基于深度学习的文本分类任务

一、项目简介

本项目主要内容为使用深度学习技术实现文本分类任务，具体实现了包括基于BERT和TextCNN的代码。本次实践是基于课程实践部分内容的代码上进行改进，并在一个文本分类数据集上进行了测试。本报告将介绍数据集、模型实现以及实验结果，最后将提及自己实现过程遇到的问题和感悟。项目主要目的是想学习预训练模型的使用以及一些简单的语言模型的实现，锻炼自己的代码能力，经过学习后自己对于python语言及pytorch框架、数据预处理、语言模型的输入输出等多方面都有了更深刻的认识。

二、数据集及数据预处理

1. 数据集简介

本次实验使用的数据集来源由网上搜集（数据来源链接），共含有8个类别的文，其标签如下所示：

标签	类别
0	电脑
1	水果
2	平板
3	书籍
4	衣服
5	酒店
6	蒙牛
7	洗浴

其中训练集包括402条数据，验证集包括63条数据，样例如下图所示：

```
1    买回来才知道潘苹果原来是这个意思，没想到与金融大鳄也有个亲密接触。包装不错，苹果挺甜的，个头也大。
7    沙宣的东西非常不错，买了几次了，真的可以，还要买买买
2    总体感觉就是 做不出性价比高的平板来 就别做 也不能 忽悠人吧
7    一直用这个牌子，很好，第二次买了
```

2. 预处理

该数据集非常简单，在使用使用BERT模型时，无需特殊预处理，只需要构建好相关数据集，并经过tokenizer即可。

但在使用TextCNN模型时，我首先使用`jieba`进行了中文分词处理，然后构建了词表，再转化为向量，代码详见`data.py`。

其中语言模型的数据集类`LanguageModelDataset`代码如下所示，`DataLoader`使用默认即可，值得注意的是，`label`的标签在计算交叉熵损失前需要转化为`int64`，实践过程中由于类似的对于数据类型和维度的马虎，代码出现了不少的bug。

```
class LanguageModelDataset(Dataset):
    def __init__(self, data, max_seq_len):
        self.vocab = get_vocab()
        self.vocab_size = len(self.vocab)
        self.lines = []
        with open(data, 'r', encoding='utf8') as f:
            self.lines += f.readlines()
        self.max_seq_len = max_seq_len

    def __getitem__(self, index):
        item = {
            'input_ids': None,
            'labels': None
        }
        example = self.lines[index]
        label, content = example.split('\t')
        content = jieba.cut(content, cut_all=False)
        input_ids = [self.vocab[w] for w in content]
        if len(input_ids) > self.max_seq_len:
            input_ids = input_ids[:self.max_seq_len]
        if len(input_ids) < self.max_seq_len:
            input_ids = input_ids + [0 for _ in range(self.max_seq_len - len(input_ids))]

        item['input_ids']=input_ids
        item['labels']=int(label)

        for k, v in item.items():
            item[k] = np.array(v)
        return item

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

三、模型实现及结果

1. BERT

BERT模型的强大已经在无数应用中得到证实，为了对预训练模型有进一步的了解，本次实践采用了`huggingface`提供的预训练`bert-case-chinese`模型，在其基础上，进行微调，实现下游的文本分类任务。

总的来说，基于BERT的文本分类模型只需要在原始的BERT模型后加上一个分类层即可，本文使用`transformer`库中的`AutoModelForSequenceClassification`进行实现，则更为简便，只需要修改类别参数，模型核心部分如图所示，具体代码可以参见`bert.py`。

```
config = AutoConfig.from_pretrained(args.model)
config.num_labels = args.n_labels  # 设置类别数
model = AutoModelForSequenceClassification.from_pretrained(args.model,
config=config)
tokenizer = AutoTokenizer.from_pretrained(args.model)
```

值得一提的是在本次实践过程中，学习到了一种较为简便的数据处理操作，通过使用`datasets`库的`load_dataset`、`functools`库的`partial`等方法，较之自己之前实现的方式有所不同，其中`convert_exp`函数实现在`utils.py`文件中，主要功能为将文本通过tokenizer转变为向量输入到模型中。

```
    # loading data
    logger.info("Loading dataset from {} ...".format(args.train_path + ' ' +
args.dev_path))
    dataset = load_dataset('text', data_files={'train': args.train_path, 'dev':
args.dev_path})
    logger.info(dataset)
    # convert the text data into tensors
    convert_func = partial(convert_exp, tokenizer=tokenizer,
max_seq_len=args.max_seq_len)
    dataset = dataset.map(convert_func, batched=True)
    # data loader
    train_dataset, eval_dataset = dataset["train"], dataset["dev"]
    print(type(train_dataset))
    train_dataloader = DataLoader(train_dataset, shuffle=True,
collate_fn=default_data_collator,
                                  batch_size=args.batch_size)
    eval_dataloader = DataLoader(eval_dataset, collate_fn=default_data_collator,
batch_size=args.batch_size)
```

在训练20个epoch后，模型结果如下图所示，准确率达到了87%，f1则为90%，召回率为91%，在样本稀少的情况下，BERT通过快速的微调就取得了不错的结果。（注：其中global optim是指最好的f1值，下同）

```
Valid | acc: 0.87302, f1: 0.90699, recall: 0.91927, global optim: 0.90699, loss: 0.41622
| epoch  16 |    10/   26 batches | ms/batch 217.70 | loss  0.01
| epoch  16 |    20/   26 batches | ms/batch 194.51 | loss  0.01
| End of epoch  16 | time:  5.45s |
| epoch  17 |     0/   26 batches | ms/batch 11.60 | loss  0.01
Valid | acc: 0.87302, f1: 0.90699, recall: 0.91927, global optim: 0.90699, loss: 0.41906
| epoch  17 |    10/   26 batches | ms/batch 218.62 | loss  0.01
| epoch  17 |    20/   26 batches | ms/batch 194.39 | loss  0.01
| End of epoch  17 | time:  5.46s |
| epoch  18 |     0/   26 batches | ms/batch 11.70 | loss  0.01
Valid | acc: 0.87302, f1: 0.90699, recall: 0.91927, global optim: 0.90699, loss: 0.42136
| epoch  18 |    10/   26 batches | ms/batch 219.23 | loss  0.01
| epoch  18 |    20/   26 batches | ms/batch 194.23 | loss  0.01
| End of epoch  18 | time:  5.47s |
| epoch  19 |     0/   26 batches | ms/batch 12.00 | loss  0.01
Valid | acc: 0.87302, f1: 0.90699, recall: 0.91927, global optim: 0.90699, loss: 0.42430
| epoch  19 |    10/   26 batches | ms/batch 220.97 | loss  0.01
| epoch  19 |    20/   26 batches | ms/batch 196.04 | loss  0.01
| End of epoch  19 | time:  5.51s |
| epoch  20 |     0/   26 batches | ms/batch 11.90 | loss  0.01
Valid | acc: 0.87302, f1: 0.90699, recall: 0.91927, global optim: 0.90699, loss: 0.42377
| epoch  20 |    10/   26 batches | ms/batch 219.97 | loss  0.01
| epoch  20 |    20/   26 batches | ms/batch 196.15 | loss  0.01
| End of epoch  20 | time:  5.50s |
```

2. TextCNN

CNN模型常被用于计算机视觉方面的工作，而《Convolutional Neural Networks for Sentence Classification》一问提出的TextCNN使得CNN同样可以被应用于文本分析。TextCNN利用多个不同size的kernel来提取句子中的关键信息，从而能够更好地捕捉局部相关性，其结构也简单易实现，如下图所示，包含一层卷积、一层max-pooling，最后将输出外接softmax来n分类。

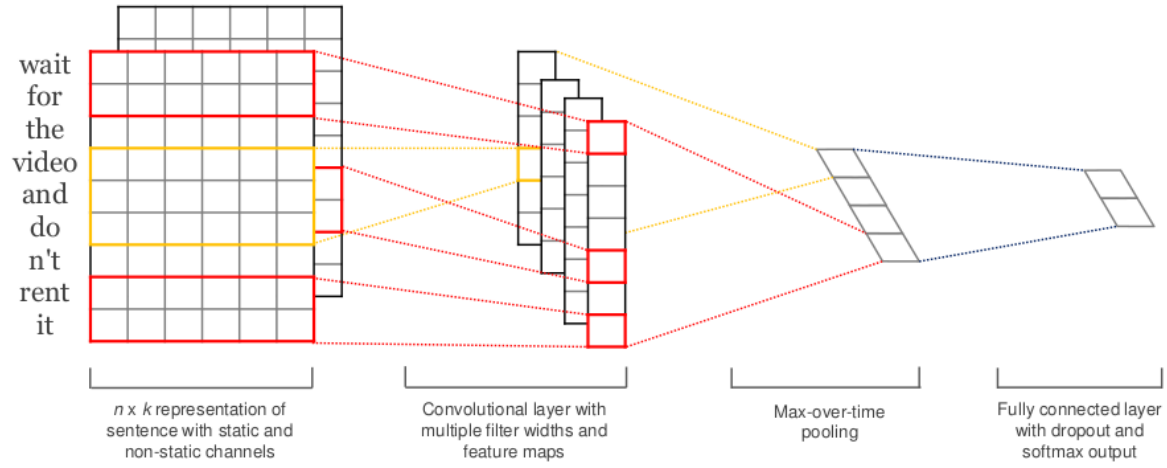

Figure 1: Model architecture with two channels for an example sentence.

其代码如下：

```python
class TextCNN(nn.Module):
    def __init__(self, n_labels, vocab_size, embed_dim, kernel_sizes, chanel_num, kernel_num):
        super(TextCNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.convs = nn.ModuleList(
            [nn.Conv2d(chanel_num, kernel_num, (size, embed_dim)) for size in kernel_sizes])
        self.dropout = nn.Dropout(args.dropout)
```

```
        self.output = nn.Linear(len(kernel_sizes) * kernel_num, n_labels)

    def forward(self, x):
        x = self.embedding(x).unsqueeze(1)
        x = [F.relu(conv(x)).squeeze(3) for conv in self.convs]
        x = [F.max_pool1d(item, item.size(2)).squeeze(2) for item in x]
        x = torch.cat(x, 1)
        x = self.dropout(x)
        logits = self.output(x)
        return logits
```

在经过100个epoch的训练后，其准确度在60左右徘徊，对比BERT模型来说，效果差了许多。

```
Valid | acc: 0.60317, f1: 0.47484, recall: 0.49107, global optim: 0.49134, loss: 1.18718
Valid | acc: 0.61905, f1: 0.48434, recall: 0.49888, global optim: 0.49134, loss: 1.18568
| End of epoch  97 | time:  1.11s |
Valid | acc: 0.61905, f1: 0.48434, recall: 0.49888, global optim: 0.48434, loss: 1.18532
| epoch  98 |    10/   26 batches | ms/batch 38.06 | loss  0.00
Valid | acc: 0.61905, f1: 0.48434, recall: 0.49888, global optim: 0.48434, loss: 1.18353
Valid | acc: 0.61905, f1: 0.48434, recall: 0.49888, global optim: 0.48434, loss: 1.18265
| epoch  98 |    20/   26 batches | ms/batch 41.01 | loss  0.00
Valid | acc: 0.61905, f1: 0.48434, recall: 0.49888, global optim: 0.48434, loss: 1.18180
Valid | acc: 0.61905, f1: 0.48434, recall: 0.49888, global optim: 0.48434, loss: 1.18182
| End of epoch  98 | time:  1.11s |
Valid | acc: 0.60317, f1: 0.45325, recall: 0.45722, global optim: 0.45325, loss: 1.18130
| epoch  99 |    10/   26 batches | ms/batch 37.96 | loss  0.00
Valid | acc: 0.60317, f1: 0.45325, recall: 0.45722, global optim: 0.45325, loss: 1.18166
Valid | acc: 0.60317, f1: 0.45325, recall: 0.45722, global optim: 0.45325, loss: 1.18273
| epoch  99 |    20/   26 batches | ms/batch 41.11 | loss  0.00
Valid | acc: 0.61905, f1: 0.48077, recall: 0.49888, global optim: 0.48077, loss: 1.18872
Valid | acc: 0.60317, f1: 0.47082, recall: 0.49107, global optim: 0.48077, loss: 1.19522
| End of epoch  99 | time:  1.11s |
Valid | acc: 0.58730, f1: 0.44799, recall: 0.48326, global optim: 0.44799, loss: 1.20060
| epoch 100 |    10/   26 batches | ms/batch 39.16 | loss  0.00
Valid | acc: 0.58730, f1: 0.44799, recall: 0.48326, global optim: 0.44799, loss: 1.20363
Valid | acc: 0.58730, f1: 0.44799, recall: 0.48326, global optim: 0.44799, loss: 1.20446
| epoch 100 |    20/   26 batches | ms/batch 40.51 | loss  0.00
Valid | acc: 0.58730, f1: 0.45076, recall: 0.48326, global optim: 0.45076, loss: 1.20487
Valid | acc: 0.58730, f1: 0.45076, recall: 0.48326, global optim: 0.45076, loss: 1.20669
| End of epoch 100 | time:  1.13s |
```

四、实践总结

在完成本次实践的过程中，我阅读了许多的代码和教程，学习到了许多。尽管BERT模型的实践网上也有许多的代码，但在这次自己实现的过程中，学习到了更多的库和函数，并且重新整理了自己的编程习惯。在课上实践部分学习了TextRNN后，实践过程中，我自学了TextCNN的实现，并且在实现针对模型需要的输入数据对数据进行预处理的过程中，对语言模型的输入输出有了更深的了解，代码能力也有所提高。

自己在实践过程中遇到了不少问题，有的通过查资料就解决了，但有的现在都没有理解因此后面改变了代码转换了思路：即BERT的数据处理操作，在针对TextCNN修改后依旧无法运行，在确认代码基本无误后重启电脑运行成功过，但再次运行依旧没有调用原本应当调用的`convert_exp`函数，调试许久无果，因此在TextCNN中换回了原本的`Dataset`、`DataLoader`函数。

由于学习还在初级阶段，并不深入，因此实践显得较为浅显，另外对于实验结果的分析也不到位，多次调参后TextCNN的结果也并不理想，词表的构建部分代码存在缺陷。总体来说有较大改进空间。

感谢老师和学长学姐们的指导，预祝新年快乐！

参考链接

1. transformers_tasks/text_classification at 9099c0766d41e06403470d901ca6fed1d7b18f6e · HarderThenHarder/transformers_tasks
2. 课上的 `rnnlm.py` 代码