# 源码阅读作业-BERT源码阅读

潘煜2001210387

阅读工具：Notepad++

背景说明：课程项目的第一个作业就是用MindSpore实现BERT，故进行BERT源码阅读。本报告对代码最重要部分BertModel进行分析。

## 1、配置类（BertConfig）

这部分代码主要定义了BERT模型的一些默认参数，另外包括了一些文件处理函数。

**class** **BertConfig**(object):

"""BERT模型的配置类."""

**def** \_\_init\_\_(self,

vocab\_size,

hidden\_size**=**768,

num\_hidden\_layers**=**12,

num\_attention\_heads**=**12,

intermediate\_size**=**3072,

hidden\_act**=**"gelu",

hidden\_dropout\_prob**=**0.1,

attention\_probs\_dropout\_prob**=**0.1,

max\_position\_embeddings**=**512,

type\_vocab\_size**=**16,

initializer\_range**=**0.02):

self**.**vocab\_size **=** vocab\_size

self**.**hidden\_size **=** hidden\_size

self**.**num\_hidden\_layers **=** num\_hidden\_layers

self**.**num\_attention\_heads **=** num\_attention\_heads

self**.**hidden\_act **=** hidden\_act

self**.**intermediate\_size **=** intermediate\_size

self**.**hidden\_dropout\_prob **=** hidden\_dropout\_prob

self**.**attention\_probs\_dropout\_prob **=** attention\_probs\_dropout\_prob

self**.**max\_position\_embeddings **=** max\_position\_embeddings

self**.**type\_vocab\_size **=** type\_vocab\_size

self**.**initializer\_range **=** initializer\_range

@classmethod

**def** **from\_dict**(cls, json\_object):

"""Constructs a `BertConfig` from a Python dictionary of parameters."""

config **=** BertConfig(vocab\_size**=**None)

**for** (key, value) **in** six**.**iteritems(json\_object):

config**.**\_\_dict\_\_[key] **=** value

**return** config

@classmethod

**def** **from\_json\_file**(cls, json\_file):

"""Constructs a `BertConfig` from a json file of parameters."""

**with** tf**.**gfile**.**GFile(json\_file, "r") **as** reader:

text **=** reader**.**read()

**return** cls**.**from\_dict(json**.**loads(text))

**def** **to\_dict**(self):

"""Serializes this instance to a Python dictionary."""

output **=** copy**.**deepcopy(self**.**\_\_dict\_\_)

**return** output

**def** **to\_json\_string**(self):

"""Serializes this instance to a JSON string."""

**return** json**.**dumps(self**.**to\_dict(), indent**=**2, sort\_keys**=**True) **+** "\n"

**参数具体含义**

vocab\_size：词表大小  
hidden\_size：隐藏层神经元数  
num\_hidden\_layers：Transformer encoder中的隐藏层数  
num\_attention\_heads：multi-head attention 的head数  
intermediate\_size：encoder的“中间”隐层神经元数（例如feed-forward layer）  
hidden\_act：隐藏层激活函数  
hidden\_dropout\_prob：隐层dropout率  
attention\_probs\_dropout\_prob：注意力部分的dropout  
max\_position\_embeddings：最大位置编码  
type\_vocab\_size：token\_type\_ids的词典大小  
initializer\_range：truncated\_normal\_initializer初始化方法的stdev

这里要注意一点，可能刚看的时候对type\_vocab\_size这个参数会有点不理解，其实就是在next sentence prediction任务里的Segment A和 Segment B。在下载的bert\_config.json文件里也有说明，默认值应该为2。

## 2、获取词向量（Embedding\_lookup）

对于输入word\_ids，返回embedding table。可以选用one-hot或者tf.gather()

**def** **embedding\_lookup**(input\_ids, *# word\_id：【batch\_size, seq\_length】*

vocab\_size,

embedding\_size**=**128,

initializer\_range**=**0.02,

word\_embedding\_name**=**"word\_embeddings",

use\_one\_hot\_embeddings**=**False):

*# 该函数默认输入的形状为【batch\_size, seq\_length, input\_num】*

*# 如果输入为2D的【batch\_size, seq\_length】，则扩展到【batch\_size, seq\_length, 1】*

**if** input\_ids**.**shape**.**ndims **==** 2:

input\_ids **=** tf**.**expand\_dims(input\_ids, axis**=**[**-**1])

embedding\_table **=** tf**.**get\_variable(

name**=**word\_embedding\_name,

shape**=**[vocab\_size, embedding\_size],

initializer**=**create\_initializer(initializer\_range))

flat\_input\_ids **=** tf**.**reshape(input\_ids, [**-**1]) *#【batch\_size\*seq\_length\*input\_num】*

**if** use\_one\_hot\_embeddings:

one\_hot\_input\_ids **=** tf**.**one\_hot(flat\_input\_ids, depth**=**vocab\_size)

output **=** tf**.**matmul(one\_hot\_input\_ids, embedding\_table)

**else**: *# 按索引取值*

output **=** tf**.**gather(embedding\_table, flat\_input\_ids)

input\_shape **=** get\_shape\_list(input\_ids)

*# output：[batch\_size, seq\_length, num\_inputs]*

*# 转成:[batch\_size, seq\_length, num\_inputs\*embedding\_size]*

output **=** tf**.**reshape(output,

input\_shape[0:**-**1] **+** [input\_shape[**-**1] **\*** embedding\_size])

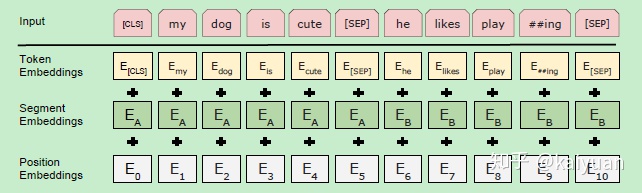
**return** (output, embedding\_table)

**参数具体含义**

input\_ids：word id 【batch\_size, seq\_length】  
vocab\_size：embedding词表  
embedding\_size：embedding维度  
initializer\_range：embedding初始化范围  
word\_embedding\_name：embeddding table命名  
use\_one\_hot\_embeddings：是否使用one-hotembedding  
Return：【batch\_size, seq\_length, embedding\_size】

## 3、词向量的后续处理（embedding\_postprocessor）

BERT模型的输入有三部分：token embedding ，segment embedding以及position embedding。上一节中只获得了token embedding，这部分代码对其完善信息，正则化，dropout之后输出最终embedding。  
注意到，在Transformer论文中的position embedding是由sin/cos函数生成的固定的值，而在这里代码实现中是跟普通word embedding一样随机生成的，可以训练的。作者这里这样选择的原因可能是BERT训练的数据比Transformer那篇大很多，完全可以让模型自己去学习。



**def** **embedding\_postprocessor**(input\_tensor, *# [batch\_size, seq\_length, embedding\_size]*

use\_token\_type**=**False,

token\_type\_ids**=**None,

token\_type\_vocab\_size**=**16, *# 一般是2*

token\_type\_embedding\_name**=**"token\_type\_embeddings",

use\_position\_embeddings**=**True,

position\_embedding\_name**=**"position\_embeddings",

initializer\_range**=**0.02,

max\_position\_embeddings**=**512, *#最大位置编码，必须大于等于max\_seq\_len*

dropout\_prob**=**0.1):

input\_shape **=** get\_shape\_list(input\_tensor, expected\_rank**=**3) *#【batch\_size,seq\_length,embedding\_size】*

batch\_size **=** input\_shape[0]

seq\_length **=** input\_shape[1]

width **=** input\_shape[2]

output **=** input\_tensor

*# Segment position信息*

**if** use\_token\_type:

**if** token\_type\_ids **is** None:

**raise** **ValueError**("`token\_type\_ids` must be specified if"

"`use\_token\_type` is True.")

token\_type\_table **=** tf**.**get\_variable(

name**=**token\_type\_embedding\_name,

shape**=**[token\_type\_vocab\_size, width],

initializer**=**create\_initializer(initializer\_range))

*# 由于token-type-table比较小，所以这里采用one-hot的embedding方式加速*

flat\_token\_type\_ids **=** tf**.**reshape(token\_type\_ids, [**-**1])

one\_hot\_ids **=** tf**.**one\_hot(flat\_token\_type\_ids, depth**=**token\_type\_vocab\_size)

token\_type\_embeddings **=** tf**.**matmul(one\_hot\_ids, token\_type\_table)

token\_type\_embeddings **=** tf**.**reshape(token\_type\_embeddings,

[batch\_size, seq\_length, width])

output **+=** token\_type\_embeddings

*# Position embedding信息*

**if** use\_position\_embeddings:

*# 确保seq\_length小于等于max\_position\_embeddings*

assert\_op **=** tf**.**assert\_less\_equal(seq\_length, max\_position\_embeddings)

**with** tf**.**control\_dependencies([assert\_op]):

full\_position\_embeddings **=** tf**.**get\_variable(

name**=**position\_embedding\_name,

shape**=**[max\_position\_embeddings, width],

initializer**=**create\_initializer(initializer\_range))

*# 这里position embedding是可学习的参数，[max\_position\_embeddings, width]*

*# 但是通常实际输入序列没有达到max\_position\_embeddings*

*# 所以为了提高训练速度，使用tf.slice取出句子长度的embedding*

position\_embeddings **=** tf**.**slice(full\_position\_embeddings, [0, 0],

[seq\_length, **-**1])

num\_dims **=** len(output**.**shape**.**as\_list())

*# word embedding之后的tensor是[batch\_size, seq\_length, width]*

*# 因为位置编码是与输入内容无关，它的shape总是[seq\_length, width]*

*# 我们无法把位置Embedding加到word embedding上*

*# 因此我们需要扩展位置编码为[1, seq\_length, width]*

*# 然后就能通过broadcasting加上去了。*

position\_broadcast\_shape **=** []

**for** \_ **in** range(num\_dims **-** 2):

position\_broadcast\_shape**.**append(1)

position\_broadcast\_shape**.**extend([seq\_length, width])

position\_embeddings **=** tf**.**reshape(position\_embeddings,

position\_broadcast\_shape)

output **+=** position\_embeddings

output **=** layer\_norm\_and\_dropout(output, dropout\_prob)

**return** output

## 4、构造attention\_mask

该部分代码的作用是构造attention可视域的attention\_mask，因为每个样本都经过padding过程，在做self-attention的时候padding的部分不能attend到其他部分上。  
输入为形状为【batch\_size, from\_seq\_length,…】的padding好的input\_ids和形状为【batch\_size, to\_seq\_length】的mask标记向量。

**def** **create\_attention\_mask\_from\_input\_mask**(from\_tensor, to\_mask):

from\_shape **=** get\_shape\_list(from\_tensor, expected\_rank**=**[2, 3])

batch\_size **=** from\_shape[0]

from\_seq\_length **=** from\_shape[1]

to\_shape **=** get\_shape\_list(to\_mask, expected\_rank**=**2)

to\_seq\_length **=** to\_shape[1]

to\_mask **=** tf**.**cast(

tf**.**reshape(to\_mask, [batch\_size, 1, to\_seq\_length]), tf**.**float32)

broadcast\_ones **=** tf**.**ones(

shape**=**[batch\_size, from\_seq\_length, 1], dtype**=**tf**.**float32)

mask **=** broadcast\_ones **\*** to\_mask

**return** mask

## 5、注意力层（attention layer）

这部分代码是**multi-head attention**的实现，主要来自《Attention is all you need》这篇论文。考虑key-query-value形式的attention，输入的from\_tensor当做是query， to\_tensor当做是key和value，当两者相同的时候即为self-attention。

**def** **attention\_layer**(from\_tensor, *# 【batch\_size, from\_seq\_length, from\_width】*

to\_tensor, *#【batch\_size, to\_seq\_length, to\_width】*

attention\_mask**=**None, *#【batch\_size,from\_seq\_length, to\_seq\_length】*

num\_attention\_heads**=**1, *# attention head numbers*

size\_per\_head**=**512, *# 每个head的大小*

query\_act**=**None, *# query变换的激活函数*

key\_act**=**None, *# key变换的激活函数*

value\_act**=**None, *# value变换的激活函数*

attention\_probs\_dropout\_prob**=**0.0, *# attention层的dropout*

initializer\_range**=**0.02, *# 初始化取值范围*

do\_return\_2d\_tensor**=**False, *# 是否返回2d张量。*

*#如果True，输出形状【batch\_size\*from\_seq\_length,num\_attention\_heads\*size\_per\_head】*

*#如果False，输出形状【batch\_size, from\_seq\_length, num\_attention\_heads\*size\_per\_head】*

batch\_size**=**None, *#如果输入是3D的，*

*#那么batch就是第一维，但是可能3D的压缩成了2D的，所以需要告诉函数batch\_size*

from\_seq\_length**=**None, *# 同上*

to\_seq\_length**=**None): *# 同上*

**def** **transpose\_for\_scores**(input\_tensor, batch\_size, num\_attention\_heads,

seq\_length, width):

output\_tensor **=** tf**.**reshape(

input\_tensor, [batch\_size, seq\_length, num\_attention\_heads, width])

output\_tensor **=** tf**.**transpose(output\_tensor, [0, 2, 1, 3]) *#[batch\_size, num\_attention\_heads, seq\_length, width]*

**return** output\_tensor

from\_shape **=** get\_shape\_list(from\_tensor, expected\_rank**=**[2, 3])

to\_shape **=** get\_shape\_list(to\_tensor, expected\_rank**=**[2, 3])

**if** len(from\_shape) **!=** len(to\_shape):

**raise** **ValueError**(

"The rank of `from\_tensor` must match the rank of `to\_tensor`.")

**if** len(from\_shape) **==** 3:

batch\_size **=** from\_shape[0]

from\_seq\_length **=** from\_shape[1]

to\_seq\_length **=** to\_shape[1]

**elif** len(from\_shape) **==** 2:

**if** (batch\_size **is** None **or** from\_seq\_length **is** None **or** to\_seq\_length **is** None):

**raise** **ValueError**(

"When passing in rank 2 tensors to attention\_layer, the values "

"for `batch\_size`, `from\_seq\_length`, and `to\_seq\_length` "

"must all be specified.")

*# 为了方便备注shape，采用以下简写:*

*# B = batch size (number of sequences)*

*# F = `from\_tensor` sequence length*

*# T = `to\_tensor` sequence length*

*# N = `num\_attention\_heads`*

*# H = `size\_per\_head`*

*# 把from\_tensor和to\_tensor压缩成2D张量*

from\_tensor\_2d **=** reshape\_to\_matrix(from\_tensor) *# 【B\*F, hidden\_size】*

to\_tensor\_2d **=** reshape\_to\_matrix(to\_tensor) *# 【B\*T, hidden\_size】*

*# 将from\_tensor输入全连接层得到query\_layer*

*# `query\_layer` = [B\*F, N\*H]*

query\_layer **=** tf**.**layers**.**dense(

from\_tensor\_2d,

num\_attention\_heads **\*** size\_per\_head,

activation**=**query\_act,

name**=**"query",

kernel\_initializer**=**create\_initializer(initializer\_range))

*# 将from\_tensor输入全连接层得到query\_layer*

*# `key\_layer` = [B\*T, N\*H]*

key\_layer **=** tf**.**layers**.**dense(

to\_tensor\_2d,

num\_attention\_heads **\*** size\_per\_head,

activation**=**key\_act,

name**=**"key",

kernel\_initializer**=**create\_initializer(initializer\_range))

*# 同上*

*# `value\_layer` = [B\*T, N\*H]*

value\_layer **=** tf**.**layers**.**dense(

to\_tensor\_2d,

num\_attention\_heads **\*** size\_per\_head,

activation**=**value\_act,

name**=**"value",

kernel\_initializer**=**create\_initializer(initializer\_range))

*# query\_layer转成多头：[B\*F, N\*H]==>[B, F, N, H]==>[B, N, F, H]*

query\_layer **=** transpose\_for\_scores(query\_layer, batch\_size,

num\_attention\_heads, from\_seq\_length,

size\_per\_head)

*# key\_layer转成多头：[B\*T, N\*H] ==> [B, T, N, H] ==> [B, N, T, H]*

key\_layer **=** transpose\_for\_scores(key\_layer, batch\_size, num\_attention\_heads,

to\_seq\_length, size\_per\_head)

*# 将query与key做点积，然后做一个scale，公式可以参见原始论文*

*# `attention\_scores` = [B, N, F, T]*

attention\_scores **=** tf**.**matmul(query\_layer, key\_layer, transpose\_b**=**True)

attention\_scores **=** tf**.**multiply(attention\_scores,

1.0 **/** math**.**sqrt(float(size\_per\_head)))

**if** attention\_mask **is** **not** None:

*# `attention\_mask` = [B, 1, F, T]*

attention\_mask **=** tf**.**expand\_dims(attention\_mask, axis**=**[1])

*# 如果attention\_mask里的元素为1，则通过下面运算有（1-1）\*-10000，adder就是0*

*# 如果attention\_mask里的元素为0，则通过下面运算有（1-0）\*-10000，adder就是-10000*

adder **=** (1.0 **-** tf**.**cast(attention\_mask, tf**.**float32)) **\*** **-**10000.0

*# 我们最终得到的attention\_score一般不会很大，*

*#所以上述操作对mask为0的地方得到的score可以认为是负无穷*

attention\_scores **+=** adder

*# 负无穷经过softmax之后为0，就相当于mask为0的位置不计算attention\_score*

*# `attention\_probs` = [B, N, F, T]*

attention\_probs **=** tf**.**nn**.**softmax(attention\_scores)

*# 对attention\_probs进行dropout，这虽然有点奇怪，但是Transforme原始论文就是这么做的*

attention\_probs **=** dropout(attention\_probs, attention\_probs\_dropout\_prob)

*# `value\_layer` = [B, T, N, H]*

value\_layer **=** tf**.**reshape(

value\_layer,

[batch\_size, to\_seq\_length, num\_attention\_heads, size\_per\_head])

*# `value\_layer` = [B, N, T, H]*

value\_layer **=** tf**.**transpose(value\_layer, [0, 2, 1, 3])

*# `context\_layer` = [B, N, F, H]*

context\_layer **=** tf**.**matmul(attention\_probs, value\_layer)

*# `context\_layer` = [B, F, N, H]*

context\_layer **=** tf**.**transpose(context\_layer, [0, 2, 1, 3])

**if** do\_return\_2d\_tensor:

*# `context\_layer` = [B\*F, N\*H]*

context\_layer **=** tf**.**reshape(

context\_layer,

[batch\_size **\*** from\_seq\_length, num\_attention\_heads **\*** size\_per\_head])

**else**:

*# `context\_layer` = [B, F, N\*H]*

context\_layer **=** tf**.**reshape(

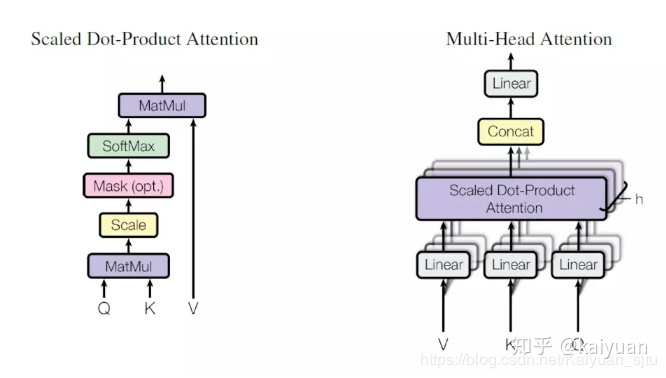
context\_layer,

[batch\_size, from\_seq\_length, num\_attention\_heads **\*** size\_per\_head])

**return** context\_layer

总结一下，attention layer的主要流程：

* 对输入的tensor进行形状校验，提取batch\_size、from\_seq\_length 、to\_seq\_length
* 输入如果是3d张量则转化成2d矩阵
* from\_tensor作为query， to\_tensor作为key和value，经过一层全连接层后得到query\_layer、key\_layer 、value\_layer
* 将上述张量通过transpose\_for\_scores转化成multi-head
* 根据论文公式计算attention\_score以及attention\_probs（注意attention\_mask的trick）：
* 将得到的attention\_probs与value相乘，返回2D或3D张量



## 6、Transformer

接下来的代码就是大名鼎鼎的Transformer的核心代码了，可以认为是"Attention is All You Need"原始代码重现。

**def** **transformer\_model**(input\_tensor, *# 【batch\_size, seq\_length, hidden\_size】*

attention\_mask**=**None, *# 【batch\_size, seq\_length, seq\_length】*

hidden\_size**=**768,

num\_hidden\_layers**=**12,

num\_attention\_heads**=**12,

intermediate\_size**=**3072,

intermediate\_act\_fn**=**gelu, *# feed-forward层的激活函数*

hidden\_dropout\_prob**=**0.1,

attention\_probs\_dropout\_prob**=**0.1,

initializer\_range**=**0.02,

do\_return\_all\_layers**=**False):

*# 这里注意，因为最终要输出hidden\_size， 我们有num\_attention\_head个区域，*

*# 每个head区域有size\_per\_head多的隐层*

*# 所以有 hidden\_size = num\_attention\_head \* size\_per\_head*

**if** hidden\_size **%** num\_attention\_heads **!=** 0:

**raise** **ValueError**(

"The hidden size (%d) is not a multiple of the number of attention "

"heads (%d)" **%** (hidden\_size, num\_attention\_heads))

attention\_head\_size **=** int(hidden\_size **/** num\_attention\_heads)

input\_shape **=** get\_shape\_list(input\_tensor, expected\_rank**=**3)

batch\_size **=** input\_shape[0]

seq\_length **=** input\_shape[1]

input\_width **=** input\_shape[2]

*# 因为encoder中有残差操作，所以需要shape相同*

**if** input\_width **!=** hidden\_size:

**raise** **ValueError**("The width of the input tensor (%d) != hidden size (%d)" **%**

(input\_width, hidden\_size))

*# reshape操作在CPU/GPU上很快，但是在TPU上很不友好*

*# 所以为了避免2D和3D之间的频繁reshape，我们把所有的3D张量用2D矩阵表示*

prev\_output **=** reshape\_to\_matrix(input\_tensor)

all\_layer\_outputs **=** []

**for** layer\_idx **in** range(num\_hidden\_layers):

**with** tf**.**variable\_scope("layer\_%d" **%** layer\_idx):

layer\_input **=** prev\_output

**with** tf**.**variable\_scope("attention"):

*# multi-head attention*

attention\_heads **=** []

**with** tf**.**variable\_scope("self"):

*# self-attention*

attention\_head **=** attention\_layer(

from\_tensor**=**layer\_input,

to\_tensor**=**layer\_input,

attention\_mask**=**attention\_mask,

num\_attention\_heads**=**num\_attention\_heads,

size\_per\_head**=**attention\_head\_size,

attention\_probs\_dropout\_prob**=**attention\_probs\_dropout\_prob,

initializer\_range**=**initializer\_range,

do\_return\_2d\_tensor**=**True,

batch\_size**=**batch\_size,

from\_seq\_length**=**seq\_length,

to\_seq\_length**=**seq\_length)

attention\_heads**.**append(attention\_head)

attention\_output **=** None

**if** len(attention\_heads) **==** 1:

attention\_output **=** attention\_heads[0]

**else**:

*# 如果有多个head，将他们拼接起来*

attention\_output **=** tf**.**concat(attention\_heads, axis**=-**1)

*# 对attention的输出进行线性映射, 目的是将shape变成与input一致*

*# 然后dropout+residual+norm*

**with** tf**.**variable\_scope("output"):

attention\_output **=** tf**.**layers**.**dense(

attention\_output,

hidden\_size,

kernel\_initializer**=**create\_initializer(initializer\_range))

attention\_output **=** dropout(attention\_output, hidden\_dropout\_prob)

attention\_output **=** layer\_norm(attention\_output **+** layer\_input)

*# feed-forward*

**with** tf**.**variable\_scope("intermediate"):

intermediate\_output **=** tf**.**layers**.**dense(

attention\_output,

intermediate\_size,

activation**=**intermediate\_act\_fn,

kernel\_initializer**=**create\_initializer(initializer\_range))

*# 对feed-forward层的输出使用线性变换变回‘hidden\_size’*

*# 然后dropout + residual + norm*

**with** tf**.**variable\_scope("output"):

layer\_output **=** tf**.**layers**.**dense(

intermediate\_output,

hidden\_size,

kernel\_initializer**=**create\_initializer(initializer\_range))

layer\_output **=** dropout(layer\_output, hidden\_dropout\_prob)

layer\_output **=** layer\_norm(layer\_output **+** attention\_output)

prev\_output **=** layer\_output

all\_layer\_outputs**.**append(layer\_output)

**if** do\_return\_all\_layers:

final\_outputs **=** []

**for** layer\_output **in** all\_layer\_outputs:

final\_output **=** reshape\_from\_matrix(layer\_output, input\_shape)

final\_outputs**.**append(final\_output)

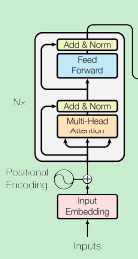
**return** final\_outputs

**else**:

final\_output **=** reshape\_from\_matrix(prev\_output, input\_shape)

**return** final\_output

配上下图一同使用效果更佳，因为BERT里只有encoder，所有decoder没有姓名。



## 7、函数入口（init）

BertModel类的构造函数，有了上面的铺垫，就可以来实现BERT模型了。

**def** \_\_init\_\_(self,

config, *# BertConfig对象*

is\_training,

input\_ids, *# 【batch\_size, seq\_length】*

input\_mask**=**None, *# 【batch\_size, seq\_length】*

token\_type\_ids**=**None, *# 【batch\_size, seq\_length】*

use\_one\_hot\_embeddings**=**False, *# 是否使用one-hot；否则tf.gather()*

scope**=**None):

config **=** copy**.**deepcopy(config)

**if** **not** is\_training:

config**.**hidden\_dropout\_prob **=** 0.0

config**.**attention\_probs\_dropout\_prob **=** 0.0

input\_shape **=** get\_shape\_list(input\_ids, expected\_rank**=**2)

batch\_size **=** input\_shape[0]

seq\_length **=** input\_shape[1]

*# 不做mask，即所有元素为1*

**if** input\_mask **is** None:

input\_mask **=** tf**.**ones(shape**=**[batch\_size, seq\_length], dtype**=**tf**.**int32)

**if** token\_type\_ids **is** None:

token\_type\_ids **=** tf**.**zeros(shape**=**[batch\_size, seq\_length], dtype**=**tf**.**int32)

**with** tf**.**variable\_scope(scope, default\_name**=**"bert"):

**with** tf**.**variable\_scope("embeddings"):

*# word embedding*

(self**.**embedding\_output, self**.**embedding\_table) **=** embedding\_lookup(

input\_ids**=**input\_ids,

vocab\_size**=**config**.**vocab\_size,

embedding\_size**=**config**.**hidden\_size,

initializer\_range**=**config**.**initializer\_range,

word\_embedding\_name**=**"word\_embeddings",

use\_one\_hot\_embeddings**=**use\_one\_hot\_embeddings)

*# 添加position embedding和segment embedding*

*# layer norm + dropout*

self**.**embedding\_output **=** embedding\_postprocessor(

input\_tensor**=**self**.**embedding\_output,

use\_token\_type**=**True,

token\_type\_ids**=**token\_type\_ids,

token\_type\_vocab\_size**=**config**.**type\_vocab\_size,

token\_type\_embedding\_name**=**"token\_type\_embeddings",

use\_position\_embeddings**=**True,

position\_embedding\_name**=**"position\_embeddings",

initializer\_range**=**config**.**initializer\_range,

max\_position\_embeddings**=**config**.**max\_position\_embeddings,

dropout\_prob**=**config**.**hidden\_dropout\_prob)

**with** tf**.**variable\_scope("encoder"):

*# input\_ids是经过padding的word\_ids： [25, 120, 34, 0, 0]*

*# input\_mask是有效词标记： [1, 1, 1, 0, 0]*

attention\_mask **=** create\_attention\_mask\_from\_input\_mask(

input\_ids, input\_mask)

*# transformer模块叠加*

*# `sequence\_output` shape = [batch\_size, seq\_length, hidden\_size].*

self**.**all\_encoder\_layers **=** transformer\_model(

input\_tensor**=**self**.**embedding\_output,

attention\_mask**=**attention\_mask,

hidden\_size**=**config**.**hidden\_size,

num\_hidden\_layers**=**config**.**num\_hidden\_layers,

num\_attention\_heads**=**config**.**num\_attention\_heads,

intermediate\_size**=**config**.**intermediate\_size,

intermediate\_act\_fn**=**get\_activation(config**.**hidden\_act),

hidden\_dropout\_prob**=**config**.**hidden\_dropout\_prob,

attention\_probs\_dropout\_prob**=**config**.**attention\_probs\_dropout\_prob,

initializer\_range**=**config**.**initializer\_range,

do\_return\_all\_layers**=**True)

*# `self.sequence\_output`是最后一层的输出，shape为【batch\_size, seq\_length, hidden\_size】*

self**.**sequence\_output **=** self**.**all\_encoder\_layers[**-**1]

*# ‘pooler’部分将encoder输出【batch\_size, seq\_length, hidden\_size】*

*# 转成【batch\_size, hidden\_size】*

**with** tf**.**variable\_scope("pooler"):

*# 取最后一层的第一个时刻[CLS]对应的tensor， 对于分类任务很重要*

*# sequence\_output[:, 0:1, :]得到的是[batch\_size, 1, hidden\_size]*

*# 我们需要用squeeze把第二维去掉*

first\_token\_tensor **=** tf**.**squeeze(self**.**sequence\_output[:, 0:1, :], axis**=**1)

*# 然后再加一个全连接层，输出仍然是[batch\_size, hidden\_size]*

self**.**pooled\_output **=** tf**.**layers**.**dense(

first\_token\_tensor,

config**.**hidden\_size,

activation**=**tf**.**tanh,

kernel\_initializer**=**create\_initializer(config**.**initializer\_range))