

# 介绍

bert模型是谷歌2018年10月底公布的，反响巨大，效果不错，在各大比赛上面出类拔萃，它的提出主要是针对word2vec等模型的不足，在之前的预训练模型（包括word2vec，ELMo等）都会生成词向量，这种类别的预训练模型属于domain transfer。而近一两年提出的ULMFiT，GPT，BERT等都属于模型迁移，说白了BERT模型是将预训练模型和下游任务模型结合在一起的，核心目的就是：是把下游具体NLP任务的活逐渐移到预训练产生词向量上。

# 资料

## 1. 官方源码

<https://github.com/google-research/bert>

## 2. Service 实现

<https://github.com/hanxiao/bert-as-service>

## 3. Paper

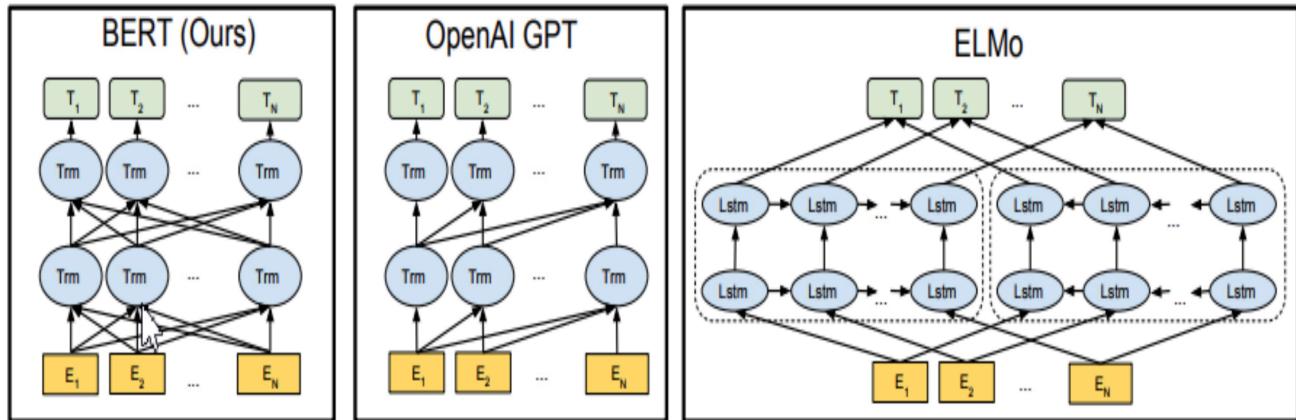
<https://arxiv.org/abs/1810.04805>

## 4. 参考博文：

<https://www.cnblogs.com/rucwxb/p/10277217.html>

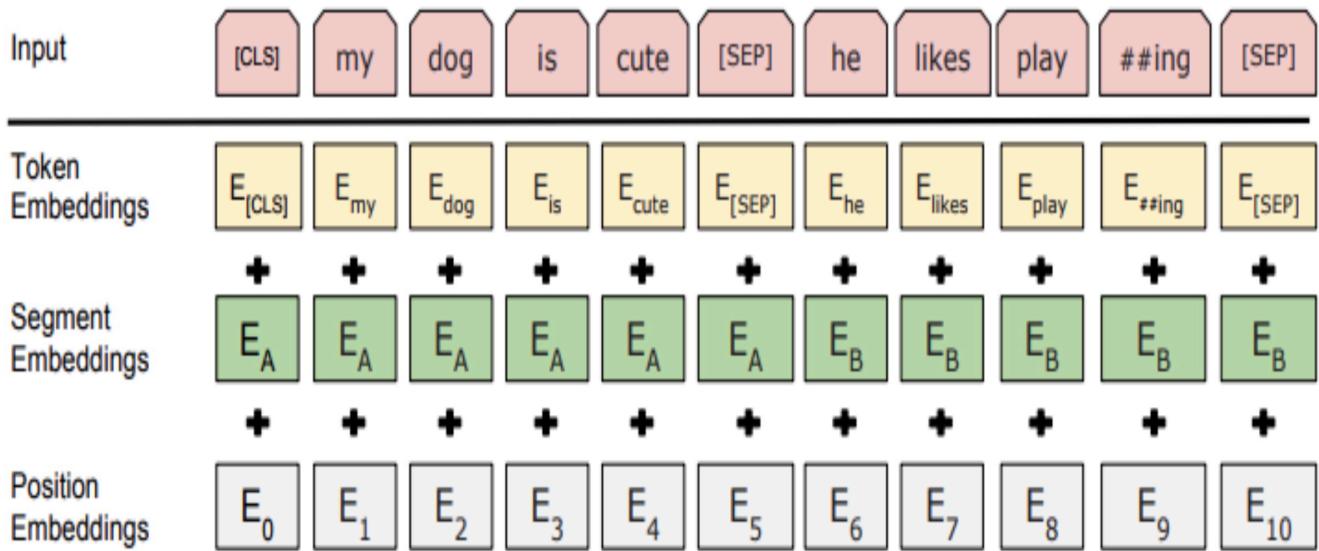
# 关键内容

## 1. 双向Transformers



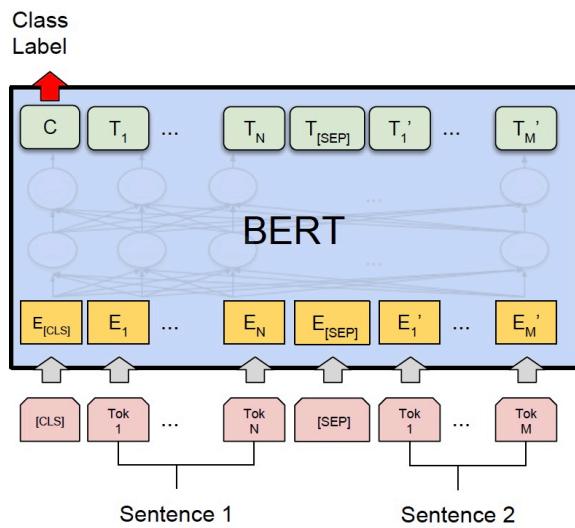
正如论文中所讲，目前的主要限制是当前模型不能同时考虑上下文，像上图的GPT只是一个从左到右，ELMo虽然有考虑从左到右和从右到左，但是是两个分开的网络，只有BERT是真正意义上的同时考虑了上下文

## 2. 句子级别的应用

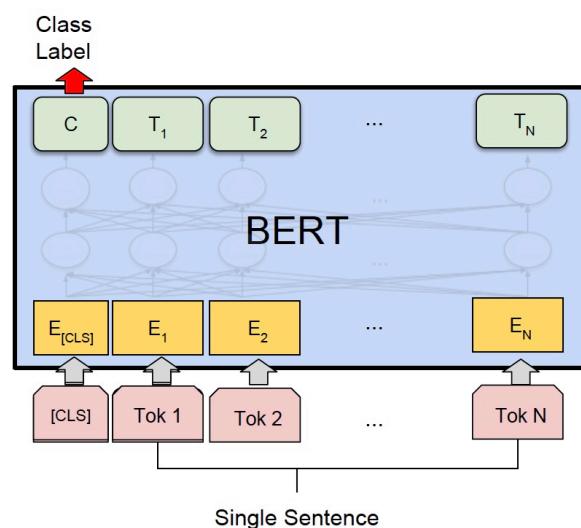


通过使用segment同时考虑了句子级别的预测，具体下面实践会看到其具体是怎么做的

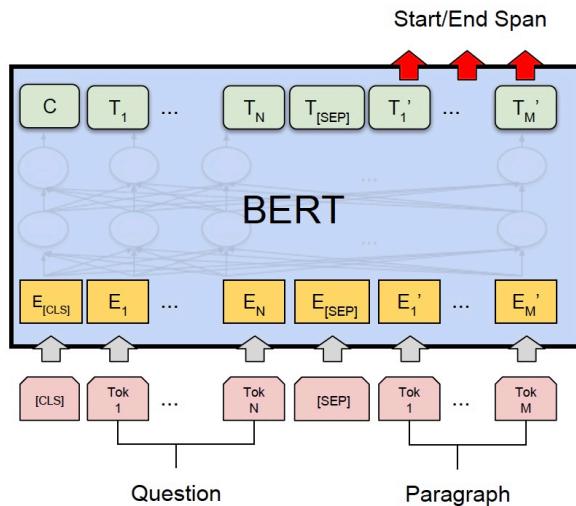
### 3. 解决多种任务



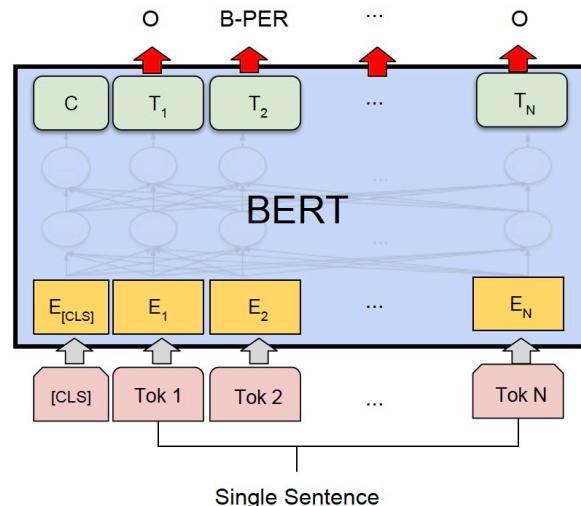
(a) Sentence Pair Classification Tasks:  
MNLI, QQP, QNLI, STS-B, MRPC,  
RTE, SWAG



(b) Single Sentence Classification Tasks:  
SST-2, CoLA



(c) Question Answering Tasks:  
SQuAD v1.1



(d) Single Sentence Tagging Tasks:  
CoNLL-2003 NER

Figure 4: Illustrations of Fine-tuning BERT on Different Tasks.

google已经预训练好了模型，我们要做的就是根据不同的任务，按照bert的输入要求（后面会看到）输入我们的数据，然后获取输出，在输出层加一层（通常情况下）全连接层就OK啦，整个训练过程就是基于预训练模型的微调，上述图片是其可以完成的几大类任务，ab都是sentence级别的（文本分类，关系抽取等等），cd是tokens级别的（如命名实体识别，知识问答）

## 实践

### 1. 使用的脚本

```
modeling.py  
optimization.py  
tokenization.py
```

`tokenization` 是对原始句子内容的解析，分为 `BasicTokenizer` 和 `WordpieceTokenizer` 两个，一般来说 `BasicTokenizer`

主要是进行 `unicode` 转换、标点符号分割、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），`WordpieceTokenizer` 的目的是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"]` 这么做的目的是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。`FullTokenizer` 的作用就很显而易见了，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。上述来源，更多该脚本的内容可以看该链接，下面主要用到 `FullTokenizer` 这个类

参考：<https://www.jianshu.com/p/22e462f01d8c>

## 2. 需要修改的脚本

```
run_classifier.py
```

分别是解决分类，读理解任务，其实套路差不多，我们具体来看一下 `run_classifier.py`

首先BERT主要分为两个部分。一个是训练语言模型 (`language model`) 的预训练 (`run_pretraining.py`) 部分。另一个是训练具体任务 (`task`) 的 `fine-tune` 部分，预训练部分巨大的运算资源，但是其已经公布了BERT的预训练模型。

The links to the models are here (right-click, 'Save link as...' on the name):

- **BERT-Large, Uncased (Whole Word Masking)** : 24-layer, 1024-hidden, 16-heads, 340M parameters
- **BERT-Large, Cased (Whole Word Masking)** : 24-layer, 1024-hidden, 16-heads, 340M parameters
- **BERT-Base, Uncased** : 12-layer, 768-hidden, 12-heads, 110M parameters
- **BERT-Large, Uncased** : 24-layer, 1024-hidden, 16-heads, 340M parameters
- **BERT-Base, Cased** : 12-layer, 768-hidden, 12-heads, 110M parameters
- **BERT-Large, Cased** : 24-layer, 1024-hidden, 16-heads, 340M parameters
- **BERT-Base, Multilingual Cased (New, recommended)** : 104 languages, 12-layer, 768-hidden, 12-heads, 110M parameters
- **BERT-Base, Multilingual Uncased (Orig, not recommended)** (Not recommended, use `Multilingual Cased` instead): 102 languages, 12-layer, 768-hidden, 12-heads, 110M parameters
- **BERT-Base, Chinese** : Chinese Simplified and Traditional, 12-layer, 768-hidden, 12-heads, 110M parameters

这里需要中文 [`BERT-Base, Chinese`]，直接下载就行，总得来说，我们要做的就是自己的数据集上进行 `fine-tune`

## 修改代码

run\_classifier.py 源码

```
Structure run_classifier.py
175
176
177 class DataProcessor(object):
178     """Base class for data converters for sequence classification data sets."""
179
180     def get_train_examples(self, data_dir):
181         """Gets a collection of `InputExample`s for the train set."""
182         raise NotImplementedError()
183
184     def get_dev_examples(self, data_dir):
185         """Gets a collection of `InputExample`s for the dev set."""
186         raise NotImplementedError()
187
188     def get_test_examples(self, data_dir):
189         """Gets a collection of `InputExample`s for prediction."""
190         raise NotImplementedError()
191
192     def get_labels(self):
193         """Gets the list of labels for this data set."""
194         raise NotImplementedError()
195
196     @classmethod
197     def _read_tsv(cls, input_file, quotechar=None):
198         """Reads a tab separated value file."""
199         with tf.gfile.Open(input_file, "r") as f:
200             reader = csv.reader(f, delimiter="\t", quotechar=quotechar)
201             lines = []
202             for line in reader:
203                 lines.append(line)
204
205         return lines
```

run\_classifier.py 修改后

```

Structure
model/.../run_classifier.py × modeling.py × bert/.../run_classifier.py × run.sh
129
130 class InputExample(object):...
149
150
151 class PaddingInputExample(object):...
162
163
164 class InputFeatures(object):...
178
179
180 class DataProcessor(object):...
208
209
210 class Baidu_95_Multi_Label_Classification_Processor(DataProcessor):...
272
273
274 def predicate_label_to_id(predicate_label, predicate_label_map):...
280
281
282 def convert_single_example(ex_index, example, label_list, max_seq_length,
283                         tokenizer):...

```

## 1. InputExample

```

127 class InputExample(object):
128     """A single training/test example for simple sequence classification."""
129
130     def __init__(self, guid, text_a, text_b=None, label=None):
131         """Constructs a InputExample.
132
133         Args:
134             guid: Unique id for the example.
135             text_a: string. The untokenized text of the first sequence. For single
136                 sequence tasks, only this sequence must be specified.
137             text_b: (Optional) string. The untokenized text of the second sequence.
138                 Only must be specified for sequence pair tasks.
139             label: (Optional) string. The label of the example. This should be
140                 specified for train and dev examples, but not for test examples.
141
142         """
143         self.guid = guid
144         self.text_a = text_a
145         self.text_b = text_b
146         self.label = label

```

`guid` 就是一个 `id` 号，一般将数据处理成 `train`、`dev`、`test` 数据集，那么这里定义方式就可以是相应数据集+行号（句子）

`text_a` 就是当前的句子，`text_b` 是另一个句子，因为有的任务需要两个句子，如果任务中没有的话，可以将 `text_b` 设为 `None`

`guid` 是该样本的唯一ID，`text_a` 和 `text_b` 表示句子对，`label` 表示句子对关系，如果是 `test` 数据集则 `label` 统一为0。

`label` 就是标签

## 2. InputFeatures

```
164     class InputFeatures(object):
165         """A single set of features of data."""
166
167         def __init__(self,
168             input_ids,
169             input_mask,
170             segment_ids,
171             label_ids,
172             is_real_example=True):
173             self.input_ids = input_ids
174             self.input_mask = input_mask
175             self.segment_ids = segment_ids
176             self.label_ids = label_ids
177             self.is_real_example = is_real_example
```

`InputFeatures` 类主要是定义了bert的输入格式，形象化点就是特征,即上面的格式使我们需要将原始数据处理成的格式，但并不是bert使用的最终格式，且还会通过一些代码将`InputExample`转化为`InputFeatures`，这才是bert最终使用的数据格式，当然啦这里根据自己的需要还可以自定义一些字段作为中间辅助字段，但bert最基本的输入字段就需要 `input_ids`，`input_mask` 和 `segment_ids` 这三个字段，`label_id` 是计算 `loss` 时候用到的，`input_ids`，`segment_ids` 分别对应单词id和句子(上下句标示)，`Input_mask` 就是记录的是填充信息，具体看下面

`tokenization` 过后的样本数据结构，`input_ids` 其实就是tokens的索引，`input_mask` 不用解释，`segment_ids` 对应模型的 `token_type_ids` 以上三者构成模型输入的 `X`，`label_id` 是标签，对应 `Y`

## 3. DataProcessor

`DataProcessor`，这是一个数据预处理的基类，里面定义了一些基本方法

```

180 ① class DataProcessor(object):
181     """Base class for data converters for sequence classification data sets."""
182
183 ②     def get_train_examples(self, data_dir):
184         """Gets a collection of `InputExample`s for the train set."""
185         raise NotImplementedError()
186
187 ②     def get_dev_examples(self, data_dir):
188         """Gets a collection of `InputExample`s for the dev set."""
189         raise NotImplementedError()
190
191 ②     def get_test_examples(self, data_dir):
192         """Gets a collection of `InputExample`s for prediction."""
193         raise NotImplementedError()
194
195 ②     def get_labels(self):
196         """Gets the list of labels for this data set."""
197         raise NotImplementedError()
198
199     @classmethod
200     def _read_tsv(cls, input_file, quotechar=None):
201         """Reads a tab separated value file."""
202         with tf.gfile.Open(input_file, "r") as f:
203             reader = csv.reader(f, delimiter="\t", quotechar=quotechar)
204             lines = []
205             for line in reader:
206                 lines.append(line)
207             return lines

```

`XnliProcessor`、`MnliProcessor`、`MrpcProcessor`、`ColaProcessor` 四个类是对 `DataProcessor` 的具体实现，这里之所以列举了四个是尽可能多的给用户呈现出各种demo，具体到实际使用的时候我们只需要参考其写法，定义一个自己的数据预处理类即可，其中一般包括以下几个方法：`get_train_examples`，`get_dev_examples`，`get_test_examples`，`get_labels`，`_create_examples`

```

processors = {
    "cola": ColaProcessor,
    "mnli": MnliProcessor,
    "mrpc": MrpcProcessor,
    "xnli": XnliProcessor,
    "selfsim": SelfProcessor #添加自己的processor
}

```

其中前三个都通过调用 `_create_examples` 返回一个 `InputExample` 类数据结构，`get_labels` 就是返回类别，所以重点就是以下两个函数：也很简单

```
274  def get_labels(self):
275      """See base class."""
276      return ["contradiction", "entailment", "neutral"]
277
278  def _create_examples(self, lines, set_type):
279      """Creates examples for the training and dev sets."""
280      examples = []
281      for (i, line) in enumerate(lines):
282          if i == 0:
283              continue
284          guid = "%s-%s" % (set_type, tokenization.convert_to_unicode(line[0]))
285          text_a = tokenization.convert_to_unicode(line[8])
286          text_b = tokenization.convert_to_unicode(line[9])
287          if set_type == "test":
288              label = "contradiction"
289          else:
290              label = tokenization.convert_to_unicode(line[-1])
291          examples.append(
292              InputExample(guid=guid, text_a=text_a, text_b=text_b, label=label))
293
294  return examples
```

---

多标签百度试题分类【实战】：

```

210     class Baidu_95_Multi_Label_Classification_Processor(DataProcessor):
211         """Processor for the Baidu_95 data set"""
212
213         def __init__(self):
214             self.language = "zh"
215
216         @staticmethod
217         def load_examples(data_dir):...
218
219         def get_train_examples(self, data_dir):
220             return self.create_example(self.load_examples(os.path.join(data_dir, "train")), "train")
221
222         def get_dev_examples(self, data_dir):
223             return self.create_example(self.load_examples(os.path.join(data_dir, "valid")), "valid")
224
225         def get_test_examples(self, data_dir):
226             return self.create_example(self.load_examples(os.path.join(data_dir, "test")), "test")
227
228         def get_labels(self):
229             """
230                 21 labels and 95 labels
231             :return:
232             """
233
234             return ['生物性污染', '细胞有丝分裂不同时期的特点', '液泡的结构和功能', '组成细胞的化学元素', '兴奋在神经纤维上的传导',
235                 '不完全显性', '免疫系统的组成', '生物技术在其他方面的应用', '群落的结构', '中央官制—三公九卿制', '核糖体的结构和功能',
236                 '人体免疫系统在维持稳态中的作用', '皇帝制度', '激素调节', '伴性遗传', '地球运动的地理意义', '宇宙中的地球', '地球运动的基本形式',
237                 '基因工程的原理及技术', '体液免疫的概念和过程', '基因的分离规律的实质及应用', '蛋白质的合成', '地球的内部圈层结构及特点',
238                 '人口增长与人口问题', '经济学常识', '劳动就业与守法经营', '器官移植', '生物技术实践', '垄断组织的出现', '基因工程的概念',
239                 '神经调节和体液调节的比较', '人口与城市', '组成细胞的化合物', '地理', '文艺的春天', '生物工程技术',
240                 '基因的自由组合规律的实质及应用', '郡县制', '人体水盐平衡调节', '内质网的结构和功能', '人体的体温调节',
241                 '免疫系统的功能', '科学社会主义常识', '与细胞分裂有关的细胞器', '太阳对地球的影响', '古代史', '清末民主革命风潮',
242                 '复等位基因', '人工授精、试管婴儿等生殖技术', '重农抑商政策', '生态系统的营养结构', '减数分裂的概念',
243                 '地球的外部圈层结构及特点', '细胞的多样性和统一性', '政治', '工业区位因素', '细胞大小与物质运输的关系',
244                 '夏商两代的政治制度', '农业区位因素', '溶酶体的结构和功能', '生产活动与地域联系', '内环境的稳态', '遗传与进化',
245                 '胚胎移植', '生物科学与社会', '近代史', '第三产业的兴起和“新经济”的出现', '公民道德与伦理常识', '中心体的结构和功能',
246                 '社会主义市场经济的伦理要求', '高中', '选官、用官制度的变化', '减数分裂与有丝分裂的比较', '遗传的细胞基础',
247                 '地球所处的宇宙环境', '培养基与无菌技术', '生活中的法律常识', '高尔基体的结构和功能', '社会主义是中国人民的历史性选择',
248                 '人口迁移与人口流动', '现代史', '地球与地图', '走进细胞', '生物', '避孕的原理和方法', '血糖平衡的调节',
249                 '现代生物技术专题', '海峡两岸关系的发展', '生命活动离不开细胞', '兴奋在神经元之间的传递', '历史', '分子与细胞',
250                 '拉马克的进化学说', '遗传的分子基础', '稳态与环境']
251
252
253         @staticmethod
254         def create_example(lines, set_type):
255             """
256                 Creates examples for the training and dev sets.
257             """
258             examples = []
259             for i, line in enumerate(lines):
260                 guid = "%s-%s" % (set_type, i)
261                 text_str = line[0]
262                 predicate_label_str = line[1]
263                 examples.append(
264                     InputExample(guid=guid, text_a=text_str, text_b=None, label=predicate_label_str))
265
266             return examples
267
268

```

上述就是数据预处理过程，也是需要我们自己根据自己的数据定义的，其实呢，这并不是Bert使用的最终样子，其还得经过一系列过程才能变成其能处理的数据格式，该过程是通过接下来的四个方法完成的：

```

convert_single_example
file_based_convert_examples_to_features
file_based_input_fn_builder
truncate_seq_pair

```

只不过一般情况下我们不需要修改，它都是一个固定的流程

## BERT 输入

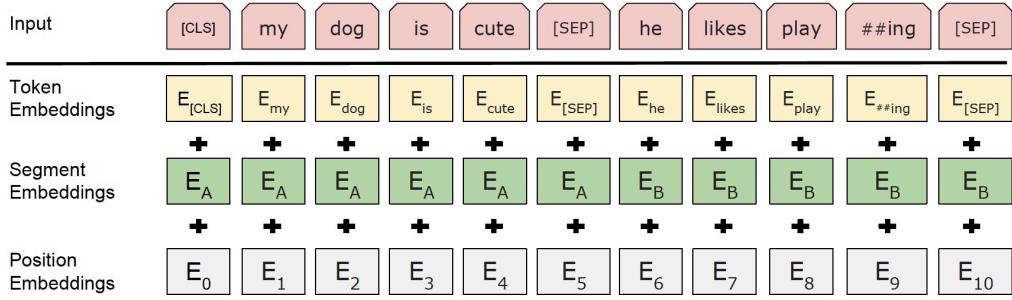


Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

代码中的 `input_ids` , `segment_ids` 分别代表 `token` , `segment` ,同时其还在句子的开头结尾加上了 `[CLS]` 和 `[SEP]` 标示

```

426     tokens = []
427     segment_ids = []
428     tokens.append("[CLS]")
429     segment_ids.append(0)
430     for token in tokens_a:
431         tokens.append(token)
432         segment_ids.append(0)
433     tokens.append("[SEP]")
434     segment_ids.append(0)
435
436     if tokens_b:
437         for token in tokens_b:
438             tokens.append(token)
439             segment_ids.append(1)
440     tokens.append("[SEP]")
441     segment_ids.append(1)

```

`input_ids` 中就是记录的是使用 `FullTokenizer` 类 `convert_tokens_to_ids` 方法将 `tokens` 转化成单个字的 `id`

`segment_ids` 就是句子级别（上下句）的标签，大概形式：

```

# The convention in BERT is:
# (a) For sequence pairs:
#   tokens: [CLS] is this jack ##son ##ville ? [SEP] no it is not . [SEP]
#   type_ids: 0 0 0 0 0 0 0 0 1 1 1 1 1 1
# (b) For single sequences:
#   tokens: [CLS] the dog is hairy . [SEP]
#   type_ids: 0 0 0 0 0 0

```

当没有`text_b`的时候，就都是0啦

还有一个 `input_mask` ，其就是和最大长度有关，假设我们定义句子的最大长度是 `120` ，当前句子长度是

100，那么 input\_mask 前 100 个元素都是 1，其余 20 个就是 0

百度试题多标签分类【实战】

## single sequences

## 4. file based convert examples to features

很简单啦，因为在训练的时候为了读写快速方便便将数据制作成TFrecords 数据格式，该函数主要就是将上述返回的InputFeatures类数据，保存成一个TFrecords数据格式，关于TFrecords数据格式的制作可以参考另一篇博客。

参考:[https://blog.csdn.net/weixin\\_42001089/article/details/90236241](https://blog.csdn.net/weixin_42001089/article/details/90236241)

- `file_based_input_fn_builder` 对应的就是从 `TFrecords` 解析读取数据
  - `truncate_seq_pair` 就是来限制 `text_a` 和 `text_b` 总长度的，当超过的话，会轮番 `pop` 掉 `tokens`

至此整个数据的预处理才算处理好，其实最后最关键的就是得到了那个TFrecords文件

## 模型

下面看模型部分

- `create_model`
  - `model_fn_builder`

整个模型过程采用了 `tf.contrib.tpu.TPUEstimator` 这一高级封装的 API

`model_fn_builder` 是壳，`create_model` 是核心，其内部定义了 `loss`、预测概率以及预测结果等等。

- `model_fn_builder` 其首先调用 `create_model` 得到 `total_loss`, `per_example_loss`, `logits`, `probabilities` 等等, 然后针对不同的状态返回不同的结果 (`output_spec`), 如果是 `train` 则返回 `loss`, `train_op` 等, 如果是 `dev` 则返回一些评价指标如 `accuracy`, 如果是 `test` 则返回预测结果

所以我们如果想看一下别的指标什么的，可以在这里改，需要注意的是指标的定义这里因为使用了 estimator API 使得其必须返回一个 operation，至于怎么定义 f1 什么的可以看下：

参考：<https://www.cnblogs.com/jiangxinyang/p/10341392.html>

- create\_model

这里可以说整个 Bert 使用的最关键的地方，我们使用 Bert 大多数情况无非进行在定义自己的下游工作进行 fine-tune，就是在这里定义的

把这段代码贴出来吧

```
def create_model(bert_config, is_training, input_ids, input_mask, segment_ids,
                 labels, num_labels, use_one_hot_embeddings):
    """Creates a classification model."""
    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)

    # In the demo, we are doing a simple classification task on the entire
    # segment.
    #
    # If you want to use the token-level output, use model.get_sequence_output()
    # instead.
    output_layer = model.get_pooled_output()

    hidden_size = output_layer.shape[-1].value

    output_weights = tf.get_variable(
        "output_weights", [num_labels, hidden_size],
        initializer=tf.truncated_normal_initializer(stddev=0.02))

    output_bias = tf.get_variable(
        "output_bias", [num_labels], initializer=tf.zeros_initializer())

    with tf.variable_scope("loss"):
        if is_training:
            # I.e., 0.1 dropout
            output_layer = tf.nn.dropout(output_layer, keep_prob=0.9)

        logits_wx = tf.matmul(output_layer, output_weights, transpose_b=True)
        logits = tf.nn.bias_add(logits_wx, output_bias)

        probabilities = tf.sigmoid(logits)
        label_ids = tf.cast(labels, tf.float32)
        per_example_loss = tf.reduce_sum(
```

```
    tf.nn.sigmoid_cross_entropy_with_logits(logits=logits, labels=label_ids), axis=-1)
loss = tf.reduce_mean(per_example_loss)

return loss, per_example_loss, logits, probabilities
```

### 多标签分类 `probabilities = tf.sigmoid(logits)`

首先调用 `modeling.BertModel` 得到 `bert` 模型

`bert`模型的输入: `input_ids` , `input_mask` , `segment_ids`

```
def create_model(bert_config, is_training, input_ids, input_mask, segment_ids,
                 labels, num_labels, use_one_hot_embeddings):
    """Creates a classification model."""
    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)
```

`config` 是 `bert` 的配置文件，在开头下载的中文模型中里面有，直接加载即可

`use_one_hot_embeddings` 是根据是不是用GPU而定的

`bert`模型的输出：

其有两种情况

```
model.get_sequence_output()
model.get_pooled_output()
```

第一种输出结果是 `[batch_size, seq_length, embedding_size]`

第二种输出结果是 `[batch_size, embedding_size]`

第二种结果是第一种结果在第二个维度上面进行了池化，要是形象点比喻的话，第一种结果得到是 `tokens` 级别的结果，第二种是句子级别的，其实就是一个池化

我们定义部分

这部分就是需要我们根据自己的任务自己具体定义啦，假设是一个简单的分类，那么就是定义一个全连接层将其转化为 `[batch_size, num_classes]` 对吧

`output_weights` 和 `output_bias` 就是对应全连接成的权值，后面就是 `loss`，使用了 `tf.nn.log_softmax` 应该是一个多分类，多标签的话可以使用 `tf.nn.sigmoid`，比较简单就不再说啦

总的来说，使用 `bert` 进行自己任务的时候，可以千变万化，变的就是这里这个下游

- `main`

最后就是主函数，主要就是通过人为定义的一些配置值是通过人为定义FLAGE将上面的流程整个组合起来

这里大体说一下流程：

```
processors = {
    "cola": ColaProcessor,
    "mnli": MnliProcessor,
    "mrpc": MrpcProcessor,
    "xnli": XnliProcessor,
}
```

```
622     processors = {
623         "baidu_95": Baidu_95_Multi_Label_Classification_Processor,
624     }
```

数据预处理完了，就使用 `tf.contrib.tpu.TPUEstimator` 定义模型

```
857     # If TPU is not available, this will fall back to normal Estimator on CPU
858     # or GPU.
859     estimator = tf.contrib.tpu.TPUEstimator(
860         use_tpu=FLAGS.use_tpu,
861         model_fn=model_fn,
862         config=run_config,
863         train_batch_size=FLAGS.train_batch_size,
864         eval_batch_size=FLAGS.eval_batch_size,
865         predict_batch_size=FLAGS.predict_batch_size)
```

最后就是根据不同模式（`train/dev/test`，这也是运行时可以指定的）运行 `estimator.train`，  
`estimator.evaluate`，`estimator.predict`

## 总结

1. 总体来说，在进行具体

- `DataProcessor` 定义一个自己的数据预处理类
- 在 `create_model` 中定义自己的具体下游工作

2. 把握住 `bert` 的输入：`input_ids`，`input_mask`，`segment_ids`（主要在 `convert_single_example` 生成），剩下的就是一些零零碎碎的小地方啦，也很简单

3. BERT的输出：`model.get_sequence_output()`，`model.get_pooled_output()`

4. 多标签 `sigmoid`，多分类 `softmax`