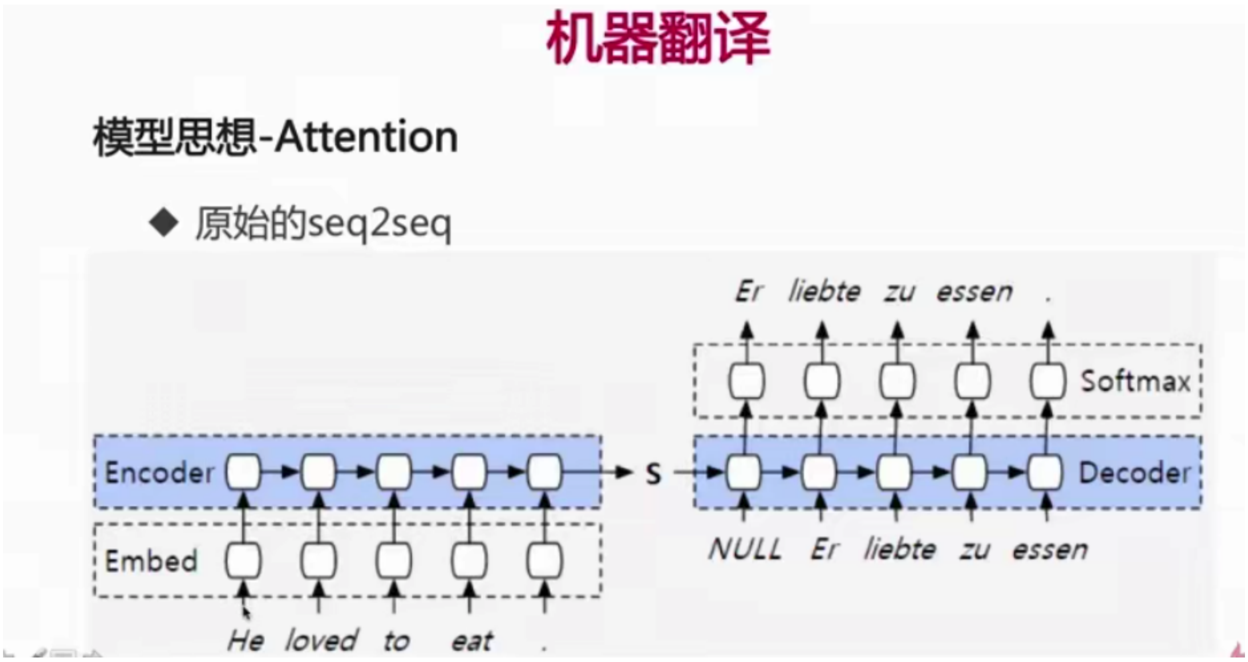
1.Attention + Seq2Seq



机器翻译

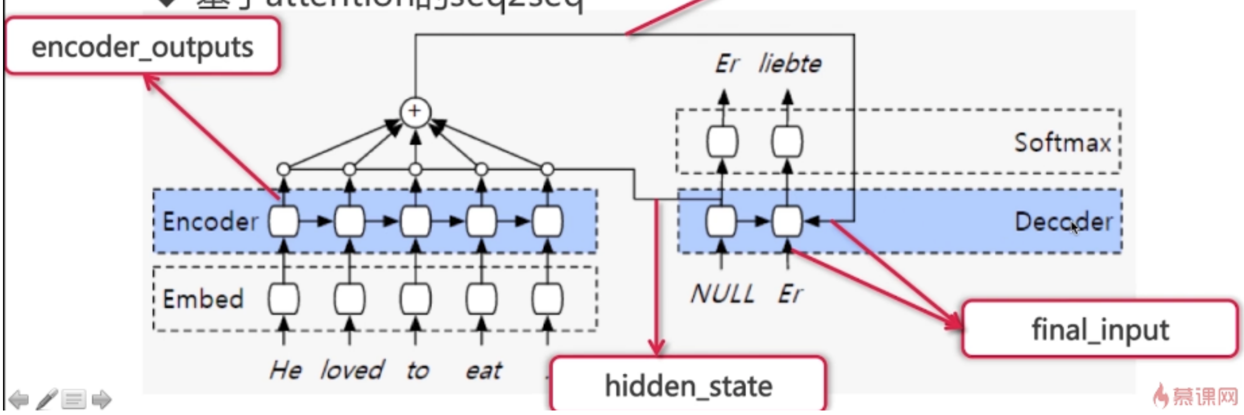
模型思想-Attention

- 原始的seq2seq
  - Encoder-Decoder结构
  - 缺点
    - 定长编码是信息瓶颈
    - 长度越长，前面输入进RNN的信息就越被稀释

机器翻译

模型思想-Attention

- 基于attention的seq2seq



机器翻译

模型思想-Attention

- EO: encoder各个位置的输出
- H: decoder某一步的隐含状态
- FC: 全连接层
- X: decoder的一个输入

- score = FC(tanh(FC(EO) + FC(H))) [Bahdanau注意力]
- 另一选项：score = EO*W*H [luong注意力]
- attention_weights = softmax(score, axis = 1)
- context = sum(attention_weights * EO, axis = 1)
- final_input = concat(context, embed(x))

```python
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import sklearn
import sys
import tensorflow as tf
import time
from tensorflow import keras

# 1. preprocessing data
# 2. build model
# 2.1 encoder
# 2.2 attention
# 2.3 decoder
# 2.4 loss & optimizer
# 2.5 train
# 3. evaluation
# 3.1 give sentence, return translated results
# 3.2 visualize results (attention)

```

```python
"1. preprocessing data"
import unicodedata
import re
from sklearn.model_selection import train_test_split

# 全为unicode，词表过大，转为ascii码，则只有255个字符
# NFD 如果某个unicode由多个ascii码组成，则将其拆开
# Mn为注音
def unicode_to_ascii(s):
    return ''.join(c for c in unicodedata.normalize('NFD', s) if unicodedata.category(c) != 'Mn')

def preprocess_sentence(w):
    w = unicode_to_ascii(w.lower().strip())
    w = re.sub(r"([?.!,¿])", r" \1 ", w)
    # 去除重复空格
    w = re.sub(r'[" "]+', " ", w)
    # replacing everything with space except (a-z, A-Z, ".", "?", "!", ",")
    w = re.sub(r"[^a-zA-Z?.!,¿]+", " ", w)
    # 去除前后空格
    w = w.rstrip().strip()
    # adding a start and an end token to the sentence
    # so that the model know when to start and stop predicting.
    w = '<start> ' + w + ' <end>'
    return w

data_path = './spa.txt'
# 1. Remove the accents
# 2. Clean the sentences
# 3. Return word pairs in the format: [ENGLISH, SPANISH]
def create_dataset(path, num_examples):
    lines = open(path, encoding='UTF-8').read().strip().split('\n')
    word_pairs = [[preprocess_sentence(w) for w in line.split('\t')] for line in lines[:num_examples]]
```

```python
    # 解包 [(1,2),(3,4)] -> [(1,3),(2,4)]
    return zip(*word_pairs)

en, sp = create_dataset(data_path, None)

def max_length(tensor):
    return max(len(t) for t in tensor)

def tokenize(lang):
    lang_tokenizer = tf.keras.preprocessing.text.Tokenizer(filters='',split=' ')
    # 统计词频，生成词表
    lang_tokenizer.fit_on_texts(lang)
    # word -> id
    tensor = lang_tokenizer.texts_to_sequences(lang)
    tensor = tf.keras.preprocessing.sequence.pad_sequences(tensor, padding='post')
    return tensor, lang_tokenizer

def load_dataset(path, num_examples=None):
    # creating cleaned input, output pairs
    targ_lang, inp_lang = create_dataset(path, num_examples)
    input_tensor, inp_lang_tokenizer = tokenize(inp_lang)
    target_tensor, targ_lang_tokenizer = tokenize(targ_lang)
    return input_tensor, target_tensor, inp_lang_tokenizer, targ_lang_tokenizer

# Try experimenting with the size of that dataset
num_examples = 30000
input_tensor, target_tensor, inp_lang, targ_lang = load_dataset(data_path, num_examples)

# Calculate max_length of the target tensors
max_length_targ, max_length_inp = max_length(target_tensor), max_length(input_tensor)

```

```python
# Creating training and validation sets using an 80-20 split
input_tensor_train, input_tensor_val, target_tensor_train, target_tensor_val = train_test_split(input_tensor, target_tensor, test_size=0.2)

# Show length
len(input_tensor_train), len(target_tensor_train), len(input_tensor_val), len(target_tensor_val)

BUFFER_SIZE = len(input_tensor_train)
BATCH_SIZE = 64
steps_per_epoch = len(input_tensor_train)//BATCH_SIZE
embedding_dim = 256
units = 1024
vocab_inp_size = len(inp_lang.word_index)+1
vocab_tar_size = len(targ_lang.word_index)+1

dataset = tf.data.Dataset.from_tensor_slices((input_tensor_train, target_tensor_train)).shuffle(BUFFER_SIZE)
dataset = dataset.batch(BATCH_SIZE, drop_remainder=True)

"2. build model"
"2.1 encoder"
class Encoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, encoding_units, batch_size):
    super(Encoder, self).__init__()
    self.batch_size = batch_size
    self.encoding_units = encoding_units
    self.embedding = keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = keras.layers.GRU(self.encoding_units, return_sequences=True, return_state=True, recurrent_initializer='glorot_uniform')

    def call(self, x, hidden):
    x = self.embedding(x)
```

```
 output, state = self.gru(x, initial_state = hidden)
 return output, state

 def initialize_hidden_state(self):
 return tf.zeros((self.batch_size, self.encoding_units))
encoder = Encoder(vocab_inp_size, embedding_dim, units, BATCH_SIZE)

"2.2 attention"
class BahdanauAttention(tf.keras.Model):
 def __init__(self, units):
 super(BahdanauAttention, self).__init__()
 self.W1 = tf.keras.layers.Dense(units)
 self.W2 = tf.keras.layers.Dense(units)
 self.V = tf.keras.layers.Dense(1)

 def call(self, query, values):
 # hidden shape == (batch_size, hidden size)
 # hidden_with_time_axis shape == (batch_size, 1, hidden size)
 # we are doing this to perform addition to calculate the score
 hidden_with_time_axis = tf.expand_dims(query, 1)

 # score shape == (batch_size, max_length, 1)
 # we get 1 at the last axis because we are applying score to self.V
 # the shape of the tensor before applying self.V is (batch_size, max_length, units)
 score = self.V(tf.nn.tanh(self.W1(values) + self.W2(hidden_with_time_axis)))

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

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

```python

    return context_vector, attention_weights
# attention_layer = BahdanauAttention(10)

"2.3 decoder"
class Decoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, decoding_units, batch_size):
    super(Decoder, self).__init__()
    self.batch_size = batch_size
    self.decoding_units = decoding_units
    self.embedding = keras.layers.Embedding(vocab_size, embedding_dim)
    self.gru = keras.layers.GRU(self.decoding_units, return_sequences=True, return_state=True, recurrent_initializer='glorot_uniform')
    self.fc = keras.layers.Dense(vocab_size)
    # used for attention
    self.attention = BahdanauAttention(self.decoding_units)

    def call(self, x, hidden, encoding_output):
    # enc_output shape == (batch_size, max_length, hidden_size)
    context_vector, attention_weights = self.attention(hidden, encoding_output)

    # x shape after passing through embedding == (batch_size, 1, embedding_dim)
    x = self.embedding(x)

    # x shape after concatenation == (batch_size, 1, embedding_dim + hidden_size)
    x = tf.concat([tf.expand_dims(context_vector, 1), x], axis=-1)

    # passing the concatenated vector to the GRU
    output, state = self.gru(x)

    # output shape == (batch_size * 1, hidden_size)
```

```
 output = tf.reshape(output, (-1, output.shape[2]))

 # output shape == (batch_size, vocab)
 x = self.fc(output)
 return x, state, attention_weights
decoder = Decoder(vocab_tar_size, embedding_dim, units, BATCH_SIZE)

"2.4 loss & optimizer"
optimizer = keras.optimizers.Adam()
# reduction='none': 损失函数如何聚合，此处不聚合
loss_object = keras.losses.SparseCategoricalCrossentropy(from_logits=True, reduction='none')

def loss_function(real, pred):
 # padding 填充的是0
 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_)

checkpoint_dir = './10-1_checkpoints'
if not os.path.exists(checkpoint_dir):
 os.mkdir(checkpoint_dir)
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(optimizer=optimizer, encoder=encoder, decoder=decoder)

"2.5 train"
@tf.function
def train_step(inp, targ, encoding_hidden):
 loss = 0
 with tf.GradientTape() as tape:
 encoding_output, encoding_hidden = encoder(inp, encoding_hidden)
```

```
 decoding_hidden = encoding_hidden
 # eg: <start> i am here <end>
# 1. <start> -> i
# 2. i -> am
# 3. am -> here
# 4. here -> <end>

 # (batch_size,1)
 decoding_input = tf.expand_dims([targ_lang.word_index['<start>']] * BATCH_SIZE, 1)
 # Teacher forcing - feeding the target as the next input
 for t in range(1, targ.shape[1]):
 # passing enc_output to the decoder
 predictions, decoding_hidden, _ = decoder(decoding_input, decoding_hidden, encoding_output)
 loss += loss_function(targ[:, t], predictions)
 # using teacher forcing
 decoding_input = tf.expand_dims(targ[:, t], 1)

 batch_loss = (loss / int(targ.shape[0]))
 variables = encoder.trainable_variables + decoder.trainable_variables
 gradients = tape.gradient(loss, variables)
 optimizer.apply_gradients(zip(gradients, variables))
 return batch_loss

EPOCHS = 10
for epoch in range(EPOCHS):
 start = time.time()
 encoding_hidden = encoder.initialize_hidden_state()
 total_loss = 0
 for (batch, (inp, targ)) in enumerate(dataset.take(steps_per_epoch)):
 batch_loss = train_step(inp, targ, encoding_hidden)
 total_loss += batch_loss
```

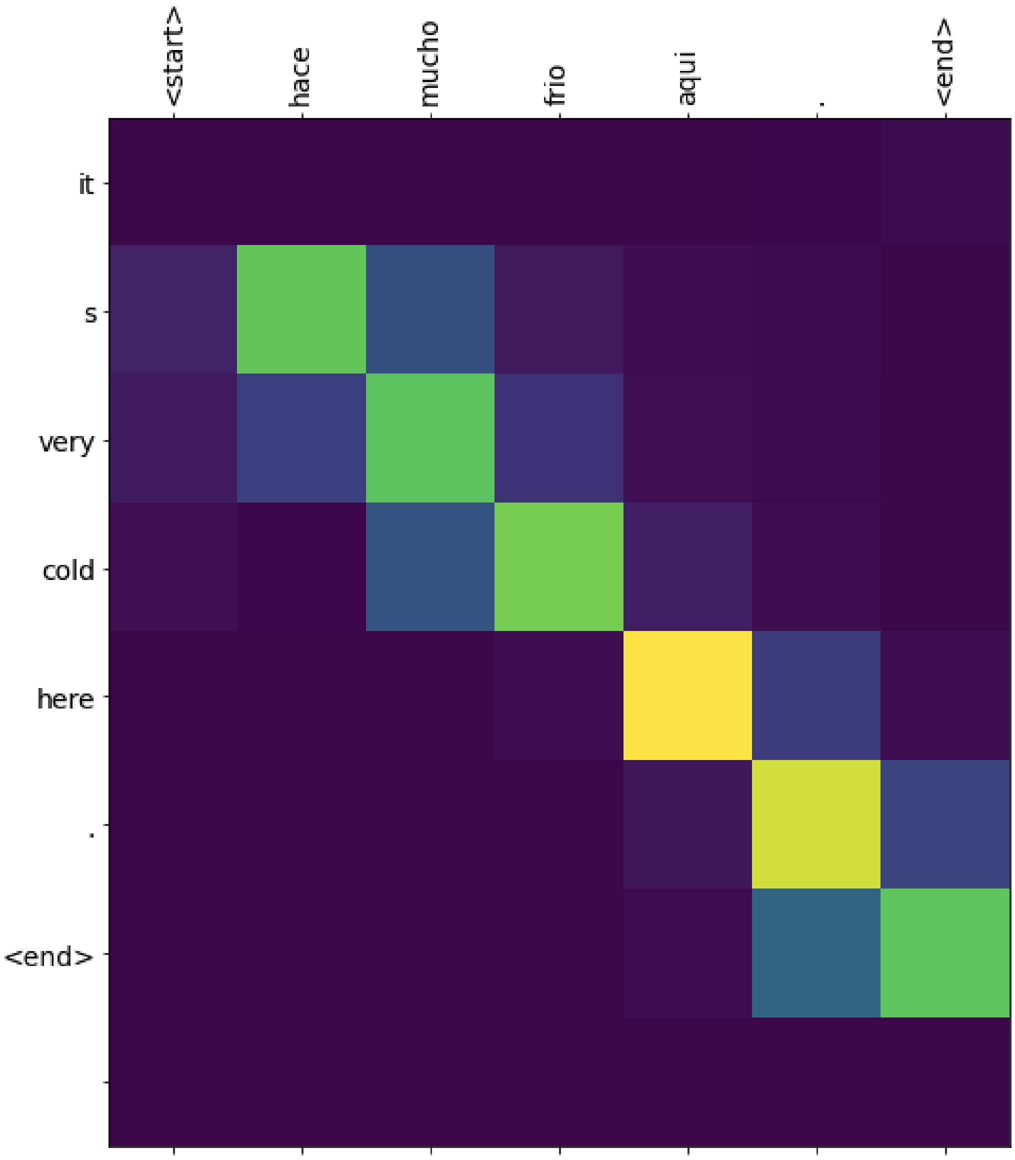
```
if batch % 100 == 0:
print('Epoch {} Batch {} Loss {:.4f}'.format(epoch + 1, batch
batch_loss.numpy()))
# saving (checkpoint) the model every 2 epochs
if (epoch + 1) % 2 == 0:
checkpoint.save(file_prefix = checkpoint_prefix)

print('Epoch {} Loss {:.4f}'.format(epoch + 1, total_loss / st
eps_per_epoch))
print('Time taken for 1 epoch {} sec\n'.format(time.time() - s
tart))

"3. evaluation"
"3.1 give sentence, return translated results"
"3.2 visualize results (attention)"
def evaluate(sentence):
# attention_plot: (11,16)
attention_plot = np.zeros((max_length_targ, max_length_inp))
sentence = preprocess_sentence(sentence)
inputs = [inp_lang.word_index[i] for i in sentence.split(' ')]
inputs = keras.preprocessing.sequence.pad_sequences([inputs],
maxlen=max_length_inp, padding='post')
inputs = tf.convert_to_tensor(inputs)

result = ''
hidden = tf.zeros((1, units))
encoding_out, encoding_hidden = encoder(inputs, hidden)
decoding_hidden = encoding_hidden
# eg: <start> -> A
# A -> B -> C -> D
# decoding_input.shape: (1,1)
decoding_input = tf.expand_dims([targ_lang.word_index['<start
>']], 0)

for t in range(max_length_targ):
predictions, decoding_hidden, attention_weights = decoder(deco
ding_input, decoding_hidden, encoding_out)
```

```
 # storing the attention weights to plot later on
 # attention_weights: (1,16,1) -> 16维向量
 attention_weights = tf.reshape(attention_weights, (-1, ))
 attention_plot[t] = attention_weights.numpy()
 # prediction.shape: (batch_size, vocab_size) (1, 4935)
 predicted_id = tf.argmax(predictions[0]).numpy()
 result += targ_lang.index_word[predicted_id] + ' '
 if targ_lang.index_word[predicted_id] == '<end>':
 return result, sentence, attention_plot
 # the predicted ID is fed back into the model
 decoding_input = tf.expand_dims([predicted_id], 0)
 return result, sentence, attention_plot

# function for plotting the attention weights
def plot_attention(attention, sentence, predicted_sentence):
 fig = plt.figure(figsize=(10,10))
 ax = fig.add_subplot(1, 1, 1)
 ax.matshow(attention, cmap='viridis')
 fontdict = {'fontsize': 14}
 ax.set_xticklabels([''] + sentence, fontdict=fontdict, rotation=90)
 ax.set_yticklabels([''] + predicted_sentence, fontdict=fontdict)
 plt.show()

def translate(sentence):
 result, sentence, attention_plot = evaluate(sentence)
 print('Input: %s' % (sentence))
 print('Predicted translation: {}'.format(result))
 attention_plot = attention_plot[:len(result.split(' ')), :len(sentence.split(' '))]
 plot_attention(attention_plot, sentence.split(' '), result.split(' '))

checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
translate(u'hace mucho frio aqui.')
```

```
translate(u'hace mucho frio aqui.')
```

```
Input: <start> hace mucho frio aqui . <end>
Predicted translation: it s very cold here . <end>
```

```
E:\Anaconda3\lib\site-packages\ipykernel_launcher.py:51: UserWarning: FixedFormatter sh
ould only be used together with FixedLocator
E:\Anaconda3\lib\site-packages\ipykernel_launcher.py:52: UserWarning: FixedFormatter sh
ould only be used together with FixedLocator
```



2.Transformer

机器翻译

模型思想-Attention

- ◆ 基于attention的seq2seq
  - ◆ 去除定长编码瓶颈，信息无损从Encoder传到Decoder
- ◆ 但是
  - ◆ 采用GRU，计算依然有瓶颈，并行度不高
  - ◆ 只有Encoder和Decoder之间有attention

机器翻译

模型思想-Attention

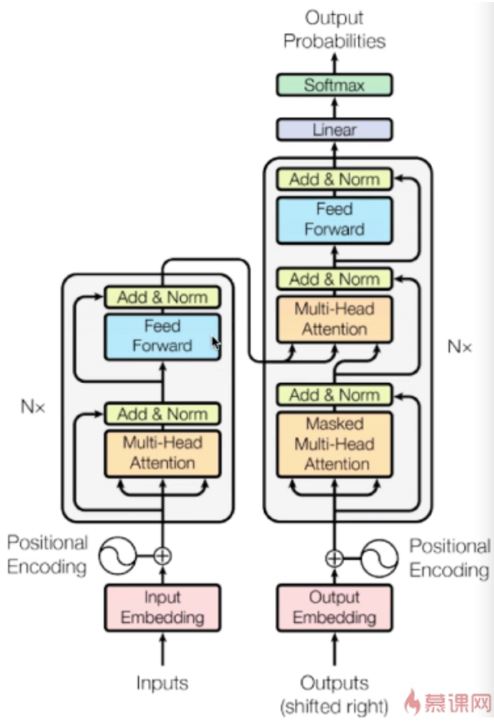
- ◆ 能否去掉RNN？
- ◆ 能否给输入和输出分别加上self attention?

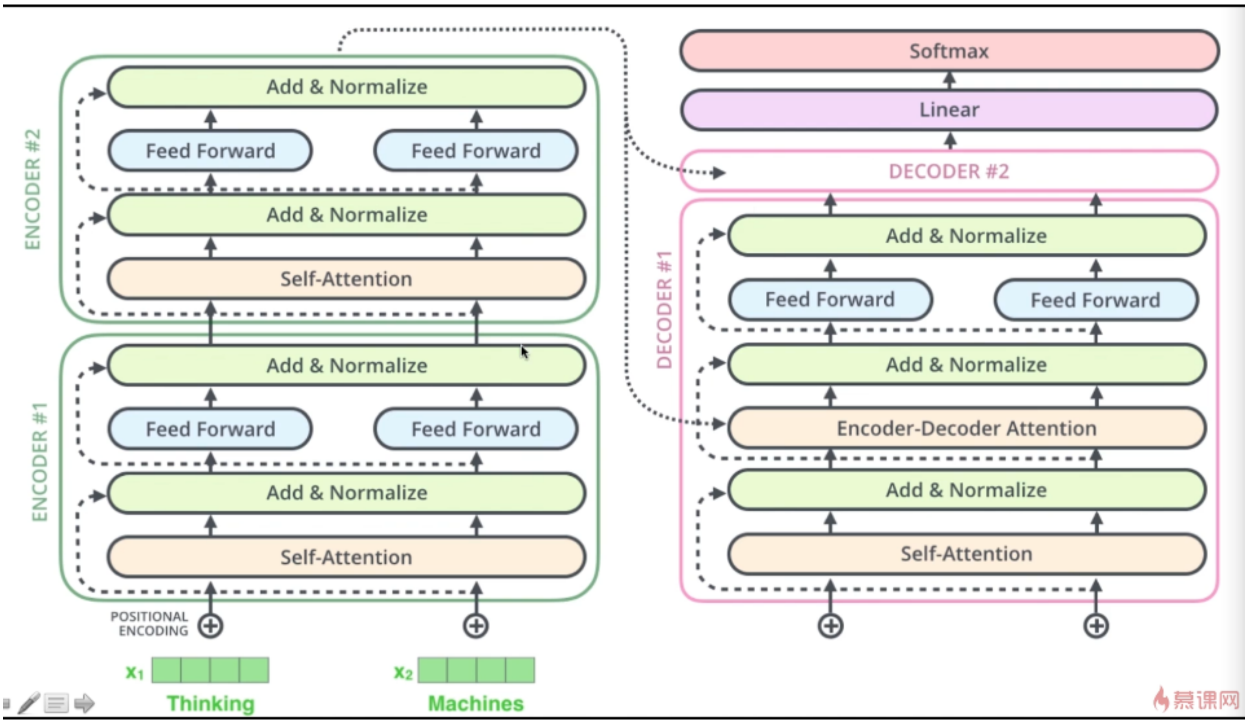
- **GRU中顺序处理序列，而Transformer中不知道句子的顺序，因此需要添加Positional Encoding**

Transformer模型

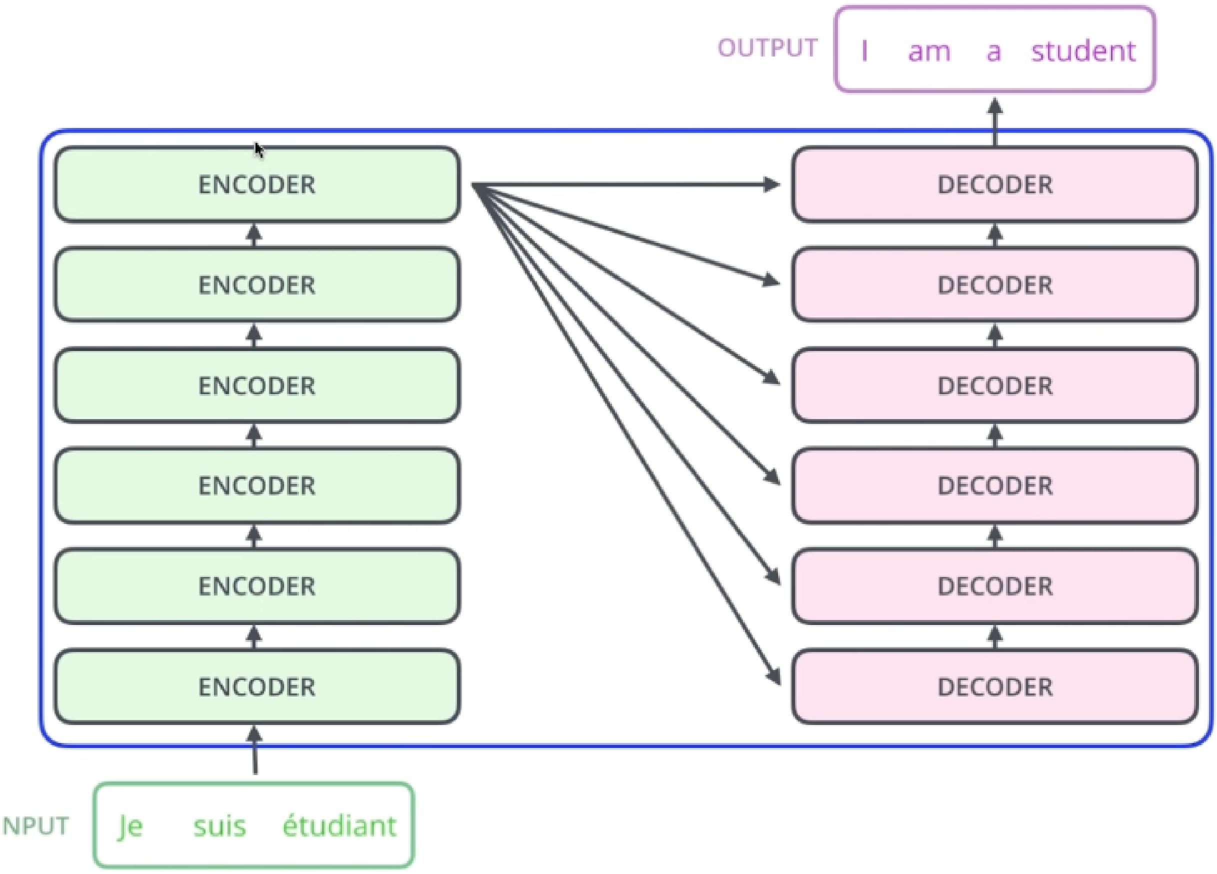
模型结构-Transformer

- ◆ Encoder-Decoder
- ◆ 多层Encoder-Decoder
- ◆ 位置编码
- ◆ 多头注意力
  - ◆ 缩放点积注意力
- ◆ Add & norm

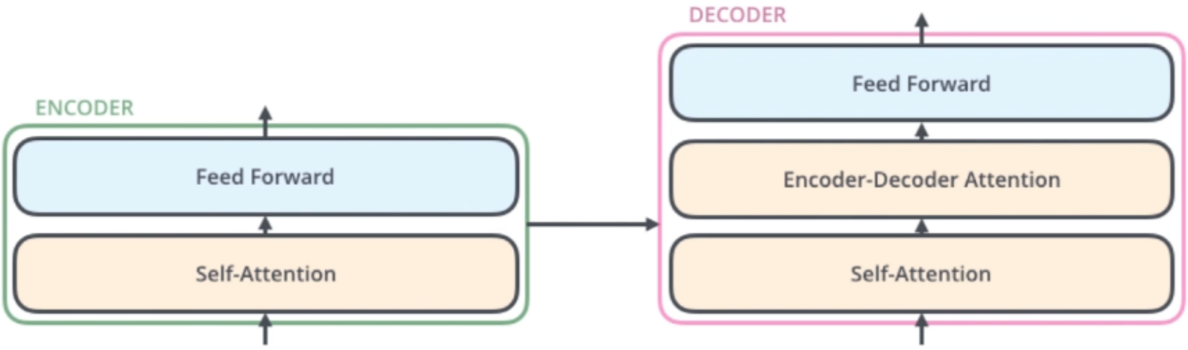




Softmax
Linear
DECODER #2
ENCODER #2
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
DECODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Encoder-Decoder Attention
Add & Normalize
Self-Attention
ENCODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
POSITIONAL ENCODING
x_1
x_2
Thinking
Machines
慕课网



OUTPUT
I am a student
ENCODER
ENCODER
ENCODER
ENCODER
ENCODER
ENCODER
DECODER
DECODER
DECODER
DECODER
DECODER
DECODER
NPUT
Je suis étudiant



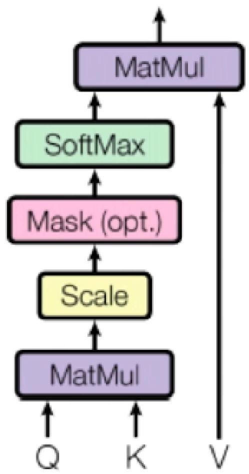
Transformer模型

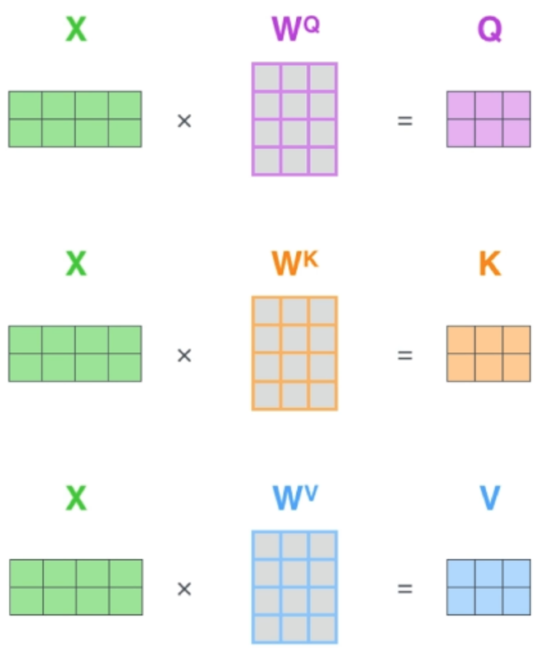
模型结构-Attention

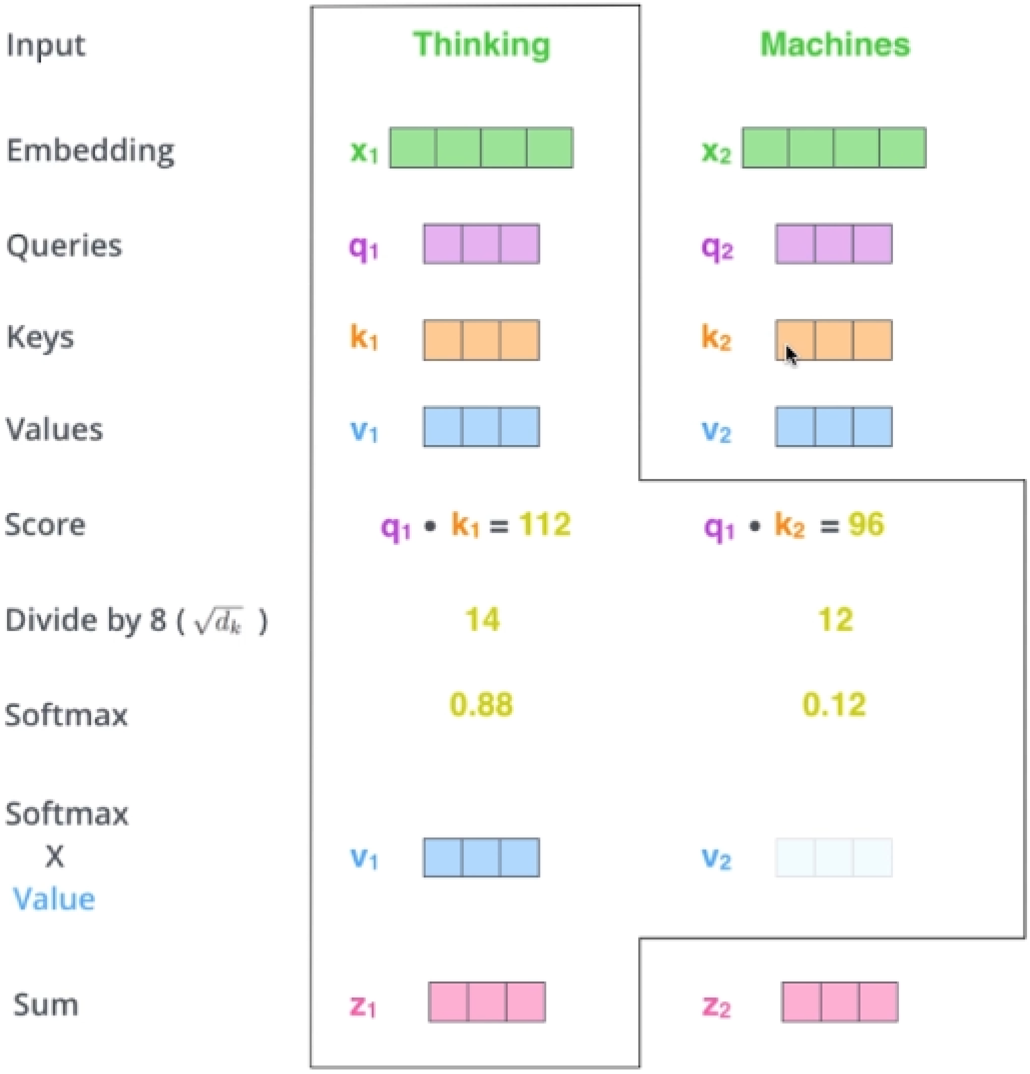
- ◆ 缩放点积attention

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

Scaled Dot-Product Attention

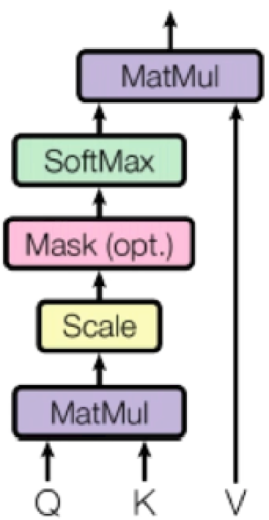




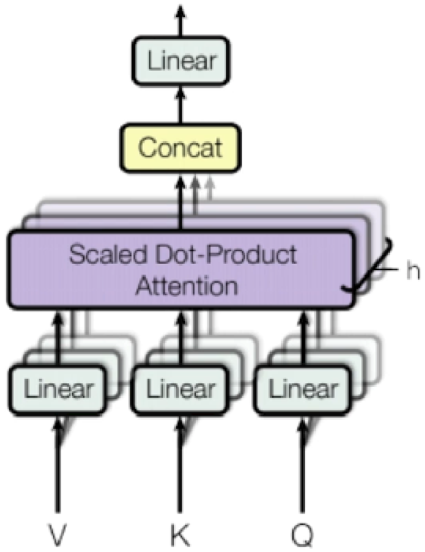


Transformer模型

Scaled Dot-Product Attention



Multi-Head Attention



Transformer模型

模型结构-Transformer

- ◆ 位置编码

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

- ◆ Pos = 1, d = 128
- ◆ [sin(1/10000^(0/128)), cos(1/10000^(1/128)), sin(1/10000^(2/128)), cos(1/10000^(3/128), ...]

Transformer模型

模型结构-Decoder

- ◆ Train的时候并行化
- ◆ Inference的时候仍然要序列式完成
- ◆ Self attention时前词不能见后词
  - ◆ Mask来实现

```python
import matplotlib as mpl
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import sklearn
import pandas as pd
import os
import sys
import time
import tensorflow as tf
from tensorflow import keras
```

```
print(tf.__version__)
print(sys.version_info)
for module in mpl, np, pd, sklearn, tf, keras:
 print(module.__name__, module.__version__)

# 1. loads data
# 2. preprocesses data -> dataset
# 3. tools
# 3.1 generates position embedding
# 3.2 create mask. (a. padding, b. decoder)
# 3.3 scaled_dot_product_attention
# 4. builds model
# 4.1 MultiheadAttention
# 4.2 EncoderLayer
# 4.3 DecoderLayer
# 4.4 EncoderModel
# 4.5 DecoderModel
# 4.6 Transformer
# 5. optimizer & loss
# 6. train step -> train
# 7. Evaluate and Visualize

"1. loads data"
import tensorflow_datasets as tfds
examples, info = tfds.load('ted_hrlr_translate/pt_to_en',with_info=True,as_supervised = True)
train_examples, val_examples = examples['train'], examples['validation']

"2. preprocesses data -> dataset"
en_tokenizer = tfds.deprecated.text.SubwordTextEncoder.build_from_corpus((en.numpy() for pt, en in train_examples), target_vocab_size = 2 ** 13)
pt_tokenizer = tfds.deprecated.text.SubwordTextEncoder.build_from_corpus((pt.numpy() for pt, en in train_examples), target_vocab_size = 2 ** 13)
buffer_size = 20000
```

```
batch_size = 64
max_length = 40

def encode_to_subword(pt_sentence, en_sentence):
 # pt_tokenizer.vocab_size: 用以指代<start>
 # pt_tokenizer.vocab_size + 1: 用以指代<end>
 pt_sequence = [pt_tokenizer.vocab_size] \
 + pt_tokenizer.encode(pt_sentence.numpy()) \
 + [pt_tokenizer.vocab_size + 1]
 en_sequence = [en_tokenizer.vocab_size] \
 + en_tokenizer.encode(en_sentence.numpy()) \
 + [en_tokenizer.vocab_size + 1]
 return pt_sequence, en_sequence

def filter_by_max_length(pt, en):
 return tf.logical_and(tf.size(pt) <= max_length, tf.size(en) <= max_length)

# dataset的map无法直接调用python函数，需要转为py_function
def tf_encode_to_subword(pt_sentence, en_sentence):
 return tf.py_function(encode_to_subword, [pt_sentence, en_sentence], [tf.int64, tf.int64])
train_dataset = train_examples.map(tf_encode_to_subword)
train_dataset = train_dataset.filter(filter_by_max_length)
# [-1], [-1]: 数据由两个维度，每个维度都扩展到当前维度下最高的值
train_dataset = train_dataset.shuffle(buffer_size).padded_batch(batch_size, padded_shapes=([-1], [-1]))

valid_dataset = val_examples.map(tf_encode_to_subword)
valid_dataset = valid_dataset.filter(filter_by_max_length).padded_batch(batch_size, padded_shapes=([-1], [-1]))
```

```
"3. tools"
"3.1 generates position embedding"
# PE(pos, 2i) = sin(pos / 10000^(2i/d_model))
```

```
# PE(pos, 2i+1) = cos(pos / 10000^(2i/d_model))

# pos.shape: [sentence_length, 1] 词在句子中的位置
# i.shape : [1, d_model] 词在embed中的位置
# result.shape: [sentence_length, 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 get_position_embedding(sentence_length, d_model):
  angle_rads = get_angles(np.arange(sentence_length)[:, np.newaxis], np.arange(d_model)[np.newaxis, :], d_model)
  # sines.shape: [sentence_length, d_model / 2]
  # cosines.shape: [sentence_length, d_model / 2]
  sines = np.sin(angle_rads[:, 0::2])
  cosines = np.cos(angle_rads[:, 1::2])
  # position_embedding.shape: [sentence_length, d_model]
  position_embedding = np.concatenate([sines, cosines], axis = -1)
  # position_embedding.shape: [1, sentence_length, d_model]
  position_embedding = position_embedding[np.newaxis, ...]
  return tf.cast(position_embedding, dtype=tf.float32)

"3.2 create mask. (a. padding, b. decoder)"
# 1. padding mask, 2. look ahead

# batch_data.shape: [batch_size, seq_len]
def create_padding_mask(batch_data):
  padding_mask = tf.cast(tf.math.equal(batch_data, 0), tf.float32)
  # [batch_size, 1, 1, seq_len]
  return padding_mask[:, tf.newaxis, tf.newaxis, :]

# attention_weights.shape: [3,3]
# [[1, 0, 0],
```

```
# [4, 5, 0], 第二个单词分别与第一、二、三个单词的attention
# [7, 8, 9]]
# 由于mask时padding为1，其余为0；所以需要创建一个上三角矩阵
def create_look_ahead_mask(size):
 mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)
 return mask # (seq_len, seq_len)

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

"3.3 scaled_dot_product_attention"
def scaled_dot_product_attention(q, k, v, mask):
 """
 Args:
 - q: shape == (..., seq_len_q, depth)
 - k: shape == (..., seq_len_k, depth)
 - v: shape == (..., seq_len_v, depth_v)
 - seq_len_k == seq_len_v
 - mask: shape == (..., seq_len_q, seq_len_k)
 Returns:
 - output: weighted sum
 - attention_weights: weights of attention
 """

 # matmul_qk.shape: (..., seq_len_q, seq_len_k)
 # transpose_b = True: 第二个矩阵做转置
 matmul_qk = tf.matmul(q, k, transpose_b = True)
 dk = tf.cast(tf.shape(k)[-1], tf.float32)
 scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)
 if mask is not None:
 # 使得在softmax后值趋近于0（一个数加上一个极小值，softmax后趋向于0）
```

```
    scaled_attention_logits += (mask * -1e9)
    # attention_weights.shape: (..., seq_len_q, seq_len_k)
    attention_weights = tf.nn.softmax(scaled_attention_logits, axis = -1)
    # output.shape: (..., seq_len_q, depth_v)
    output = tf.matmul(attention_weights, v)
    return output, attention_weights

def print_scaled_dot_product_attention(q, k, v):
    temp_out, temp_att = scaled_dot_product_attention(q, k, v, None)
    print("Attention weights are:")
    print(temp_att)
    print("Output is:")
    print(temp_out)
```

```
"4. builds model"
"4.1 MultiheadAttention"
class MultiHeadAttention(keras.layers.Layer):
 """
 理论上:
 x -> Wq0 -> q0
 x -> Wk0 -> k0
 x -> Wv0 -> v0

  实战中:
  q -> Wq0 -> q0
  k -> Wk0 -> k0
  v -> Wv0 -> v0

  实战中技巧:
  q -> Wq -> Q -> split -> q0, q1, q2...
  """
  def __init__(self, d_model, num_heads):
  super(MultiHeadAttention, self).__init__()
  self.num_heads = num_heads
```

```
self.d_model = d_model
assert self.d_model % self.num_heads == 0

self.depth = self.d_model // self.num_heads

self.WQ = keras.layers.Dense(self.d_model)
self.WK = keras.layers.Dense(self.d_model)
self.WV = keras.layers.Dense(self.d_model)

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

def split_heads(self, x, batch_size):
# x.shape: (batch_size, seq_len, d_model)
# d_model = num_heads * depth
# x -> (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, q, k, v, mask):
batch_size = tf.shape(q)[0]

q = self.WQ(q) # q.shape: (batch_size, seq_len_q, d_model)
k = self.WK(k) # k.shape: (batch_size, seq_len_k, d_model)
v = self.WV(v) # v.shape: (batch_size, seq_len_v, d_model)

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

# scaled_attention_outputs.shape: (batch_size, num_heads, seq_len_q, depth)
```

```
 # attention_weights.shape: (batch_size, num_heads, seq_len_q, seq_len_k)
 scaled_attention_outputs, attention_weights = \
 scaled_dot_product_attention(q, k, v, mask)

 # scaled_attention_outputs.shape: (batch_size, seq_len_q, num_heads, depth)
 scaled_attention_outputs = tf.transpose(
 scaled_attention_outputs, perm = [0, 2, 1, 3])
 # concat_attention.shape: (batch_size, seq_len_q, d_model)
 concat_attention = tf.reshape(scaled_attention_outputs,
 (batch_size, -1, self.d_model))

 # output.shape : (batch_size, seq_len_q, d_model)
 output = self.dense(concat_attention)
 return output, attention_weights

def feed_forward_network(d_model, dff):
 # dff: dim of feed forward network.
 return keras.Sequential([
 keras.layers.Dense(dff, activation='relu'),
 keras.layers.Dense(d_model)
 ])

"4.2 EncoderLayer"
class EncoderLayer(keras.layers.Layer):
 """
 x -> self attention -> add & normalize & dropout
 -> feed_forward -> add & normalize & dropout
 """
 def __init__(self, d_model, num_heads, dff, rate=0.1):
 super(EncoderLayer, self).__init__()
 self.mha = MultiHeadAttention(d_model, num_heads)
 self.ffn = feed_forward_network(d_model, dff)

 self.layer_norm1 = keras.layers.LayerNormalization(epsilon = 1e-6)
```

```
self.layer_norm2 = keras.layers.LayerNormalization(epsilon = 1e-6)

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

def call(self, x, training, encoder_padding_mask):
# x.shape : (batch_size, seq_len, dim=d_model)
# attn_output.shape: (batch_size, seq_len, d_model)
# out1.shape : (batch_size, seq_len, d_model)
attn_output, _ = self.mha(x, x, x, encoder_padding_mask)
attn_output = self.dropout1(attn_output, training=training)
out1 = self.layer_norm1(x + attn_output)

# ffn_output.shape: (batch_size, seq_len, d_model)
# out2.shape : (batch_size, seq_len, d_model)
ffn_output = self.ffn(out1)
ffn_output = self.dropout2(ffn_output, training=training)
out2 = self.layer_norm2(out1 + ffn_output)
return out2

"4.3 DecoderLayer"
class DecoderLayer(keras.layers.Layer):
"""
x -> self attention -> add & normalize & dropout -> out1
out1 , encoding_outputs -> attention -> add & normalize & dropout -> out2
out2 -> ffn -> add & normalize & dropout -> out3
"""
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 = feed_forward_network(d_model, dff)

```

```
    self.layer_norm1 = keras.layers.LayerNormalization(epsilon = 1e-6)
    self.layer_norm2 = keras.layers.LayerNormalization(epsilon = 1e-6)
    self.layer_norm3 = keras.layers.LayerNormalization(epsilon = 1e-6)

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

    def call(self, x, encoding_outputs, training,
    decoder_mask, encoder_decoder_padding_mask):
    # decoder_mask: 由look_ahead_mask和decoder_padding_mask合并而来
    # x.shape: (batch_size, target_seq_len, d_model)
    # encoding_outputs.shape: (batch_size, input_seq_len, d_model)

    # attn1, out1.shape : (batch_size, target_seq_len, d_model)
    attn1, attn_weights1 = self.mha1(x, x, x, decoder_mask)
    attn1 = self.dropout1(attn1, training = training)
    out1 = self.layer_norm1(attn1 + x)

    # attn2, out2.shape : (batch_size, target_seq_len, d_model)
    attn2, attn_weights2 = self.mha2(out1, encoding_outputs, encoding_outputs,encoder_decoder_padding_mask)
    attn2 = self.dropout2(attn2, training = training)
    out2 = self.layer_norm2(attn2 + out1)

    # ffn_output, out3.shape: (batch_size, target_seq_len, d_model)
    ffn_output = self.ffn(out2)
    ffn_output = self.dropout3(ffn_output, training=training)
    out3 = self.layer_norm3(ffn_output + out2)
    return out3, attn_weights1, attn_weights2

"4.4 EncoderModel"
class EncoderModel(keras.layers.Layer):
```

```
def __init__(self, num_layers, input_vocab_size, max_length,d_model, num_heads, dff, rate=0.1):
super(EncoderModel, self).__init__()
self.d_model = d_model
self.num_layers = num_layers
self.max_length = max_length

self.embedding = keras.layers.Embedding(input_vocab_size,self.d_model)
# position_embedding.shape: (1, max_length, d_model)
self.position_embedding = get_position_embedding(max_length,self.d_model)

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

def call(self, x, training, encoder_padding_mask):
# x.shape: (batch_size, input_seq_len)
input_seq_len = tf.shape(x)[1]
tf.debugging.assert_less_equal(input_seq_len, self.max_length,"input_seq_len should be less or equal to self.max_length")

# x.shape: (batch_size, input_seq_len, d_model)
x = self.embedding(x)
# 扩大x的值，使得与position_embedding相加后，x起的作用大一些
x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
x += self.position_embedding[:, :input_seq_len, :]
x = self.dropout(x, training = training)

for i in range(self.num_layers):
x = self.encoder_layers[i](x, training,encoder_padding_mask)
# x.shape: (batch_size, input_seq_len, d_model)
return x

```

```
"4.5 DecoderModel"
class DecoderModel(keras.layers.Layer):
  def __init__(self, num_layers, target_vocab_size, max_length,d_model, num_heads, dff, rate=0.1):
  super(DecoderModel, self).__init__()
  self.num_layers = num_layers
  self.max_length = max_length
  self.d_model = d_model
  self.embedding = keras.layers.Embedding(target_vocab_size,d_model)
  self.position_embedding = get_position_embedding(max_length,d_model)

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

  def call(self, x, encoding_outputs, training,decoder_mask, encoder_decoder_padding_mask):
  # x.shape: (batch_size, output_seq_len)
  output_seq_len = tf.shape(x)[1]
  tf.debugging.assert_less_equal(output_seq_len,
self.max_length,"output_seq_len should be less or equal to self.max_length")

  attention_weights = {}
  # x.shape: (batch_size, output_seq_len, d_model)
  x = self.embedding(x)
  x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
  x += self.position_embedding[:, :output_seq_len, :]

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

  for i in range(self.num_layers):
  x, attn1, attn2 = self.decoder_layers[i](x, encoding_outputs, training, decoder_mask, encoder_decoder_padding_mask)
  attention_weights['decoder_layer{}_att1'.format(i+1)] = attn1
  attention_weights['decoder_layer{}_att2'.format(i+1)] = attn2
```

```
 # x.shape: (batch_size, output_seq_len, d_model)
 return x, attention_weights

"4.6 Transformer"
class Transformer(keras.Model):
 def __init__(self, num_layers, input_vocab_size, target_vocab_
size,max_length, d_model, num_heads, dff, rate=0.1):
 super(Transformer, self).__init__()

 self.encoder_model = EncoderModel(num_layers,
input_vocab_size, max_length,d_model, num_heads, dff, rate)

 self.decoder_model = DecoderModel(num_layers, target_vocab_si
e, max_length,d_model, num_heads, dff, rate)

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

 def call(self, inp, tar, training, encoder_padding_mask,decode
r_mask, encoder_decoder_padding_mask):
 # encoding_outputs.shape: (batch_size, input_seq_len, d_model)
 encoding_outputs = self.encoder_model(inp, training, encoder_p
adding_mask)

 # decoding_outputs.shape: (batch_size, output_seq_len, d_mode
l)
 decoding_outputs, attention_weights = self.decoder_model(tar,
encoding_outputs, training,decoder_mask, encoder_decoder_padding_m
ask)

 # predictions.shape: (batch_size, output_seq_len, target_vocab
_size)
 predictions = self.final_layer(decoding_outputs)

 return predictions, attention_weights

```

```
"5. optimizer & loss"
num_layers = 4
d_model = 128
```

```
dff = 512
num_heads = 8

input_vocab_size = pt_tokenizer.vocab_size + 2
target_vocab_size = en_tokenizer.vocab_size + 2

dropout_rate = 0.1

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


learning_rate = 0.001
optimizer = keras.optimizers.Adam(learning_rate,beta_1 = 0.9,beta_2 = 0.98,epsilon = 1e-9)

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

def loss_function(real, pred):
 mask = tf.math.logical_not(tf.math.equal(real, 0))
 loss_ = loss_object(real, pred)

 mask = tf.cast(mask, dtype=loss_.dtype)
 loss_ *= mask

 return tf.reduce_mean(loss_)

def create_masks(inp, tar):
 """
 Encoder:
 - encoder_padding_mask (self attention of EncoderLayer)
 Decoder:
 - look_ahead_mask (self attention of DecoderLayer)
 - encoder_decoder_padding_mask (encoder-decoder attention of DecoderLayer)
 - decoder_padding_mask (self attention of DecoderLayer)
```

```
    """
    encoder_padding_mask = create_padding_mask(inp)
    encoder_decoder_padding_mask = create_padding_mask(inp)

    look_ahead_mask = create_look_ahead_mask(tf.shape(tar)[1])
    decoder_padding_mask = create_padding_mask(tar)
    decoder_mask = tf.maximum(decoder_padding_mask,look_ahead_mask)

    return encoder_padding_mask, decoder_mask, encoder_decoder_padding_mask

```

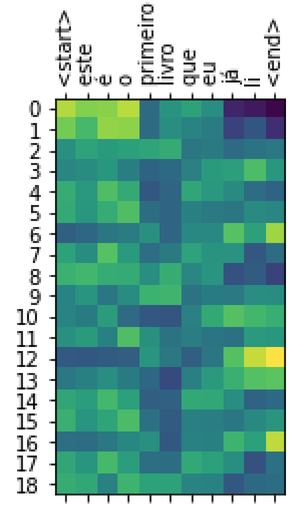
```
"6. train step -> train"
train_loss = keras.metrics.Mean(name = 'train_loss')
train_accuracy = keras.metrics.SparseCategoricalAccuracy(name = 'train_accuracy')

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

    encoder_padding_mask, decoder_mask, encoder_decoder_padding_mask = create_masks(inp, tar_inp)

    with tf.GradientTape() as tape:
    predictions, _ = transformer(inp, tar_inp, True,encoder_padding_mask,decoder_mask,encoder_decoder_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(tar_real, predictions)

```

```
epochs = 2 # 0
for epoch in range(epochs):
 start = time.time()
 train_loss.reset_states()
 train_accuracy.reset_states()

 for (batch, (inp, tar)) in enumerate(train_dataset):
 train_step(inp, tar)
 if batch % 100 == 0:
 print('Epoch {} Batch {} Loss {:.4f} Accuracy {:.4f}'.format(epoch + 1, batch, train_loss.result(),train_accuracy.result()))

 print('Epoch {} Loss {:.4f} Accuracy {:.4f}'.format(epoch + 1, train_loss.result(), train_accuracy.result()))
 print('Time take for 1 epoch: {} secs\n'.format(time.time() - start))
```

```
"7. Evaluate and Visualize"
"""
eg: A B C D -> E F G H.
Train: A B C D, E F G -> F G H
Eval: A B C D -> E
 A B C D, E -> F
 A B C D, E F -> G
 A B C D, E F G -> H
"""
def evaluate(inp_sentence):
 input_id_sentence = [pt_tokenizer.vocab_size] + pt_tokenizer.encode(inp_sentence) + [pt_tokenizer.vocab_size + 1]
 # encoder_input.shape: (1, input_sentence_length)
 encoder_input = tf.expand_dims(input_id_sentence, 0)

 # decoder_input.shape: (1, 1)
 decoder_input = tf.expand_dims([en_tokenizer.vocab_size], 0)

 for i in range(max_length):
```

```
    encoder_padding_mask, decoder_mask, encoder_decoder_padding_mask = create_masks(encoder_input, decoder_input)
    # predictions.shape: (batch_size, output_target_len, target_vocab_size)
    predictions, attention_weights = transformer(encoder_input,decoder_input,False,encoder_padding_mask,decoder_mask,encoder_decoder_padding_mask)
    # predictions.shape: (batch_size, target_vocab_size)
    predictions = predictions[:, -1, :]

    predicted_id = tf.cast(tf.argmax(predictions, axis = -1),tf.int32)

    if tf.equal(predicted_id, en_tokenizer.vocab_size + 1):
    return tf.squeeze(decoder_input, axis = 0), attention_weights

    decoder_input = tf.concat([decoder_input, [predicted_id]],axis = -1)
    return tf.squeeze(decoder_input, axis = 0), attention_weights

def plot_encoder_decoder_attention(attention, input_sentence,result, layer_name):
    fig = plt.figure(figsize = (16, 8))

    input_id_sentence = pt_tokenizer.encode(input_sentence)

    # attention.shape: (num_heads, tar_len, input_len)
    attention = tf.squeeze(attention[layer_name], axis = 0)

    for head in range(attention.shape[0]):
    ax = fig.add_subplot(2, 4, head + 1)

    # :-1: <end>的id没有加入attention矩阵
    ax.matshow(attention[head][:-1, :])

    fontdict = {'fontsize': 10}

    ax.set_xticks(range(len(input_id_sentence) + 2))
    ax.set_yticks(range(len(result)))
```

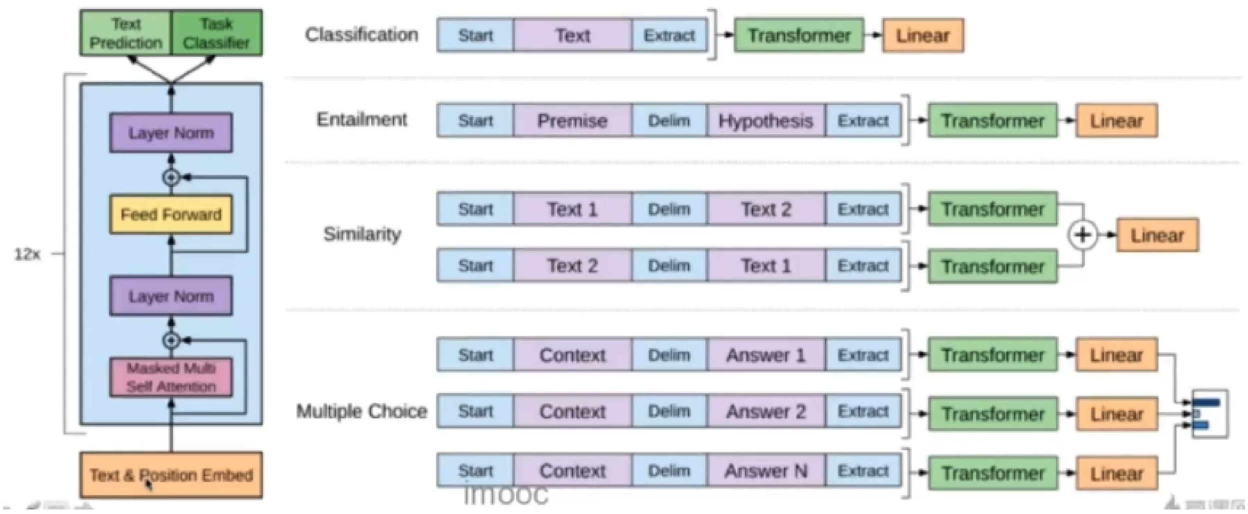
```
    ax.set_ylim(len(result) - 1.5, -0.5)

    ax.set_xticklabels(['<start>'] + [pt_tokenizer.decode([i]) for i in input_id_sentence] + ['<end>'],fontdict = fontdict, rotation = 90)
    ax.set_yticklabels([en_tokenizer.decode([i]) for i in result if i < en_tokenizer.vocab_size],fontdict = fontdict)
    ax.set_xlabel('Head {}'.format(head + 1))
    plt.tight_layout()
    plt.show()

def translate(input_sentence, layer_name = ''):
    result, attention_weights = evaluate(input_sentence)

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

    print("Input: {}".format(input_sentence))
    print("Predicted translation: {}".format(predicted_sentence))

    if layer_name:
    plot_encoder_decoder_attention(attention_weights, input_sentence,result, layer_name)

translate('este é o primeiro livro que eu já li',layer_name = 'decoder_layer4_att2')
# Input: este é o primeiro livro que eu já li
# Predicted translation: it 's a lot of the world 's going to be a lot of the world .
```



Transformer模型

- GPT模型
  - Generative Pre-training
  - Transformer的Decoder部分
  - 预训练+fine-tune到具体任务

Generative Pre-Training模型



Transformer模型

- Bert模型
  - Bidirectional Encoder Representations from Transformers
  - 预训练+fine-tune到特殊任务
  - 双向网络
  - 随机mask
  - 预测下一个句子

Bert模型