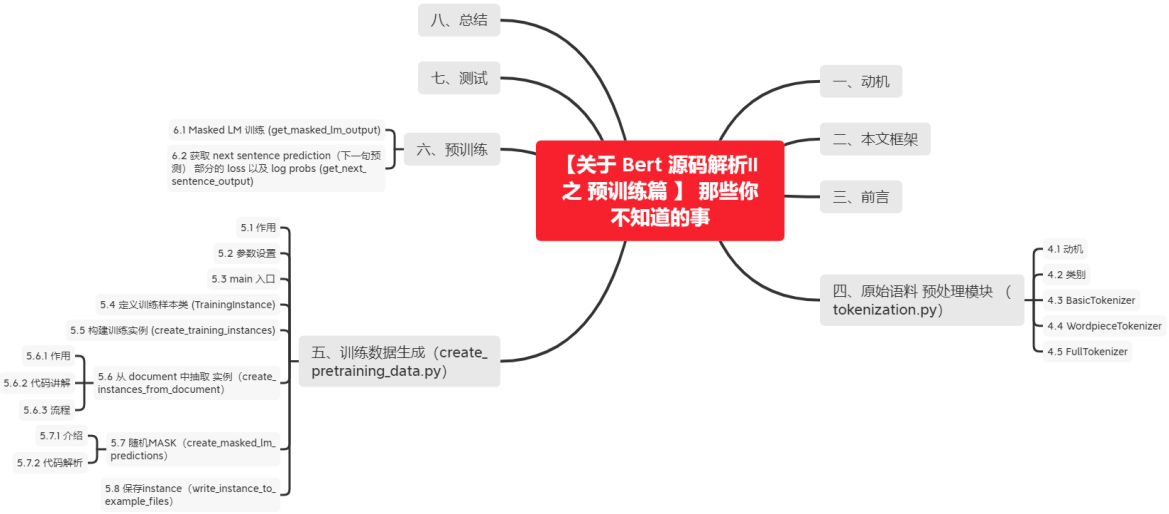
【关于 Bert 源码解析II 之 预训练篇 】 那些你不知道的事



一、前言

本文 主要 解读 Bert 模型的 预训练 模块代码：

- tokenization.py：主要用于对原始句子内容进行解析，分为 BasicTokenizer和 WordpieceTokenizer 两种；
- create_pretraining_data.py：用于将 原始语料 转化为 模型 所需要的 训练格式；
- run_pretraining.py：模型预训练；

二、原始语料 预处理模块 （tokenization.py）

2.1 动机

由于 原始语料 可能多种多样，所以需要将 原始语料 转化为 Bert 所需要的训练数据格式。

2.2 类别

预处理模块主要分为：

- BasicTokenizer
- WordpieceTokenizer

2.3 BasicTokenizer

- 动机：对原始语料进行处理
- 处理操作：unicode转换、标点符号分割、小写转换、中文字符分割、去除重音符号等操作，最后返回的是关于词的数组（中文是字的数组）；
- 代码解析

```
class BasicTokenizer(object):
  """Runs basic tokenization (punctuation splitting, lower casing, etc.)."""

  def __init__(self, do_lower_case=True):
    """Constructs a BasicTokenizer.

    Args:
      do_lower_case: 是否将 query 字母都转化为小写
    """
    self.do_lower_case = do_lower_case

  def tokenize(self, text):
    """Tokenizes a piece of text."""
    # step 1：将 text 从 Unicode 转化为 utf-8
    text = convert_to_unicode(text)
    # step 2：去除无意义字符以及空格
    text = self._clean_text(text)
    # step 3：增加中文支持
    text = self._tokenize_chinese_chars(text)
    # step 4：在一段文本上运行基本的空格清除和拆分
    orig_tokens = whitespace_tokenize(text)
    # step 5：用标点切分
    split_tokens = []
    for token in orig_tokens:
      # 是否转小写
      if self.do_lower_case:
        token = token.lower()
        # 对text进行归一化
        token = self._run_strip_accents(token)
      # 用标点切分
      split_tokens.extend(self._run_split_on_punc(token))
    # step 5：在一段文本上运行基本的空格清除和拆分
    output_tokens = whitespace_tokenize(" ".join(split_tokens))
    return output_tokens

  def _run_strip_accents(self, text):
    """这个函数去除掉text中的非间距字符"""
    # step 1：  对text进行归一化
    # 标准化对于任何需要以一致的方式处理Unicode文本的程序都是非常重要的。
    # 当处理来自用户输入的字符串而你很难去控制编码的时候尤其如此。
    # normalize() 将文本标准化,第一个参数指定字符串标准化的方式,NFD表示字符应该分解为多个组合字符表示
    text = unicodedata.normalize("NFD", text)
    output = []
    for char in text:
      # # category() 返回字符在UNICODE里分类的类型
      cat = unicodedata.category(char)
      # 判断cat 是否为 Mn，Mark, Nonspacing 指示字符是非间距字符，这指示基字符的修改。
      if cat == "Mn":
        continue
      output.append(char)
    return "".join(output)

  def _run_split_on_punc(self, text):
    """用标点对 文本 进行切分，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]

  def _tokenize_chinese_chars(self, text):
    """ 按字切分中文，实现就是在字两侧添加空格
    Adds whitespace around any CJK character. """
    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
    Checks whether CP is the codepoint of a CJK character."""
    # This defines a "chinese character" as anything in the CJK Unicode block:
    #   https://en.wikipedia.org/wiki/CJK_Unified_Ideographs_(Unicode_block)
    #
    # Note that the CJK Unicode block is NOT all Japanese and Korean characters,
    # despite its name. The modern Korean Hangul alphabet is a different block,
    # as is Japanese Hiragana and Katakana. Those alphabets are used to write
    # space-separated words, so they are not treated specially and handled
    # like the all of the other languages.
    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
      Performs invalid character removal and whitespace cleanup on text.
    """
    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)
```

2.4 WordpieceTokenizer

- 动机：对于 词中 可能是 未登录词、时态问题等；
- 操作：将BasicTokenizer的结果进一步做更细粒度的切分，将合成词分解成类似词根一样的词片。例如将"unwanted"分解成["un", "##want", "##ed"]
- 目的：去除未登录词对模型效果的影响。防止因为词的过于生僻没有被收录进词典最后只能以[UNK]代替的局面，因为英语当中这样的合成词非常多，词典不可能全部收录。
- 代码讲解

```
class WordpieceTokenizer(object):
  """Runs WordPiece tokenziation."""

  def __init__(self, vocab, unk_token="[UNK]", max_input_chars_per_word=200):
    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:
    For example:
      input = \"unaffable\"
      output = [\"un\", \"##aff\", \"##able\"]
    Args:
      text: A single token or whitespace separated tokens. This should have
        already been passed through `BasicTokenizer.

    Returns:
      A list of wordpiece tokens.
    """
    # step 1: 将 text 从 Unicode 转化为 utf-8
    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
```

> 举例说明：
>
> 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。

2.5 FullTokenizer

- 功能：对一个文本段进行以上两种解析，最后返回词（字）的数组，同时还提供token到id的索引以及id到token的索引。这里的token可以理解为文本段处理过后的最小单元。
- 代码讲解

```
class FullTokenizer(object):
  """Runs end-to-end tokenziation."""

  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)
```

三、训练数据生成（create_pretraining_data.py）

3.1 作用

将原始输入语料转换成模型预训练所需要的数据格式TFRecoed。

3.2 参数设置

```
flags.DEFINE_string("input_file", None,
                    "Input raw text file (or comma-separated list of files).")
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 on.")
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 sequence.")
flags.DEFINE_integer("random_seed", 12345, "Random seed for data generation.")
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 length.")
```

- 参数介绍
  - input_file：：代表输入的源语料文件地址
  - output_file：代表处理过的预料文件地址

- do_lower_case：是否全部转为小写字母
- vocab_file：词典文件
- max_seq_length：最大序列长度
- dupe_factor： 重复参数，默认重复10次，目的是可以生成不同情况的masks；举例：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。

3.3 main 入口

- 思路:

1. 构造tokenizer，构造tokenizer对输入语料进行分词处理；
2. 构造instances，经过create_training_instances函数构造训练instance；
3. 保存instances，调用write_instance_to_example_files函数以TFRecord格式保存数据；

- 代码讲解

```
def main(_):
  tf.logging.set_verbosity(tf.logging.INFO)
  # step 1：构造tokenizer，构造tokenizer对输入语料进行分词处理 ；
  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)

  # step 2：构造instances，经过create_training_instances函数构造训练instance；
  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_per_seq,
      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)
  # step 3：保存instances，调用write_instance_to_example_files函数以TFRecord格式保存数据
;
  write_instance_to_example_files(instances, tokenizer, FLAGS.max_seq_length,
                                  FLAGS.max_predictions_per_seq, output_files)
```

3.4 定义训练样本类 (TrainingInstance)

- 作用：构建训练样本
- 代码讲解：

```
class TrainingInstance(object):
  """A single training instance (sentence pair)."""

  def __init__(self, tokens, segment_ids, masked_lm_positions, masked_lm_labels,
               is_random_next):
    self.tokens = tokens
    self.segment_ids = segment_ids   # 指的形式为[0,0,0...1,1,111] 0的个数为i+1个，1的个数为max_seq_length - (i+1) 对应到模型输入就是token_type
    self.is_random_next = is_random_next  # 其实就是上图的Label，0.5的概率为True（和当只有一个segment的时候），如果为True则B和A不属于同一document。剩下的情况为False，则B为A同一document的后续句子。
    self.masked_lm_positions = masked_lm_positions  # 序列里被[MASK]的位置；
    self.masked_lm_labels = masked_lm_labels   # 序列里被[MASK]的token

  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_ids]))
    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__()
```

3.5 构建训练实例 (create_training_instances)

- 功能：读取数据，并构建实例
- 训练样本输入格式说明（sample_text.txt）：

```
This text is included to make sure Unicode is handled properly: 力加勝北区ᴵᴺᵀᵃছজটডণত
Text should be one-sentence-per-line, with empty lines between documents.
This sample text is public domain and was randomly selected from Project Guttenberg.

The rain had only ceased with the gray streaks of morning at Blazing Star, and the settlement awoke to a moral sense of cleanliness, and the finding of forgotten knives, tin cups, and smaller camp utensils, where the heavy showers had washed away the debris and dust heaps before the cabin doors.
Indeed, it was recorded in Blazing Star that a fortunate early riser had once picked up on the highway a solid chunk of gold quartz which the rain had freed from its incumbering soil, and washed into immediate and glittering popularity.
...
```

- 说明：
  - 不同句子 用 换行符 分割，也就是一个句子一行；
  - 不同 文档 中间 用 两个换行符 分割；
  - 同一篇文档的上下句 之间 操作关系；
- 代码讲解：

```
def create_training_instances(input_files, tokenizer, max_seq_length,
                              dupe_factor, short_seq_prob, masked_lm_prob,
                              max_predictions_per_seq, rng):
  """Create `TrainingInstance`s from raw text."""
  all_documents = [[]]
  # step 1：加载数据
  # Input file format:
  # (1) 每行一句话。理想情况下，这些应该是实际句子，而不是整个段落或文本的任意跨度。 （因为我们将句子边界用于“下一句预测”任务）。
  # (2) 文档之间的空白行。需要文档边界，以便“下一个句子预测”任务不会跨越文档之间。
  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()

        # Empty lines are used as document delimiters
        if not line:
          all_documents.append([])
        tokens = tokenizer.tokenize(line)
        if tokens:
          all_documents[-1].append(tokens)

  # step 2：清除 空文档
  all_documents = [x for x in all_documents if x]
  rng.shuffle(all_documents)

  vocab_words = list(tokenizer.vocab.keys())
  instances = []
  # step 3：重复dupe_factor次，目的是可以生成不同情况的masks
  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_prob,
              masked_lm_prob, max_predictions_per_seq, vocab_words, rng))
  # step 4：数据打乱
  rng.shuffle(instances)
  return instances
```

3.6 从 document 中抽取 实例 (create_instances_from_document)

3.6.1 作用

- 作用：实现从一个文档中抽取多个训练样本

3.6.2 代码讲解

```
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):
  """Creates `TrainingInstance`s for a single document."""
  document = all_documents[document_index]

  # step 1：为[CLS], [SEP], [SEP]预留三个空位
  max_num_tokens = max_seq_length - 3

  # step 2：以short_seq_prob的概率随机生成（2~max_num_tokens）的长度
  # 我们*通常*想要填充整个序列，因为无论如何我们都要填充到“ max_seq_length”，因此短序列通常会浪费计算量。但是，我们有时*（即short_seq_prob == 0.1 == 10%的时间）希望使用较短的序列来最大程度地减少预训练和微调之间的不匹配。但是，“ target_seq_length”只是一个粗略的目标，而“ max_seq_length”是一个硬限制。
  target_seq_length = max_num_tokens
  if rng.random() < short_seq_prob:
    target_seq_length = rng.randint(2, max_num_tokens)

  # step 3：根据用户输入提供的实际“句子”将输入分为“ A”和“ B”两段
  # 我们不只是将文档中的所有标记连接成一个较长的序列，并选择一个任意的分割点，因为这会使下一个句子的预测任务变得太容易了。相反，我们根据用户输入提供的实际“句子”将输入分为“ A”和“ B”两段。
  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])

        # step 4：构建 NSP 任务
        tokens_b = []
        # 是否随机next
        is_random_next = False
```

```
        # 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)

        # step 5：构建 MLN 任务
        # 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, rng)
        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
```

3.6.3 流程

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

- 模板

> [CLS] A [SEP] B [SEP]
>
> A = [token_0, token_1, ..., token_i]
>
> B = [token_i+1, token_i+2, ... , token_n-1]
>
> 其中:
>
> 2<= n < max_seq_length - 3 (in short_seq_prob)
>
> n=max_seq_length - 3 (in 1-short_seq_prob)

- 结果举例

> Input = [CLS] the man went to [MASK] store [SEP] he bought a gallon [MASK] milk [SEP]
>
> Label = IsNext
>
> Input = [CLS] the man [MASK] to the store [SEP] he penguin [MASK] are flight ##less birds [SEP]
>
> Label = NotNext

4. 在create_masked_lm_predictions函数里，一个序列在指定MASK数量之后，有80%被真正MASK，10%还是保留原来token，10%被随机替换成其他token。

3.7 随机MASK（create_masked_lm_predictions）

3.7.1 介绍

- 创新点：对Tokens进行随机mask
- 原因：为了防止模型在双向循环训练的过程中“预见自身”；
- 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。
- 操作

> 原始句子：the man went to a store, he bought a gallon milk.
>
> 对于 句子中 15% 的 Token，采用以下一种方式优化：
>
> 1. 以80%的概率用[MASK]替换：
>
>   the man went to a [MASK], he bought a gallon milk.
> 2. 以10%的概率随机替换：
>
>   the man went to a shop, he bought a gallon milk.
> 3. 以10%的概率保持原状：
>
>   the man went to a store, he bought a gallon milk.

3.7.2 代码解析

```
def create_masked_lm_predictions(tokens, masked_lm_prob,
                                 max_predictions_per_seq, vocab_words, rng):
  """Creates the predictions for the masked LM objective."""

  cand_indexes = []
  # step 1: [CLS]和[SEP]不能用于MASK
  for (i, token) in enumerate(tokens):
    if token == "[CLS]" or token == "[SEP]":
      continue
    # Whole Word Masking means that if we mask all of the wordpieces
    # corresponding to an original word. When a word has been split into
    # WordPieces, the first token does not have any marker and any subsequence
    # tokens are prefixed with ##. So whenever we see the ## token, we
    # append it to the previous set of word indexes.
    #
    # Note that Whole Word Masking does *not* change the training code
    # at all -- we still predict each WordPiece independently, softmaxed
    # over the entire vocabulary.
    if (FLAGS.do_whole_word_mask and len(cand_indexes) >= 1 and
        token.startswith("##")):
      cand_indexes[-1].append(i)
    else:
      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))))
  # step 2 ：  mask 操作
  masked_lms = []
  covered_indexes = set()
  for index_set in cand_indexes:
    if len(masked_lms) >= num_to_predict:
      break
    # If adding a whole-word mask would exceed the maximum number of
    # predictions, then just skip this candidate.
    if len(masked_lms) + len(index_set) > num_to_predict:
      continue
    is_any_index_covered = False
    for index in index_set:
      if index in covered_indexes:
        is_any_index_covered = True
        break
    if is_any_index_covered:
      continue
    for index in index_set:
      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]))
  assert len(masked_lms) <= num_to_predict
  # step 3: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)
```

3.8 保存instance（write_instance_to_example_files）

- 代码讲解

```
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_labels)
    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_positions)
    features["masked_lm_ids"] = create_int_feature(masked_lm_ids)
    features["masked_lm_weights"] = create_float_feature(masked_lm_weights)
    features["next_sentence_labels"] = create_int_feature([next_sentence_label])

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

```
    # 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)
```

四、预训练

4.1 Masked LM 训练 (get_masked_lm_output)

- 作用:针对的是语言模型对MASK起来的标签的预测，即上下文语境预测当前词，并计算 MLM 的 训练 loss

```
def get_masked_lm_output(
  bert_config,            # bert 配置
  input_tensor,           # BertModel的最后一层sequence_output输出（[batch_size,
seq_length, hidden_size]）
  output_weights,         # embedding_table，用来反embedding，这样就映射到token的学习
了
  positions,
  label_ids,
  label_weights):
  """ 获取 MLM 的 loss 和 log probs
  Get loss and log probs for the masked LM.
  """
  # step 1:在一个小批量的特定位置收集向量。
  input_tensor = gather_indexes(input_tensor, positions)

  with tf.variable_scope("cls/predictions"):
    # step 2: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)

    # step 3:output_weights是和传入的word embedding一样的
    # 但是在输出中有一个对应每个 token 的权重
    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)

    # step 4:label_ids表示mask掉的Token的id
    label_ids = tf.reshape(label_ids, [-1])
    label_weights = tf.reshape(label_weights, [-1])
    # step 5:关于 label 的一些格式处理，处理完之后把 label 转化成 one hot 类型的输出。
    one_hot_labels = tf.one_hot(
        label_ids, depth=bert_config.vocab_size, dtype=tf.float32)

    # step 6: `positions` tensor可以补零（如果序列太短而无法获得最大预测数）。 对于每个真实的预测，`label_weights` tensor 的值为1.0，对于填充预测，其值为0.0。
    # 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，计算loss要去掉。
    per_example_loss = -tf.reduce_sum(log_probs * one_hot_labels, axis=[-1])
    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)
```

4.2 获取 next sentence prediction（下一句预测） 部分的 loss 以及 log probs (get_next_sentence_output)

- 作用:用于计算 NSP 的训练loss

```
def get_next_sentence_output(bert_config, input_tensor, labels):
  """获取 NSP 的 loss 和 log probs"""

  # 二分类任务
  #  0 is "next sentence" and 1 is "random sentence".
  # 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_range))
    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)
```

五、测试

```
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
```

六、总结

本章 主要介绍了 利用 Bert fineturn，代码比较简单。

参考文档

1. Bert系列（三）——源码解读之Pre-train
2. BERT源码分析PART II