道路工程深度学习技术

第七周 Transformer 实践 1—时序数据处理

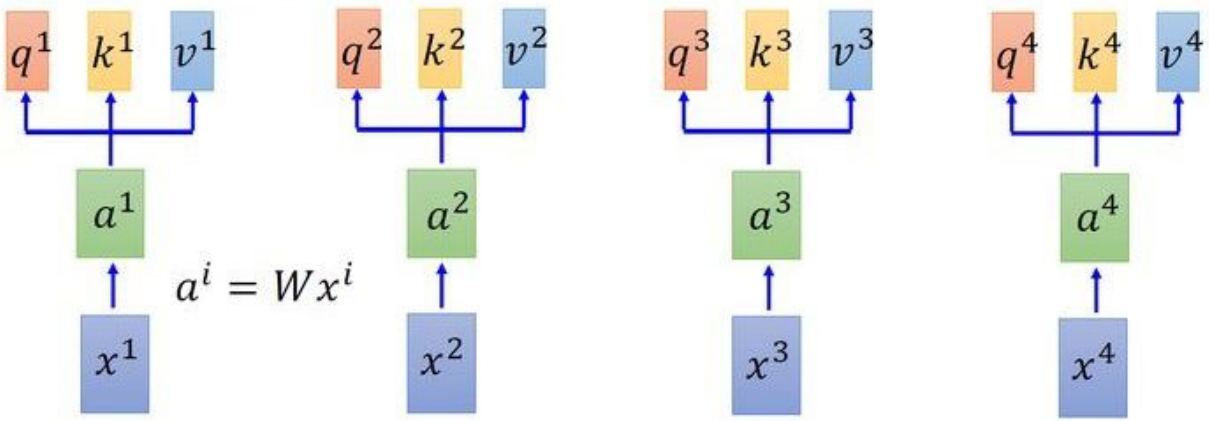
童峥

东南大学 交通学院

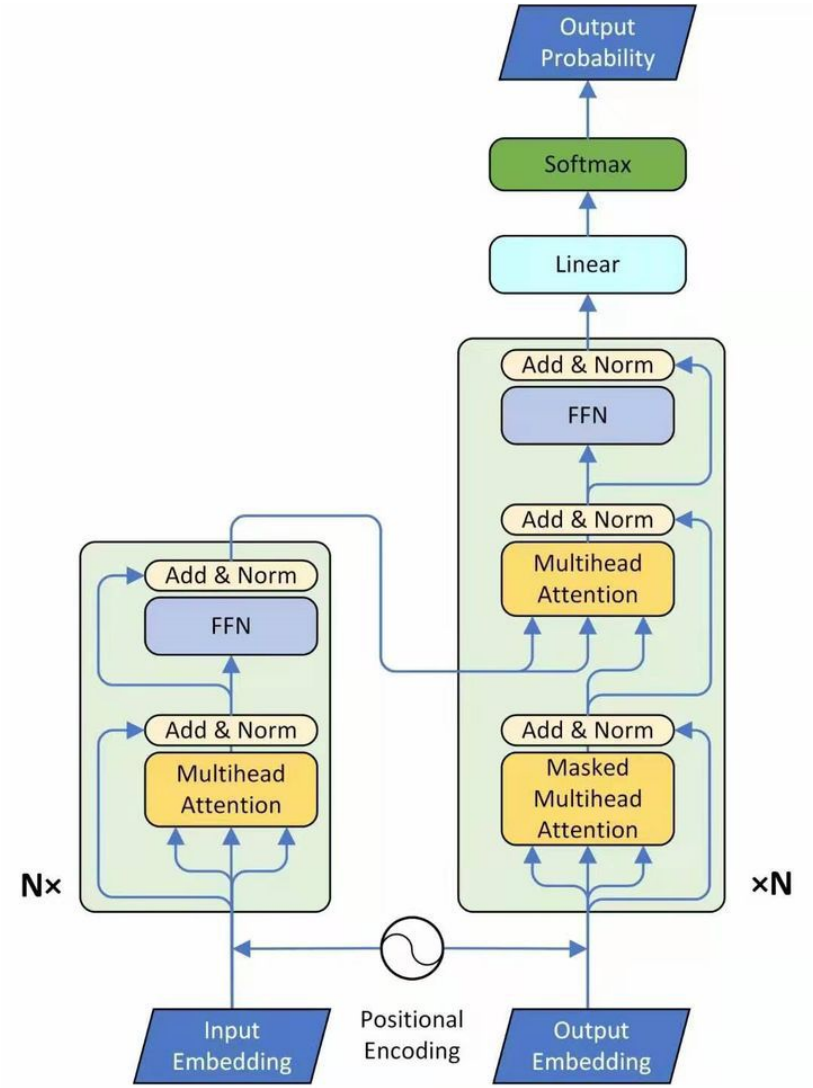
2023 年 4 月 25 日

自注意力机制

- *Attention Is All You Need*
- `https://arxiv.org/abs/1706.03762`



Transformer 模型



文本向量化

- 将字符串映射到数字表示值
- 创建两个查找表格：一个将字符映射到数字，另一个将数字映射到字符
- 将每个句子填充（pad）到最大长度

```python
# 将 unicode 文件转换为 ascii
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())

    # 在单词与跟在其后的标点符号之间插入一个空格
    # 例如： "he is a boy." => "he is a boy ."
    # 参考：https://stackoverflow.com/questions/3645931/python-padding-punctuation-with-white-
    w = re.sub(r"([?.!,¿])", r" \1 ", w)
    w = re.sub(r'[" "]+', " ", w)

    # 除了 (a-z, A-Z, ".", "?", "!", ","), 将所有字符替换为空格
    w = re.sub(r"[^a-zA-Z?.!,¿]+", " ", w)

    w = w.rstrip().strip()

    # 给句子加上开始和结束标记
    # 以便模型知道何时开始和结束预测
    w = '<start> ' + w + ' <end>'
    return w
```

```python
en_sentence = u"May I borrow this book?"
sp_sentence = u"¿Puedo tomar prestado este libro?"
print(preprocess_sentence(en_sentence))
print(preprocess_sentence(sp_sentence).encode('utf-8'))
```

```python
# 1. 去除重音符号
# 2. 清理句子
# 3. 返回这样格式的单词对：[ENGLISH, SPANISH]
def create_dataset(path, num_examples):
    lines = io.open(path, encoding='UTF-8').read().strip().split('\n')

    word_pairs = [[preprocess_sentence(w) for w in l.split('\t')]  for l in lines[:num_examp

    return zip(*word_pairs)
```

```python
en, sp = create_dataset(path_to_file, None)
print(en[-1])
print(sp[-1])
```

```
<start> if you want to sound like a native speaker , you must be willing to practice saying t
<start> si quieres sonar como un hablante nativo , debes estar dispuesto a practicar diciendo
```

文本向量化

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

```python
def tokenize(lang):
  lang_tokenizer = tf.keras.preprocessing.text.Tokenizer(
      filters='')
  lang_tokenizer.fit_on_texts(lang)

  tensor = lang_tokenizer.texts_to_sequences(lang)

  tensor = tf.keras.preprocessing.sequence.pad_sequences(tensor,
                                                         padding='post')

  return tensor, lang_tokenizer
```

```python
def load_dataset(path, num_examples=None):
    # 创建清理过的输入输出对
    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
```

```python
print ("Input Language; index to word mapping")
convert(inp_lang, input_tensor_train[0])
print ()
print ("Target Language; index to word mapping")
convert(targ_lang, target_tensor_train[0])
```

```
Input Language; index to word mapping
1 ----> <start>
139 ----> vamos
10 ----> a
1569 ----> seguir
10 ----> a
4 ----> tom
3 ----> .
2 ----> <end>

Target Language; index to word mapping
1 ----> <start>
16 ----> we
38 ----> ll
```

其他的文本向量化方法

- Word-level：基于空格的分词器
- Character-level：基于字符的分词器
- Subword-level：基于子词的分词器

```
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)
print ('Tokenized string is {}'.format(tokenized_string))

original_string = tokenizer_en.decode(tokenized_string)
print ('The original string: {}'.format(original_string))

assert original_string == sample_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 ----> .
```

葡萄牙语-英语翻译数据集

- 使用 TFDS 来导入葡萄牙语-英语翻译数据集
- 该数据集包含来约 50000 条训练样本，1100 条验证样本，2000 条测试样本
- 该数据集来自于TED 演讲开放翻译项目

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

```
Downloading and preparing dataset ted_hrlr_translate/pt_to_en/1.0.0 (download: 124.9
Shuffling and writing examples to /home/kbuilder/tensorflow_datasets/ted_hrlr_transl
Shuffling and writing examples to /home/kbuilder/tensorflow_datasets/ted_hrlr_transl
Shuffling and writing examples to /home/kbuilder/tensorflow_datasets/ted_hrlr_transl
Dataset ted_hrlr_translate downloaded and prepared to /home/kbuilder/tensorflow_data
```

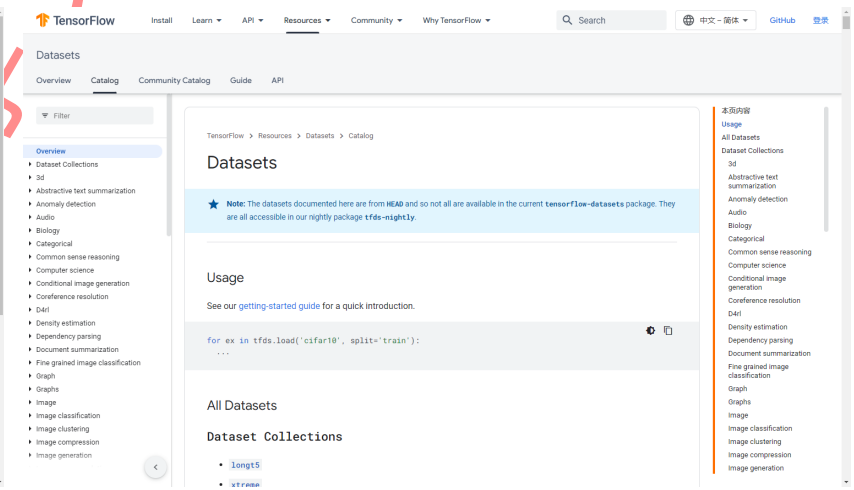
```
def tf_encode(pt, en):
  result_pt, result_en = tf.py_function(encode, [pt, en], [tf.int64, tf.int64])
  result_pt.set_shape([None])
  result_en.set_shape([None])

  return result_pt, result_en
```

```
train_dataset = train_examples.map(tf_encode)
train_dataset = train_dataset.filter(filter_max_length)
# 将数据集缓存到内存中以加快读取速度。
train_dataset = train_dataset.cache()
train_dataset = train_dataset.shuffle(BUFFER_SIZE).padded_batch(BATCH_SIZE)
train_dataset = train_dataset.prefetch(tf.data.experimental.AUTOTUNE)


val_dataset = val_examples.map(tf_encode)
val_dataset = val_dataset.filter(filter_max_length).padded_batch(BATCH_SIZE)
```

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

Tensorflow 数据库

- TensorFlow Datasets 是可用于 TensorFlow 或其他 Python 机器学习框架的一系列数据集
- 所有数据集都作为 tf.data.Datasets 提供
- 包含信号、文本、图像、视频、三维点云等各类公开数据集



位置编码

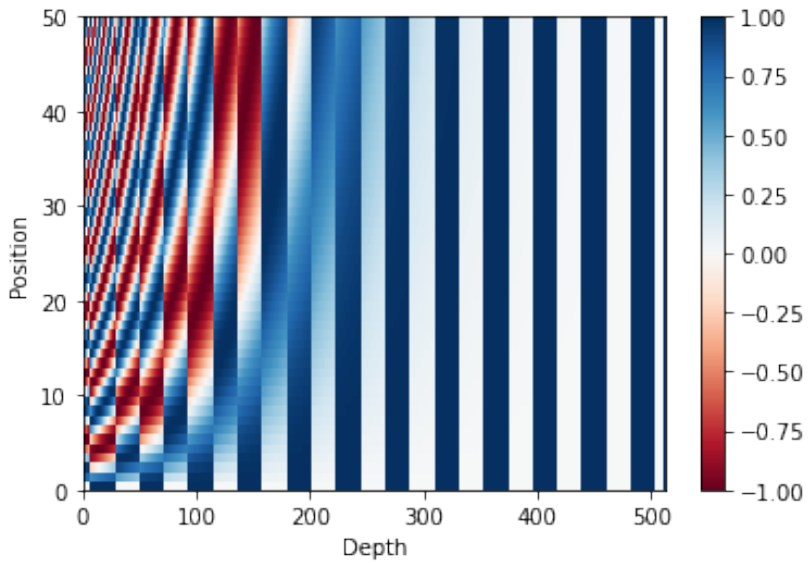
- 将位置编码嵌入文本向量
- 计算位置编码的公式

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

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

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

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

  # 将 sin 应用于数组中的偶数索引（indices）；2i
  angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])

  # 将 cos 应用于数组中的奇数索引；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)
```



填充遮挡（pad mask）

- 遮挡一批序列中所有的填充标记（pad tokens）
- 确保模型不会将填充作为输入
- 该 mask 表明填充值 0 出现的位置：在这些位置 mask 输出 1，否则输出 0。


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

  # 添加额外的维度来将填充加到
  # 注意力对数（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)>
```

前瞻遮挡（look-ahead mask）

- 遮挡一个序列中的后续标记（future tokens）
- 确保模型不会使用不应该使用的向量
- 预测第三个词，将仅使用第一个和第二个词；预测第四个词，仅使用第一个，第二个和第三个词，依此类推


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

```python
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)>
```

点积注意力（Scaled dot product attention）

Transformer 使用的注意力函数有三个输入：Q（query）、K（key）、V（value）

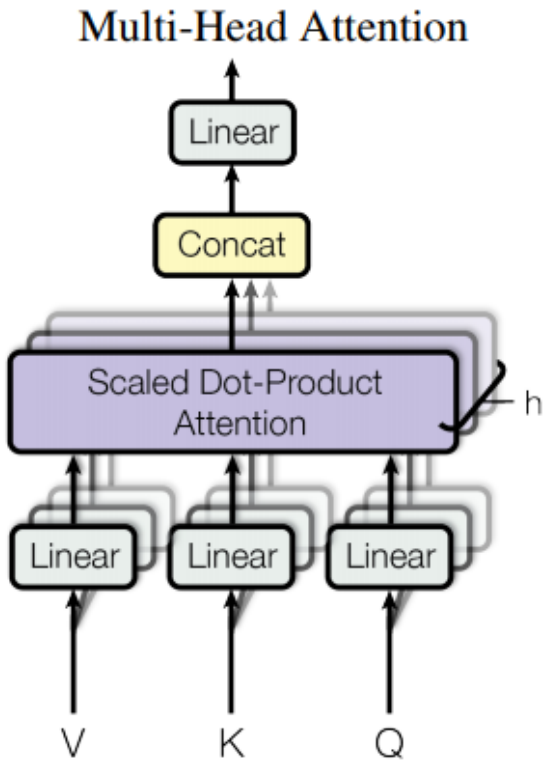
$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_K}}) \cdot V$$

```python
def scaled_dot_product_attention(q, k, v, mask):
  """计算注意力权重。
  q, k, v 必须具有匹配的前置维度。
  k, v 必须有匹配的倒数第二个维度，例如：seq_len_k = seq_len_v。
  虽然 mask 根据其类型（填充或前瞻）有不同的形状，
  但是 mask 必须能进行广播转换以便求和。

  参数：
    q: 请求的形状 == (..., seq_len_q, depth)
    k: 主键的形状 == (..., seq_len_k, depth)
    v: 数值的形状 == (..., seq_len_v, depth_v)
    mask: Float 张量，其形状能转换成
          (..., seq_len_q, seq_len_k)。默认为None。

  返回值：
    输出，注意力权重
  """
```

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

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

# 将 mask 加入到缩放的张量上。
if mask is not None:
  scaled_attention_logits += (mask * -1e9)

# softmax 在最后一个轴（seq_len_k）上归一化，因此分数
# 相加等于1。
attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)  # (...,

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

return output, attention_weights
```

多头注意力（Multi-head attention）

- 多头注意力由四部分组成：线性层并分拆成多头、按比缩放的点积注意力、多头及联、线性层
- 每个多头注意力块有三个输入：Q、K、V



多头注意力（Multi-head attention）

```python
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):
    """分拆最后一个维度到 (num_heads, depth).
    转置结果使得形状为 (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])
```

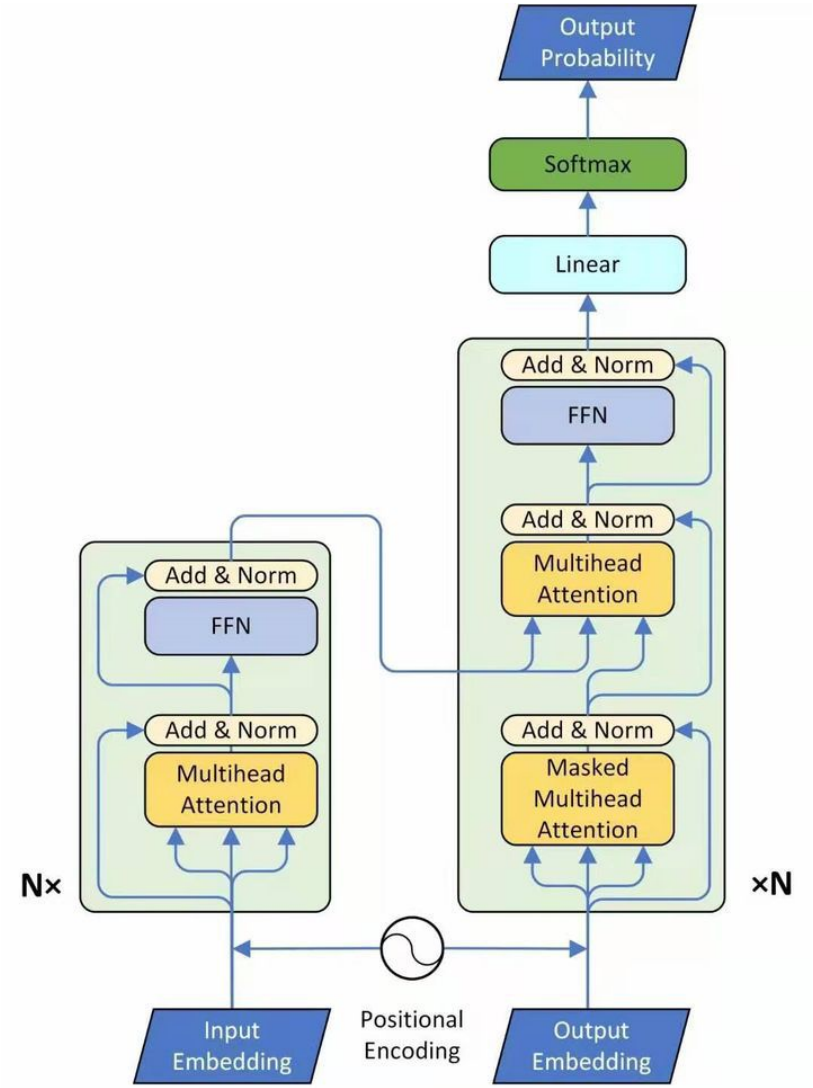
```python
def call(self, v, k, q, mask):
  batch_size = tf.shape(q)[0]

  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_si

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

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

  return output, attention_weights
```

点式前馈网络（Point wise feed forward network）

- 点式前馈网络由两层全联接层组成
- 两层之间有一个 ReLU 激活函数

```
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)
  ])
```

概览


Output Probability
Softmax
Linear
Add & Norm
FFN
Add & Norm
Multihead Attention
Add & Norm
Masked Multihead Attention
Add & Norm
FFN
Add & Norm
Multihead Attention
N×
×N
Input Embedding
Positional Encoding
Output Embedding

编码器层（Encoder layer）

- 多头注意力
- 点式前馈网络

```python
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
```

编码器（Encoder ）

- 输入嵌入（Input Embedding）
- 位置编码（Positional Encoding）
- N 个编码器层（encoder layers）

```
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]

  # 将嵌入和位置编码相加。
  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)
```

解码器层（Decoder layer）

- 遮挡的多头注意力（前瞻遮挡和填充遮挡）
- 点式前馈网络

```
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_
  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
```

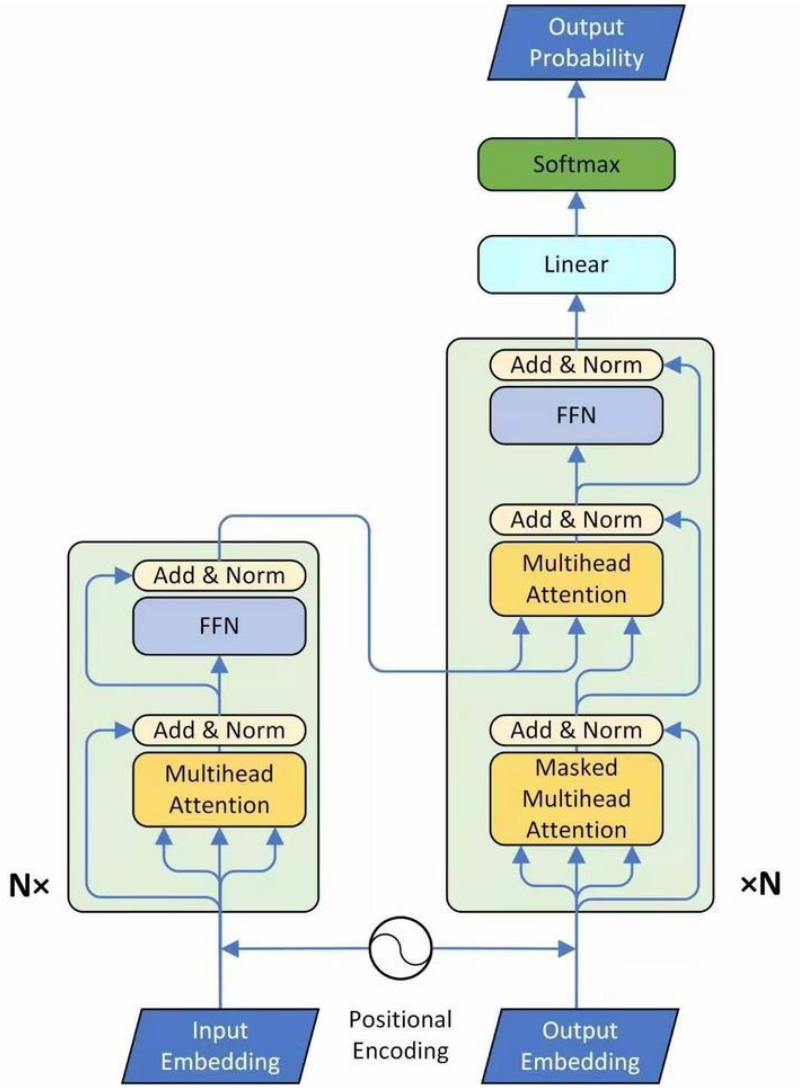
解码器（Decoder ）

- 输出嵌入（Output Embedding）
- 位置编码（Positional Encoding）
- N 个解码器层（decoder layers）


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

    self.d_model = d_model
    self.num_layers = num_layers

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

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

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

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

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

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

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

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

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

Transformer 模型




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

    self.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, inp, tar, training, enc_padding_mask,
           look_ahead_mask, dec_padding_mask):

    enc_output = self.encoder(inp, training, enc_padding_mask)  # (batch_siz

    # 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,

    return final_output, attention_weights
```

配置超参数

```
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, 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)

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

```
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
```

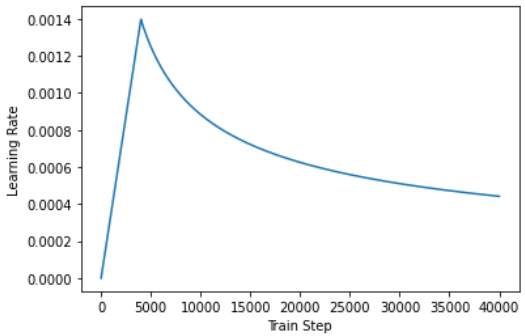
Adam 优化器（Optimizer）

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

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

    self.warmup_steps = warmup_steps

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

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

```
learning_rate = CustomSchedule(d_model)

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

```
temp_learning_rate_schedule = CustomSchedule(d_model)

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

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



损失函数与指标（Loss and metrics）

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

```python
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_)
```

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

训练与检查点（Training and checkpointing）

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

```
def create_masks(inp, tar):
  # 编码器填充遮挡
  enc_padding_mask = create_padding_mask(inp)

  # 在解码器的第二个注意力模块使用。
  # 该填充遮挡用于遮挡编码器的输出。
  dec_padding_mask = create_padding_mask(inp)

  # 在解码器的第一个注意力模块使用。
  # 用于填充（pad）和遮挡（mask）解码器获取到的输入的后续标记（future tokens）。
  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)

# 如果检查点存在，则恢复最新的检查点。
if ckpt_manager.latest_checkpoint:
  ckpt.restore(ckpt_manager.latest_checkpoint)
  print ('Latest checkpoint restored!!')
```

训练

```python
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_dataset):
    train_step(inp, tar)

    if batch % 50 == 0:
      print ('Epoch {} Batch {} Loss {:.4f} Accuracy {:.4f}'.format(
          epoch + 1, batch, train_loss.result(), train_accuracy.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_accuracy.result()))

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

```
Epoch 1 Batch 0 Loss 4.3163 Accuracy 0.0004
Epoch 1 Batch 50 Loss 4.2825 Accuracy 0.0018
Epoch 1 Batch 100 Loss 4.2022 Accuracy 0.0134
Epoch 1 Batch 150 Loss 4.1526 Accuracy 0.0180
Epoch 1 Batch 200 Loss 4.0717 Accuracy 0.0204
Epoch 1 Batch 250 Loss 3.9939 Accuracy 0.0224
Epoch 1 Batch 300 Loss 3.9193 Accuracy 0.0242
Epoch 1 Batch 350 Loss 3.8341 Accuracy 0.0264
Epoch 1 Batch 400 Loss 3.7555 Accuracy 0.0301
Epoch 1 Batch 450 Loss 3.6772 Accuracy 0.0335
Epoch 1 Batch 500 Loss 3.6066 Accuracy 0.0367
Epoch 1 Batch 550 Loss 3.5419 Accuracy 0.0403
Epoch 1 Batch 600 Loss 3.4774 Accuracy 0.0440
Epoch 1 Batch 650 Loss 3.4262 Accuracy 0.0476
```

评估步骤

- 用葡萄牙语分词器（tokenizer_pt）编码输入语句
- 解码器输入为 start token == tokenizer_en.vocab_size
- 计算填充遮挡和前瞻遮挡
- 解码器通过查看编码器输出和它自身的输出（自注意力）给出预测
- 选择最后一个词并计算它的 argmax
- 将预测的词连接到解码器输入，然后传递给解码器
- 在这种方法中，解码器根据它预测的之前的词预测下一个

评估代码

```
def evaluate(inp_sentence):
  start_token = [tokenizer_pt.vocab_size]
  end_token = [tokenizer_pt.vocab_size + 1]

  # 输入语句是葡萄牙语，增加开始和结束标记
  inp_sentence = start_token + tokenizer_pt.encode(inp_sentence) + end_token
  encoder_input = tf.expand_dims(inp_sentence, 0)

  # 因为目标是英语，输入 transformer 的第一个词应该是
  # 英语的开始标记。
  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,
```

评估代码

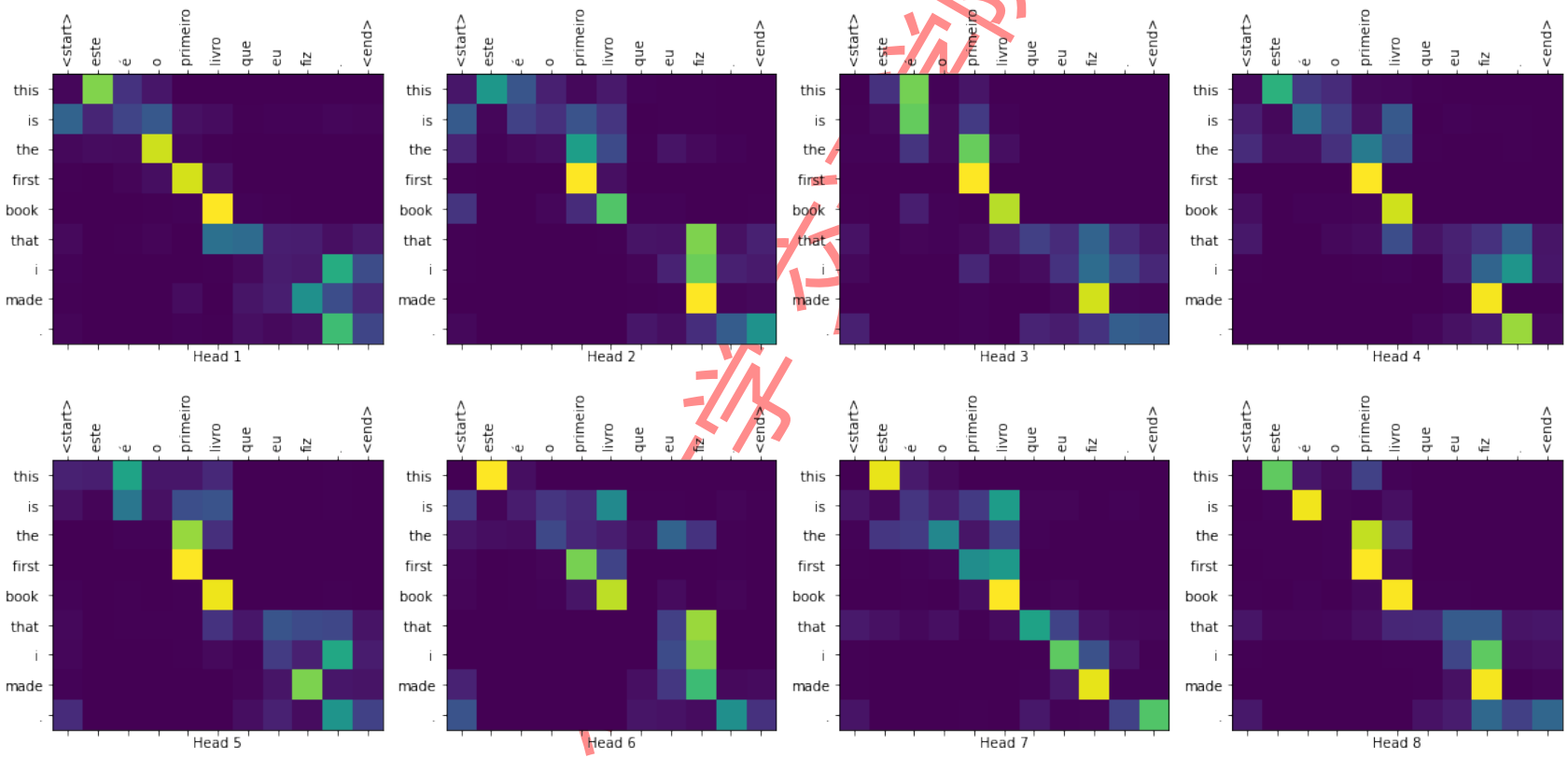
```
def plot_attention_weights(attention, sentence, result, layer):
  fig = plt.figure(figsize=(16, 8))

  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)

    # 画出注意力权重
    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(
```

评估结果

```
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)
```

```
translate("este é um problema que temos que resolver.")
print ("Real translation: this is a problem we have to solve .")
```

评估结果


<start> este é o primeiro livro que eu fiz <end>
this is the first book that i made .
Head 1
Head 2
Head 3
Head 4
Head 5
Head 6
Head 7
Head 8