【关于 Transformer 代码实战（文本摘要任务篇）】 那些你不知道的事

作者：杨夕

项目地址：https://github.com/km1994/nlp\_paper\_study

个人介绍：大佬们好，我叫杨夕，该项目主要是本人在研读顶会论文和复现经典论文过程中，所见、所思、所想、所闻，可能存在一些理解错误，希望大佬们多多指正。

目录

* [【关于 Transformer 代码实战（文本摘要任务篇）】 那些你不知道的事](#%E5%85%B3%E4%BA%8E-transformer-%E4%BB%A3%E7%A0%81%E5%AE%9E%E6%88%98%E6%96%87%E6%9C%AC%E6%91%98%E8%A6%81%E4%BB%BB%E5%8A%A1%E7%AF%87-%E9%82%A3%E4%BA%9B%E4%BD%A0%E4%B8%8D%E7%9F%A5%E9%81%93%E7%9A%84%E4%BA%8B)
  + [目录](#%E7%9B%AE%E5%BD%95)
  + [引言](#%E5%BC%95%E8%A8%80)
  + [一、文本摘要数据集介绍](#%E4%B8%80%E6%96%87%E6%9C%AC%E6%91%98%E8%A6%81%E6%95%B0%E6%8D%AE%E9%9B%86%E4%BB%8B%E7%BB%8D)
  + [二、数据集加载介绍](#%E4%BA%8C%E6%95%B0%E6%8D%AE%E9%9B%86%E5%8A%A0%E8%BD%BD%E4%BB%8B%E7%BB%8D)
    - [2.1 数据加载](#21-%E6%95%B0%E6%8D%AE%E5%8A%A0%E8%BD%BD)
    - [2.2 数据字段抽取](#22-%E6%95%B0%E6%8D%AE%E5%AD%97%E6%AE%B5%E6%8A%BD%E5%8F%96)
  + [三、 数据预处理](#%E4%B8%89-%E6%95%B0%E6%8D%AE%E9%A2%84%E5%A4%84%E7%90%86)
    - [3.1 summary 数据 处理](#31-summary-%E6%95%B0%E6%8D%AE-%E5%A4%84%E7%90%86)
    - [3.2 编码处理](#32-%E7%BC%96%E7%A0%81%E5%A4%84%E7%90%86)
    - [3.3 获取 encoder 词典 和 decoder 词典 长度](#33-%E8%8E%B7%E5%8F%96-encoder-%E8%AF%8D%E5%85%B8-%E5%92%8C-decoder-%E8%AF%8D%E5%85%B8-%E9%95%BF%E5%BA%A6)
    - [3.4 确定 encoder 和 decoder 的 maxlen](#34-%E7%A1%AE%E5%AE%9A-encoder-%E5%92%8C-decoder-%E7%9A%84-maxlen)
    - [3.5 序列 填充/裁剪](#35-%E5%BA%8F%E5%88%97-%E5%A1%AB%E5%85%85%E8%A3%81%E5%89%AA)
  + [四、创建数据集 pipeline](#%E5%9B%9B%E5%88%9B%E5%BB%BA%E6%95%B0%E6%8D%AE%E9%9B%86-pipeline)
  + [五、组件构建](#%E4%BA%94%E7%BB%84%E4%BB%B6%E6%9E%84%E5%BB%BA)
    - [5.1 位置编码](#51-%E4%BD%8D%E7%BD%AE%E7%BC%96%E7%A0%81)
      * [5.1.1 问题](#511-%E9%97%AE%E9%A2%98)
      * [5.1.2 目的](#512-%E7%9B%AE%E7%9A%84)
      * [5.1.3 思路](#513-%E6%80%9D%E8%B7%AF)
      * [5.1.4 位置向量的作用](#514-%E4%BD%8D%E7%BD%AE%E5%90%91%E9%87%8F%E7%9A%84%E4%BD%9C%E7%94%A8)
      * [5.1.5 步骤](#515-%E6%AD%A5%E9%AA%A4)
      * [5.1.6 计算公式](#516-%E8%AE%A1%E7%AE%97%E5%85%AC%E5%BC%8F)
      * [5.1.7 代码实现](#517-%E4%BB%A3%E7%A0%81%E5%AE%9E%E7%8E%B0)
    - [5.2 Masking 操作](#52-masking-%E6%93%8D%E4%BD%9C)
      * [5.2.1 介绍](#521-%E4%BB%8B%E7%BB%8D)
      * [5.2.3 类别：padding mask and sequence mask](#523-%E7%B1%BB%E5%88%ABpadding-mask-and-sequence-mask)
        + [padding mask](#padding-mask)
        + [sequence mask](#sequence-mask)
  + [六、模型构建](#%E5%85%AD%E6%A8%A1%E5%9E%8B%E6%9E%84%E5%BB%BA)
    - [6.1 self-attention](#61-self-attention)
      * [6.1.1 动机](#611-%E5%8A%A8%E6%9C%BA)
      * [6.1.2 传统 Attention](#612-%E4%BC%A0%E7%BB%9F-attention)
      * [6.1.3 核心思想](#613-%E6%A0%B8%E5%BF%83%E6%80%9D%E6%83%B3)
      * [6.1.4 目的](#614-%E7%9B%AE%E7%9A%84)
      * [6.1.5 公式](#615-%E5%85%AC%E5%BC%8F)
      * [6.1.6 步骤](#616-%E6%AD%A5%E9%AA%A4)
      * [6.1.7 代码实现](#617-%E4%BB%A3%E7%A0%81%E5%AE%9E%E7%8E%B0)
    - [6.2 Multi-Headed Attention](#62--multi-headed-attention)
      * [思路](#%E6%80%9D%E8%B7%AF)
      * [步骤](#%E6%AD%A5%E9%AA%A4)
      * [代码实现](#%E4%BB%A3%E7%A0%81%E5%AE%9E%E7%8E%B0)
    - [6.3 前馈网络](#63-%E5%89%8D%E9%A6%88%E7%BD%91%E7%BB%9C)
      * [思路](#%E6%80%9D%E8%B7%AF-1)
      * [目的](#%E7%9B%AE%E7%9A%84)
      * [代码实现](#%E4%BB%A3%E7%A0%81%E5%AE%9E%E7%8E%B0-1)
    - [6.4 Transformer encoder 单元](#64-transformer-encoder-%E5%8D%95%E5%85%83)
      * [结构](#%E7%BB%93%E6%9E%84)
      * [代码实现](#%E4%BB%A3%E7%A0%81%E5%AE%9E%E7%8E%B0-2)
    - [6.5 Transformer decoder 单元](#65-transformer-decoder-%E5%8D%95%E5%85%83)
      * [结构](#%E7%BB%93%E6%9E%84-1)
      * [代码实现](#%E4%BB%A3%E7%A0%81%E5%AE%9E%E7%8E%B0-3)
  + [七、Encoder 和 Decoder 模块构建](#%E4%B8%83encoder-%E5%92%8C-decoder-%E6%A8%A1%E5%9D%97%E6%9E%84%E5%BB%BA)
    - [7.1 Encoder 模块构建](#71-encoder-%E6%A8%A1%E5%9D%97%E6%9E%84%E5%BB%BA)
    - [7.2 Dncoder 模块构建](#72-dncoder-%E6%A8%A1%E5%9D%97%E6%9E%84%E5%BB%BA)
  + [八、Transformer 构建](#%E5%85%ABtransformer-%E6%9E%84%E5%BB%BA)
  + [九、模型训练](#%E4%B9%9D%E6%A8%A1%E5%9E%8B%E8%AE%AD%E7%BB%83)
    - [9.1 配置类](#91-%E9%85%8D%E7%BD%AE%E7%B1%BB)
    - [9.2 优化函数定义](#92-%E4%BC%98%E5%8C%96%E5%87%BD%E6%95%B0%E5%AE%9A%E4%B9%89)
    - [9.3 Loss 损失函数 和 评测指标 定义](#93--loss-%E6%8D%9F%E5%A4%B1%E5%87%BD%E6%95%B0-%E5%92%8C-%E8%AF%84%E6%B5%8B%E6%8C%87%E6%A0%87-%E5%AE%9A%E4%B9%89)
      * [9.3.1 Loss 损失函数 定义](#931--loss-%E6%8D%9F%E5%A4%B1%E5%87%BD%E6%95%B0-%E5%AE%9A%E4%B9%89)
    - [9.4 Transformer 实例化](#94-transformer-%E5%AE%9E%E4%BE%8B%E5%8C%96)
    - [9.5 Mask 实现](#95-mask-%E5%AE%9E%E7%8E%B0)
    - [9.6 模型结果保存](#96-%E6%A8%A1%E5%9E%8B%E7%BB%93%E6%9E%9C%E4%BF%9D%E5%AD%98)
    - [9.7 Training Steps](#97-training-steps)
    - [9.8 训练](#98-%E8%AE%AD%E7%BB%83)

引言

之前给 小伙伴们 写过 一篇 【[【关于Transformer】 那些的你不知道的事](https://github.com/km1994/nlp_paper_study/tree/master/transformer_study/Transformer)】后，有一些小伙伴联系我，并和我请教了蛮多细节性问题，针对该问题，小菜鸡的我 也 想和小伙伴 一起 学习，所以就 找到了 一篇【[Transformer 在文本摘要任务 上的应用](https://github.com/rojagtap/abstractive_summarizer)】作为本次学习的 Coding！

一、文本摘要数据集介绍

本任务采用的 文本摘要数据集 为 [Kaggle 比赛 之 Inshorts Dataset](https://www.kaggle.com/shashichander009/inshorts-news-data)，该数据集 包含以下字段：

<table>

<tr>

<td>序号</td><td>字段名</td><td>字段介绍 </td><td>举例</td>

</tr>

<tr>

<td>1</td><td>Headline</td><td>标题 </td><td>4 ex-bank officials booked for cheating bank of ₹209 crore</td>

</tr>

<tr>

<td>2</td><td>Short</td><td>短文 </td><td>The CBI on Saturday booked four former officials of Syndicate Bank and six others for cheating, forgery, criminal conspiracy and causing ₹209 crore loss to the state-run bank. The accused had availed home loans and credit from Syndicate Bank on the basis of forged and fabricated documents. These funds were fraudulently transferred to the companies owned by the accused persons.</td>

</tr>

<tr>

<td>3</td><td>Source</td><td>数据来源 </td><td>The New Indian Express </td>

</tr>

<tr>

<td>4</td><td>Time</td><td>发表时间 </td><td>9:25:00</td>

</tr>

<tr>

<td>5</td><td>Publish Date</td><td>发表日期 </td><td>2017/3/26 </td>

</tr>

</table>

注：这里我们只 用到 Headline摘要 作为 文本摘要任务 实验数据

二、数据集加载介绍

2.1 数据加载

本文将数据集存储在 Excel 文件中，通过 pandas 的 read\_excel() 方法 获取数据集，代码如下：

news = pd.read\_excel("data/news.xlsx")

2.2 数据字段抽取

在 [一、文本摘要数据集介绍](#%E4%B8%80%E6%96%87%E6%9C%AC%E6%91%98%E8%A6%81%E6%95%B0%E6%8D%AE%E9%9B%86%E4%BB%8B%E7%BB%8D) 中，我们说过，我们只用到 Headline[摘要] [长文本] 作为 文本摘要任务 实验数据，所以我们需要 清除 其他字段。代码如下：

news.drop(['Source ', 'Time ', 'Publish Date'], axis=1, inplace=True)

可以采用以下命令，查看结果：

news.head()

news.shape # (55104, 2)



方便后期操作，我们这里直接 从 DataFrame 中分别抽取 出 Headline[摘要] [长文本] 数据：

document = news['Short']

summary = news['Headline']

document[30], summary[30]

>>>

('According to the Guinness World Records, the most generations alive in a single family have been seven. The difference between the oldest and the youngest person in the family was about 109 years, when Augusta Bunge&#39;s great-great-great-great grandson was born on January 21, 1989. The family belonged to the United States of America.',

'The most generations alive in a single family have been 7')

三、 数据预处理

3.1 summary 数据 处理

summary 数据 作为 decoder 序列数据，我们需要做一些小处理【前后分别加一个标识符】，如下所示：

# for decoder sequence

summary = summary.apply(lambda x: '<go> ' + x + ' <stop>')

summary[0]

>>>

'<go> 4 ex-bank officials booked for cheating bank of ₹209 crore <stop>'

3.2 编码处理

在 进行 文本摘要任务 之前，我们需要 将 文本进行编码：

1. 变量定义

# since < and > from default tokens cannot be removed

filters = '!"#$%&()\*+,-./:;=?@[\\]^\_`{|}~\t\n' # 文本中特殊符号清洗

oov\_token = '<unk>' # 未登录词 表示

1. 定义 文本预处理 tf.keras.preprocessing.text.Tokenizer() 编码类【用于后期 文本编码处理】

document\_tokenizer = tf.keras.preprocessing.text.Tokenizer(oov\_token=oov\_token)

summary\_tokenizer = tf.keras.preprocessing.text.Tokenizer(filters=filters, oov\_token=oov\_token)

Tokenizer : 一个将文本向量化，转换成序列的类。用来文本处理的分词、嵌入 。

keras.preprocessing.text.Tokenizer(num\_words=None,

filters='!"#$%&()\*+,-./:;<=>?@[\\]^\_`{|}~\t\n',

lower=True,

split=' ',

char\_level=False,

oov\_token=None,

document\_count=0)

* 参数说明：
  + num\_words: 默认是None处理所有字词，但是如果设置成一个整数，那么最后返回的是最常见的、出现频率最高的num\_words个字词。一共保留 num\_words-1 个词。
  + filters: 过滤一些特殊字符，默认上文的写法就可以了。
  + lower: 是否全部转为小写。
  + split: 分词的分隔符字符串，默认为空格。因为英文分词分隔符就是空格。
  + char\_level: 分字。
  + oov\_token: if given, it will be added to word\_index and used to replace out-of-vocabulary words during text\_to\_sequence calls

参考文档：[Keras分词器 Tokenizer](http://codewithzhangyi.com/2019/04/23/keras-tokenizer/)

1. 构建词典库

# 构建词典库

document\_tokenizer.fit\_on\_texts(document)

summary\_tokenizer.fit\_on\_texts(summary)

1. 文本列表 转 序列的列表 【列表中每个序列对应于一段输入文本】

# 文本列表 转 序列的列表 【列表中每个序列对应于一段输入文本】

inputs = document\_tokenizer.texts\_to\_sequences(document)

targets = summary\_tokenizer.texts\_to\_sequences(summary)

# 举例测试

summary\_tokenizer.texts\_to\_sequences(["This is a test"]) # [[184, 22, 12, 71]]

summary\_tokenizer.sequences\_to\_texts([[184, 22, 12, 71]]) # ['this is a test']

3.3 获取 encoder 词典 和 decoder 词典 长度

encoder\_vocab\_size = len(document\_tokenizer.word\_index) + 1

decoder\_vocab\_size = len(summary\_tokenizer.word\_index) + 1

# vocab\_size

encoder\_vocab\_size, decoder\_vocab\_size

>>>

(76362, 29661)

3.4 确定 encoder 和 decoder 的 maxlen

1. 分别进行 documents 和 summarys 中每个 序列长度

document\_lengths = pd.Series([len(x) for x in document])

summary\_lengths = pd.Series([len(x) for x in summary])

1. 对 document\_lengths 和 summary\_lengths 进行 统计分析

* 对 document 进行 统计分析

document\_lengths.describe()

>>>

count 55104.000000

mean 368.003049

std 26.235510

min 280.000000

25% 350.000000

50% 369.000000

75% 387.000000

max 469.000000

dtype: float64

* 对 summary 进行 统计分析

summary\_lengths.describe()

>>>

count 55104.000000

mean 63.620282

std 7.267463

min 20.000000

25% 59.000000

50% 63.000000

75% 69.000000

max 96.000000

dtype: float64

1. 确定 encoder 和 decoder 的 maxlen

# 取值>并同时四舍五入到第75个百分位数，而不会留下高方差

encoder\_maxlen = 400

decoder\_maxlen = 75

3.5 序列 填充/裁剪

# 对 序列 进行 填充/裁剪 ，是所有序列长度 都 等于 maxlen

inputs = tf.keras.preprocessing.sequence.pad\_sequences(inputs, maxlen=encoder\_maxlen, padding='post', truncating='post')

targets = tf.keras.preprocessing.sequence.pad\_sequences(targets, maxlen=decoder\_maxlen, padding='post', truncating='post')

四、创建数据集 pipeline

对数据集的顺序进行打乱，并 进行分 batch

# 数据类型 转为 为 tf.int32

inputs = tf.cast(inputs, dtype=tf.int32)

targets = tf.cast(targets, dtype=tf.int32)

BUFFER\_SIZE = 20000

BATCH\_SIZE = 64

dataset = tf.data.Dataset.from\_tensor\_slices((inputs, targets)).shuffle(BUFFER\_SIZE).batch(BATCH\_SIZE)

五、组件构建

5.1 位置编码

5.1.1 问题

* 介绍：缺乏 一种 表示 输入序列中 单词顺序 的方法
* 说明：因为模型不包括Recurrence/Convolution，因此是无法捕捉到序列顺序信息的，例如将K、V按行进行打乱，那么Attention之后的结果是一样的。但是序列信息非常重要，代表着全局的结构，因此必须将序列的分词相对或者绝对position信息利用起来

5.1.2 目的

加入词序信息，使 Attention 能够分辨出不同位置的词

5.1.3 思路

在 encoder 层和 decoder 层的输入添加了一个额外的向量Positional Encoding，维度和embedding的维度一样，让模型学习到这个值

5.1.4 位置向量的作用

* 决定当前词的位置；
* 计算在一个句子中不同的词之间的距离

5.1.5 步骤

* 将每个位置编号，
* 然后每个编号对应一个向量，
* 通过将位置向量和词向量相加，就给每个词都引入了一定的位置信息。

5.1.6 计算公式



* 论文的位置编码是使用三角函数去计算的。好处：
  + 值域只有[-1,1]
  + 容易计算相对位置。



注：<br/>

pos 表示当前词在句子中的位置<br/>

i 表示向量中每个值 的 index<br/>

在偶数位置：使用 正弦编码 sin();<br/>

在奇数位置：使用 余弦编码 cos();<br/>

5.1.7 代码实现

# 位置编码 类

class Positional\_Encoding():

def \_\_init\_\_(self):

pass

# 功能：计算角度 函数

def get\_angles(self, position, i, d\_model):

'''

功能：计算角度 函数

input:

position 单词在句子中的位置

i 维度

d\_model 向量维度

'''

angle\_rates = 1 / np.power(10000, (2 \* (i // 2)) / np.float32(d\_model))

return position \* angle\_rates

# 功能：位置编码 函数

def positional\_encoding(self, position, d\_model):

'''

功能：位置编码 函数

input:

position 单词在句子中的位置

d\_model 向量维度

'''

angle\_rads = self.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)

5.2 Masking 操作

5.2.1 介绍

掩盖某些值的信息，让模型信息不到该信息；

5.2.3 类别：padding mask and sequence mask

padding mask

* 作用域：每一个 scaled dot-product attention 中
* 动机：输入句子的长度不一问题
* 方法：
  + 短句子：后面 采用 0 填充
  + 长句子：只截取 左边 部分内容，其他的丢弃
* 原因：对于 填充 的位置，其所包含的信息量 对于 模型学习 作用不大，所以 self-attention 应该 抛弃对这些位置 进行学习；
* 做法：在这些位置上加上 一个 非常大 的负数（负无穷），使 该位置的值经过 Softmax 后，值近似 0，利用 padding mask 标记哪些值需要做处理；
* 实现：

# 功能： padding mask

def create\_padding\_mask(seq):

'''

功能： padding mask

input:

seq 序列

'''

seq = tf.cast(tf.math.equal(seq, 0), tf.float32)

return seq[:, tf.newaxis, tf.newaxis, :]

sequence mask

* 作用域：只作用于 decoder 的 self-attention 中
* 动机：不可预测性；
* 目标：sequence mask 是为了使得 decoder 不能看见未来的信息。也就是对于一个序列，在 time\_step 为 t 的时刻，我们的解码输出应该只能依赖于 t 时刻之前的输出，而不能依赖 t 之后的输出。因此我们需要想一个办法，把 t 之后的信息给隐藏起来。
* 做法：产生一个上三角矩阵，上三角的值全为0。把这个矩阵作用在每一个序列上，就可以达到我们的目的
* 实现：

# 功能：sequence mask

def create\_look\_ahead\_mask(size):

'''

功能： sequence mask

input:

seq 序列

'''

mask = 1 - tf.linalg.band\_part(tf.ones((size, size)), -1, 0)

return mask

六、模型构建

6.1 self-attention

6.1.1 动机

* CNN 所存在的长距离依赖问题；
* RNN 所存在的无法并行化问题【虽然能够在一定长度上缓解 长距离依赖问题】；

6.1.2 传统 Attention

* 方法：基于源端和目标端的隐向量计算Attention，
* 结果：源端每个词与目标端每个词间的依赖关系 【源端->目标端】
* 问题：忽略了 远端或目标端 词与词间 的依赖关系

6.1.3 核心思想

* 介绍： self-attention的结构在计算每个token时，总是会考虑整个序列其他token的表达；
* 举例：“我爱中国”这个序列，在计算"我"这个词的时候，不但会考虑词本身的embedding，也同时会考虑其他词对这个词的影响

6.1.4 目的

学习句子内部的词依赖关系，捕获句子的内部结构。

6.1.5 公式





6.1.6 步骤

建议阅读 [Transformer#self-attention-长怎么样](https://github.com/km1994/nlp_paper_study/tree/master/transformer_study/Transformer#self-attention-%E9%95%BF%E6%80%8E%E4%B9%88%E6%A0%B7)

6.1.7 代码实现

def scaled\_dot\_product\_attention(q, k, v, mask):

# s1：权重 score 计算：查询向量 query 点乘 key

matmul\_qk = tf.matmul(q, k, transpose\_b=True)

# s2：scale 操作：除以 sqrt(dk)，将 Softmax 函数推入梯度极小的区域

dk = tf.cast(tf.shape(k)[-1], tf.float32)

scaled\_attention\_logits = matmul\_qk / tf.math.sqrt(dk)

# s3：

if mask is not None:

scaled\_attention\_logits += (mask \* -1e9)

# s4：Softmax 归一化

attention\_weights = tf.nn.softmax(scaled\_attention\_logits, axis=-1)

# s5：加权求和

output = tf.matmul(attention\_weights, v)

return output, attention\_weights

6.2 Multi-Headed Attention

思路

* 相当于 图片 个 不同的 self-attention 的集成
* 就是把self-attention做 n 次，取决于 head 的个数；论文里面是做了8次。

步骤

* step 1 : 初始化 N 组 图片矩阵(论文为 8组)；
* step 2 : 每组 分别 进行 self-attention;
* step 3：
  + 问题：多个 self-attention 会得到 多个 矩阵，但是前馈神经网络没法输入8个矩阵；
  + 目标：把8个矩阵降为1个
  + 步骤：
    - 每次self-attention都会得到一个 Z 矩阵，把每个 Z 矩阵拼接起来，
    - 再乘以一个Wo矩阵，
    - 得到一个最终的矩阵，即 multi-head Attention 的结果；  
      

代码实现

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

# 初始化 Q，K，V 矩阵

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):

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]

# step 1：利用矩阵计算 q,k,v

q = self.wq(q)

k = self.wk(k)

v = self.wv(v)

# step 2：

q = self.split\_heads(q, batch\_size)

k = self.split\_heads(k, batch\_size)

v = self.split\_heads(v, batch\_size)

# step 3：每组 分别 进行 self-attention

scaled\_attention, attention\_weights = scaled\_dot\_product\_attention(

q, k, v, mask)

# step 4：矩阵拼接

scaled\_attention = tf.transpose(scaled\_attention, perm=[0, 2, 1, 3])

concat\_attention = tf.reshape(scaled\_attention, (batch\_size, -1, self.d\_model))

# step 5：全连接层

output = self.dense(concat\_attention)

return output, attention\_weights

6.3 前馈网络

思路

经过一层前馈网络以及 Add&Normalize，（线性转换+relu+线性转换 如下式）



目的

增加非线性的表达能力，毕竟之前的结构基本都是简单的矩阵乘法。若前馈网络的隐向量是512维，则结构最后输出100\*512；

代码实现

def point\_wise\_feed\_forward\_network(d\_model, dff):

return tf.keras.Sequential([

tf.keras.layers.Dense(dff, activation='relu'),

tf.keras.layers.Dense(d\_model)

])

6.4 Transformer encoder 单元

结构



代码实现

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):

# step 1：多头自注意力

attn\_output, \_ = self.mha(x, x, x, mask)

# step 2：前馈网络

attn\_output = self.dropout1(attn\_output, training=training)

# step 3：Layer Norml

out1 = self.layernorm1(x + attn\_output)

# step 4：前馈网络

ffn\_output = self.ffn(out1)

ffn\_output = self.dropout2(ffn\_output, training=training)

# step 5：Layer Norml

out2 = self.layernorm2(out1 + ffn\_output)

return out2

6.5 Transformer decoder 单元

结构



代码实现

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):

# step 1：带 sequence mask 的 多头自注意力

attn1, attn\_weights\_block1 = self.mha1(x, x, x, look\_ahead\_mask)

attn1 = self.dropout1(attn1, training=training)

# step 2：Layer Norm

out1 = self.layernorm1(attn1 + x)

# step 3：带 padding mask 的 多头自注意力

attn2, attn\_weights\_block2 = self.mha2(enc\_output, enc\_output, out1, padding\_mask)

attn2 = self.dropout2(attn2, training=training)

# step 4：Layer Norm

out2 = self.layernorm2(attn2 + out1)

# step 5：前馈网络

ffn\_output = self.ffn(out2)

ffn\_output = self.dropout3(ffn\_output, training=training)

# step 6：Layer Norm

out3 = self.layernorm3(ffn\_output + out2)

return out3, attn\_weights\_block1, attn\_weights\_block2

七、Encoder 和 Decoder 模块构建

7.1 Encoder 模块构建

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 # encoder 层数

# 词嵌入

self.embedding = tf.keras.layers.Embedding(input\_vocab\_size, d\_model)

# 位置编码

self.positional\_encoding\_obj = Positional\_Encoding()

self.pos\_encoding = self.positional\_encoding\_obj.positional\_encoding(maximum\_position\_encoding, self.d\_model)

# Encoder 模块构建

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]

# step 1：词嵌入

x = self.embedding(x)

x \*= tf.math.sqrt(tf.cast(self.d\_model, tf.float32))

# step 2：位置编码

x += self.pos\_encoding[:, :seq\_len, :]

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

# step 3：Encoder 模块构建

for i in range(self.num\_layers):

x = self.enc\_layers[i](x, training, mask)

return x

7.2 Dncoder 模块构建

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 # encoder 层数

# 词嵌入

self.embedding = tf.keras.layers.Embedding(target\_vocab\_size, d\_model)

# 位置编码

self.positional\_encoding\_obj = Positional\_Encoding()

self.pos\_encoding = self.positional\_encoding\_obj.positional\_encoding(maximum\_position\_encoding, d\_model)

# Dncoder 模块构建

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 = {}

# step 1：词嵌入

x = self.embedding(x)

x \*= tf.math.sqrt(tf.cast(self.d\_model, tf.float32))

# step 2：位置编码

x += self.pos\_encoding[:, :seq\_len, :]

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

# step 3：Dncoder 模块构建

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

return x, attention\_weights

八、Transformer 构建

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\_\_()

# Encoder 模块

self.encoder = Encoder(num\_layers, d\_model, num\_heads, dff, input\_vocab\_size, pe\_input, rate)

# Decoder 模块

self.decoder = Decoder(num\_layers, d\_model, num\_heads, dff, target\_vocab\_size, pe\_target, rate)

# 全连接层

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):

# step 1： encoder

enc\_output = self.encoder(inp, training, enc\_padding\_mask)

# step 2：decoder

dec\_output, attention\_weights = self.decoder(tar, enc\_output, training, look\_ahead\_mask, dec\_padding\_mask)

# step 3：全连接层

final\_output = self.final\_layer(dec\_output)

return final\_output, attention\_weights

九、模型训练

9.1 配置类

# hyper-params

class Config():

def \_\_init\_\_(self):

self.num\_layers = 4 # encoder 和 decoder 层数

self.d\_model = 128 # 向量维度

self.dff = 512 # 序列维度

self.num\_heads = 8 # 多头自注意力 头数

self.EPOCHS = 10 # 训练 次数

config = Config()

9.2 优化函数定义

Adam optimizer with custom learning rate scheduling

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(config.d\_model)

optimizer = tf.keras.optimizers.Adam(learning\_rate, beta\_1=0.9, beta\_2=0.98, epsilon=1e-9)

9.3 Loss 损失函数 和 评测指标 定义

9.3.1 Loss 损失函数 定义

# 功能：损失函数 定义

def loss\_function(real, pred):

mask = tf.math.logical\_not(tf.math.equal(real, 0))

# 稀疏分类交叉熵：将数字编码转化成one-hot编码格式，然后对one-hot编码格式的数据（真实标签值）与预测出的标签值使用交叉熵损失函数。

loss\_object = tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True, reduction='none')

loss\_ = loss\_object(real, pred)

mask = tf.cast(mask, dtype=loss\_.dtype)

loss\_ \*= mask

return tf.reduce\_sum(loss\_)/tf.reduce\_sum(mask)

# 实例化

train\_loss = tf.keras.metrics.Mean(name='train\_loss')

[稀疏分类交叉熵与稀疏分类交叉熵的使用差异,sparsecategoricalcrossentropy,和,SparseCategoricalCrossentropy,用法,区别](https://www.pythonf.cn/read/111510)

9.4 Transformer 实例化

transformer = Transformer(

config.num\_layers,

config.d\_model,

config.num\_heads,

config.dff,

encoder\_vocab\_size,

decoder\_vocab\_size,

pe\_input=encoder\_vocab\_size,

pe\_target=decoder\_vocab\_size,

)

9.5 Mask 实现

def create\_masks(inp, tar):

enc\_padding\_mask = create\_padding\_mask(inp)

dec\_padding\_mask = create\_padding\_mask(inp)

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

9.6 模型结果保存

checkpoint\_path = "checkpoints"

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

ckpt\_manager = tf.train.CheckpointManager(ckpt, checkpoint\_path, max\_to\_keep=5)

if ckpt\_manager.latest\_checkpoint:

ckpt.restore(ckpt\_manager.latest\_checkpoint)

print ('Latest checkpoint restored!!')

9.7 Training Steps

@tf.function

def train\_step(inp, tar):

tar\_inp = tar[:, :-1]

tar\_real = tar[:, 1:]

enc\_padding\_mask, combined\_mask, dec\_padding\_mask = create\_masks(inp, tar\_inp)

with tf.GradientTape() as tape:

predictions, \_ = transformer(

inp, tar\_inp,

True,

enc\_padding\_mask,

combined\_mask,

dec\_padding\_mask

)

loss = loss\_function(tar\_real, predictions)

gradients = tape.gradient(loss, transformer.trainable\_variables)

optimizer.apply\_gradients(zip(gradients, transformer.trainable\_variables))

train\_loss(loss)

9.8 训练

for epoch in range(config.EPOCHS):

start = time.time()

train\_loss.reset\_states()

for (batch, (inp, tar)) in enumerate(dataset):

train\_step(inp, tar)

# 55k samples

# we display 3 batch results -- 0th, middle and last one (approx)

# 55k / 64 ~ 858; 858 / 2 = 429

if batch % 429 == 0:

print (f'Epoch {epoch + 1} Batch {batch} Loss {train\_loss.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}'.format(epoch + 1, train\_loss.result()))

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

>>>

Epoch 1 Batch 0 Loss 2.4681

Epoch 1 Batch 429 Loss 2.4650

Epoch 1 Batch 858 Loss 2.5071

Epoch 1 Loss 2.5077

Time taken for 1 epoch: 308.9519073963165 secs

Epoch 2 Batch 0 Loss 2.3482

Epoch 2 Batch 429 Loss 2.4071

Epoch 2 Batch 858 Loss 2.4461

Epoch 2 Loss 2.4464

Time taken for 1 epoch: 299.0744743347168 secs

...