BERT源码分析完整版

@Author：NewBeeNLP

目录

---

*BERT*模型也出来很久了， 之前有看过论文和一些博客对其做了解读：*NLP*大杀器*BERT*模型解读[1]，但是一直没有细致地去看源码具体实现。最近有用到就抽时间来仔细看看记录下来，和大家一起讨论。

注意，源码阅读系列需要提前对*NLP*相关知识有所了解，比如*attention*机制、*transformer*框架以及*python*和*tensorflow*基础等，关于*BERT*的原理不是本文的重点。

附上关于*BERT*的资料汇总：*BERT*相关论文、文章和代码资源汇总[2]

源码地址[3]：这里
如有解读不正确，请务必指出~

一、整体模型

第一部分要介绍的是*BERT*最主要的模型实现部分-----*BertModel*，代码位于

- *modeling.py*模块[4]

1、BERT基础配置（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
```

```python
    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*。参考这个*Issue*[5]

2、BERT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()*

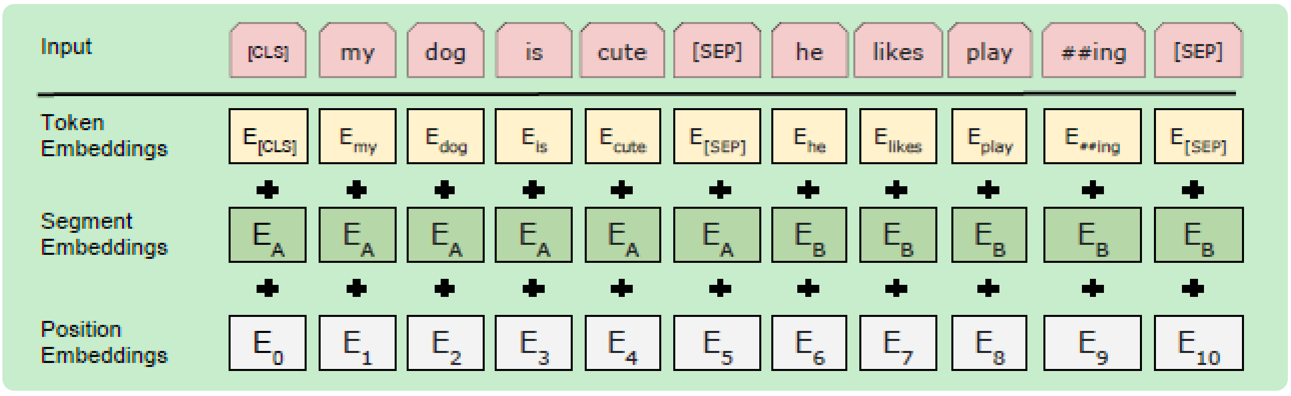
```
def embedding_lookup(input_ids,      # word_id：【batch_size, seq_length】
                     vocab_size,
                     embedding_size=128,
                     initializer_range=0.02,
                     word_embedding_name="word_embeddings",
                     use_one_hot_embeddings=False):

  # 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*i
  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] * embe
  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）

我们知道*BERT*模型的输入有三部分： `token embedding` ， `segment embedding` 以及 `position embedding` 。上一节中我们只获得了*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，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, embe
                            use_token_type=False,
                            token_type_ids=None,
                            token_type_vocab_size=16,        # 一般是2
                            token_type_embedding_name="token_type_embeddi
                            use_position_embeddings=True,
                            position_embedding_name="position_embeddings"
                            initializer_range=0.02,
                            max_position_embeddings=512,     #最大位置编码，
                            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_
  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*， 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*的是*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、Attention实现

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*。关于*attention*更详细的介绍可以转到理解*Attention*机制原理及模型[6]。

```
def attention_layer(from_tensor,   # 【batch_size, from_seq_length, from_w
                    to_tensor,       #【batch_size, to_seq_length, to_widt
                    attention_mask=None,         #【batch_size,from_seq_l
                    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
                    initializer_range=0.02,
                    do_return_2d_tensor=False,               # 是否
#如果True，输出形状【batch_size*from_seq_length,num_attention_heads*size_per
#如果False，输出形状【batch_size, from_seq_length, num_attention_heads*size_
                    batch_size=None,                         #如果输
#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])      #[batc
    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
      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, hid
  to_tensor_2d = reshape_to_matrix(to_tensor)                 # 【B*T, hid

  # 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_h
                                   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-10
    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
  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]
```

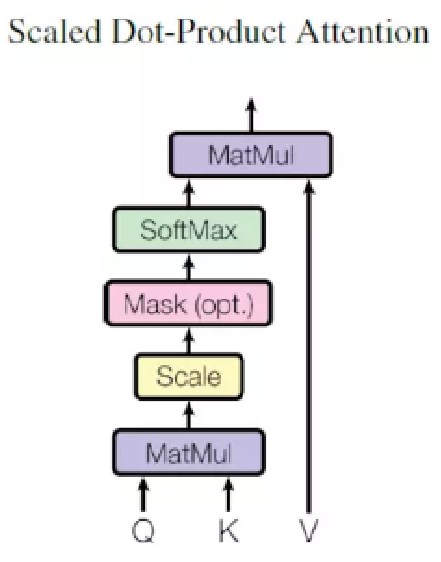
```
  value_layer = tf.transpose(value_layer, [0, 2, 1, 3])

  # `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_hea
  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
```

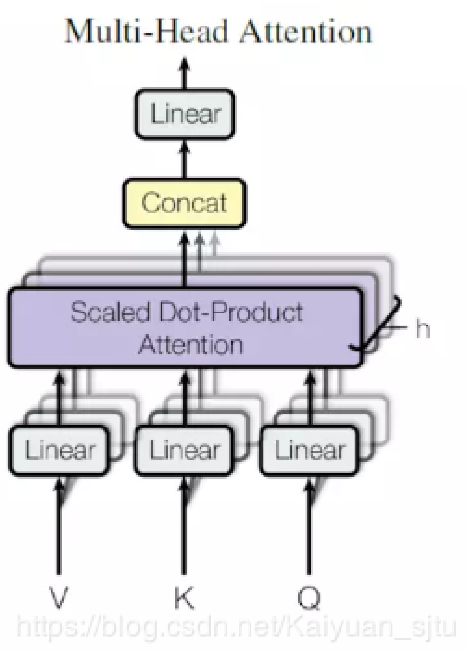
总结一下，*attention layer*的主要流程：

- 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*）：

$$(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- 将得到的*attention_probs*与*value*相乘，返回*2D*或*3D*张量




Multi-Head Attention
Linear
Concat
Scaled Dot-Product Attention
h
Linear
Linear
Linear
V
K
Q
https://blog.csdn.net/Kaiyuan_sjtu

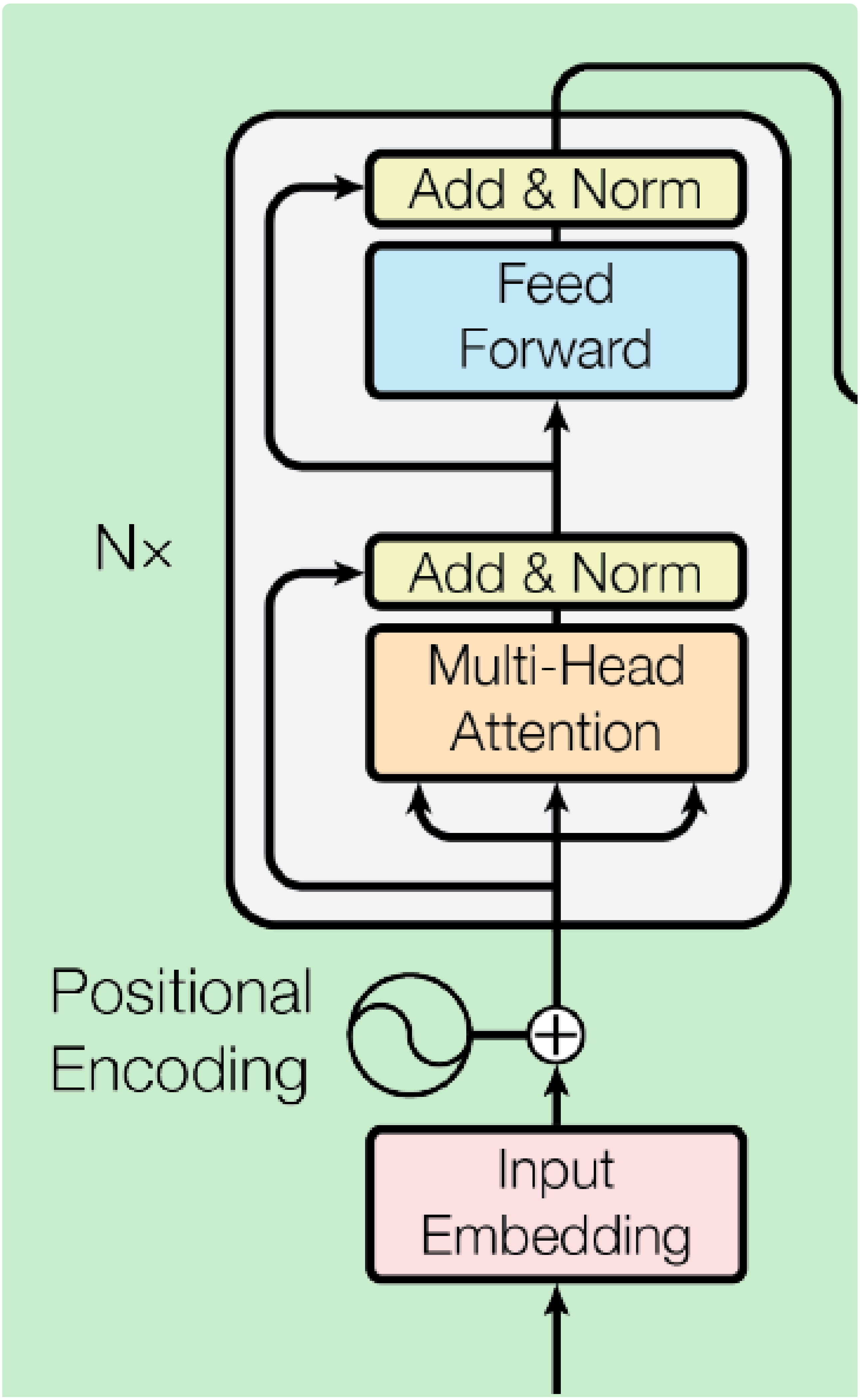
6、Transformer实现

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。可以参见原始论文[7]和原始代码[8]。

```
def transformer_model(input_tensor,                                  # 【ba
                      attention_mask=None,                      # 【batch_si
                      hidden_size=768,
                      num_hidden_layers=12,
                      num_attention_heads=12,
                      intermediate_size=3072,
                      intermediate_act_fn=gelu,                 # feed-forwa
                      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 attentio
        "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 (
                   (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_pro
        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


Add & Norm
Feed Forward
Nx
Add & Norm
Multi-Head Attention
Positional Encoding
Input Embedding

7、BERT函数入口（init）

*BertModel*类的构造函数，有了上面几节的铺垫，我们就可以来实现*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_leng
               token_type_ids=None,    # 【batch_size, seq_length】
               use_one_hot_embeddings=False,       # 是否使用one-hot；否则tf.g
               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.

    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_p
          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_le
    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_siz
        # 我们需要用squeeze把第二维去掉
      first_token_tensor = tf.squeeze(self.sequence_output[:, 0:1, :],
      # 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_rang
```

8、总结一哈

有了以上对源码的深入了解之后，我们在使用*BertModel*的时候就会更加得心应手。举个模型使用的简单栗子：

```
# 假设输入已经经过分词变成word_ids. shape=[2, 3]
input_ids = tf.constant([[31, 51, 99], [15, 5, 0]])
input_mask = tf.constant([[1, 1, 1], [1, 1, 0]])
# segment_emebdding. 表示第一个样本前两个词属于句子1，后一个词属于句子2.
# 第二个样本的第一个词属于句子1， 第二次词属于句子2，第三个元素0表示padding
# 原始代码是下面这样的，但是感觉么必要用 2，不知道是不是我哪里没理解
token_type_ids = tf.constant([[0, 0, 1], [0, 2, 0]])

# 创建BertConfig实例
config = modeling.BertConfig(vocab_size=32000, hidden_size=512,
         num_hidden_layers=8, num_attention_heads=6, intermediate_size=10

# 创建BertModel实例
model = modeling.BertModel(config=config, is_training=True,
     input_ids=input_ids, input_mask=input_mask, token_type_ids=token_typ


label_embeddings = tf.get_variable(...)
#得到最后一层的第一个Token也就是[CLS]向量表示，可以看成是一个句子的embedding
pooled_output = model.get_pooled_output()
logits = tf.matmul(pooled_output, label_embeddings)
```

在*BERT*模型构建这一块的主要流程：

- 对输入序列进行*Embedding*（三个），接下去就是*'Attention is all you need'*的内容了

- 简单一点就是将*embedding*输入*transformer*得到输出结果
- 详细一点就是*embedding --> N *【multi-head attention --> Add(Residual) &Norm--> Feed-Forward --> Add(Residual) &Norm*】
- 源码中还有一些其他的辅助函数，不是很难理解，这里就不再啰嗦。

二、数据处理

*BERT*的使用可以分为两个步骤：「**pre-training**」和「**fine-tuning**」。*pre-training*的话可以很好地适用于自己特定的任务，但是训练成本很高（*four days on 4 to 16 Cloud TPUs*），对于大对数从业者而言不太好实现从零开始*(from scratch)*。不过*Google*已经发布了各种预训练好的模型可供选择，只需要进行对特定任务的*Fine-tuning*即可。 这一部分我们就继续按照原始论文的框架，来一起读读*BERT*预训练的源码。*BERT*预训练过程分为两个具体子任务：「**Masked LM**」 和 「**Next Sentence Prediction**」

- *tokenization.py*[9]
- *create_pretraining_data.py*[10]
- *run_pretraining*[11]

2.1 分词器（tokenization.py）

*tokenization.py*是对原始文本语料的处理，分为*BasicTokenizer*和*WordpieceTokenizer*两类。

2.1.1 BasicTokenizer

根据空格，标点进行普通的分词，最后返回的是关于词的列表，对于中文而言是关于字的列表。

```
class BasicTokenizer(object):
  def __init__(self, do_lower_case=True):
    self.do_lower_case = do_lower_case

  def tokenize(self, text):
    text = convert_to_unicode(text)
    text = self._clean_text(text)
      # 增加中文支持
    text = self._tokenize_chinese_chars(text)

    orig_tokens = whitespace_tokenize(text)
    split_tokens = []
    for token in orig_tokens:
      if self.do_lower_case:
        token = token.lower()
        token = self._run_strip_accents(token)
      split_tokens.extend(self._run_split_on_punc(token))

    output_tokens = whitespace_tokenize(" ".join(split_tokens))
    return output_tokens

  def _run_strip_accents(self, text):
    # 对text进行归一化
    text = unicodedata.normalize("NFD", text)
    output = []
    for char in text:
      cat = unicodedata.category(char)
      # 把category为Mn的去掉
      # refer: https://www.fileformat.info/info/unicode/category/Mn/list.
      if cat == "Mn":
        continue
      output.append(char)
    return "".join(output)

  def _run_split_on_punc(self, text):
    # 用标点切分，返回list
    chars = list(text)
    i = 0
    start_new_word = True
    output = []
    while i < len(chars):
      char = chars[i]
      if _is_punctuation(char):
        output.append([char])
        start_new_word = True
      else:
        if start_new_word:
          output.append([])
        start_new_word = False
        output[-1].append(char)
      i += 1

    return ["".join(x) for x in output]
```

```python
  def _tokenize_chinese_chars(self, text):
    # 按字切分中文，实现就是在字两侧添加空格
    output = []
    for char in text:
      cp = ord(char)
      if self._is_chinese_char(cp):
        output.append(" ")
        output.append(char)
        output.append(" ")
      else:
        output.append(char)
    return "".join(output)

  def _is_chinese_char(self, cp):
    # 判断是否是汉字
    # refer: https://www.cnblogs.com/straybirds/p/6392306.html
    if ((cp >= 0x4E00 and cp <= 0x9FFF) or  #
        (cp >= 0x3400 and cp <= 0x4DBF) or  #
        (cp >= 0x20000 and cp <= 0x2A6DF) or  #
        (cp >= 0x2A700 and cp <= 0x2B73F) or  #
        (cp >= 0x2B740 and cp <= 0x2B81F) or  #
        (cp >= 0x2B820 and cp <= 0x2CEAF) or
        (cp >= 0xF900 and cp <= 0xFAFF) or  #
        (cp >= 0x2F800 and cp <= 0x2FA1F)):  #
      return True

    return False

  def _clean_text(self, text):
    # 去除无意义字符以及空格
    output = []
    for char in text:
      cp = ord(char)
      if cp == 0 or cp == 0xfffd or _is_control(char):
        continue
      if _is_whitespace(char):
        output.append(" ")
      else:
        output.append(char)
    return "".join(output)
```

2.1.2 WordpieceTokenizer

*WordpieceTokenizer*是将*BasicTokenizer*的结果进一步做更细粒度的切分。做这一步的目的主要是为了去除未登录词对模型效果的影响。这一过程对中文没有影响，因为在前面*BasicTokenizer*里面已经切分成以字为单位的了。

```
class WordpieceTokenizer(object):
  def __init__(self, vocab, unk_token="[UNK]", max_input_chars_per_word=2
    self.vocab = vocab
    self.unk_token = unk_token
    self.max_input_chars_per_word = max_input_chars_per_word

  def tokenize(self, text):
    """使用贪心的最大正向匹配算法
    例如:
      input = "unaffable"
      output = ["un", "##aff", "##able"]
    """
    text = convert_to_unicode(text)

    output_tokens = []
    for token in whitespace_tokenize(text):
      chars = list(token)
      if len(chars) > self.max_input_chars_per_word:
        output_tokens.append(self.unk_token)
        continue

      is_bad = False
      start = 0
      sub_tokens = []
      while start < len(chars):
        end = len(chars)
        cur_substr = None
        while start < end:
          substr = "".join(chars[start:end])
          if start > 0:
            substr = "##" + substr
          if substr in self.vocab:
            cur_substr = substr
            break
          end -= 1
        if cur_substr is None:
          is_bad = True
          break
        sub_tokens.append(cur_substr)
        start = end

      if is_bad:
        output_tokens.append(self.unk_token)
      else:
        output_tokens.extend(sub_tokens)
    return output_tokens
```

我们用一个例子来看代码的执行过程。比如假设输入是”*unaffable*”。我们跳到*while*循环部分，这是*start=0*，*end=len(chars)=9*，也就是先看看*unaffable*在不在词典里，如果在，那么直接作为一个*WordPiece*，如果不再，那么*end-=1*，也就是看*unaffabl*在不在词典里，最终发现”*un*”在词典里，把*un*加到结果里。

接着*start=2*，看*affable*在不在，不在再看*affabl*，…，最后发现 *##aff* 在词典里。注意：*##*表示这个词是接着前面的，这样使得*WordPiece*切分是可逆的——我们可以恢复出“真正”的词。

2.1.3 FullTokenizer

*BERT*分词的主要接口，包含了上述两种实现。

```
class FullTokenizer(object):
  def __init__(self, vocab_file, do_lower_case=True):
    # 加载词表文件为字典形式
    self.vocab = load_vocab(vocab_file)
    self.inv_vocab = {v: k for k, v in self.vocab.items()}
    self.basic_tokenizer = BasicTokenizer(do_lower_case=do_lower_case)
    self.wordpiece_tokenizer = WordpieceTokenizer(vocab=self.vocab)

  def tokenize(self, text):
    split_tokens = []
    # 调用BasicTokenizer粗粒度分词
    for token in self.basic_tokenizer.tokenize(text):
      # 调用WordpieceTokenizer细粒度分词
      for sub_token in self.wordpiece_tokenizer.tokenize(token):
        split_tokens.append(sub_token)

    return split_tokens
  def convert_tokens_to_ids(self, tokens):
    return convert_by_vocab(self.vocab, tokens)

  def convert_ids_to_tokens(self, ids):
    return convert_by_vocab(self.inv_vocab, ids)
```

2.2 数据整体流程

首先来看一下参数设置，

```
flags.DEFINE_string("input_file", None,
                    "Input raw text file (or comma-separated list of file

flags.DEFINE_string("output_file", None,
    "Output TF example file (or comma-separated list of files).")

flags.DEFINE_string("vocab_file", None,
                    "The vocabulary file that the BERT model was trained

flags.DEFINE_bool( "do_lower_case", True,
    "Whether to lower case the input text. Should be True for uncased "
    "models and False for cased models.")

flags.DEFINE_integer("max_seq_length", 128, "Maximum sequence length.")

flags.DEFINE_integer("max_predictions_per_seq", 20,
                     "Maximum number of masked LM predictions per sequenc

flags.DEFINE_integer("random_seed", 12345, "Random seed for data generati

flags.DEFINE_integer( "dupe_factor", 10,
    "Number of times to duplicate the input data (with different masks)."

flags.DEFINE_float("masked_lm_prob", 0.15, "Masked LM probability.")

flags.DEFINE_float("short_seq_prob", 0.1,
    "Probability of creating sequences which are shorter than the maximum
```

这里就说几个参数

- *dupe_factor*： 重复参数，即对于同一个句子，我们可以设置不同位置的【*MASK*】次数。比如对于句子 `Hello world, this is bert.`，为了充分利用数据，第一次可以*mask*成 `Hello [MASK], this is bert.`，第二次可以变成 `Hello world, this is [MASK[.`
- *max_predictions_per_seq*： 一个句子里最多有多少个*[MASK]*标记
- *masked_lm_prob*： 多少比例的*Token*被*MASK*掉
- *short_seq_prob*： 长度小于"*max_seq_length*"的样本比例。因为在*fine-tune*过程里面输入的*target_seq_length*是可变的（小于等于*max_seq_length*），那么为了防止过拟合也需要在*pre-train*的过程当中构造一些短的样本。

接着来看构造数据的整体流程，

```
def main(_):
  tf.logging.set_verbosity(tf.logging.INFO)

  tokenizer = tokenization.FullTokenizer(
      vocab_file=FLAGS.vocab_file, do_lower_case=FLAGS.do_lower_case)

  input_files = []
  for input_pattern in FLAGS.input_file.split(","):
    input_files.extend(tf.gfile.Glob(input_pattern))

  tf.logging.info("*** Reading from input files ***")
  for input_file in input_files:
    tf.logging.info("  %s", input_file)

  rng = random.Random(FLAGS.random_seed)
  instances = create_training_instances(
      input_files, tokenizer, FLAGS.max_seq_length, FLAGS.dupe_factor,
      FLAGS.short_seq_prob, FLAGS.masked_lm_prob, FLAGS.max_predictions_p
      rng)

  output_files = FLAGS.output_file.split(",")
  tf.logging.info("*** Writing to output files ***")
  for output_file in output_files:
    tf.logging.info("  %s", output_file)

  write_instance_to_example_files(instances, tokenizer, FLAGS.max_seq_len
                                  FLAGS.max_predictions_per_seq, output_f
```

- 构造*tokenizer*对输入语料进行分词处理（*Tokenizer*部分之前已经说明）
- 经过 `create_training_instances` 函数构造训练*instance*

- 调用 `write_instance_to_example_files` 函数以*TFRecord*格式保存数据

下面我们一一解析这些函数。

2.3 构造训练数据（create_pretraining_data.py）

这个文件的这作用就是将原始输入语料转换成模型预训练所需要的数据格式*TFRecoed*。

2.3.1 构造训练样本

首先定义了一个训练样本的类

```
class TrainingInstance(object):

  def __init__(self, tokens, segment_ids, masked_lm_positions, masked_lm_
               is_random_next):
    self.tokens = tokens
    self.segment_ids = segment_ids
    self.is_random_next = is_random_next
    self.masked_lm_positions = masked_lm_positions
    self.masked_lm_labels = masked_lm_labels

  def __str__(self):
    s = ""
    s += "tokens: %s\n" % (" ".join(
        [tokenization.printable_text(x) for x in self.tokens]))
    s += "segment_ids: %s\n" % (" ".join([str(x) for x in self.segment_id
    s += "is_random_next: %s\n" % self.is_random_next
    s += "masked_lm_positions: %s\n" % (" ".join(
        [str(x) for x in self.masked_lm_positions]))
    s += "masked_lm_labels: %s\n" % (" ".join(
        [tokenization.printable_text(x) for x in self.masked_lm_labels]))
    s += "\n"
    return s

  def __repr__(self):
    return self.__str__()
```

构造训练样本的代码如下。在源码包中*Google*提供了一个实例训练样本输入（「**sample_text.txt**」），输入文件格式为：

- 每行一个句子，这应该是实际的句子，不应该是整个段落或者段落的随机片段(*span*)，因为我们需要使用句子边界来做下一个句子的预测。
- 不同文档之间用一个空行分割。
- 我们认为同一文档的句子之间是有关系的，不同文档句子之间没有关系。

```
def create_training_instances(input_files, tokenizer, max_seq_length,
                              dupe_factor, short_seq_prob, masked_lm_prob
                              max_predictions_per_seq, rng):
  all_documents = [[]]
  # all_documents是list的list，第一层list表示document，
  # 第二层list表示document里的多个句子。
  for input_file in input_files:
    with tf.gfile.GFile(input_file, "r") as reader:
      while True:
        line = tokenization.convert_to_unicode(reader.readline())
        if not line:
          break
        line = line.strip()

        # 空行表示文档分割
        if not line:
          all_documents.append([])
        tokens = tokenizer.tokenize(line)
        if tokens:
          all_documents[-1].append(tokens)

  # 删除空文档
  all_documents = [x for x in all_documents if x]
  rng.shuffle(all_documents)

  vocab_words = list(tokenizer.vocab.keys())
  instances = []
  # 重复dupe_factor次
  for _ in range(dupe_factor):
    for document_index in range(len(all_documents)):
      instances.extend(
          create_instances_from_document(
              all_documents, document_index, max_seq_length, short_seq_pr
              masked_lm_prob, max_predictions_per_seq, vocab_words, rng))

  rng.shuffle(instances)
  return instances
```

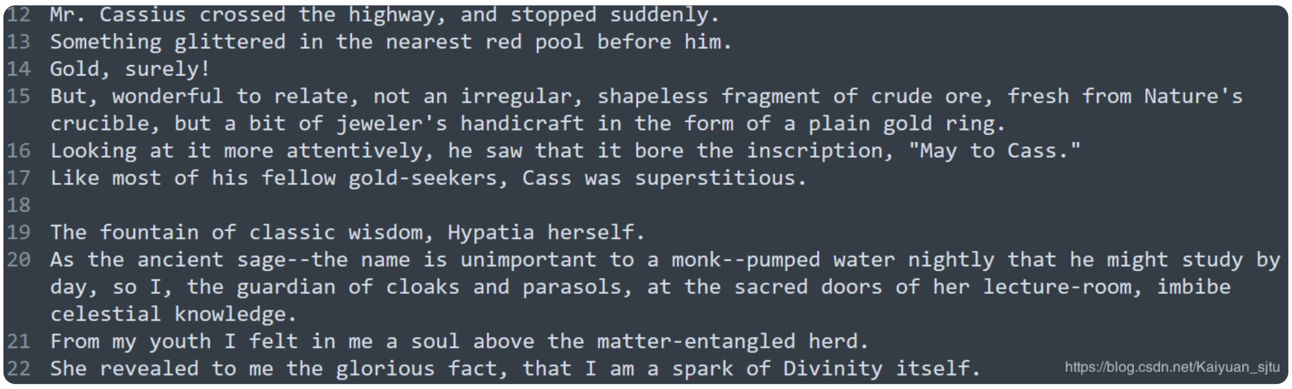
上面的函数会调用 `create_instances_from_document` 来实现从一个文档中抽取多个训练样本。

```
def create_instances_from_document(
    all_documents, document_index, max_seq_length, short_seq_prob,
    masked_lm_prob, max_predictions_per_seq, vocab_words, rng):

  document = all_documents[document_index]

  # 为[CLS], [SEP], [SEP]预留三个空位
  max_num_tokens = max_seq_length - 3

  target_seq_length = max_num_tokens
  # 以short_seq_prob的概率随机生成（2~max_num_tokens）的长度
  if rng.random() < short_seq_prob:
    target_seq_length = rng.randint(2, max_num_tokens)

  #
  instances = []
  current_chunk = []
  current_length = 0
  i = 0
  while i < len(document):
    segment = document[i]
    current_chunk.append(segment)
    current_length += len(segment)
    # 将句子依次加入current_chunk中，直到加完或者达到限制的最大长度
    if i == len(document) - 1 or current_length >= target_seq_length:
      if current_chunk:
        # `a_end`是第一个句子A结束的下标
        a_end = 1
        # 随机选取切分边界
        if len(current_chunk) >= 2:
          a_end = rng.randint(1, len(current_chunk) - 1)

        tokens_a = []
        for j in range(a_end):
          tokens_a.extend(current_chunk[j])

        tokens_b = []
        # 是否随机next
        is_random_next = False
```

```python
# 构建随机的下一句
if len(current_chunk) == 1 or rng.random() < 0.5:
  is_random_next = True
  target_b_length = target_seq_length - len(tokens_a)

  # 随机的挑选另外一篇文档的随机开始的句子
  # 但是理论上有可能随机到的文档就是当前文档，因此需要一个while循环
  # 这里只while循环10次，理论上还是有重复的可能性，但是我们忽略
  for _ in range(10):
    random_document_index = rng.randint(0, len(all_documents) - 1
    if random_document_index != document_index:
      break

  random_document = all_documents[random_document_index]
  random_start = rng.randint(0, len(random_document) - 1)
  for j in range(random_start, len(random_document)):
    tokens_b.extend(random_document[j])
    if len(tokens_b) >= target_b_length:
      break
  # 对于上述构建的随机下一句，我们并没有真正地使用它们
  # 所以为了避免数据浪费，我们将其“放回”
  num_unused_segments = len(current_chunk) - a_end
  i -= num_unused_segments
# 构建真实的下一句
else:
  is_random_next = False
  for j in range(a_end, len(current_chunk)):
    tokens_b.extend(current_chunk[j])
# 如果太多了，随机去掉一些
truncate_seq_pair(tokens_a, tokens_b, max_num_tokens, rng)

assert len(tokens_a) >= 1
assert len(tokens_b) >= 1

tokens = []
segment_ids = []
# 处理句子A
tokens.append("[CLS]")
segment_ids.append(0)
for token in tokens_a:
  tokens.append(token)
  segment_ids.append(0)
```

```
          # 句子A结束，加上【SEP】
      tokens.append("[SEP]")
      segment_ids.append(0)
          # 处理句子B
      for token in tokens_b:
        tokens.append(token)
        segment_ids.append(1)
      # 句子B结束，加上【SEP】
      tokens.append("[SEP]")
      segment_ids.append(1)

          # 调用 create_masked_lm_predictions来随机对某些Token进行mask
      (tokens, masked_lm_positions,
       masked_lm_labels) = create_masked_lm_predictions(
           tokens, masked_lm_prob, max_predictions_per_seq, vocab_words
      instance = TrainingInstance(
          tokens=tokens,
          segment_ids=segment_ids,
          is_random_next=is_random_next,
          masked_lm_positions=masked_lm_positions,
          masked_lm_labels=masked_lm_labels)
      instances.append(instance)
    current_chunk = []
    current_length = 0
  i += 1

return instances
```

上面代码有点长，在关键的地方我都注释上了。下面我们结合一个具体的例子来看代码的实现过程。以提供的「**sample_text.txt**」中语料为例，只截取了一部分，下图包含了两个文档，第一个文档中有6个句子，第二个有4个句子：



```
Mr. Cassius crossed the highway, and stopped suddenly.
Something glittered in the nearest red pool before him.
Gold, surely!
But, wonderful to relate, not an irregular, shapeless fragment of crude ore, fresh from Nature's crucible, but a bit of jeweler's handicraft in the form of a plain gold ring.
Looking at it more attentively, he saw that it bore the inscription, "May to Cass."
Like most of his fellow gold-seekers, Cass was superstitious.

The fountain of classic wisdom, Hypatia herself.
As the ancient sage--the name is unimportant to a monk--pumped water nightly that he might study by day, so I, the guardian of cloaks and parasols, at the sacred doors of her lecture-room, imbibe celestial knowledge.
From my youth I felt in me a soul above the matter-entangled herd.
She revealed to me the glorious fact, that I am a spark of Divinity itself.
```

`create_instances_from_document` 分析的是一个文档，我们就以上述第一个为例。

1. 算法首先会维护一个*chunk*，不断加入*document*中的元素，也就是句子（*segment*），直到加载完或者*chunk*中*token*数大于等于最大限制，这样做的目的是使得*padding*的尽量少，训练效率更高。
2. 现在*chunk*建立完毕之后，假设包括了前三个句子，算法会随机选择一个切分点，比如*2*。接下来构建 `predict next` 判断：*(1)* 如果是正样本，前两个句子当成是句子*A*，后一个句子当成是句子*B*；*(2)* 如果是负样本，前两个句子当成是句子*A*，无关的句子从其他文档中随机抽取
3. 得到句子*A*和句子*B*之后，对其填充*tokens*和*segment_ids*，这里会加入特殊的*[CLS]*和*[SEP]*标记
4. 对句子进行*mask*操作（下一节中描述）

2.3.2 随机MASK

对*Tokens*进行随机*mask*是*BERT*的一大创新点。使用*mask*的原因是为了防止模型在双向循环训练的过程中"预见自身"。于是，文章中选取的策略是对输入序列中*15%*的词使用*[MASK]*标记掩盖掉，然后通过上下文去预测这些被*mask*的*token*。但是为了防止模型过拟合地学习到【*MASK*】这个标记，对*15%mask*掉的词进一步优化：

- 以*80%*的概率用*[MASK]*替换：
  - *hello world， this is bert. ----> hello world, this is [MASK].*
- 以*10%*的概率随机替换：
  - *hello world， this is bert. ----> hello world, this is python.*
- 以*10%*的概率不进行替换：
  - *hello world， this is bert. ----> hello world, this is bert.*

```
def create_masked_lm_predictions(tokens, masked_lm_prob,
                                 max_predictions_per_seq, vocab_words, rn

  cand_indexes = []
  # [CLS]和[SEP]不能用于MASK
  for (i, token) in enumerate(tokens):
    if token == "[CLS]" or token == "[SEP]":
      continue
    cand_indexes.append(i)

  rng.shuffle(cand_indexes)

  output_tokens = list(tokens)

  num_to_predict = min(max_predictions_per_seq,
                       max(1, int(round(len(tokens) * masked_lm_prob))))

  masked_lms = []
  covered_indexes = set()
  for index in cand_indexes:
    if len(masked_lms) >= num_to_predict:
      break
    if index in covered_indexes:
      continue
    covered_indexes.add(index)

    masked_token = None
    # 80% of the time, replace with [MASK]
    if rng.random() < 0.8:
      masked_token = "[MASK]"
    else:
      # 10% of the time, keep original
      if rng.random() < 0.5:
        masked_token = tokens[index]
      # 10% of the time, replace with random word
      else:
        masked_token = vocab_words[rng.randint(0, len(vocab_words) - 1)]

    output_tokens[index] = masked_token

    masked_lms.append(MaskedLmInstance(index=index, label=tokens[index]))

  # 按照下标重排，保证是原来句子中出现的顺序
  masked_lms = sorted(masked_lms, key=lambda x: x.index)

  masked_lm_positions = []
  masked_lm_labels = []
  for p in masked_lms:
    masked_lm_positions.append(p.index)
    masked_lm_labels.append(p.label)

  return (output_tokens, masked_lm_positions, masked_lm_labels)
```

2.3.3 保存tfrecord数据

最后是将上述步骤处理好的数据保存为*tfrecord*文件。整体逻辑比较简单，代码如下

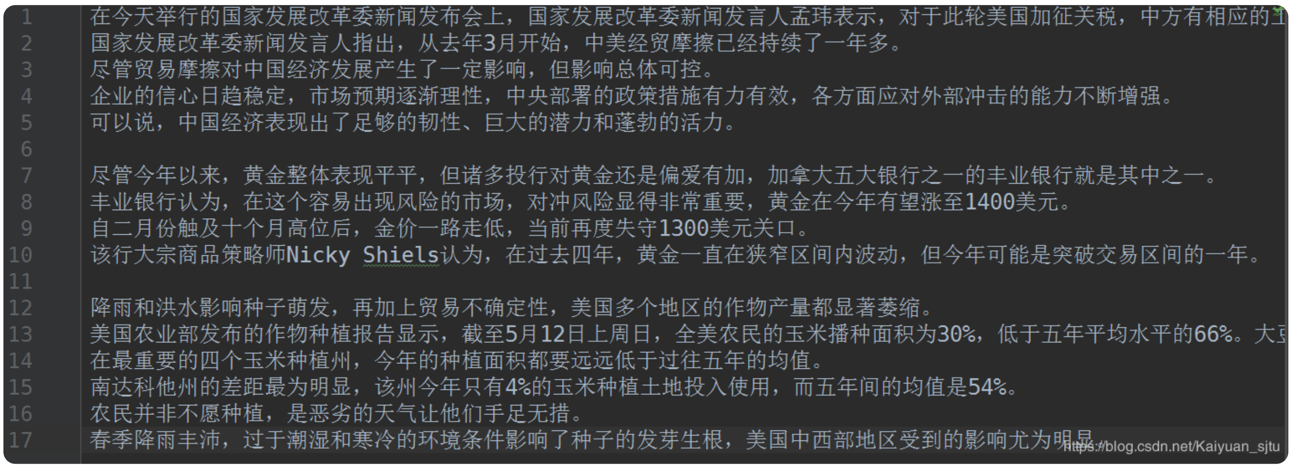
```
def write_instance_to_example_files(instances, tokenizer, max_seq_length,
                                    max_predictions_per_seq, output_files

  writers = []
  for output_file in output_files:
    writers.append(tf.python_io.TFRecordWriter(output_file))

  writer_index = 0

  total_written = 0
  for (inst_index, instance) in enumerate(instances):
      # 将输入转成word-ids
    input_ids = tokenizer.convert_tokens_to_ids(instance.tokens)
    # 记录实际句子长度
    input_mask = [1] * len(input_ids)
    segment_ids = list(instance.segment_ids)
    assert len(input_ids) <= max_seq_length

    # padding
    while len(input_ids) < max_seq_length:
      input_ids.append(0)
      input_mask.append(0)
      segment_ids.append(0)

    assert len(input_ids) == max_seq_length
    assert len(input_mask) == max_seq_length
    assert len(segment_ids) == max_seq_length

    masked_lm_positions = list(instance.masked_lm_positions)
    masked_lm_ids = tokenizer.convert_tokens_to_ids(instance.masked_lm_la
    masked_lm_weights = [1.0] * len(masked_lm_ids)

    while len(masked_lm_positions) < max_predictions_per_seq:
      masked_lm_positions.append(0)
      masked_lm_ids.append(0)
      masked_lm_weights.append(0.0)

    next_sentence_label = 1 if instance.is_random_next else 0

    features = collections.OrderedDict()
    features["input_ids"] = create_int_feature(input_ids)
    features["input_mask"] = create_int_feature(input_mask)
```

```
      features["segment_ids"] = create_int_feature(segment_ids)
      features["masked_lm_positions"] = create_int_feature(masked_lm_positi
      features["masked_lm_ids"] = create_int_feature(masked_lm_ids)
      features["masked_lm_weights"] = create_float_feature(masked_lm_weight
      features["next_sentence_labels"] = create_int_feature([next_sentence_

        # 生成训练样本
      tf_example = tf.train.Example(features=tf.train.Features(feature=feat

      # 输出到文件
      writers[writer_index].write(tf_example.SerializeToString())
      writer_index = (writer_index + 1) % len(writers)

      total_written += 1

        # 打印前20个样本
      if inst_index < 20:
        tf.logging.info("*** Example ***")
        tf.logging.info("tokens: %s" % " ".join(
            [tokenization.printable_text(x) for x in instance.tokens]))

        for feature_name in features.keys():
          feature = features[feature_name]
          values = []
          if feature.int64_list.value:
            values = feature.int64_list.value
          elif feature.float_list.value:
            values = feature.float_list.value
          tf.logging.info(
              "%s: %s" % (feature_name, " ".join([str(x) for x in values]))

    for writer in writers:
      writer.close()

    tf.logging.info("Wrote %d total instances", total_written)
```

2.4 测试代码

```
python create_pretraining_data.py \
  --input_file=./sample_text_zh.txt \
  --output_file=/tmp/tf_examples.tfrecord \
  --vocab_file=$BERT_BASE_DIR/vocab.txt \
  --do_lower_case=True \
  --max_seq_length=128 \
  --max_predictions_per_seq=20 \
  --masked_lm_prob=0.15 \
  --random_seed=12345 \
  --dupe_factor=5
```

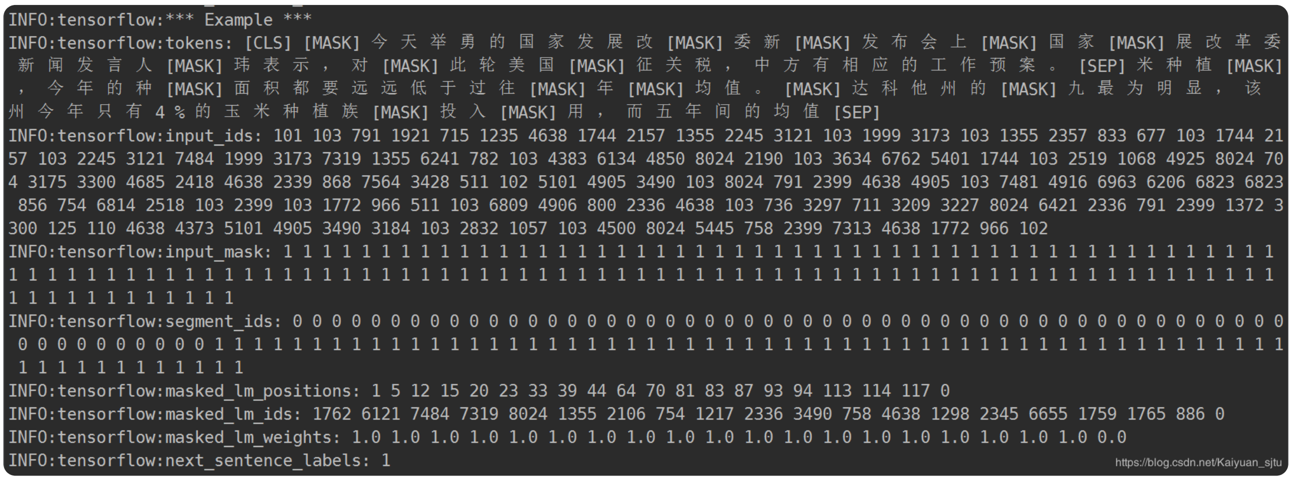
因为我之前下载的词表是中文的，所以就网上随便找了几篇新闻进行测试。结果如下



```
在今天举行的国家发展改革委新闻发布会上，国家发展改革委新闻发言人孟玮表示，对于此轮美国加征关税，中方有相应的
国家发展改革委新闻发言人指出，从去年3月开始，中美经贸摩擦已经持续了一年多。
尽管贸易摩擦对中国经济发展产生了一定影响，但影响总体可控。
企业的信心日趋稳定，市场预期逐渐理性，中央部署的政策措施有力有效，各方面应对外部冲击的能力不断增强。
可以说，中国经济表现出了足够的韧性、巨大的潜力和蓬勃的活力。

尽管今年以来，黄金整体表现平平，但诸多投行对黄金还是偏爱有加，加拿大五大银行之一的丰业银行就是其中之一。
丰业银行认为，在这个容易出现风险的市场，对冲风险显得非常重要，黄金在今年有望涨至1400美元。
自二月份触及十个月高位后，金价一路走低，当前再度失守1300美元关口。
该行大宗商品策略师Nicky Shiels认为，在过去四年，黄金一直在狭窄区间内波动，但今年可能是突破交易区间的一年。

降雨和洪水影响种子萌发，再加上贸易不确定性，美国多个地区的作物产量都显著萎缩。
美国农业部发布的作物种植报告显示，截至5月12日上周日，全美农民的玉米播种面积为30%，低于五年平均水平的66%。大
在最重要的四个玉米种植州，今年的种植面积都要远远低于过往五年的均值。
南达科他州的差距最为明显，该州今年只有4%的玉米种植土地投入使用，而五年间的均值是54%。
农民并非不愿种植，是恶劣的天气让他们手足无措。
春季降雨丰沛，过于潮湿和寒冷的环境条件影响了种子的发芽生根，美国中西部地区受到的影响尤为明显
```

https://blog.csdn.net/Kaiyuan_sjtu

这是其中的一个样例：



```
INFO:tensorflow:*** Example ***
INFO:tensorflow:tokens: [CLS] [MASK] 今 天 举 勇 的 国 家 发 展 改 [MASK] 委 新 [MASK] 发 布 会 上 [MASK] 国 家 [MASK] 展 改 革 委
 新 闻 发 言 人 [MASK] 玮 表 示 ， 对 [MASK] 此 轮 美 国 [MASK] 征 关 税 ， 中 方 有 相 应 的 工 作 预 案 。 [SEP] 米 种 植 [MASK]
 ， 今 年 的 种 [MASK] 面 积 都 要 远 远 低 于 过 往 [MASK] 年 [MASK] 均 值 。 [MASK] 达 科 他 州 的 [MASK] 九 最 为 明 显 ， 该
州 今 年 只 有 4 % 的 玉 米 种 植 族 [MASK] 投 入 [MASK] 用 ， 而 五 年 间 的 均 值 [SEP]
INFO:tensorflow:input_ids: 101 103 791 1921 715 1235 4638 1744 2157 1355 2245 3121 103 1999 3173 103 1355 2357 833 677 103 1744 21
57 103 2245 3121 7484 1999 3173 7319 1355 6241 782 103 4383 6134 4850 8024 2190 103 3634 6762 5401 1744 103 2519 1068 4925 8024 70
4 3175 3300 4685 2418 4638 2339 868 7564 3428 511 102 5101 4905 3490 103 8024 791 2399 4638 4905 103 7481 4916 6963 6206 6823 6823
 856 754 6814 2518 103 2399 103 1772 966 511 103 6809 4906 800 2336 4638 103 736 3297 711 3209 3227 8024 6421 2336 791 2399 1372 3
300 125 110 4638 4373 5101 4905 3490 3184 103 2832 1057 103 4500 8024 5445 758 2399 7313 4638 1772 966 102
INFO:tensorflow:input_mask: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
INFO:tensorflow:segment_ids: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1
INFO:tensorflow:masked_lm_positions: 1 5 12 15 20 23 33 39 44 64 70 81 83 87 93 94 113 114 117 0
INFO:tensorflow:masked_lm_ids: 1762 6121 7484 7319 8024 1355 2106 754 1217 2336 3490 758 4638 1298 2345 6655 1759 1765 886 0
INFO:tensorflow:masked_lm_weights: 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 0.0
INFO:tensorflow:next_sentence_labels: 1
```

https://blog.csdn.net/Kaiyuan_sjtu

2.5 小结一哈

主要介绍*BERT*的自带分词组件以及*pretraining*数据生成过程，属于整个项目的准备部分。没想到代码这么多，*pretraining*训练的部分就不放在这一篇里了，请见下篇~

三、BERT预训练

继续之前没有介绍完的 *Pre-training* 部分，在上一部分中我们已经完成了对输入数据的处理，接下来看看 *BERT* 是怎么完成「**Masked LM**」和「**Next Sentence Prediction**」

两个任务的训练。

- *run_pretraining*[12]

3.1 任务#1：Masked LM

`get_masked_lm_output` 函数用于计算「任务#1」的训练 *loss*。输入为 *BertModel* 的最后一层 *sequence_output* 输出（*[batch_size, seq_length, hidden_size]*），因为对一个序列的 *MASK* 标记的预测属于标注问题，需要整个 *sequence* 的输出状态。

```python
def get_masked_lm_output(bert_config, input_tensor, output_weights, posit
                         label_ids, label_weights):
  """Get loss and log probs for the masked LM."""
  # 获取mask词的encode
  input_tensor = gather_indexes(input_tensor, positions)

  with tf.variable_scope("cls/predictions"):
    # 在输出之前添加一个非线性变换，只在预训练阶段起作用
    with tf.variable_scope("transform"):
      input_tensor = tf.layers.dense(
          input_tensor,
          units=bert_config.hidden_size,
          activation=modeling.get_activation(bert_config.hidden_act),
          kernel_initializer=modeling.create_initializer(
              bert_config.initializer_range))
      input_tensor = modeling.layer_norm(input_tensor)

    # output_weights是和传入的word embedding一样的
    # 这里再添加一个bias
    output_bias = tf.get_variable(
        "output_bias",
        shape=[bert_config.vocab_size],
        initializer=tf.zeros_initializer())
    logits = tf.matmul(input_tensor, output_weights, transpose_b=True)
    logits = tf.nn.bias_add(logits, output_bias)
    log_probs = tf.nn.log_softmax(logits, axis=-1)

      # label_ids表示mask掉的Token的id
    label_ids = tf.reshape(label_ids, [-1])
    label_weights = tf.reshape(label_weights, [-1])

    one_hot_labels = tf.one_hot(
        label_ids, depth=bert_config.vocab_size, dtype=tf.float32)
    # 但是由于实际MASK的可能不到20，比如只MASK18，那么label_ids有2个0(padding)
    # 而label_weights=[1, 1, ...., 0, 0]，说明后面两个label_id是padding的，计
    per_example_loss = -tf.reduce_sum(log_probs * one_hot_labels, axis=[-
    numerator = tf.reduce_sum(label_weights * per_example_loss)
    denominator = tf.reduce_sum(label_weights) + 1e-5
    loss = numerator / denominator

  return (loss, per_example_loss, log_probs)
```

3.2 任务#2 Next Sentence Prediction

`get_next_sentence_output` 函数用于计算「任务#2」的训练 *loss*。输入为 *BertModel* 的最后一层 *pooled_output* 输出（*[batch_size, hidden_size]*），因为该任务属于二分类问题，所以只需要每个序列的第一个 *token*【*CLS*】即可。

```
def get_next_sentence_output(bert_config, input_tensor, labels):
  """Get loss and log probs for the next sentence prediction."""

 # 标签0表示 下一个句子关系成立； 标签1表示 下一个句子关系不成立。
 # 这个分类器的参数在实际Fine-tuning阶段会丢弃掉
  with tf.variable_scope("cls/seq_relationship"):
    output_weights = tf.get_variable(
        "output_weights",
        shape=[2, bert_config.hidden_size],
        initializer=modeling.create_initializer(bert_config.initializer_r
    output_bias = tf.get_variable(
        "output_bias", shape=[2], initializer=tf.zeros_initializer())

    logits = tf.matmul(input_tensor, output_weights, transpose_b=True)
    logits = tf.nn.bias_add(logits, output_bias)
    log_probs = tf.nn.log_softmax(logits, axis=-1)
    labels = tf.reshape(labels, [-1])
    one_hot_labels = tf.one_hot(labels, depth=2, dtype=tf.float32)
    per_example_loss = -tf.reduce_sum(one_hot_labels * log_probs, axis=-1
    loss = tf.reduce_mean(per_example_loss)
    return (loss, per_example_loss, log_probs)
```

3.3 预训练主代码

module_fn_builder 函数，用于构造 *Estimator* 使用的 model_fn 。定义好了上述两个训练任务，就可以写出训练过程，之后将训练集传入自动训练。

```
def model_fn_builder(bert_config, init_checkpoint, learning_rate,
                     num_train_steps, num_warmup_steps, use_tpu,
                     use_one_hot_embeddings):

  def model_fn(features, labels, mode, params):

    tf.logging.info("*** Features ***")
    for name in sorted(features.keys()):
      tf.logging.info("  name = %s, shape = %s" % (name, features[name].s

    input_ids = features["input_ids"]
    input_mask = features["input_mask"]
    segment_ids = features["segment_ids"]
    masked_lm_positions = features["masked_lm_positions"]
    masked_lm_ids = features["masked_lm_ids"]
    masked_lm_weights = features["masked_lm_weights"]
    next_sentence_labels = features["next_sentence_labels"]

    is_training = (mode == tf.estimator.ModeKeys.TRAIN)

      # 创建Transformer实例对象
    model = modeling.BertModel(
        config=bert_config,
        is_training=is_training,
        input_ids=input_ids,
        input_mask=input_mask,
        token_type_ids=segment_ids,
        use_one_hot_embeddings=use_one_hot_embeddings)

      # 获得MASK LM任务的批损失，平均损失以及预测概率矩阵
    (masked_lm_loss,
     masked_lm_example_loss, masked_lm_log_probs) = get_masked_lm_output(
         bert_config, model.get_sequence_output(), model.get_embedding_ta
         masked_lm_positions, masked_lm_ids, masked_lm_weights)

      # 获得NEXT SENTENCE PREDICTION任务的批损失，平均损失以及预测概率矩阵
    (next_sentence_loss, next_sentence_example_loss,
     next_sentence_log_probs) = get_next_sentence_output(
         bert_config, model.get_pooled_output(), next_sentence_labels)

      # 总的损失定义为两者之和
```

```
total_loss = masked_lm_loss + next_sentence_loss

  # 获取所有变量
tvars = tf.trainable_variables()

initialized_variable_names = {}
scaffold_fn = None
# 如果有之前保存的模型，则进行恢复
if init_checkpoint:
  (assignment_map, initialized_variable_names
  ) = modeling.get_assignment_map_from_checkpoint(tvars, init_checkpo
  if use_tpu:

    def tpu_scaffold():
      tf.train.init_from_checkpoint(init_checkpoint, assignment_map)
      return tf.train.Scaffold()

    scaffold_fn = tpu_scaffold
  else:
    tf.train.init_from_checkpoint(init_checkpoint, assignment_map)

tf.logging.info("**** Trainable Variables ****")
for var in tvars:
  init_string = ""
  if var.name in initialized_variable_names:
    init_string = ", *INIT_FROM_CKPT*"
  tf.logging.info("  name = %s, shape = %s%s", var.name, var.shape,
                  init_string)

output_spec = None
# 训练过程，获得spec
if mode == tf.estimator.ModeKeys.TRAIN:
  train_op = optimization.create_optimizer(
      total_loss, learning_rate, num_train_steps, num_warmup_steps, u

  output_spec = tf.contrib.tpu.TPUEstimatorSpec(
      mode=mode,
      loss=total_loss,
      train_op=train_op,
      scaffold_fn=scaffold_fn)
# 验证过程spec
elif mode == tf.estimator.ModeKeys.EVAL:

  def metric_fn(masked_lm_example_loss, masked_lm_log_probs, masked_l
                  masked_lm_weights, next_sentence_example_loss,
                  next_sentence_log_probs, next_sentence_labels):
        """计算损失和准确率"""
        masked_lm_log_probs = tf.reshape(masked_lm_log_probs,
                                         [-1, masked_lm_log_probs.shape[-
        masked_lm_predictions = tf.argmax(
            masked_lm_log_probs, axis=-1, output_type=tf.int32)
        masked_lm_example_loss = tf.reshape(masked_lm_example_loss, [-1])
        masked_lm_ids = tf.reshape(masked_lm_ids, [-1])
        masked_lm_weights = tf.reshape(masked_lm_weights, [-1])
        masked_lm_accuracy = tf.metrics.accuracy(
            labels=masked_lm_ids,
            predictions=masked_lm_predictions,
            weights=masked_lm_weights)
        masked_lm_mean_loss = tf.metrics.mean(
            values=masked_lm_example_loss, weights=masked_lm_weights)

        next_sentence_log_probs = tf.reshape(
            next_sentence_log_probs, [-1, next_sentence_log_probs.shape[-
        next_sentence_predictions = tf.argmax(
            next_sentence_log_probs, axis=-1, output_type=tf.int32)
        next_sentence_labels = tf.reshape(next_sentence_labels, [-1])
        next_sentence_accuracy = tf.metrics.accuracy(
            labels=next_sentence_labels, predictions=next_sentence_predic
        next_sentence_mean_loss = tf.metrics.mean(
            values=next_sentence_example_loss)

        return {
            "masked_lm_accuracy": masked_lm_accuracy,
            "masked_lm_loss": masked_lm_mean_loss,
            "next_sentence_accuracy": next_sentence_accuracy,
            "next_sentence_loss": next_sentence_mean_loss,
        }

      eval_metrics = (metric_fn, [
          masked_lm_example_loss, masked_lm_log_probs, masked_lm_ids,
          masked_lm_weights, next_sentence_example_loss,
          next_sentence_log_probs, next_sentence_labels
      ])
      output_spec = tf.contrib.tpu.TPUEstimatorSpec(
          mode=mode,
          loss=total_loss,
          eval_metrics=eval_metrics,
          scaffold_fn=scaffold_fn)
    else:
      raise ValueError("Only TRAIN and EVAL modes are supported: %s" % (m

    return output_spec

  return model_fn
```

「主函数」 基于上述函数实现训练过程

```
def main(_):
  tf.logging.set_verbosity(tf.logging.INFO)
  if not FLAGS.do_train and not FLAGS.do_eval:
    raise ValueError("At least one of `do_train` or `do_eval` must be Tru
  bert_config = modeling.BertConfig.from_json_file(FLAGS.bert_config_file
  tf.gfile.MakeDirs(FLAGS.output_dir)

  input_files = []
  for input_pattern in FLAGS.input_file.split(","):
    input_files.extend(tf.gfile.Glob(input_pattern))

  tf.logging.info("*** Input Files ***")
  for input_file in input_files:
    tf.logging.info("  %s" % input_file)

  tpu_cluster_resolver = None
  if FLAGS.use_tpu and FLAGS.tpu_name:
    tpu_cluster_resolver = tf.contrib.cluster_resolver.TPUClusterResolver
        FLAGS.tpu_name, zone=FLAGS.tpu_zone, project=FLAGS.gcp_project)

  is_per_host = tf.contrib.tpu.InputPipelineConfig.PER_HOST_V2
  run_config = tf.contrib.tpu.RunConfig(
      cluster=tpu_cluster_resolver,
      master=FLAGS.master,
      model_dir=FLAGS.output_dir,
      save_checkpoints_steps=FLAGS.save_checkpoints_steps,
      tpu_config=tf.contrib.tpu.TPUConfig(
          iterations_per_loop=FLAGS.iterations_per_loop,
          num_shards=FLAGS.num_tpu_cores,
          per_host_input_for_training=is_per_host))

  # 自定义模型用于estimator训练
  model_fn = model_fn_builder(
      bert_config=bert_config,
      init_checkpoint=FLAGS.init_checkpoint,
      learning_rate=FLAGS.learning_rate,
    num_train_steps=FLAGS.num_train_steps,
    num_warmup_steps=FLAGS.num_warmup_steps,
    use_tpu=FLAGS.use_tpu,
    use_one_hot_embeddings=FLAGS.use_tpu)

# 如果没有TPU，会自动转为CPU/GPU的Estimator
estimator = tf.contrib.tpu.TPUEstimator(
    use_tpu=FLAGS.use_tpu,
    model_fn=model_fn,
    config=run_config,
    train_batch_size=FLAGS.train_batch_size,
    eval_batch_size=FLAGS.eval_batch_size)

if FLAGS.do_train:
  tf.logging.info("***** Running training *****")
  tf.logging.info("  Batch size = %d", FLAGS.train_batch_size)
  train_input_fn = input_fn_builder(
      input_files=input_files,
      max_seq_length=FLAGS.max_seq_length,
      max_predictions_per_seq=FLAGS.max_predictions_per_seq,
      is_training=True)
  estimator.train(input_fn=train_input_fn, max_steps=FLAGS.num_train_st

if FLAGS.do_eval:
  tf.logging.info("***** Running evaluation *****")
  tf.logging.info("  Batch size = %d", FLAGS.eval_batch_size)

  eval_input_fn = input_fn_builder(
      input_files=input_files,
      max_seq_length=FLAGS.max_seq_length,
      max_predictions_per_seq=FLAGS.max_predictions_per_seq,
      is_training=False)

  result = estimator.evaluate(
      input_fn=eval_input_fn, steps=FLAGS.max_eval_steps)

  output_eval_file = os.path.join(FLAGS.output_dir, "eval_results.txt")
  with tf.gfile.GFile(output_eval_file, "w") as writer:
    tf.logging.info("***** Eval results *****")
    for key in sorted(result.keys()):
      tf.logging.info("  %s = %s", key, str(result[key]))
      writer.write("%s = %s\n" % (key, str(result[key])))
```

3.4 代码测试

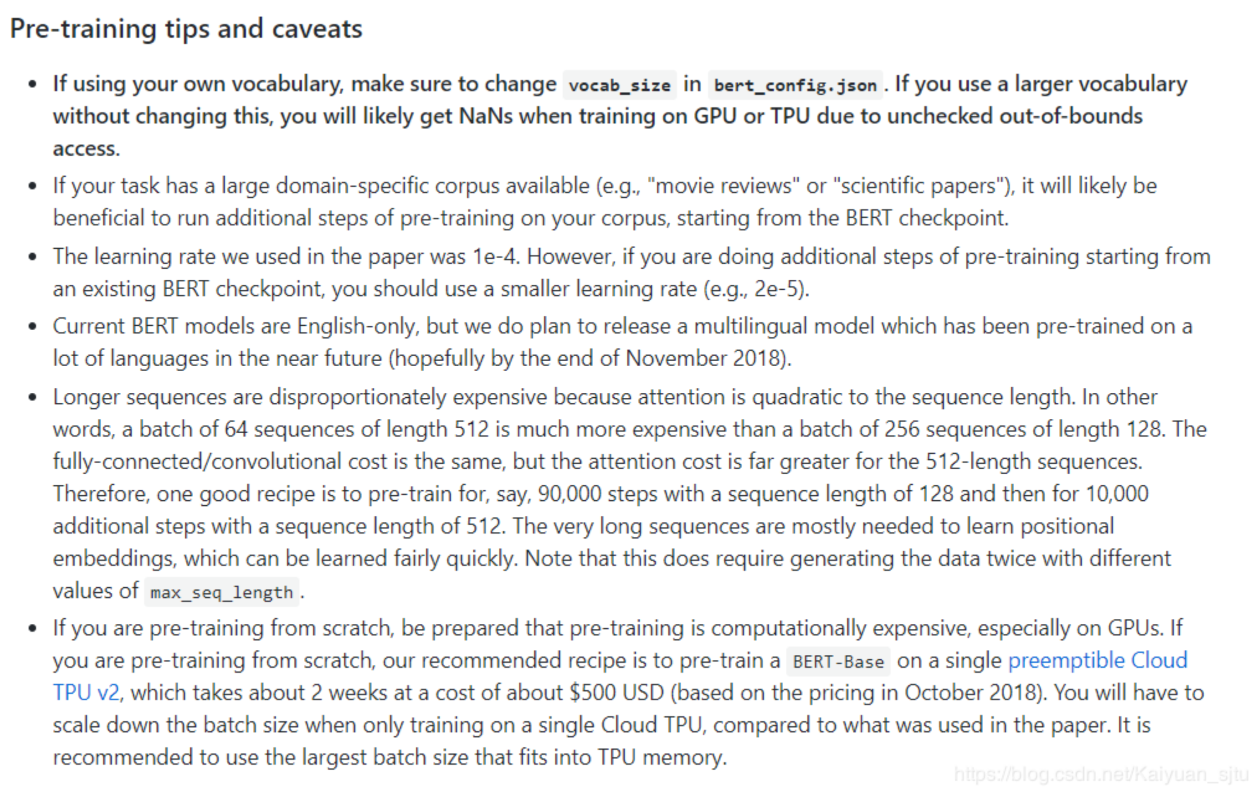
预训练运行脚本

```
python run_pretraining.py \
  --input_file=/tmp/tf_examples.tfrecord \
  --output_dir=/tmp/pretraining_output \
  --do_train=True \
  --do_eval=True \
  --bert_config_file=$BERT_BASE_DIR/bert_config.json \
  --init_checkpoint=$BERT_BASE_DIR/bert_model.ckpt \
  --train_batch_size=32 \
  --max_seq_length=128 \
  --max_predictions_per_seq=20 \
  --num_train_steps=20 \
  --num_warmup_steps=10 \
  --learning_rate=2e-5
```

之后你可以得到类似以下输出日志：

```
***** Eval results *****
  global_step = 20
  loss = 0.0979674
  masked_lm_accuracy = 0.985479
  masked_lm_loss = 0.0979328
  next_sentence_accuracy = 1.0
  next_sentence_loss = 3.45724e-05
```

最后贴一个预训练过程的 *tips*【反正我也做不了，看看就行= 。=】



Pre-training tips and caveats

- **If using your own vocabulary, make sure to change `vocab_size` in `bert_config.json`. If you use a larger vocabulary without changing this, you will likely get NaNs when training on GPU or TPU due to unchecked out-of-bounds access.**
- If your task has a large domain-specific corpus available (e.g., "movie reviews" or "scientific papers"), it will likely be beneficial to run additional steps of pre-training on your corpus, starting from the BERT checkpoint.
- The learning rate we used in the paper was 1e-4. However, if you are doing additional steps of pre-training starting from an existing BERT checkpoint, you should use a smaller learning rate (e.g., 2e-5).
- Current BERT models are English-only, but we do plan to release a multilingual model which has been pre-trained on a lot of languages in the near future (hopefully by the end of November 2018).
- Longer sequences are disproportionately expensive because attention is quadratic to the sequence length. In other words, a batch of 64 sequences of length 512 is much more expensive than a batch of 256 sequences of length 128. The fully-connected/convolutional cost is the same, but the attention cost is far greater for the 512-length sequences. Therefore, one good recipe is to pre-train for, say, 90,000 steps with a sequence length of 128 and then for 10,000 additional steps with a sequence length of 512. The very long sequences are mostly needed to learn positional embeddings, which can be learned fairly quickly. Note that this does require generating the data twice with different values of `max_seq_length`.
- If you are pre-training from scratch, be prepared that pre-training is computationally expensive, especially on GPUs. If you are pre-training from scratch, our recommended recipe is to pre-train a `BERT-Base` on a single preemptible Cloud TPU v2, which takes about 2 weeks at a cost of about $500 USD (based on the pricing in October 2018). You will have to scale down the batch size when only training on a single Cloud TPU, compared to what was used in the paper. It is recommended to use the largest batch size that fits into TPU memory.

*Over~BERT*源码系列到这里就结束啦~到目前为止，*BERT*也更新了很多比如 `Whole Word Masking` 等等，所以之前有错误的还请大家一定指出，我好及时修正~

最后，一个硬广哈哈，欢迎关注干货公众号：*NewBeeNLP*

本文参考资料

[1] **NLP大杀器BERT模型解读**：*https://blog.csdn.net/Kaiyuan_sjtu/article/details/83991186*

[2] **BERT相关论文、文章和代码资源汇总**：*http://www.52nlp.cn/bert-paper-%E8%AE%BA%E6%96%87-%E6%96%87%E7%AB%A0-%E4%BB%A3%E7%A0%81%E8%B5%84%E6%BA%90%E6%B1%87%E6%80%BB*

[3] **BERT源码地址**：*https://github.com/google-research/bert*

[4] **x] [modeling.py**模块：*https://github.com/google-research/bert/blob/master/modeling.py*

[5] 参考这个**Issue:** *https://github.com/google-research/bert/issues/16*

[6] 理解**Attention**机制原理及模型：*https://blog.csdn.net/Kaiyuan_sjtu/article/details/81806123*

[7] 原始论文：*https://arxiv.org/abs/1706.03762*

[8] 原始代码：*https://github.com/tensorflow/tensor2tensor/blob/master/tensor2tensor/models/transformer.py*

[9] **tokenization.py:** *https://github.com/google-research/bert/blob/master/tokenization.py*

[10] **create_pretraining_data.py:** *https://github.com/google-research/bert/blob/master/create_pretraining_data.py*

[11] **r un_pretraining:** *https://github.com/google-research/bert/blob/master/run_pretraining.py*

[12] **run_pretraining:** *https://github.com/google-research/bert/blob/master/run_pretraining.py*