基于Bert的文本分类

使用pytorch框架

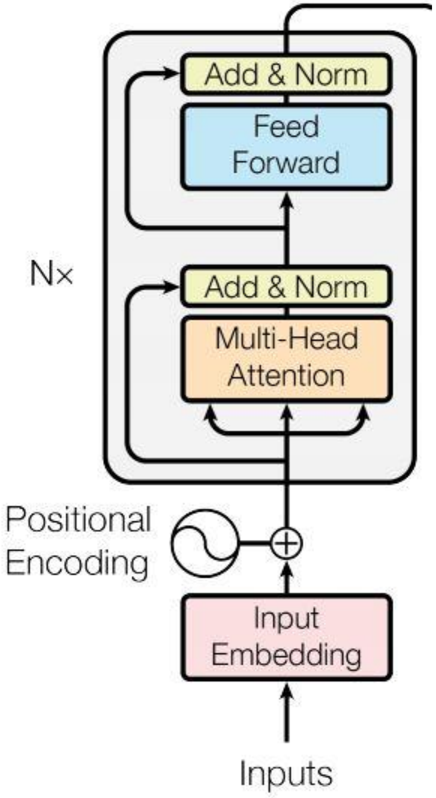
Bert模型实现

文本分类模型实现

实验以及不同数据集的预处理

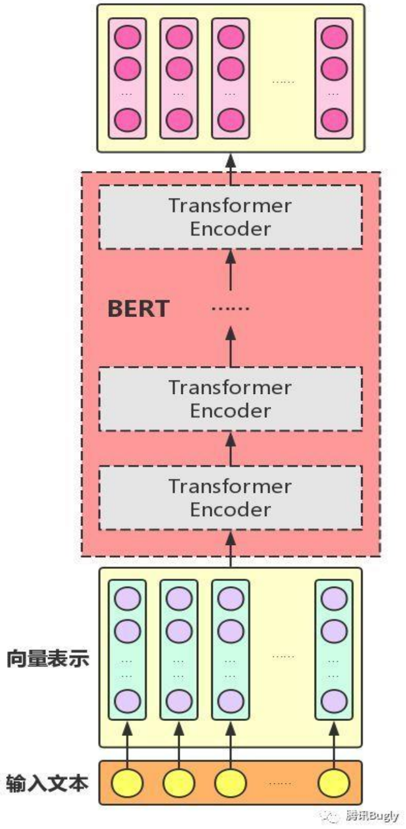
BERT结构

作为Transformer的典型应用Bert，在11种不同NLP测试中创出最佳成绩，包括将GLUE基准推至80.4%（绝对改进7.6%），MultiNLI准确度达到86.7%（绝对改进率5.6%）等。如右图展示了Bert的基本架构，Bert是由右图的基本结构一层一层堆叠而成。



BERT代码结构

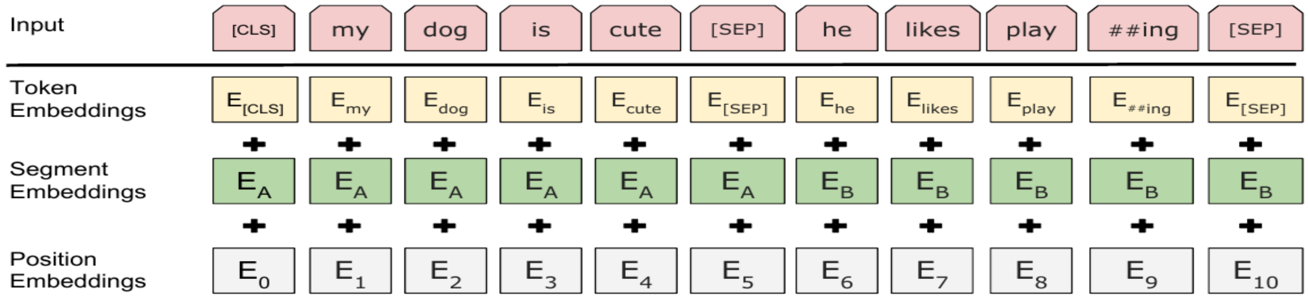
通过前面介绍的TransformerEncoder，Bert是由多个Transformer Encoder一层一层地堆叠起来，模型结构如右图所示。我们使用pytorch实现Bert模型,需要分别实现右图的三个模块,从下到上分别是BertEmbeddings类，BertEncoder类(以BertLayer类叠加组成的类)和BertPooler类。下面我们来分别介绍其实现。



BertEmbeddings类

BERT的输入表示如下图所示，由三部分组成：

1. Token Embeddings
2. Segment Embeddings
3. Position Embeddings



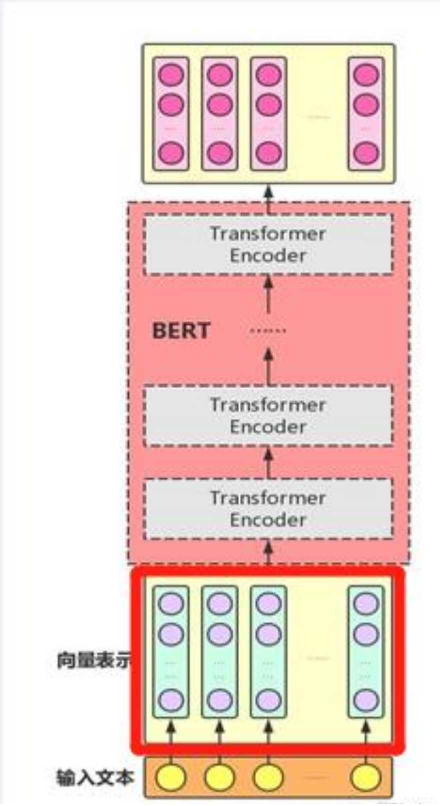
Bert模型实现

https://github.com/huggingface/pytorch-pretrained-BERT

BertEmbeddings类实现

本文只分析forward函数,完整代码见附件

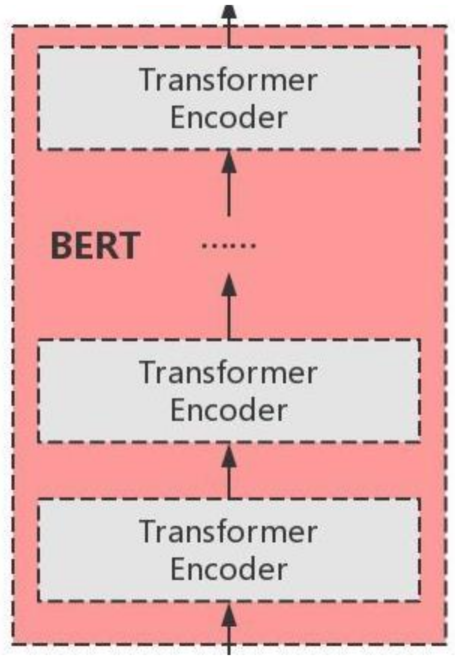
```
def forward(self, input_ids, token_type_ids=None):    #输入单词本身向量
    position_ids = torch.arange(seq_length, dtype=torch.long, device=input_ids.device)
    position_ids = position_ids.unsqueeze(0).expand_as(input_ids)  #生成单词在句子中的位
    if token_type_ids is None:              #生成句子所在单个训练文本中位置的向量
        token_type_ids = torch.zeros_like(input_ids)
    words_embeddings = self.word_embeddings(input_ids)
    position_embeddings = self.position_embeddings(position_ids)
    token_type_embeddings = self.token_type_embeddings(token_type_ids)#通过词典查询
    embeddings = words_embeddings + position_embeddings + token_type_embeddings
    embeddings = self.LayerNorm(embeddings)
    embeddings = self.dropout(embeddings)  #归一化和dropout常见处理
    return embeddings #得到输入
```



BertEncoder类

如右图所示，是由Transformer的Encoder组成，因此只需要实现一个Transformer的Encoder，然后叠加就可以实现Bert的编码部分，因此，BertEncoder类有BertLayer类组成，而BertLayer类由三部分组成：

1. BertAttention类
2. BertIntermediate类
3. BertOut类

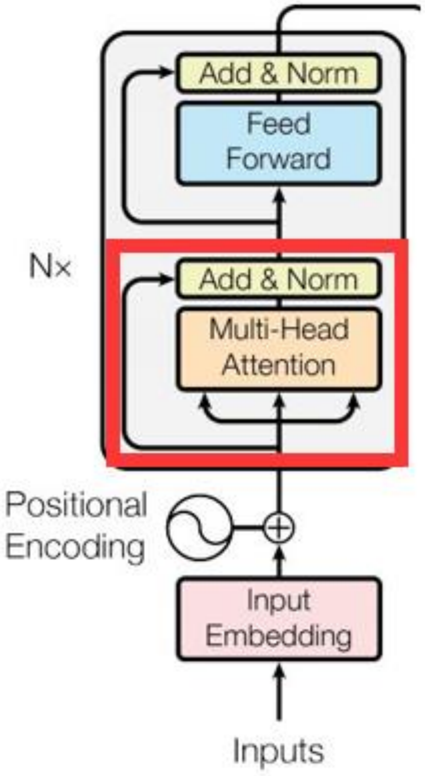


BertAttention类

BertAttention类的实现如下，由两部分组成：

BertSelfAttention类和BertSelfOutput类组成，实现右图的功能

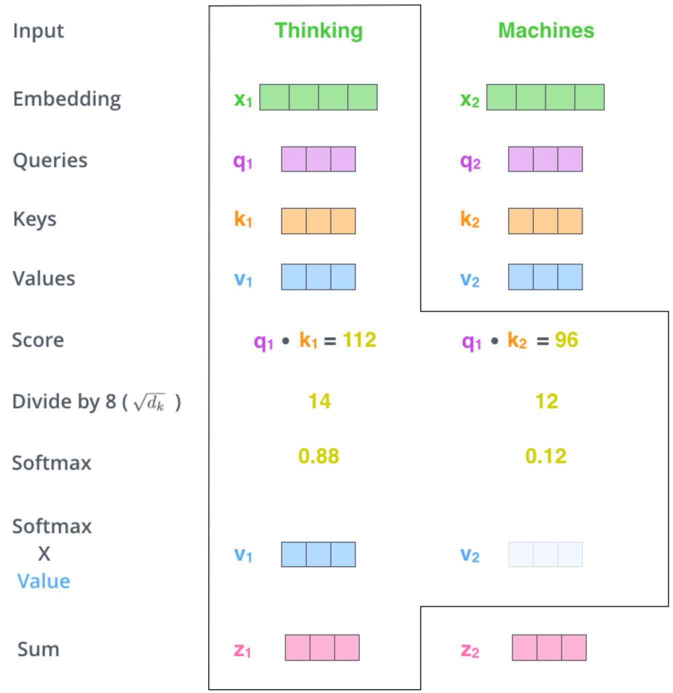
```
 def __init__(self, config):
        super(BertAttention, self).__init__()
        self.self = BertSelfAttention(config)    #实现multi_head和self_attention
        self.output = BertSelfOutput(config)   #层级归一化

    def forward(self, input_tensor, attention_mask):
        self_output = self.self(input_tensor, attention_mask)
        attention_output = self.output(self_output, input_tensor)
        return attention_output
```



Bert模型实现

BertSelfAttention类

```
def forward(self, hidden_states, attention_mask):
    mixed_query_layer = self.query(hidden_states)
    mixed_key_layer = self.key(hidden_states)
    mixed_value_layer = self.value(hidden_states)
    query_layer = self.transpose_for_scores(mixed_query_layer) #多头
    key_layer = self.transpose_for_scores(mixed_key_layer) #多头
    value_layer = self.transpose_for_scores(mixed_value_layer) #多头
    attention_scores = torch.matmul(query_layer, key_layer.transpose(-1, -2))
    attention_scores = attention_scores / math.sqrt(self.attention_head_size)
    attention_scores = attention_scores + attention_mask
    attention_probs = nn.Softmax(dim=-1)(attention_scores)
    attention_probs = self.dropout(attention_probs)
    context_layer = torch.matmul(attention_probs, value_layer)
    return context_layer
```



BertSelfOut类

BertSelfOut类是BertAttention的输出模块,主要实现层级归一化，实现如下:

```
class BertSelfOutput(nn.Module):
    def __init__(self, config):
        super(BertSelfOutput, self).__init__()
        self.dense = nn.Linear(config.hidden_size, config.hidden_size)
        self.LayerNorm = BertLayerNorm(config.hidden_size, eps=config.layer_norm_eps) #实现层级归一化
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
    def forward(self, hidden_states, input_tensor):
        hidden_states = self.dense(hidden_states)
        hidden_states = self.dropout(hidden_states)
        hidden_states = self.LayerNorm(hidden_states + input_tensor)
        return hidden_states
```

BertIntermediate类

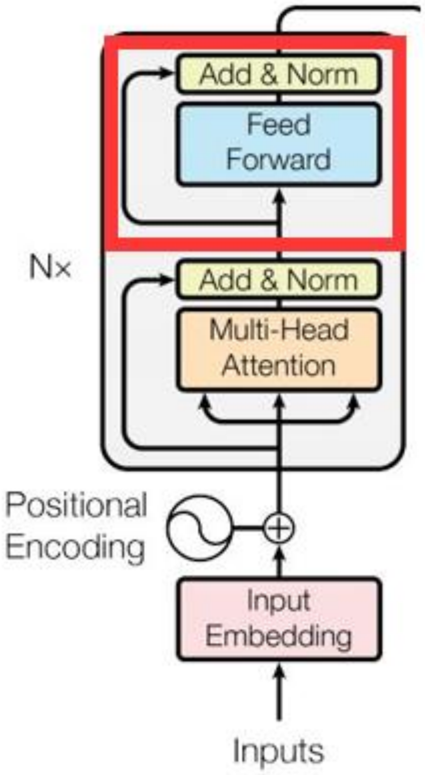
BertIntermediate类，提供激活函数的选择，实现如下：

```
def __init__(self, config):
    super(BertIntermediate, self).__init__()
    self.dense = nn.Linear(config.hidden_size, config.intermediate_size)
    if isinstance(config.hidden_act, str) or (sys.version_info[0] == 2 and isinstance(config.hidden_act, unicode)):
        self.intermediate_act_fn = ACT2FN[config.hidden_act]
    else:
        self.intermediate_act_fn = config.hidden_act
```

BertOutput类

BertEncoder类，在之前已搭建的红色方框的基础上，加入线形Linear层加激活函数ACT2FN，又接了一个Dropout和一个归一化。即完成了红色方框的搭建。实现如下:

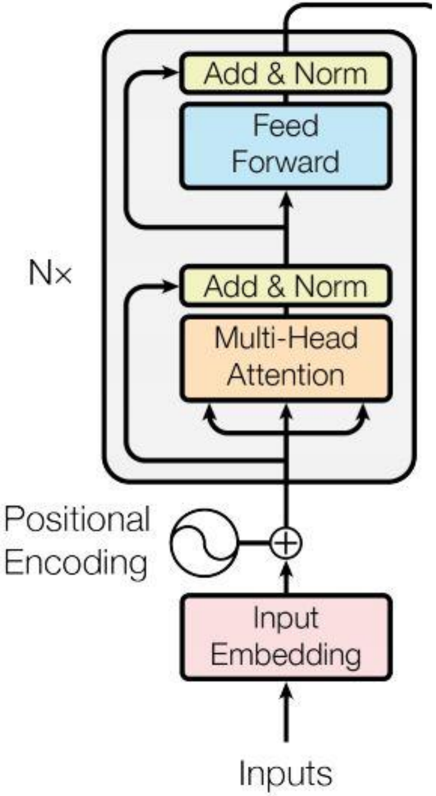
```
def forward(self, hidden_states):
    hidden_states = self.dense(hidden_states)
    hidden_states = self.intermediate_act_fn(hidden_states)
    hidden_states = self.dense(hidden_states)
    hidden_states = self.dropout(hidden_states)
    hidden_states = self.LayerNorm(hidden_states + input_tensor)
    return hidden_states
```



BertLayer类

上面介绍完了实现右图所有结构的内容，现在将其封装成一个BertLayer，方便实现Bert的多层结构，实现如下:

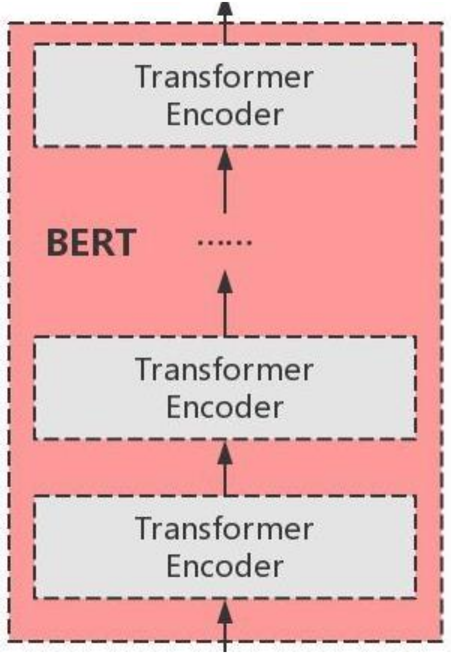
```
def forward(self, hidden_states):#将三个类依次链接
    attention_output = self.attention(hidden_states)
    intermediate_output = self.intermediate(attention_output)
    layer_output = self.output(intermediate_output, attention_output)
    return layer_output
```



BertEncoder类

上面我们已经分别介绍了实现BertLayer的三个类：BertAttention类、BertIntermediate类 、BertOutput类，现将介绍由BertLayer类实现右图的架构，实现代码如下：

```
def forward(self, hidden_states, attention_mask,
output_all_encoded_layers=True):
    all_encoder_layers = []
    for layer_module in self.layer:
        hidden_states = layer_module(hidden_states, attention_mask)
        if output_all_encoded_layers:
            all_encoder_layers.append(hidden_states)
    if not output_all_encoded_layers:
        all_encoder_layers.append(hidden_states)
    return all_encoder_layers
```

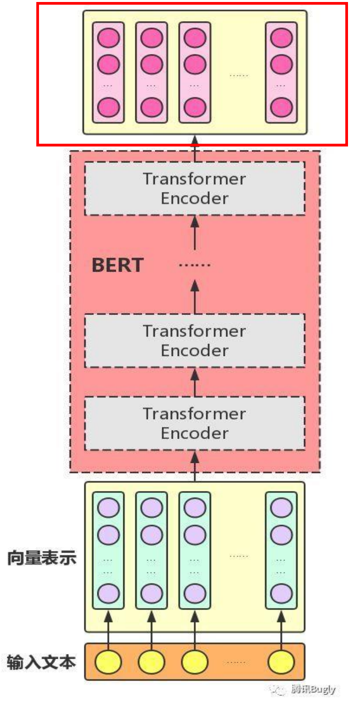


Bert模型实现

BertPooler类

BertPooler类是Bert的输出模块,用过一个linear线形层加一个Tanh()的激活函数，用来池化BertEncoder的输出，实现如下

```
def forward(self, hidden_states):
        first_token_tensor = hidden_states[:, 0]
        pooled_output = self.dense(first_token_tensor)# 通过线性层
        pooled_output = self.activation(pooled_output)#激活层
        return pooled_output
```

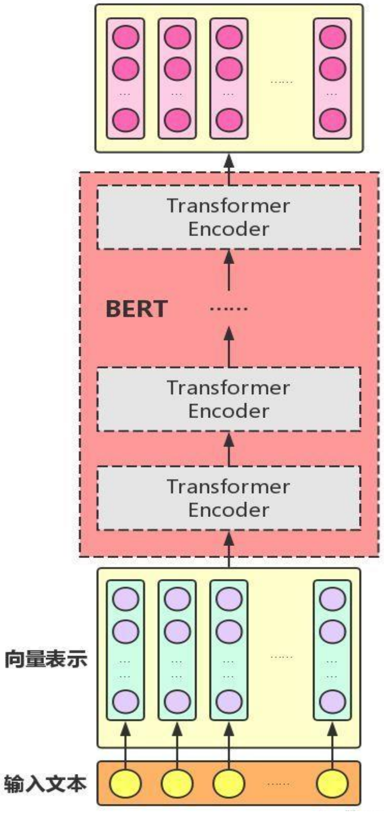


BertModel类

到此为止,我们完成了整个BertEmbeddings和BertEncoder的介绍，接下来我们介绍由BertEmbedding类、BertEncoder类和BertPooler类构建右图的bert模型，BertModel类，实现代码如下：

```
def __init__(self, config):
        super(BertModel, self).__init__(config)
        self.embeddings = BertEmbeddings(config)
        self.encoder = BertEncoder(config)
        self.pooler = BertPooler(config)
        self.apply(self.init_bert_weights)
```

以上是实现右图架构



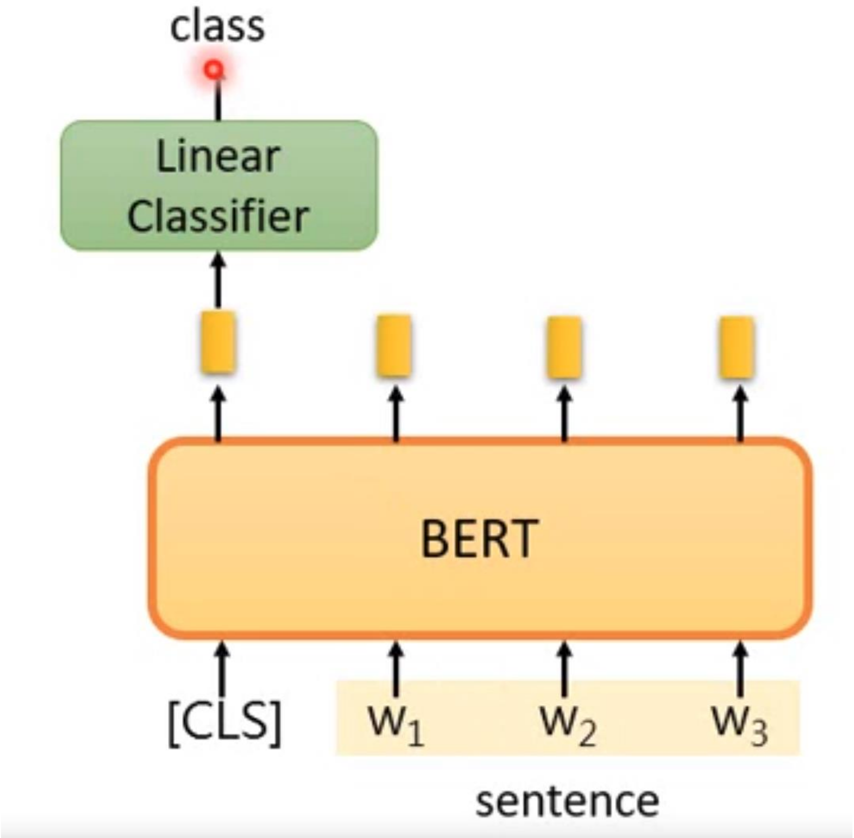
Bert模型实现

文本分类模型实现

实验以及不同数据集的预处理

基于Bert的文本分类器

对于使用Bert的文本分类器的实现方法，一般情况下，如右图在Bert的句子开始标记[CLS]对应输出处加一层线性分类器，因为词向量的输出是增强语义向量，与句子整体关系不大，可以只对[CLS]处输出训练一个对应的文本分类器。在本项目中，我们采用对Bert输出层全连接的方法加上一层线性分类器。它的输入是Bert的所有输出，输出为分类的个数



基于Bert的文本分类器的实现

```
class BertForSequenceClassification(PreTrainedBertModel):
    def __init__(self, config, num_labels=2):       #config:指定的bert模型的预训练参数 num_labels：分类的类别数量
        super(BertForSequenceClassification, self).__init__(config)
        self.num_labels = num_labels#
        self.bert = BertModel(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, num_labels)
        self.apply(self.init_bert_weights)
    def forward(self, input_ids, token_type_ids=None, attention_mask=None, labels=None):   #input_ids：训练集
        pooled_output = self.bert(input_ids, token_type_ids, attention_mask, output_all_encoded_layers=False)[]
        pooled_output = self.dropout(pooled_output)
        logits = self.classifier(pooled_output)
        return logits
```

Bert模型实现

文本分类模型实现

实验以及不同数据集的预处理

实验数据分析以及下载

本实验以CoLA标准数据集为例，CoLA是纽约大学发布的有关语法的数据集，该任务主要是对一个给定句子，判定其是否语法正确，因此，CoLA属于单个句子的文本二分类任务。右图为CoLA数据集的截图,由图可以看出第一列是label，CoLA中label一共是2类：0和1，1代表语法正确，0代表语法错误。第三列是文本训练集。每个训练集的数据中只有一句话。

数据集下载路径：https://nyu-mll.github.io/CoLA/

0	*	the more you would want , the less you would eat .
0	*	i demand that the more john eat , the more he pays .
1		mary listens to the grateful dead , she gets depressed .
1		the angrier mary got , the more she looked at pictures
1		the higher the stakes , the lower his expectations are .

Bert模型训练及预测

实验步骤

本实验以Linux环境为实验平台，具体实验步骤如下：

1. 下载pytorch的模型代码，下载命令如下：

git clone https://github.com/huggingface/pytorch-pretrained-BERT.git

2. 进入path/pytorch-pretrained-BERT/examples下，path是指定机器的路径

   cd path/pytorch-pretrained-BERT/examples

3. 建立gule/cola/目录，命令如下

   mkdir gule

   mkdir gule/cola/

4. 将数据集拷贝到gule/cola/目录下，将训练集和测试集名字改成train.tsv和dev.tsv

   cp in_domain_train.tsv train.tsv

   cp in_domain_dev.tsv   dev.tsv

5. 回到path/pytorch-pretrained-BERT/examples目录下，执行如下命令

export GLUE_DIR = ./glue

export TASK_NAME = cola #数据集名字，必须是cola

python run_classifier.py   --task_name $TASK_NAME   --do_train   --do_eval   --do_lower_case   --data_dir $GLUE_DIR/$TASK_NAME   --bert_model bert-base-uncased   --max_seq_length 128   --train_batch_size 32   --learning_rate 2e-5   --num_train_epochs 3.0   --output_dir ./$TASK_NAME/

--task_name   执行分类任务的名字

--max_seq_length 句子的最大长度，可进行修改

--train_batch_size   训练batch大小，可修改

--learning_rate   学习率，可修改

--num_train_epochs 训练epoch的轮数

--output_dir   输出结果的存储路径

Bert模型训练及预测

6. 实验结果

查看路径path/pytorch-pretrained-BERT/examples/cola/下的eval_results.txt，内容如下：

eval_loss = 0.5076644778477423
global_step = 804
loss = 0.037649269796101684
mcc = 0.6088517493277251

Bert模型训练及预测

不同数据集的预处理

对于不同数据集，我们要对数据进行不同的预处理，那么怎么样在不改变软件架构的情况下，处理这种问题呢，抱抱脸开源的用pytorch实现的bert代码，完美的解决了这个问题。现在分析如下：

1.定义了一个抽取类DataProcessor，代码如下

```
class DataProcessor(object):
    def get_train_examples(self, data_dir):
            raise NotImplementedError()
    def get_dev_examples(self, data_dir):
            raise NotImplementedError()
    def get_labels(self):
            raise NotImplementedError()
```

Bert模型训练及预测

```
@classmethod
 def _read_tsv(cls, input_file, quotechar=None):
    with open(input_file, "r", encoding="utf-8") as f:
       reader = csv.reader(f, delimiter="\t", quotechar=quotechar)
       lines = []
       for line in reader:
          if sys.version_info[0] == 2:
             line = list(unicode(cell, 'utf-8') for cell in line)
          lines.append(line)
       return lines
```

定义了三个方法和一个静态方法，接下来我们以Mrpc为例来说明不同数据集的添加方式。

Bert模型训练及预测

首先，定义了一个MrpcProcesser类，继承了DataProcesser类，然后实行了DataProcesser类的三个方法，具体的代码实现如下：

```
class MrpcProcessor(DataProcessor):
    def get_train_examples(self, data_dir):
        logger.info("LOOKING AT {}".format(os.path.join(data_dir, "train.tsv")))
        return self._create_examples(self._read_tsv(os.path.join(data_dir, "train.tsv")), "train")
    def get_dev_examples(self, data_dir):
        return self._create_examples(self._read_tsv(os.path.join(data_dir, "dev.tsv")), "dev")
    def get_labels(self):
        return ["0", "1"]
```

Bert模型训练及预测

```
def _create_examples(self, lines, set_type):
    examples = []
    for (i, line) in enumerate(lines):
        if i == 0:
            continue
        guid = "%s-%s" % (set_type, i)
        text_a = line[3]
        text_b = line[4]
        label = line[0]
        examples.append(
            InputExample(guid=guid, text_a=text_a, text_b=text_b, label=label))
    return examples
```

Bert模型训练及预测

```
def _create_examples(self, lines, set_type):
    examples = []
    for (i, line) in enumerate(lines):
        if i == 0:
            continue
        guid = "%s-%s" % (set_type, i)
        text_a = line[3]
        text_b = line[4]
        label = line[0]
        examples.append(
            InputExample(guid=guid, text_a=text_a, text_b=text_b, label=label))
    return examples
```

Bert模型训练及预测

接下来将MrpcProcessor加入到Processor，代码实现如下：

```
processors = {
    "cola": ColaProcessor,
    "mnli": MnliProcessor,
    "mnli-mm": MnliMismatchedProcessor,
    "mrpc": MrpcProcessor,
    "sem": SemProcessor,
    "sst-2": Sst2Processor,
}
```

最后将Mrpc的数据输出模式加入output_modes里面，代码实现如下

Bert模型训练及预测

```
output_modes = {
        "cola": "classification",
        "mnli": "classification",
        "mrpc": "classification",
        "sem": "classification",
        "sst-2": "classification",
}
```

执行Mrpc数据集的命令如下：

```
export GLUE_DIR = ./glue
export TASK_NAME = Mrpc  #数据集名字，必须是cola
```

Bert模型训练及预测

python run_classifier.py --task_name $TASK_NAME --do_train --do_eval --do_lower_case --data_dir $GLUE_DIR/$TASK_NAME --bert_model bert-base-uncased --max_seq_length 128 --train_batch_size 32 --learning_rate 2e-5 --num_train_epochs 3.0 --output_dir ./$TASK_NAME/

以上红色的部分，根据实际情况进行修改。

结语

感谢各位聆听!

Thanks for Listening