype=object)>
```

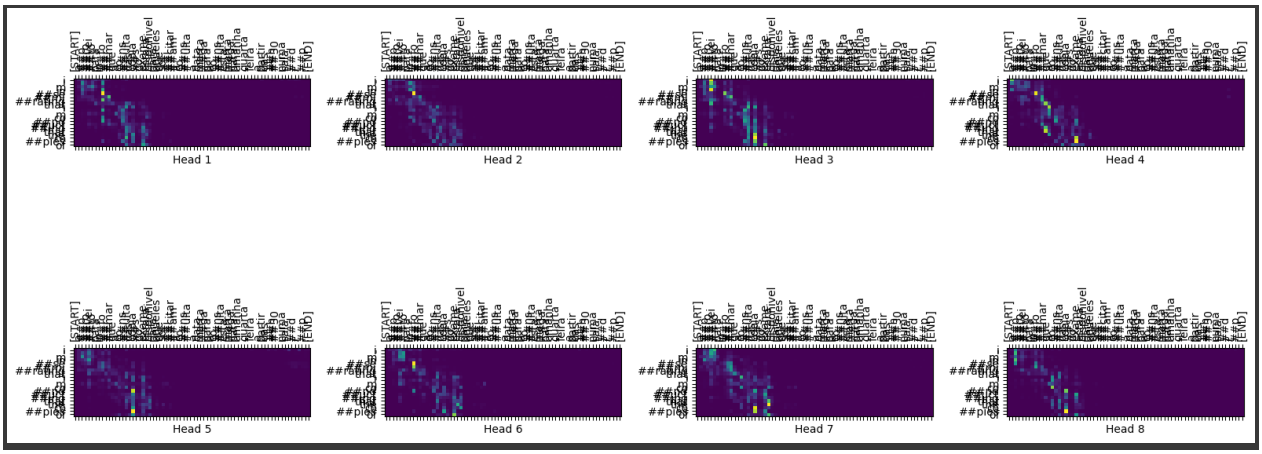
```
plot_attention_head(in_tokens, translated_tokens, attention)
```




```
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(f'Head {h+1}')

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

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



Le modèle fonctionne bien avec des mots inconnus. Ni "tricératops" ni "encyclopédie" ne figurent dans l'ensemble de données d'entrée et le modèle apprend presque à les translittérer, même sans vocabulaire partagé :

```
sentence = "Eu li sobre triceratops na enciclopédia."
ground_truth = "I read about triceratops in the encyclopedia."

translated_text, translated_tokens, attention_weights = translator(
    tf.constant(sentence))
print_translation(sentence, translated_text, ground_truth)

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

Input: : Eu li sobre triceratops na enciclopédia.

Prediction : i read about tartopathos on encyclopedia .

Ground truth : I read about triceratops in the encyclopedia.