####### predict.py

**#### 第一部分：embedding 怎么处理的？**

if \_\_name\_\_ == "\_\_main\_\_":

config = util.initialize\_from\_env()

## util.py

# sys:是我们在command下的指令：

python3.6 predict.py spanbert\_large sample1.in.json out1.txt

#下面这个函数是设定model所用的各种参数，存在config这个dictionary中。

def initialize\_from\_env(eval\_test=False):

if "GPU" in os.environ:

set\_gpus(int(os.environ["GPU"]))

name = sys.argv[1] # spanbert\_large

print("Running experiment: {}".format(name))

…

return config

log\_dir = config["log\_dir"]

# Input file in .jsonlines format.

input\_filename = sys.argv[2]

#input\_filename = "sample3.in.json"

# Predictions will be written to this file in .jsonlines format.

output\_filename = sys.argv[3]

#output\_filename="output3.txt"

**model = util.get\_model(config)**

## util.py

def get\_model(config):

if config['model\_type'] == 'independent':

return independent.CorefModel(config)

elif config['model\_type'] == 'overlap':

return overlap.CorefModel(config)

else:

raise NotImplementedError('Undefined model type')

##independent.py

**class CorefModel(object):**

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

## 下面是初始模型的各种参数

self.config = config

…

self.input\_tensors = queue.dequeue()

#主要改动的是下面这个函数的一些内容

self.predictions, self.loss = self.get\_predictions\_and\_loss(\*self.input\_tensors)

**def get\_predictions\_and\_loss(self, input\_ids, input\_mask, text\_len, speaker\_ids, genre, is\_training, gold\_starts, gold\_ends, cluster\_ids, sentence\_map):**

model = modeling.BertModel(

config=self.bert\_config,

is\_training=is\_training,

input\_ids=input\_ids,

input\_mask=input\_mask,

use\_one\_hot\_embeddings=False,

scope='bert')

all\_encoder\_layers = model.get\_all\_encoder\_layers()

# mention\_doc是要输出的word embedding

mention\_doc0 = model.get\_sequence\_output()

# 上面那个transformer\_model（）Returns:

float Tensor of shape [batch\_size, seq\_length, hidden\_size], the final

hidden layer of the Transformer.

是加入attention机制后的encoder，return 出来的内容就是我们所要的word embedding. 第一维度是batch\_size。对应于我们的句子，batch\_size 是1，以此，我们需要的是sequence\_output。 sequence\_output 是被get\_sequence\_output()函数所获取，进而赋值给

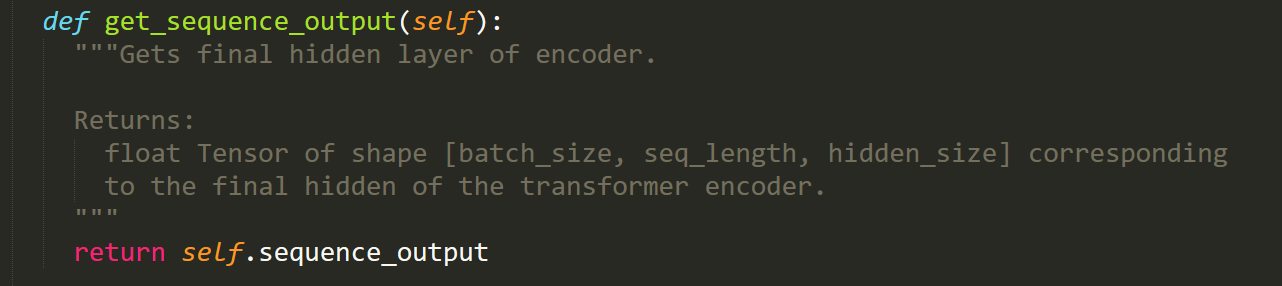
mention\_doc0。

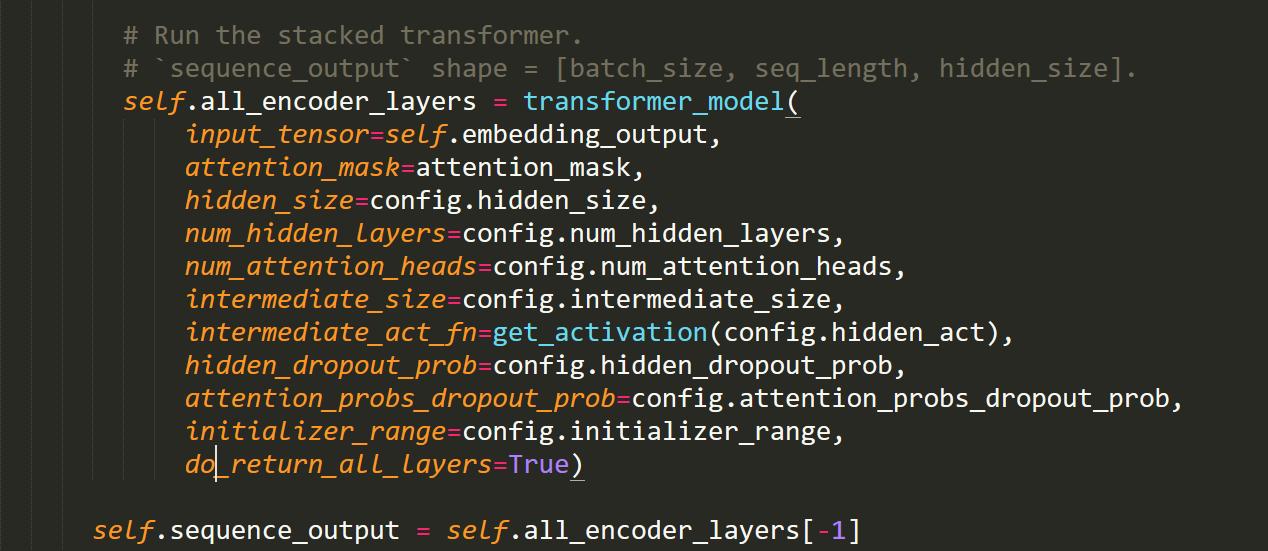
top\_antecedent\_scores,mention\_doc], loss

bert 文件夹下面的modelling.py，里面定义了下面的class，下面class里面有2个函数，如下

**class BertModel(object):**

"""BERT model ("Bidirectional Embedding Representations from a Transformer").





**### 接上面的内容，还是在get\_predictions\_and\_loss()里面**

**#def get\_predictions\_and\_loss(self, input\_ids, input\_mask, text\_len, speaker\_ids, #genre, is\_training, gold\_starts, gold\_ends, cluster\_ids, sentence\_map):**

…

self.dropout = self.get\_dropout(self.config["dropout\_rate"], is\_training)

num\_sentences = tf.shape(mention\_doc0)[0]

max\_sentence\_length = tf.shape(mention\_doc0)[1]

mention\_doc = self.flatten\_emb\_by\_sentence(mention\_doc0, input\_mask) #这边是在做shape的变化

…

在最后return时，增加一项，mention\_doc

return [candidate\_starts, candidate\_ends, candidate\_mention\_scores, top\_span\_starts, top\_span\_ends, top\_antecedents, top\_antecedent\_scores,**mention\_doc**], loss

下面再回到predict.py

# emboutput 是word embedding 向量，在session run model.predictions时，增加一个return的内容，mention\_doc，这里起了一个新的名字叫做emboutput。

\_, \_, \_, top\_span\_starts, top\_span\_ends, top\_antecedents, top\_antecedent\_scores, **emboutput** = session.run(model.predictions, feed\_dict=feed\_dict)

print(emboutput.shape)

注意这个return出来的word embedding 是我们原本句子加上mask以后的word list的embedding，比如

#inputtext是加入mask"##"的words list

#[["[CLS]", "I", "am", "hungry", ",", "please", "drive", "to", "the", "ne", "##ast", "Di", "##m", "Su", "##m",

#"restaurant", ".", "NO", "##UN", "and", "park", "next", "to", "it", "[SEP]"]],

我们还要把这个mask 的list去掉##, 并合并单词，还原成原本句子的word list，比如

# 删除mask以后，原本句子的word list

#["I am hungry, please drive to the neast Dim Sum restaurant.NOUN and park next to it"]

处理方式是把带“##”的单词与其前面一个单词合并。比如，把"Di", "##m",合并成“Dim”, 拼接回去后的词的embedding是原本词embedding的平均数。把最后embedding 所需要的vector index存到embedding\_final\_indice中。

embedding\_final\_indice=[]

for i in range(1,len(inputtext)-1):

#把"##"分割的词拼接回去,举例来说，把"Di"和 "##m"拼成"Dim"

if "##" in inputtext[i]:

inputtext[i]=''.join([inputtext[i-1],inputtext[i].replace("##",'')])

# 拼接回去后的词的embedding是原本词embedding的平均数

emboutput[i-1]=(emboutput[i-1]+emboutput[i-1])/2

#### 第二部分：

代词指代替换

代词和其指代的内容会被分到同一个cluster里面。 以下面的句子来说：

"sentences": [["[CLS]", "I", "got", "a", "fine", "two", "weeks", "ago", "because", "I", "was", "speeding", "!", "I", "need", "to", "pay", "it", "as", "##ap", "[SEP]"]],

有一个predicted cluster是((3, 4), (17, 17))

(17, 17)是指示代词部分的起始位置，对应的是it

(3, 4)是代词所指示内容的起始位置，对于的是"a", "fine"。

接下来，就是遍历每一个cluster，把代词找出来，对应的指示内容找出来，用reftext表示。然后，把代词位置替换成reftext的内容。注意，替换之前还要处理mask ##或者“.”句号单独切成一个词的问题，该合并的合并，该删除的删除。 还有一个问题是首字母大写“The apple”的问题，在替换时要变成“the apple”。

for cluster in example["predicted\_clusters"]:

print(cluster)

print(inputtext[cluster[-1][-1]])

reftext= inputtext[cluster[-2][0]:cluster[-2][-1]+1]

if cluster[-2][0]==1:

# 判断第一句话的第一个word是不是"The"

if inputtext[1]=="The":

reftext[0]="the"

print(reftext)

pop\_l0=[]

for i in range(len(reftext)):

#把"##"分割的词拼接回去

if "##" in reftext[i]:

reftext[i]=''.join([reftext[i-1],reftext[i].replace("##",'')])

pop\_l0.append(reftext[i-1])

#删除多余的词（前面已经增加拼接过的词，拼接前的词就删除掉）

for poptext0 in pop\_l0:

reftext.remove(poptext0)

pop\_l1=[]

for i in range(len(reftext)):

#把"."切割的词拼回去

if "."==reftext[i]:

reftext[i]=''.join(reftext[(i-1):(i+2)])

pop\_l1.append(reftext[i-1])

pop\_l1.append(reftext[i+1])

#删除多余的词（前面已经增加拼接过的词，拼接前的词就删除掉）

for poptext1 in pop\_l1:

reftext.remove(poptext1)

inputtext[cluster[-1][-1]] = ' '.join(reftext) #用预测的内容替换指示代词

pop\_l21=[]

embedding\_final\_indice=[]

for i in range(1,len(inputtext)-1):

#把"##"分割的词拼接回去,举例来说，把"Di"和 "##m"拼成"Dim"

if "##" in inputtext[i]:

inputtext[i]=''.join([inputtext[i-1],inputtext[i].replace("##",'')])

# 拼接回去后的词的embedding是原本词embedding的平均数

emboutput[i-1]=(emboutput[i-1]+emboutput[i-1])/2

pop\_l21.append(inputtext[i-1])

else:

embedding\_final\_indice.append(i)

#删除多余的词（前面已经增加拼接过的词，拼接前的词就删除掉）

for poptext in pop\_l21:

print(poptext)

if poptext in inputtext:

inputtext.remove(poptext)

pop\_l22=[]

for i in range(len(inputtext)):

#把"."切割的词拼回去

if "."==inputtext[i]:

inputtext[i]=''.join(inputtext[(i-1):(i+2)])

pop\_l22.append(inputtext[i-1])

pop\_l22.append(inputtext[i+1])

emboutput[i-1]=(emboutput[i-1]+emboutput[i]+emboutput[i+1])/3

embedding\_final\_indice.pop(i)

embedding\_final\_indice.pop(i+1)

#删除多余的词（前面已经增加拼接过的词，拼接前的词就删除掉）

for poptext1 in pop\_l22:

inputtext.remove(poptext1)

# for j in range(2,len(inputtext)-1):

# inputtext[j]=inputtext[j].lower()

example["output"]= ' '.join(inputtext[1:-1])

print(example["output"])