10分钟掌握Bert源码（PyTorch版）

Python中文社区 1月17日

以下文章来源于NLP从入门到放弃 ，作者DASOU

NLP从入门到放弃

积累一些平时的工作经验和思考，主要是关于NLP，搜索和推荐，只写干货！

Bert在生产环境的应用需要进行压缩，这就要求对Bert结构很了解，这个仓库会一步步解读Bert源代码（pytorch版本）。仓库地址在

https://github.com/DA-southampton/NLP_ability

代码和数据介绍

首先 对代码来说，借鉴的是这个仓库

我直接把代码clone过来，放到了本仓库，重新命名为bert_read_step_to_step。

我会使用这个代码，一步步运行bert关于文本分类的代码，然后同时记录下各种细节包括自己实现的情况。

运行之前，需要做两个事情。

准备预训练模型

一个是预训练模型的准备，我使用的是谷歌的中文预训练模型：chinese_L-12_H-768_A-12.zip，模型有点大，我就不上传了，如果本地不存在，就点击这里直接下载,或者直接命令行运行

```
wget https://storage.googleapis.com/bert_models/2018_11_03/chinese_L-12_H-768_A-12.zip
```

预训练模型下载下来之后，进行解压，然后将tf模型转为对应的pytorch版本即可。对应代码如下:

```
export BERT_BASE_DIR=/path/to/bert/chinese_L-12_H-768_A-12

python convert_tf_checkpoint_to_pytorch.py \
  --tf_checkpoint_path $BERT_BASE_DIR/bert_model.ckpt \
  --bert_config_file $BERT_BASE_DIR/bert_config.json \
  --pytorch_dump_path $BERT_BASE_DIR/pytorch_model.bin
```

转化成功之后，将模型放入到仓库对应位置：

```
Read_Bert_Code/bert_read_step_to_step/prev_trained_model/
```

并重新命名为：

```
bert-base-chinese
```

准备文本分类训练数据

第二个事情就是准备训练数据，这里我准备做一个文本分类任务，使用的是Tnews数据集，这个数据集来源是这里，分为训练，测试和开发集，我已经上传到了仓库中，具体位置在

```
Read_Bert_Code/bert_read_step_to_step/chineseGLUEdatasets/tnews
```

需要注意的一点是，因为我只是为了了解内部代码情况，所以准确度不是在我的考虑范围之内，所以我只是取其中的一部分数据，其中训练数据使用1k，测试数据使用1k，开发数据1k。

准备就绪，使用pycharm导入项目，准备调试，我的调试文件是 run_classifier.py文件，对应的参数为

```
--model_type=bert --model_name_or_path=prev_trained_model/bert-base-chinese --task_name="tnews
```

然后对run_classifier.py 进行调试，我会在下面是调试的细节

1.main函数进入

首先是主函数位置打入断点，位置在这里，然后进入看一下主函数的情况

```
##主函数打上断点
if __name__ == "__main__":
    main()##主函数进入
```

2.解析命令行参数

从这里到这里就是在解析命令行参数，是常规操作，主要是什么模型名称，模型地址，是否进行测试等等。比较简单，直接过就可以了。

3.判断一些情况

从这里到这里是一些常规的判断:

判断是否存在输出文件夹

判断是否需要远程debug

判断单机cpu训练还是单机多gpu训练，还是多机分布式gpu训练，这个有两个参数进行控制

具体可以看代码如下：

```
if args.local_rank == -1 or args.no_cuda:
    device = torch.device("cuda" if torch.cuda.is_available() and not args.no_cuda else "cpu")
    args.n_gpu = torch.cuda.device_count()
else:  # Initializes the distributed backend which will take care of sychronizing nodes/GPUs
    torch.cuda.set_device(args.local_rank)
    device = torch.device("cuda", args.local_rank)
    torch.distributed.init_process_group(backend='nccl')
    args.n_gpu = 1
```

4.获取任务对应Processor

获取任务对应的相应processor，这个对应的函数就是需要我们自己去定义的处理我们自己输入文件的函数，位置在这里，代码如下：

```
processor = processors[args.task_name]()
```

这里我们使用的是，这个结果返回的是一个类，我们使用的是如下的类：

```
TnewsProcessor(DataProcessor)
```

具体代码位置在这里,

4.1 TnewsProcessor

仔细分析一下TnewsProcessor，首先继承自DataProcessor

点击此处打开折叠代码

```
## DataProcessor在整个项目的位置：processors.utils.DataProcessor
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()

    @classmethod
    def _read_tsv(cls, input_file, quotechar=None):
        with open(input_file, "r", encoding="utf-8-sig") as f:
            reader = csv.reader(f, delimiter="\t", quotechar=quotechar)
            lines = []
            for line in reader:
                lines.append(line)
            return lines

    @classmethod
    def _read_txt(cls, input_file):
```

```
        """Reads a tab separated value file."""
        with open(input_file, "r") as f:
            reader = f.readlines()
            lines = []
            for line in reader:
                lines.append(line.strip().split("_!_"))
            return lines
```

然后它自己包含五个函数，分别是读取训练集，开发集数据，获取返回label，制作bert需要的格式的数据

接下来看一下 TnewsProcessor代码格式：

点击此处打开折叠代码

```
class TnewsProcessor(DataProcessor):

    def get_train_examples(self, data_dir):
        """See base class."""
        return self._create_examples(
            self._read_txt(os.path.join(data_dir, "toutiao_category_train.txt")), "train")

    def get_dev_examples(self, data_dir):
        """See base class."""
        return self._create_examples(
            self._read_txt(os.path.join(data_dir, "toutiao_category_dev.txt")), "dev")

    def get_test_examples(self, data_dir):
        """See base class."""
        return self._create_examples(
            self._read_txt(os.path.join(data_dir, "toutiao_category_test.txt")), "test")

    def get_labels(self):
        """See base class."""
        labels = []
        for i in range(17):
            if i == 5 or i == 11:
                continue
            labels.append(str(100 + i))
        return labels

    def _create_examples(self, lines, set_type):
        """Creates examples for the training and dev sets."""
        examples = []
```

```
        for (i, line) in enumerate(lines):
            guid = "%s-%s" % (set_type, i)
            text_a = line[3]
            if set_type == 'test':
                label = '0'
            else:
                label = line[1]
            examples.append(
                InputExample(guid=guid, text_a=text_a, text_b=None, label=label))
        return examples
```

这里有一点需要提醒大家，如果说我们使用自己的训练数据，有两个方法，第一个就是把数据格式变化成和我们测试用例一样的数据，第二个就是我们在这里更改源代码，去读取我们自己的数据格式

5.加载预训练模型

代码比较简单，就是调用预训练模型，不详细介绍了

点击此处打开折叠代码

```
config_class, model_class, tokenizer_class = MODEL_CLASSES[args.model_type]
config = config_class.from_pretrained(args.config_name if args.config_name else args.model_nam
tokenizer = tokenizer_class.from_pretrained(args.tokenizer_name if args.tokenizer_name else ar
model = model_class.from_pretrained(args.model_name_or_path, from_tf=bool('.ckpt' in args.mode
```

6.训练模型-也是最重要的部分

训练模型，从主函数这里看就是两个步骤，一个是加载需要的数据集，一个是进行训练，代码位置在这里。大概代码就是这样：

```
train_dataset = load_and_cache_examples(args, args.task_name, tokenizer, data_type='train')
global_step, tr_loss = train(args, train_dataset, model, tokenizer)
```

两个函数，我们一个个看：

6.1 加载训练集

我们先看一下第一个函数，load_and_cache_examples 就是加载训练数据集，代码位置在这里。大概看一下这个代码，核心操作有三个。

第一个核心操作，位置在这里，代码如下：

```
examples = processor.get_train_examples(args.data_dir)
```

这个代码是为了利用processor读取训练集，很简单。

这里得到的example大概是这样的(这个返回形式在上面看processor的时候很清楚的展示了)：

```
guid='train-0'
label='104'
text_a='今天股票形式不怎么样啊'
text_b=None
```

第二个核心操作是convert_examples_to_features讲数据进行转化，也很简单。

代码位置在这里。代码如下：

```
features = convert_examples_to_features(examples,tokenizer,label_list=label_list,max_length=ar
pad_token_segment_id=4 if args.model_type in ['xlnet'] else 0,
```

我们进入这个函数看一看里面究竟是咋回事，位置在：

```
processors.glue.glue_convert_examples_to_features
```

做了一个标签的映射，'100'->0 '101'->1...

接着获取输入文本的序列化表达：input_ids, token_type_ids；形式大概如此：

'input_ids'=[101, 5500, 4873, 704, 4638, 4960, 4788, 2501, 2578, 102]

'token_type_ids'=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

获取attention mask：attention_mask = [1 if mask_padding_with_zero else 0] * len(input_ids)

结果形式如下：[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

计算出当前长度，获取pading长度，比如我们现在长度是10，那么需要补到128，pad就需要118个0.

这个时候，我们的input_ids 就变成了上面的列表后面加上128个0.然后我们的attention_mask就变成了上面的形式加上118个0，因为补长的并不是我们的第二个句子，我们压根没第二个句子，所以token_type_ids是总共128个0

每操作一个数据之后，我们需要做的是

```
features.append(InputFeatures(input_ids=input_ids,
attention_mask=attention_mask,
token_type_ids=token_type_ids,
label=label,
input_len=input_len))##长度为原始长度，这里应该是10，不是128
```

InputFeatures 在这里就是将转化之后的特征存储到一个新的变量中

在将所有原始数据进行特征转化之后，我们得到了features列表，然后将其中的元素转化为tensor形式，随后

第三个是将转化之后的新数据tensor化，然后使用TensorDataset构造最终的数据集并返回，

```
dataset = TensorDataset(all_input_ids, all_attention_mask, all_token_type_ids, all_lens,all_la
```

6.2 训练模型-Train函数

我们来看第二个函数，就是train的操作。

6.2.1 常规操作

首先都是一些常规操作。

对数据随机采样：RandomSampler

DataLoader读取数据

计算总共训练步数（梯度累计），warm_up 参数设定，优化器，是否fp16等等

然后一个batch一个batch进行训练就好了。这里最核心的代码就是下面的把数据和参数送入到模型中去：

```
outputs = model(**inputs)
```

我们是在进行一个文本分类的demo操作，使用的是Bert中对应的 BertForSequenceClassification 这个类。

我们直接进入这个类看一下里面函数究竟是啥情况。

6.2.2 Bert分类模型：BertForSequenceClassification

主要代码代码如下：

点击此处打开折叠代码

```
##reference: transformers.modeling_bert.BertForSequenceClassification
class BertForSequenceClassification(BertPreTrainedModel):
    def __init__(self, config):
                ...
        ...
        self.bert = BertModel(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, self.config.num_labels)

    def forward(self, input_ids, attention_mask=None, token_type_ids=None,position_ids=None, h
        outputs = self.bert(input_ids,
                            attention_mask=attention_mask,
                            token_type_ids=token_type_ids,
                            position_ids=position_ids,
                            head_mask=head_mask)
        ##zida注解：注意看init中，定义了self.bert就是BertModel，所以我们需要的就是看一看BertModel中类
        pooled_output = outputs[1]
        pooled_output = self.dropout(pooled_output)
                ...
        ...
        return outputs  # (loss), logits, (hidden_states), (attentions)
```

这个类最核心的有两个部分，第一个部分就是使用了 BertModel 获取Bert的原始输出，然后使用 cls的输出继续做后续的分类操作。比较重要的是 BertModel，我们直接进入看 BertModel 这个类的内部情况。代码如下：

然后我们看一下BertModel这个模型究竟是怎么样的

6.2.1.1 BertModel

代码如下：

点击此处打开折叠代码

```
## reference: transformers.modeling_bert.BertModel
class BertModel(BertPreTrainedModel):
    def __init__(self, config):

        self.embeddings = BertEmbeddings(config)
        self.encoder = BertEncoder(config)
        self.pooler = BertPooler(config)
                ...
    def forward(self, input_ids, attention_mask=None, token_type_ids=None,position_ids=None, h
                ...
        ### 第一部分，对 attention_mask 进行操作,并对输入做embedding
        extended_attention_mask = attention_mask.unsqueeze(1).unsqueeze(2)
        extended_attention_mask = extended_attention_mask.to(dtype=next(self.parameters()).dty
        extended_attention_mask = (1.0 - extended_attention_mask) * -10000.0
        embedding_output = self.embeddings(input_ids, position_ids=position_ids, token_type_id
        ### 第二部分 进入 encoder 进行编码
        encoder_outputs = self.encoder(embedding_output,
                                       extended_attention_mask,
                                       head_mask=head_mask)
                ...
        return outputs
```

对于BertModel，我们可以把它分成两个部分，第一个部分是对 attention_mask 进行操作,并对输入做embedding，第二个部分是进入encoder进行编码，这里的encoder使用的是BertEncoder。我们直接进去看一下

6.2.1.1.1 BertEncoder

代码如下：

```
##reference: transformers.modeling_bert.BertEncoder
class BertEncoder(nn.Module):
    def __init__(self, config):
        super(BertEncoder, self).__init__()
        self.output_attentions = config.output_attentions
        self.output_hidden_states = config.output_hidden_states
        self.layer = nn.ModuleList([BertLayer(config) for _ in range(config.num_hidden_layers)

    def forward(self, hidden_states, attention_mask=None, head_mask=None):
        all_hidden_states = ()
        all_attentions = ()
        for i, layer_module in enumerate(self.layer):
```

```
            if self.output_hidden_states:
                all_hidden_states = all_hidden_states + (hidden_states,)

            layer_outputs = layer_module(hidden_states, attention_mask, head_mask[i])
            hidden_states = layer_outputs[0]

            if self.output_attentions:
                all_attentions = all_attentions + (layer_outputs[1],)

        # Add last layer
        if self.output_hidden_states:
            all_hidden_states = all_hidden_states + (hidden_states,)

        outputs = (hidden_states,)
        if self.output_hidden_states:
            outputs = outputs + (all_hidden_states,)
        if self.output_attentions:
            outputs = outputs + (all_attentions,)
        return outputs  # last-layer hidden state, (all hidden states), (all attentions)
```

有一个BertEncoder小细节，就是如果output_hidden_states为True，会把每一层的结果都输出，也包含词向量，所以如果十二层的话，输出是是13层，第一层为word-embedding结果，每层结果都是[batchsize,seqlength,Hidden_size]（除了第一层，[batchsize,seqlength,embedding_size]）

当然embedding_size在维度上是和隐层维度一样的。

还有一点需要注意的就是，我们需要在这里看到一个细节，就是我们是可以做head_mask，这个head_mask我记得有个论文是在做哪个head对结果的影响，这个好像能够实现。

BertEncoder 中间最重要的是BertLayer

- BertLayer

BertLayer分为两个操作，BertAttention和BertIntermediate。BertAttention分为BertSelfAttention和BertSelfOutput。我们一个个来看

- BertAttention
- BertSelfAttention

```
def forward(self, hidden_states, attention_mask=None, head_mask=None):
  ## 接受参数如上
  mixed_query_layer = self.query(hidden_states) ## 生成query [16,32,768],16是batch_size,32是这个
  mixed_key_layer = self.key(hidden_states)
  mixed_value_layer = self.value(hidden_states)
```

```
    query_layer = self.transpose_for_scores(mixed_query_layer)## 将上面生成的query进行维度转化，现在
    key_layer = self.transpose_for_scores(mixed_key_layer)
    value_layer = self.transpose_for_scores(mixed_value_layer)

    # Take the dot product between "query" and "key" to get the raw attention scores.
    attention_scores = torch.matmul(query_layer, key_layer.transpose(-1, -2))
    ## 上面操作之后 attention_scores 维度为torch.Size([16, 12, 32, 32])
    attention_scores = attention_scores / math.sqrt(self.attention_head_size)
    if attention_mask is not None:
    # Apply the attention mask is (precomputed for all layers in BertModel forward() function)
    attention_scores = attention_scores + attention_mask
    ## 这里直接就是相加了，pad的部分直接为非常大的负值，下面softmax的时候，直接就为接近0

    # Normalize the attention scores to probabilities.
    attention_probs = nn.Softmax(dim=-1)(attention_scores)

    # This is actually dropping out entire tokens to attend to, which might
    # seem a bit unusual, but is taken from the original Transformer paper.
    attention_probs = self.dropout(attention_probs)##维度torch.Size([16, 12, 32, 32])

    # Mask heads if we want to
    if head_mask is not None:
      attention_probs = attention_probs * head_mask

    context_layer = torch.matmul(attention_probs, value_layer)##维度torch.Size([16, 12, 32, 64])

    context_layer = context_layer.permute(0, 2, 1, 3).contiguous()##维度torch.Size([16, 32, 12,
    new_context_layer_shape = context_layer.size()[:-2] + (self.all_head_size,)## new_context_la
    context_layer = context_layer.view(*new_context_layer_shape)
  ##维度变成torch.Size([16, 32, 768])
    outputs = (context_layer, attention_probs) if self.output_attentions else (context_layer,)
    return outputs
```

这个时候 BertSelfAttention 返回结果维度为 torch.Size([16, 32, 768])，这个结果作为BertSelfOutput的输入

- BertSelfOutput

```
class BertSelfOutput(nn.Module):
    def __init__(self, config):
        super(BertSelfOutput, self).__init__()
        self.dense = nn.Linear(config.hidden_size, config.hidden_size)
        ## 做了一个linear 维度没变
        self.LayerNorm = BertLayerNorm(config.hidden_size, eps=config.layer_norm_eps)
```

```
        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
```

上面两个函数BertSelfAttention 和BertSelfOutput之后，返回attention的结果，接下来仍然是BertLayer的下一个操作：BertIntermediate

- BertIntermediate

这个函数比较简单，经过一个Linear，经过一个Gelu激活函数

输入结果维度为 torch.Size([16, 32, 3072])

这个结果 接下来进入 BertOutput 这个模型

- BertOutput

也比较简单，Liner+BertLayerNorm+Dropout,输出结果维度为 torch.Size([16, 32, 768])

BertOutput的输出结果返回给BertEncoder 类

BertEncoder 结果返回 BertModel 类 作为 encoder_outputs，维度大小 torch.Size([16, 32, 768])

BertModel的返回为 outputs = (sequence_output, pooled_output,) + encoder_outputs[1:]

sequence_output：torch.Size([16, 32, 768])

pooled_output：torch.Size([16, 768]) 是cls的输出经过一个pool层（其实就是linear维度不变+tanh）的输出

outputs返回给BertForSequenceClassification，也就是对pooled_output 做分类

更多阅读

2020 年最佳流行 Python 库 Top 10

2020 Python中文社区热门文章 Top 10

Top 10 沙雕又有趣的 GitHub 程序