1. 命名实体任务简介

在MUC-6中首次使用了命名实体（named entity）这一术语，由于当时关注的焦点是信息抽取（information extraction）问题，即从报章等非结构化文本中抽取关于公司活动和国防相关活动的结构化信息，而人名、地名、组织机构名、时间和数字表达（包括时间、日期、货币量和百分数等）是结构化信息的关键内容。

命名实体识别（NER）是信息提取的子任务，该任务旨在将文本中的命名实体定位和分类为预定义的类别，例如人员名称，组织，位置等。作为结构化信息提取的重要步骤，它是信息提取，问题解答系统和语法分析等应用领域的重要基础技术。

目前人类已经创建了基于语言语法的技术以及诸如机器学习之类统计模型的NER系统，这种手工制作的基于语法的系统通常可以获得更高的精度，但代价是召回率较低，并且需要让有经验的计算语言学家花费数月的时间，统计NER系统通常需要大量的手动注释训练数据。目前使用半监督方法来避免部分注释工作已成为主流。诸如BERT及其改进版本之类的预训练方法已在许多自然语言理解（NLU）任务（包括NER和其他序列标记任务）上显着提高了性能。现在已经有很多效果优异的基线模型例如BILSTM-CRF、BERT-CRF等，然而面向中国人的可公开访问的高质量细粒度NER数据集还较为缺乏。

1.1. chinese-bert-wwm-ext

本文主要采取以BERT-BILSTM-CRF的模型结构对下述两个数据集分别微调，并查看效果。其中BERT采用的是来自哈工大科大讯飞预训练的chinese-bert-wwm-ext模型。相较于谷歌提供的chinese-bert，哈工大用`全词Mask`方法让chinese-bert-wwm-ext模型在中文NLP任务上有着更好的效果。**因此本文将使用chinese-bert-wwm-ext模型而非谷歌提供的chinese-bert模型。**下述内容是对`全词Mask`的解释，摘自于哈工大讯飞联合实验室。

Whole Word Masking (wwm)，暂翻译为`全词Mask`或`整词Mask`，是谷歌在2019年5月31日发布的一项BERT的升级版本，主要更改了原预训练阶段的训练样本生成策略。 简单来说，原有基于WordPiece的分词方式会把一个完整的词切分成若干个子词，在生成训练样本时，这些被分开的子词会随机被mask。 在`全词Mask`中，如果一个完整的词的部分WordPiece子词被mask，则同属该词的其他部分也会被mask，即`全词Mask`。

同理，由于谷歌官方发布的`BERT-base, Chinese`中，中文是以**字**为粒度进行切分，没有考虑到传统NLP中的中文分词（CWS）。 我们将全词Mask的方法应用在了中文中，使用了中文维基百科（包括简体和繁体）进行训练，并且使用了哈工大LTP作为分词工具，即对组成同一个**词**的汉字全部进行Mask。

下述文本展示了`全词Mask`的生成样例。 **注意：为了方便理解，下述例子中只考虑替换成[MASK]标签的情况。**

说明	样例
原始文本	使用语言模型来预测下一个词的probability。
分词文本	使用 语言 模型 来 预测 下 一个 词 的 probability 。
原始Mask输入	使 用 语 言 [MASK] 型 来 [MASK] 测 下 一 个 词 的 pro [MASK] ##lity 。
全词Mask输入	使 用 语 言 [MASK] [MASK] 来 [MASK] [MASK] 下 一 个 词 的 [MASK] [MASK] [MASK] 。

1.2. 数据集简介

本文命名实体识别任务分别使用下述两个数据集进行了训练。MSRA微软亚洲研究院数据集数据量较大，但是只有地点、机构、人物三种标签。CLUENER2020数据集有十种不同标签，并且有leaderboard公示，能更好得实践搭建训练模型，学习在特定语料库场景下如何提升模型的准确度、召回率等指标。

1.2.1. MSRA微软亚洲研究院数据集

5 万多条中文命名实体识别标注数据（包括地点、机构、人物）， 示例如下。

```
调 O
查 O
范 O
围 O
涉 O
及 O
故 B-LOC
宫 I-LOC
、 O
历 B-LOC
博 I-LOC
、 O
古 B-ORG
研 I-ORG
所 I-ORG
、 O
```

1.2.2. CLUENER2020数据集

CLUENER2020包含10个不同的类别，包括组织，人员姓名，地址，公司，政府，书籍，游戏，电影，职位和场景。它包含13,436个带标签的样本。具体而言，每个样本都包含两部分，即输入的原始文本和带标签的序列。原始文本是一条新闻中的一两个句子。标记的序列被组织为键值对。键是类别，值是实体及其开始和结束位置。值得注意的是，在给定的示例中，一个类别可能有多个实体。与其他可用的中文数据集相比，该数据集带有更多的类别和详细信息。由于完成此任务更具挑战性和难度，区分现代模型的能力要好得多。

标签类别定义 & 标注规则：

地址（address）： **省**市**区**街**号，**路，**街道，**村等（如单独出现也标记）。地址是标记尽量完全的，标记到最细。
书名（book）： 小说，杂志，习题集，教科书，教辅，地图册，食谱，书店里能买到的一类书籍，包含电子书。
公司（company）： **公司，**集团，**银行（央行，中国人民银行除外，二者属于政府机构），如：新东方，包含新华网/中国军网等。
游戏（game）： 常见的游戏，注意有一些从小说，电视剧改编的游戏，要分析具体场景到底是不是游戏。
政府（government）： 包括中央行政机关和地方行政机关两级。 中央行政机关有国务院、国务院组成部门（包括各部、委员会、中国人民银行和审计署）、国务院直属机构（如海关、税务、工商、环保总局等），军队等。
电影（movie）： 电影，也包括拍的一些在电影院上映的纪录片，如果是根据书名改编成电影，要根据场景上下文着重区分下是电影名字还是书名。
姓名（name）： 一般指人名，也包括小说里面的人物，宋江，武松，郭靖，小说里面的人物绰号：及时雨，花和尚，著名人物的别称，通过这个别称能对应到某个具体人物。
组织机构（organization）： 篮球队，足球队，乐团，社团等，另外包含小说里面的帮派如：少林寺，丐帮，铁掌帮，武当，峨眉等。
职位（position）： 古时候的职称：巡抚，知州，国师等。现代的总经理，记者，总裁，艺术家，收藏家等。
景点（scene）： 常见旅游景点如：长沙公园，深圳动物园，海洋馆，植物园，黄河，长江等。

数据字段解释：

以train.json为例，数据分为两列：text & label，其中text列代表文本，label列代表文本中出现的所有包含在10个类别中的实体。
例如：

```
{"text": "浙商银行企业信贷部叶老桂博士则从另一个角度对五道门槛进行了解读。叶老桂认为，对目前国内商业银行而言，", "label": {"name": {"叶老桂": [[9, 11]]}, "company": {"浙商银行": [[0, 3]]}}}
{"text": "生生不息CSOL生化狂潮让你填弹狂扫", "label": {"game": {"CSOL": [[4, 7]]}}}
```

2. 模型搭建

2.1. 基于Transformers做数据预处理、调用BERT

2.1.1. Transformers简介

基于Pytorch和TensorFlow 2.0最先进的自然语言处理。🤗 Transformers（以前称为pytorch-transformers和pytorch-pretrained-bert）提供用于自然语言理解（NLU）和自然语言生成（NLG）的通用体系结构（BERT，GPT-2，RoBERTa，XLM，DistilBert，XLNet等）在100多种语言中提供超过32种以上的预训练模型，以及TensorFlow 2.0和PyTorch之间的深层互操作性。

2.1.2. 使用tokenizer对文本做预处理

tokenizer负责文本的预处理。 首先，它将给定的文本分割为单词（或单词的一部分，标点符号等），通常称为*tokens*。 有多个规则可以控制该过程（您可以在tokenizer摘要中了解有关它们的更多信息，**这就是为什么我们需要使用模型名称实例化tokenizer，以确保我们使用的tokenizer与模型创建时拥有相同的规则。**

```
>>> from transformers import BertTokenizer, BertModel
>>> model_name = "chinese-bert-wwm-ext"
>>> tokenizer = BertTokenizer.from_pretrained(model_name)
```

第二步是将这些标记转换为数字，以便能够根据它们构建张量并将其馈送到模型中。 为此，tokenizer有一个词汇库，这是我们在使用from_pretrained方法实例化时下载的部分，因为我们需要使用与模型经过预训练时相同的词汇。

要将这些步骤应用于给定的文本，我们可以将其提供给tokenizer。您可以将句子列表直接传递给tokenizer。 如果您的目标是将它们批量发送通过模型，则您可能希望将它们全部填充为相同的长度，将它们截断为模型可以接受的最大长度，并恢复张量。 您可以将所有这些指定给tokenizer：

```
>>> pt_batch = tokenizer(
...     ["We are very happy to show you the 🤗 Transformers library.", "We hope
you don't hate it."],
...     padding=True,
...     truncation=True,
...     return_tensors="pt"
... )
```

填充将自动应用到模型预期的一侧（在本例中为右侧），并使用模型经过预训练的填充令牌。 注意掩码也适用于考虑填充：

```
>>> for key, value in pt_batch.items():
...     print(f"{key}: {value.numpy().tolist()}")
input_ids: [[101, 2057, 2024, 2200, 3407, 2000, 2265, 2017, 1996, 100, 19081,
3075, 1012, 102], [101, 2057, 3246, 2017, 2123, 1005, 1056, 5223, 2009, 1012,
102, 0, 0, 0]]
attention_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, 0, 0, 0]]
```

因此使用tokenizer模块即可完成对文本内容的预处理。

2.1.3. 对命名实体标签做预处理

标签预处理的目标是让命名实体标签与文本中的token一一对应，并且要与文本剪裁成相同的长度，用tensor的形式保存下来。**这里有个细节就是tokenizer处理后文本的第一位[CLS]对应的id是'101'，因此标签预处理后第一位对应的一定是'O'。**因为要将标签存储为tensor，所以需要将标签转化为数字。下述的DECODER不仅用在预处理，模型预测中也需要使用DECODER将tensor转化成entity的形式。其中e2t指的是entity to tensor，同理t2e指的是tensor to entity。

```
class DECODER():
    def __init__(self,):
        self.all_decoder = dict()
        self.all_decoder['liu_e2t'] = {
            'O': 0,
            'B-PER': 1,
            'I-PER': 2,
            'B-ORG': 3,
            'I-ORG': 4,
            'B-LOC': 5,
            'I-LOC': 6,
        }
```

```
        self.all_decoder['liu_t2e'] = {self.all_decoder['liu_e2t'][key]:key for
key in
                                       self.all_decoder['liu_e2t'].keys()}

        self.all_decoder['glue_e2t'] = {
            'O': 0,
            'B-address': 1,
            'I-address': 2,
            'B-book': 3,
            'I-book': 4,
            'B-company': 5,
            'I-company': 6,
            'B-game': 7,
            'I-game': 8,
            'B-government': 9,
            'I-government': 10,
            'B-movie': 11,
            'I-movie': 12,
            'B-name': 13,
            'I-name': 14,
            'B-organization': 15,
            'I-organization': 16,
            'B-position': 17,
            'I-position': 18,
            'B-scene': 19,
            'I-scene': 20,
        }

        self.all_decoder['glue_t2e'] = {self.all_decoder['glue_e2t'][key]:key
for key in
                                        self.all_decoder['glue_e2t'].keys()}

    def get(self, type):
        return self.all_decoder[type]
```

通过将主体转换成tensor，我们可以得到下图中模型的输入（input_ids，token_type_ids，attention_mask）和输出。**具体的预处理方程在preprocess和utils脚本中。**

```
>>> text
"浙商银行企业信贷部叶老桂博士则从另一个角度对五道门槛进行了解读。叶老桂认为，对目前国内商业银行而言，"
>>> input_ids
tensor([ 101, 3851, 1555, 7213, 6121,  821,  689,  928, 6587, 6956, 1383, 5439,
        3424, 1300, 1894, 1156,  794, 1369,  671,  702, 6235, 2428, 2190,  758,
        6887, 7305, 3546, 6822, 6121,  749, 6237, 6438,  511, 1383, 5439, 3424,
        6371,  711, 8024, 2190, 4680, 1184, 1744, 1079, 1555,  689, 7213, 6121,
        5445, 6241, 8024,  102])
>>> token_type_ids
tensor([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])
>>> attention_mask
tensor([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])
>>> labels
```

```
tensor([ 0,  5,  6,  6,  6,  0,  0,  0,  0,  0, 13, 14, 14,  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])
```

2.1.4. 基于pytorch写Dataset类

使用Dataset是为了之后能调用pytorch自带的DataLoader，能够更好的调整batch size和使用分布式训练。根据pytorch官方文档描述，继承Dataset类，需要改写两个关键内置函数`__getitem__`和`__len__`。下述就是针对本实验编写的基于BERT NER任务所需要的Dataset。其中`__getitem__`表示的是得到第idx个数据（即item，每个item就是一个字典，包含键值input_ids、token_type_ids、attention_mask、labels）。`__len__`就是这个数据集包含item的长度。完成这两个函数，就可以把训练集的encodings和labels传入BertNerDataset，形成Dataset实例，之后就能传入到DataLoader做后续工作。

```
import torch
from torch.utils.data import Dataset

class BertNerDataset(Dataset):

    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: val[idx].clone().detach() for key, val in
self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)
```

2.1.5. 继承chinese-bert-wwm-ext

这里粘贴了来自transformers官方docs中基于BERT的字符分类模型的源码。字符分类就是对每一个token做标签，命名实体识别就可以看做是其中的一个任务。下列的代码讲述了如何用pytorch继承transformers预训练好的BERT，具体操作的操作就是在`__init__(self, config)`中的`self.bert = BertModel(config, add_pooling_layer=False)`。到这都是好理解的，之后就是微调层的编写：`self.classifier = nn.Linear(config.hidden_size, config.num_labels)`，这就是一层简单的线性分类器，将每个token的768维向量转化为`num_labels`维度的向量。到这为止其实它已经可以做最基础的NER模型了，效果也不会太差。但是这会产生不好的效果，例如生成的结果中可能会生成`O O B-PER I-ORG I-PER O O`这种不符合常理的序列。并且NER任务需要较强的序列位置信息，BERT虽然有位置信息，但是较弱，不能被充分利用。这就是当今NER任务中BERT-BILSTM-CRF结构最为流行的原因，因为它能很好得解决上述两个问题。

```
class BertForTokenClassification(BertPreTrainedModel):

    _keys_to_ignore_on_load_unexpected = [r"pooler"]

    def __init__(self, config):
```

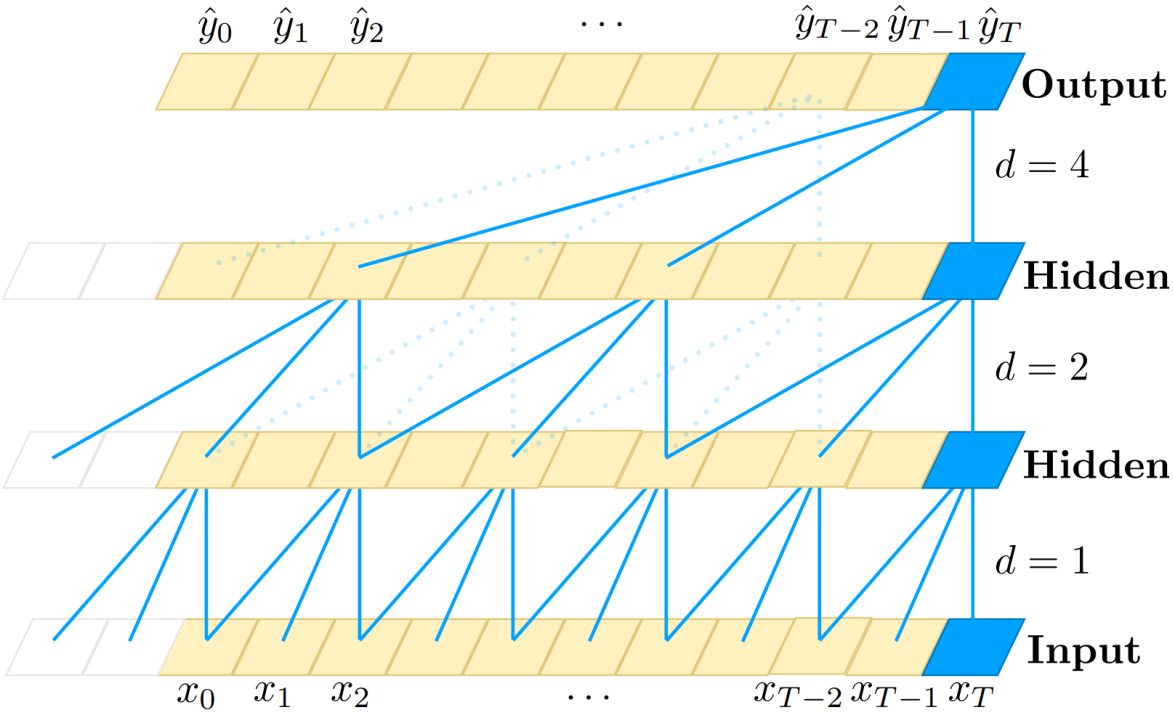
```
        super().__init__(config)
        self.num_labels = config.num_labels

        self.bert = BertModel(config, add_pooling_layer=False)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)

        self.init_weights()

[DOCS]
 @add_start_docstrings_to_model_forward(BERT_INPUTS_DOCSTRING.format("batch_size
, sequence_length"))
    @add_code_sample_docstrings(
        tokenizer_class=_TOKENIZER_FOR_DOC,
        checkpoint="bert-base-uncased",
        output_type=TokenClassifierOutput,
        config_class=_CONFIG_FOR_DOC,
    )
    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        labels=None,
        output_attentions=None,
        output_hidden_states=None,
        return_dict=None,
    ):
        r"""
        labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,
sequence_length)`, `optional`):
            Labels for computing the token classification loss. Indices should be
in ``[0, ..., config.num_labels -
            1]``.
        """
        return_dict = return_dict if return_dict is not None else
self.config.use_return_dict

        outputs = self.bert(
            input_ids,
            attention_mask=attention_mask,
            token_type_ids=token_type_ids,
            position_ids=position_ids,
            head_mask=head_mask,
            inputs_embeds=inputs_embeds,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            return_dict=return_dict,
        )

        sequence_output = outputs[0]

        sequence_output = self.dropout(sequence_output)
        logits = self.classifier(sequence_output)
```

```
        loss = None
        if labels is not None:
            loss_fct = CrossEntropyLoss()
            # Only keep active parts of the loss
            if attention_mask is not None:
                active_loss = attention_mask.view(-1) == 1
                active_logits = logits.view(-1, self.num_labels)
                active_labels = torch.where(
                    active_loss, labels.view(-1),
torch.tensor(loss_fct.ignore_index).type_as(labels)
                )
                loss = loss_fct(active_logits, active_labels)
            else:
                loss = loss_fct(logits.view(-1, self.num_labels),
labels.view(-1))

        if not return_dict:
            output = (logits,) + outputs[2:]
            return ((loss,) + output) if loss is not None else output

        return TokenClassifierOutput(
            loss=loss,
            logits=logits,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
        )
```

2.2. seq2seq模型：LSTM或TCN + 条件随机场

> **时序卷积网络（TCN）**

相比于LSTM，时序卷积网络有稳定的感受野，如下图所示，这是一个三层的膨胀系数为2的时序卷积网络，膨胀系数d使得TCN网络用较少的参数获取更多的历史信息。TCN的源代码来自[Sequence Modeling Benchmarks and Temporal Convolutional Networks (TCN)]。源代码中并没有给出双向TCN的直接调用方法，所以本文只能参照BILSTM，手写两个TCN最终连接输出形成双向TCN的输出。



> **条件随机场（CRF）**

条件随机场的核心：状态转移矩阵。通过迭代学习，状态转移矩阵会将一些不可能出现的标签序列的概率降到最低，例如序列`O I-XXX`或者`B-PER I-LOC`。本文采用的pytorch版本的CRF模型来自于[CLUENER 细粒度命名实体识别](#)。

2.2.1. BERT-BILSTM-CRF

参考transformers给出的`BertForTokenClassification`源代码，自行增添LSTM和CRF模块即可完成新一轮的模型微调，具体代码如下所示。

```
class BertBilstmCRF(BertPreTrainedModel):

    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels

        self.bert = BertModel(config, add_pooling_layer=False)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.bi_lstm = nn.LSTM(
            input_size=config.hidden_size,
            hidden_size=config.hidden_size // 2,
            batch_first=True,
            bidirectional=True,
        )
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)
        #     self.crf = CRF(target_size=config.num_labels, average_batch=True,
use_cuda=config.use_cuda)
        self.crf = CRF(num_tags=config.num_labels, batch_first=True)

        self.init_weights()

    def forward(
            self,
            input_ids=None,
            attention_mask=None,
            token_type_ids=None,
            position_ids=None,
            head_mask=None,
            inputs_embeds=None,
            labels=None,
            output_attentions=None,
            output_hidden_states=None,
            return_dict=None,
    ):

        r"""
        labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,
sequence_length)`, `optional`):
            Labels for computing the token classification loss.
            Indices should be in ``[0, ..., config.num_labels - 1]``.
        """
        return_dict = return_dict if return_dict is not None else
self.config.use_return_dict
```

```
        outputs = self.bert(
            input_ids,
            attention_mask=attention_mask,
            token_type_ids=token_type_ids,
            position_ids=position_ids,
            head_mask=head_mask,
            inputs_embeds=inputs_embeds,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            return_dict=return_dict,
        )

        sequence_output = outputs[0]

        sequence_output = self.dropout(sequence_output)
        sequence_output = self.bi_lstm(sequence_output)[0]

        logits = self.classifier(sequence_output)

        loss = None
        if labels is not None:
            loss = - self.crf(emissions=logits, tags=labels,
mask=attention_mask)

        if not return_dict:
            output = (logits,) + outputs[2:]
            return ((loss,) + output) if loss is not None else output

        return TokenClassifierOutput(
            loss=loss,
            logits=logits,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
        )
```

2.2.2. BERT-TCN-CRF

要实现双向的TCN，这里采用了`self.tcn_forward`和`self.tcn_backward`两个单向的网络，最后连接而成。与BILSTM相同，隐含层维度得设为`config.hidden_size // 2`，保证后续连接后的输出向量为768维度。

```
class BertTcnCRF(BertPreTrainedModel):

    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels

        self.bert = BertModel(config, add_pooling_layer=False)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.tcn_forward = TemporalConvNet(
            num_inputs=config.hidden_size,
            num_channels=[config.hidden_size // 2] * 3,
            kernel_size=3,
            dropout=0.1,
        )
```

```
        self.tcn_backward = TemporalConvNet(
            num_inputs=config.hidden_size,
            num_channels=[config.hidden_size // 2] * 3,
            kernel_size=3,
            dropout=0.1,
        )
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)
        #     self.crf = CRF(target_size=config.num_labels, average_batch=True,
use_cuda=config.use_cuda)
        self.crf = CRF(num_tags=config.num_labels, batch_first=True)

        self.init_weights()

    def forward(
            self,
            input_ids=None,
            attention_mask=None,
            token_type_ids=None,
            position_ids=None,
            head_mask=None,
            inputs_embeds=None,
            labels=None,
            output_attentions=None,
            output_hidden_states=None,
            return_dict=None,
    ):

        r"""
        labels (:obj:`torch.LongTensor` of shape :obj:`(batch_size,
sequence_length)`, `optional`):
            Labels for computing the token classification loss.
            Indices should be in ``[0, ..., config.num_labels - 1]``.
        """
        return_dict = return_dict if return_dict is not None else
self.config.use_return_dict

        outputs = self.bert(
            input_ids,
            attention_mask=attention_mask,
            token_type_ids=token_type_ids,
            position_ids=position_ids,
            head_mask=head_mask,
            inputs_embeds=inputs_embeds,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            return_dict=return_dict,
        )

        sequence_output = outputs[0]

        sequence_output = self.dropout(sequence_output)
        sequence_output_forward = self.tcn_forward(sequence_output.transpose(1,
2)).transpose(1, 2)
        sequence_output_backward =
self.tcn_backward(sequence_output.flip([2]).transpose(1, 2)).transpose(1, 2)
        sequence_output =
torch.cat((sequence_output_forward,sequence_output_backward),2)
```

```
        logits = self.classifier(sequence_output)

        loss = None
        if labels is not None:
            loss = - self.crf(emissions=logits, tags=labels,
mask=attention_mask)

        if not return_dict:
            output = (logits,) + outputs[2:]
            return ((loss,) + output) if loss is not None else output

        return TokenClassifierOutput(
            loss=loss,
            logits=logits,
            hidden_states=outputs.hidden_states,
            attentions=outputs.attentions,
        )
```

3. 模型训练

如果不会使用pytorch训练自定义模型，建议阅读pytorch官方教学文档https://pytorch.org/tutorials/beginner/examples_nn/two_layer_net_module.html。模型训练的核心代码是`loss.backward()`和`optim.step()`，这是pytorch的优势，对初学者十分友好。下述代码中调用logging函数和tqdm函数是为了更好的可视化和实验数据的保存，良好的规范建议时刻保持。

3.1. BERT参数的冻结

微调阶段可以冻结BERT参数也可以不冻结，若需冻结下述代码即可实现。

```
if freezing:
    for param in model.base_model.parameters():
        param.requires_grad = False
```

3.2. 模型的保存

训练完成后建议先将模型转移到CPU上再进行存储，这样下次读取模型不会造成不必要的GPU内存消耗。

3.3. 具体代码

```
def train(model, num_epochs, Train_data_loader, Val_data_loader,
          freezing=False, device_type='cuda', lr=5e-5, model_name='model',):

    logging.info('Commencing training!')
    torch.manual_seed(196)

    device = torch.device(device_type)
```

```
    model.to(device)

    if freezing:
        for param in model.base_model.parameters():
            param.requires_grad = False

    optim = AdamW(model.parameters(), lr=lr)
    train_loss, val_loss = [], []

    for epoch in range(num_epochs):
        model.train()
        stats = OrderedDict()
        stats['loss'] = 0
        stats['lr'] = 0
        stats['batch_size'] = 0
        progress_bar = tqdm(Train_data_loader, desc='| Epoch
{:03d}'.format(epoch), leave=False,
                            disable=False)

        for i, batch in enumerate(progress_bar):
            optim.zero_grad()
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            token_type_ids = batch['token_type_ids'].to(device)
            labels = batch['labels'].to(device)
            outputs = model(
                input_ids=input_ids,
                attention_mask=attention_mask,
                token_type_ids=token_type_ids,
                labels=labels,
                return_dict=True,
            )
            loss = outputs[0]
            loss.backward()
            optim.step()

            total_loss, batch_size = loss.item(), len(batch['labels'])
            stats['loss'] += total_loss
            stats['lr'] += optim.param_groups[0]['lr']
            stats['batch_size'] += batch_size
            progress_bar.set_postfix({key: '{:.4g}'.format(value / (i + 1)) for
key, value in
                                      stats.items()}, refresh=True)

        logging.info('Epoch {:03d}: {}'.format(epoch, ' | '.join(key + '
{:.4g}'.format(
            value / len(progress_bar)) for key, value in stats.items())))

        train_loss.append(stats['loss']/stats['batch_size'])
        val_loss.append(validate(model, Val_data_loader, epoch,
device_type=device_type))

    model.cpu()
    torch.save(model.state_dict(), "finetuned_models/{}.pt".format(model_name))
    make_plot(train_loss, val_loss, model_name)
```

3.4. 训练结果

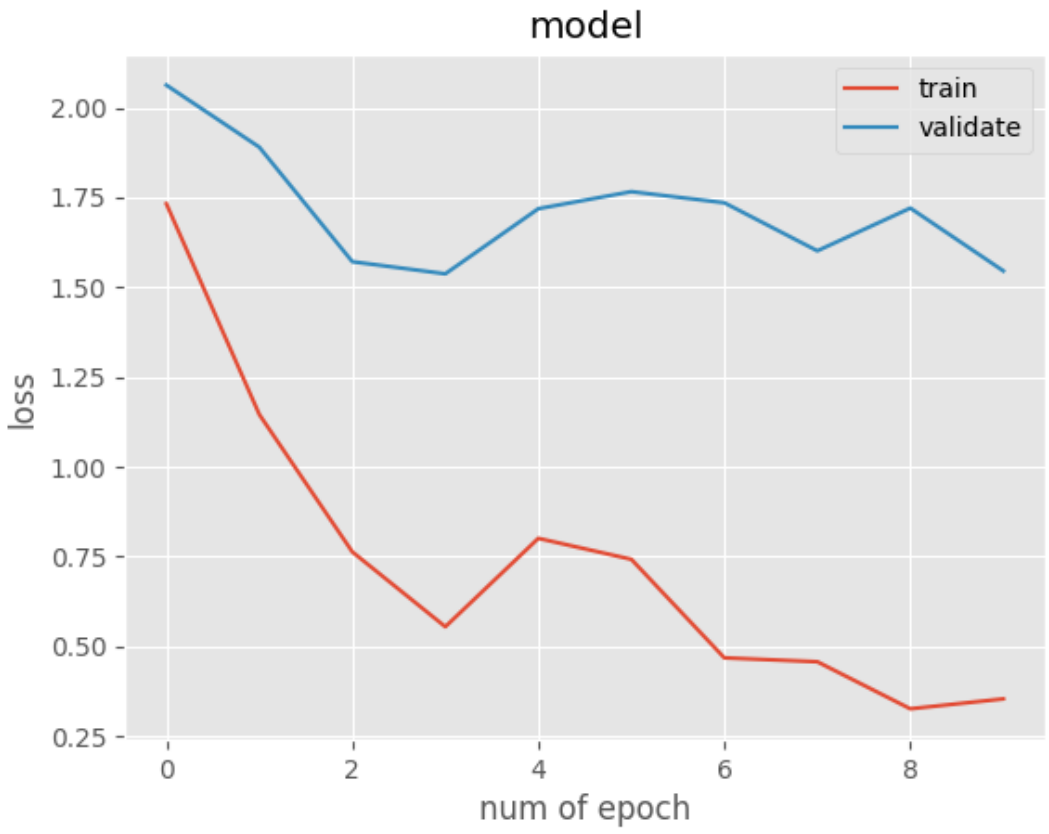
3.4.1. 训练记录log文件

如下是训练的记录文件示例。

```
INFO: Commencing training!
INFO: Epoch 000: loss 7.952 | lr 5e-05 | batch_size 100
INFO: Epoch 000: valid_loss 2.83 | batch_size 100
INFO: Epoch 001: loss 1.962 | lr 5e-05 | batch_size 100
INFO: Epoch 001: valid_loss 2.15 | batch_size 100
INFO: Epoch 002: loss 1.085 | lr 5e-05 | batch_size 100
INFO: Epoch 002: valid_loss 2.12 | batch_size 100
INFO: Epoch 003: loss 0.7195 | lr 5e-05 | batch_size 100
INFO: Epoch 003: valid_loss 2.29 | batch_size 100
```

3.4.2. 训练损失图像

这是BERT-BILSTM-CRF训练10个epochs的损失图像。一般来说，基于BERT的微调基本4个epochs就能达到最好的效果（这个结论是来自于我多次微调的总结，以及其他学者的结论）。



4. 模型分布式

4.1. 基本概念

下面是分布式系统中常用的一些概念：

- `group`：

即进程组。默认情况下，只有一个组，一个 `job` 即为一个组，也即一个 `world`。

当需要进行更加精细的通信时，可以通过 `new_group` 接口，使用 `world` 的子集，创建新组，用于集体通信等。

- `world size`：

表示全局进程个数。

- `rank`：

表示进程序号，用于进程间通讯，表征进程优先级。`rank = 0` 的主机为 `master` 节点。

- `local_rank`：

进程内，`GPU` 编号，非显式参数，由 `torch.distributed.launch` 内部指定。比方说， `rank = 3, local_rank = 0` 表示第 `3` 个进程内的第 `1` 块 `GPU`。

4.2. 具体操作

使用多进程进行分布式训练，我们需要为每个GPU启动一个进程。每个进程需要知道自己运行在哪个GPU上，以及自身在所有进程中的序号。对于多节点，我们需要在每个节点启动脚本。

首先，我们要配置基本的参数：

```
def DDP_train():
    parser = argparse.ArgumentParser()
    parser.add_argument('-n', '--nodes', default=1,
                        type=int, metavar='N')
    parser.add_argument('-g', '--gpus', default=1, type=int,
                        help='number of gpus per node')
    parser.add_argument('-nr', '--nr', default=0, type=int,
                        help='ranking within the nodes')
    parser.add_argument('--epochs', default=2, type=int,
                        metavar='N',
                        help='number of total epochs to run')
    args = parser.parse_args()
    ########################################################
    args.world_size = args.gpus * args.nodes
    os.environ['MASTER_ADDR'] = '172.27.28.196'
    os.environ['MASTER_PORT'] = '8888'
    mp.spawn(rank_train, nprocs=args.gpus, args=(args,))
    ########################################################
```

其中`args.nodes`是节点总数，而`args.gpus`是每个节点的GPU总数（每个节点GPU数是一样的），而`args.nr`是当前节点在所有节点的序号。节点总数乘以每个节点的GPU数可以得到`world_size`，也即进程总数。所有的进程需要知道进程0的IP地址以及端口，这样所有进程可以在开始时同步，一般情况下称进程0是master进程，比如我们会在进程0中打印信息或者保存模型。PyTorch提供了`mp.spawn`来在一个节点启动该节点所有进程，每个进程运行`train(i, args)`，其中`i`从0到`args.gpus - 1`。

同样，我们要修改训练函数：

```
def rank_train(gpu, args):
    ########################################################
    rank = args.nr * args.gpus + gpu
    dist.init_process_group(
```

```
        backend='nccl',
        init_method='env://',
        world_size=args.world_size,
        rank=rank
    )
    ########################################################

    torch.manual_seed(0)

    config_path = 'chinese-bert-wwm-ext/config.json'
    bert_config = BertConfig.from_json_file(config_path)
    tokenizer = BertTokenizer.from_pretrained("chinese-bert-wwm-ext")
    bert_config.num_labels = 7
    model = BertTcnCRF.from_pretrained('chinese-bert-wwm-ext',
config=bert_config)

    torch.cuda.set_device(gpu)
    model.cuda(gpu)
    batch_size = 32

    optimizer = AdamW(model.parameters(), lr=5e-5)
    ###################################################################
    # Wrap the model
    model = nn.parallel.DistributedDataParallel(model, device_ids=[gpu])
    model.train()
    ###################################################################

    # Data loading code
    train_dataset = torch.load('data//liu_data//TrainDataset.pt')
    val_dataset = torch.load('data//liu_data//ValDataset.pt')

    ###################################################################
    train_sampler = torch.utils.data.distributed.DistributedSampler(
        train_dataset,
        num_replicas=args.world_size,
        rank=rank
    )
    ###################################################################

    train_loader = torch.utils.data.DataLoader(
        dataset=train_dataset,
        batch_size=batch_size,
        ##############################
        shuffle=False,  #
        ##############################
        num_workers=0,
        pin_memory=True,
        ##############################
        sampler=train_sampler)  #
    ##############################

    start = datetime.now()
    total_step = len(train_loader)
    for epoch in range(args.epochs):
        for i, batch in enumerate(train_loader):
            input_ids = batch['input_ids'].cuda(non_blocking=True)
            attention_mask = batch['attention_mask'].cuda(non_blocking=True)
            token_type_ids = batch['token_type_ids'].cuda(non_blocking=True)
```

```
            labels = batch['labels'].cuda(non_blocking=True)
            outputs = model.forward(
                input_ids=input_ids,
                attention_mask=attention_mask,
                token_type_ids=token_type_ids,
                labels=labels,
                return_dict=True,
            )
            loss = outputs[0]

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            if (i + 1) % 10 == 0 and gpu == 0:
                print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'.format(
                    epoch + 1,
                    args.epochs,
                    i + 1,
                    total_step,
                    loss.item())
                )
    if gpu == 0:
        print("Training complete in: " + str(datetime.now() - start))
        model.cpu()
        torch.save(model.state_dict(), "finetuned_models/{}.pt".format('model-
ddp'))
```

这里我们首先计算出当前进程序号：`rank = args.nr * args.gpus + gpu`，然后就是通过`dist.init_process_group`初始化分布式环境，其中`backend`参数指定通信后端，包括`mpi, gloo, nccl`，这里选择`nccl`，这是Nvidia提供的官方多卡通信框架，相对比较高效。`mpi`也是高性能计算常用的通信协议，不过你需要自己安装MPI实现框架，比如OpenMPI。`gloo`倒是内置通信后端，但是不够高效。`init_method`指的是如何初始化，以完成刚开始的进程同步；这里我们设置的是`env://`，指的是环境变量初始化方式，需要在环境变量中配置4个参数：MASTER_PORT，MASTER_ADDR，WORLD_SIZE，RANK，前面两个参数我们已经配置，后面两个参数也可以通过`dist.init_process_group`函数中`world_size`和`rank`参数配置。其它的初始化方式还包括共享文件系统以及TCP，比如`init_method='tcp://10.1.1.20:23456'`，其实也是要提供master的IP地址和端口。注意这个调用是阻塞的，必须等待所有进程来同步，如果任何一个进程出错，就会失败。

对于模型侧，我们只需要用`DistributedDataParallel`包装一下原来的model即可，在背后它会支持梯度的All-Reduce操作。对于数据侧，我们`nn.utils.data.DistributedSampler`来给各个进程切分数据，只需要在dataloader中使用这个sampler就好，值得注意的一点是你要训练循环过程的每个epoch开始时调用`train_sampler.set_epoch(epoch)`，（主要是为了保证每个epoch的划分是不同的）其它的训练代码都保持不变。

最后就可以执行代码了，比如我们使用196节点的四块GPU，那么参数就如下：

```
python main.py -n 1 -g 4 -nr 0 --epochs 10
```

如果我们要使用195和196双节点的分布式，那么需要分别在195和196环境下运行：

```
python main.py -n 2 -g 4 -nr 0 --epochs 10
```

```
python main.py -n 2 -g 4 -nr 1 --epochs 10
```

此时195的cuda:0为主节点，195会等待196与其通信，两者完成通信后，八块GPU就能参与分布式训练。分布式训练速度很快，四块GPU的训练速度约为单GPU的3.5倍。训练完成最后在主节点的cuda:0处保存模型。

还有要注意的是，分布式训练的batch_size其实是`batch_size_per_gpu * world_size`。例如在`rank_train`函数中设定`batch_size`为32，那么只使用196单节点的分布式，最终的`batch_size = 32*4 = 128`了。

> **具体的分布式知识请看我之前写的`PyTorch分布式调研`。**

5. 模型测试

5.1. 评判标准

模型的最终输出是标签序列，这通过CRF层中的维特比解码器实现。NER模型的评判标准为准确度、召回率和F1分数。一开始天真地认为预测出的单个非O标签与单个实际标签一致就算正确，于是写了下列两个函数。这就导致了例如预测序列`B-PER I-PER O O`与标准序列`B-PER I-PER I-PER O`算出来准确率100%，召回率66.6%。

```
def compute_precision(tags, labels):
    """
    :param tags: torch.tensor (batch_size, seq_len, num_labels)
    :param labels: torch.tensor (batch_size, seq_len, num_labels)
    :return: torch.tensor (1)
    """
    try:
        return int(torch.sum((tags != 0) * (tags == labels))) /
int(torch.sum(tags != 0))
    except:
        return 0


def compute_recall(tags, labels):
    """
    :param tags: torch.tensor (batch_size, seq_len, num_labels)
    :param labels: torch.tensor (batch_size, seq_len, num_labels)
    :return: torch.tensor (1)
    """
    try:
        return int(torch.sum((labels != 0) * (tags == labels))) /
int(torch.sum(labels != 0))
    except:
        return 0
```

但实际上预测的实体需要完全对应上标准答案，上述的预测序列`B-PER I-PER O O`与标准序列`B-PER I-PER I-PER O`的真实准确率和召回率其实均为0%。于是标准的计算准确率、召回率和F1分数的方程如下所示。

```
def split_entity(label_sequence):
```

```
    entity_mark = dict()
    entity_pointer = None
    for index, label in enumerate(label_sequence):
        if label.startswith('B'):
            category = label.split('-')[1]
            entity_pointer = (index, category)
            entity_mark.setdefault(entity_pointer, [label])
        elif label.startswith('I'):
            if entity_pointer is None: continue
            if entity_pointer[1] != label.split('-')[1]: continue
            entity_mark[entity_pointer].append(label)
        else:
            entity_pointer = None
    return entity_mark


def evaluate(LABELS, TAGS):
    """
    LABELS和TAGS的格式范例：
    [
    ['O','B-PER','I-PER','O','O'],
    ['O','B-PER','I-PER','O','O','B-LOC','I-LOC','O'],
    ]
    """
    real_entity_num = 0
    predict_entity_num = 0
    true_entity_num = 0
    for i in range(len(TAGS)):
        real_label = LABELS[i]
        predict_label = TAGS[i]
        real_entity_mark = split_entity(real_label)
        predict_entity_mark = split_entity(predict_label)

        true_entity_mark = dict()
        key_set = real_entity_mark.keys() & predict_entity_mark.keys()
        for key in key_set:
            real_entity = real_entity_mark.get(key)
            predict_entity = predict_entity_mark.get(key)
            if tuple(real_entity) == tuple(predict_entity):
                true_entity_mark.setdefault(key, real_entity)

        real_entity_num += len(real_entity_mark)
        predict_entity_num += len(predict_entity_mark)
        true_entity_num += len(true_entity_mark)

    precision = true_entity_num / predict_entity_num if predict_entity_num != 0
else 0
    recall = true_entity_num / real_entity_num if real_entity_num != 0 else 0
    f1 = 2 * precision * recall / (precision + recall) if precision + recall !=
0 else 0

    return precision, recall, f1
```

5.2. MSRA数据集训练结果

具体的预测方程请见在`train.py`中的`predict`函数，在这就不多进行说明。按照训练集验证集测试集64:16:20的比例训练两种NER模型得到结果：bert-lstm-crf f1: 91.04%; precision: 91.76%; recall: 90.34%。bert-tcn-crf f1: 91.28%; precision: 91.05%; recall: 91.50%，时序卷积网络稍占上风。MSRA数据集较为规整，不存在单个实体多标签的情况，所以模型的预测结果相当好。

5.3. CLUENER2020数据集训练结果

保持模型结构不变，将MSRA数据集换做CLUNER数据集后，初步预测结果准确度和召回率均为62%左右，与leaderboard还有差距。目前还未对数据集分析和进一步的预处理，刘算法描述该数据集存在单实体多标签的情况。由于该数据集有10种标签，场景难度增加，因此后续需要进一步学习，数据预处理和改进模型。

6. 模型优化方案

这一部分在针对CLUENER数据集优化模型后再进行编写。

7. 引用

1. Chinese-BERT-wwm
2. Transformers
3. MSRA微软亚洲研究院数据集
4. CLUENER2020
5. Sequence Modeling Benchmarks and Temporal Convolutional Networks (TCN)
6. Pytorch 分布式训练
7. PyTorch分布式训练简明教程