PRML Assignment 4

Part 1 文本分类任务：RNN与CNN

0. 实验概述

0.1 实验目的

文本分类是自然语言处理的一个常见任务，它把一段不定长的文本序列变换为文本的类别。本次实验将通过文本分类任务，熟悉NLP中CNN和RNN模型的搭建，并掌握fastNLP的使用方法。

0.2 使用模型及训练方法

RNN: 在Bi-directional LSTM的基础上进行改进。
CNN: 在TextCNN的基础上进行改进。

使用fastNLP的Trainer进行模型的训练，使用交叉熵损失进行梯度下降，梯度下降主要使用Adam优化算法。在Trainer中加入EarlyStop机制，并借助Fitlog实现训练过程可视化和训练结果记录。

使用验证集准确率挑选具有合适超参数的模型，使用测试集准确率评价模型性能。

Trainer的参数设置如下:

```
trainer = Trainer(
    train_data=train_data,
    model=model,
    loss=CrossEntropyLoss(pred=Const.OUTPUT, target=Const.TARGET),
    metrics=AccuracyMetric(),
    n_epochs=80,
    batch_size=batch_size,
    print_every=10,
    validate_every=-1,
    dev_data=dev_data,
    optimizer=torch.optim.Adam(model.parameters(), lr=learning_rate),
    check_code_level=2,
    metric_key='acc',
    use_tqdm=True,
    callbacks=[FitlogCallback(), EarlyStopCallback(train_patience)],
    device=device,
)

tester = Tester(test_data, model, metrics=AccuracyMetric())
```

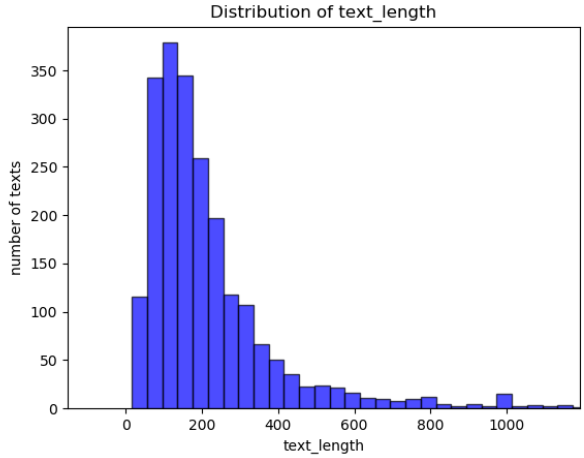
1. 数据处理

使用的数据集来自sklearn.datasets的fetch_20newsgroups: 训练数据集有11314条文本，测试数据集有7532条文本，文本类别总数为20。

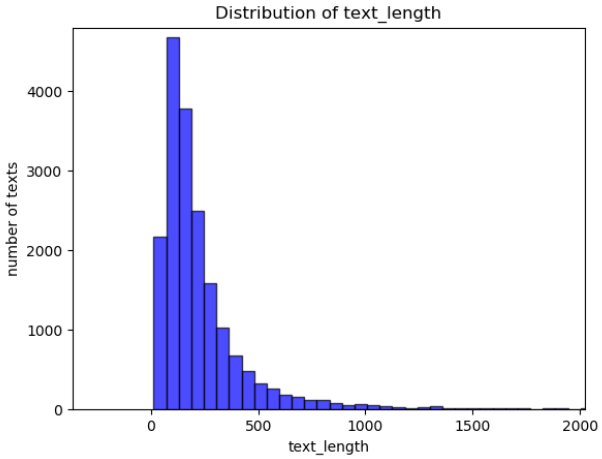
本实验将先在实验2中的4分类小数据集上进行训练，然后在整个20分类大数据集上进行训练。

文本数据处理方式：

1. 将文本逐条存入Dataset， 去除文本中的标点符号，将不可见符替换为空白符，根据空白符分词。
2. 为方便之后对文本向量进行Batch，且尽量减少padding，需要设置合适的最大文本长度。所以我们需要观察数据集中文本长度的分布：



4分类小数据集的文本长度分布



20分类大数据集的文本长度分布

可知90%以上的文本长度在400以下。因此我们设置**最大文本长度为400**。根据最大文本长度，截断数据集中的文本。之后做Batch的时候会在长度不足的文本后添加padding字符。

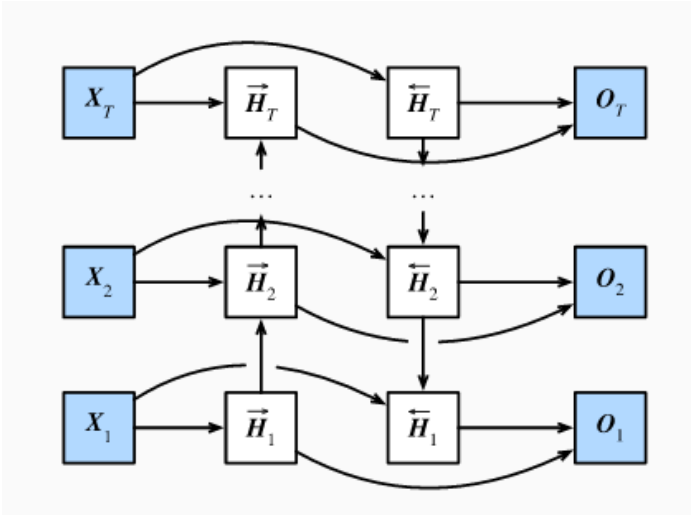
3. 将训练集分出20%作为验证集。
4. 基于训练集中文本的所有词构建词表，设置最低词频为10，以减小词表维度。将数据集文本中的单词替换为词表中单词索引。

2. 文本分类：使用双向循环神经网络

2.1 双向循环神经网络Bi-directional RNN

一般的循环神经网络模型假设当前时间步由较早时间步的序列决定，因此它们都将信息通过隐藏状态(hidden state)从前往后传递。但很多时候，当前时间步也可能由后面时间步决定。双向循环神经网络通过增加从后往前传递信息的隐藏层来更灵活地处理这类信息。

一个含单隐藏层的双向循环神经网络的架构:



含单隐藏层的双向循环神经网络

在本实验使用的模型中，每个词先通过嵌入层得到特征向量，然后BiLSTM对特征序列进一步编码得到序列信息。最后，我们将编码的序列信息通过全连接层变换为输出，即，将双向长短期记忆在最初时间步和最终时间步的隐藏状态连结，作为特征序列的表征，传递给输出层生成分类信息。

以下是BiLSTM的代码，默认层数=2：

```
class BiRNNText(nn.Module):
    def __init__(self, vocab_size, embedding_dim, output_dim, hidden_dim=64, num_layers=2, dropout=0.5):
        super(BiRNNText, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=num_layers, bidirectional=True,
                            batch_first=True, dropout=dropout)
                            # batch_first: input and output tensors are provided as (batch, seq, feature).
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, words):
        # (input) words : (batch_size, seq_len)
        embedded = self.dropout(self.embedding(words))

        # embedded : (batch_size, seq_len, embedding_dim)
        output, (hidden, cell) = self.lstm(embedded)
        # output: (batch_size, seq_len, hidden_dim * 2)
        # hidden / cell: (num_layers * 2, batch_size, hidden_dim)

        # dropout
        hidden = torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1)
        hidden = self.dropout(hidden)
        # hidden: (batch_size, hidden_dim * 2)

        pred = self.fc(hidden)
        # pred: (batch_size, output_dim)
        return {Const.OUTPUT:pred}
```

2.2 在小数据集上训练

数据集规模信息：

> 单词表: 4604条
> 训练集: 1798条
> 验证集: 449条

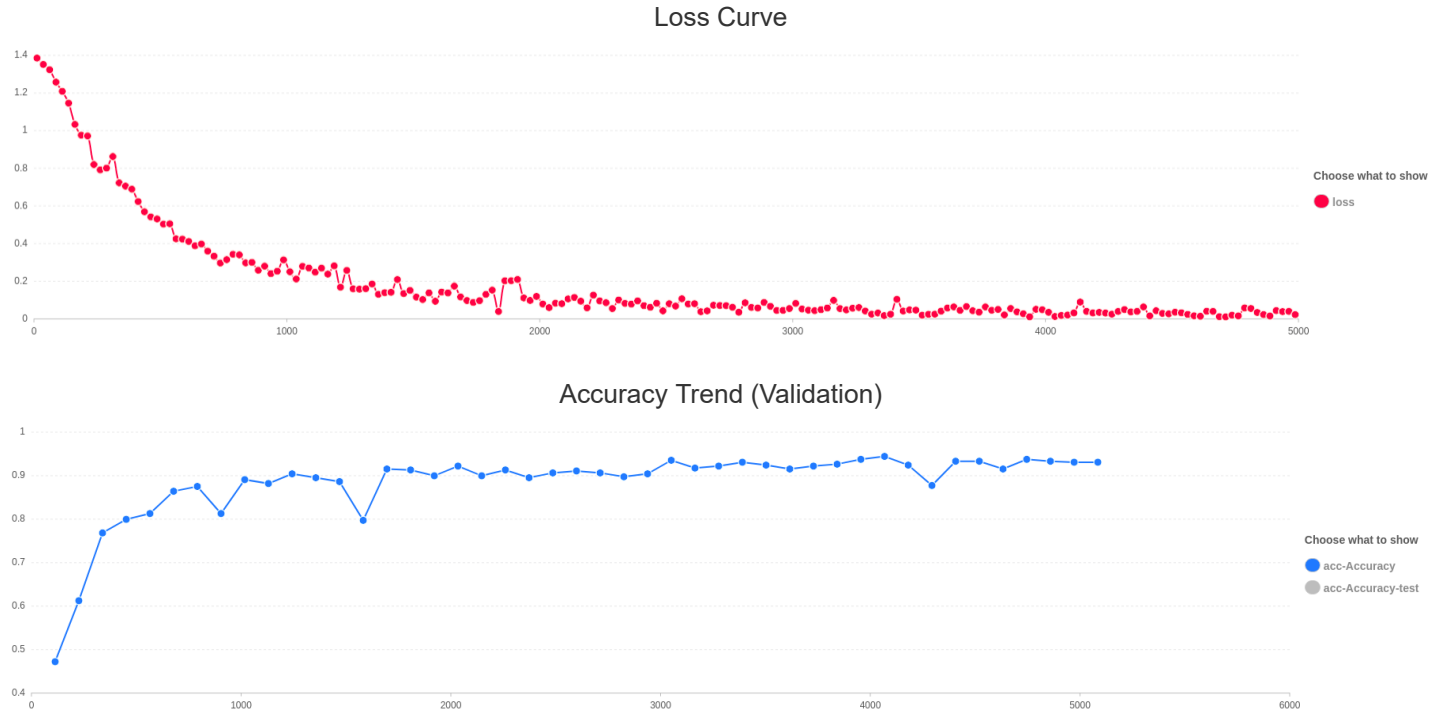
> 测试集: 1496条
> 类别数： 4类

对超参数——嵌入层维度和LSTM隐藏状态维度，根据模型在验证集上准确率，进行调参：

No.	Embedding dim	Hidden dim	batch size	Epochs	Accuracy(Validation)
1	128	64	16	8	0.819599
2	128	128	16	23	0.944321
3	128	256	16	27	0.942094
4	128	512	16	19	0.919822
5	256	256	16	19	0.926503

选择第2组超参数，训练和测试结果如下：

model	Embedding dim	Hidden dim	Batch size	Epochs	Accuracy(Validation)	Accuracy(Test)
BiLSTM	128	128	16	36	0.944321	0.904412



2.3 在更大的数据集上训练

数据集规模信息：

> 单词表: 13908条
> 训练集: 9052条
> 验证集: 2262条
> 测试集: 7532条
> 类别数: 20类

超参数与训练结果：

model	Embedding dim	Hidden dim	Batch size	Epochs	Accuracy(Validation)	Accuracy(Test)
BiLSTM	128	128	64	71	0.904951	0.797531

2.4 改进1：处理LSTM的序列输出：最大池化层与平均池化层

在之前的BiLSTM模型中，我们取最初时间步和最终时间步的隐藏状态输出到线性层生成分类信息，取得了不错的效果。但我们想探究，若通过一些方法处理LSTM的所有序列输出来整合特征，是否能提升模型的表现?

我们尝试了2种方法：最大池化层（max-pooling），平均池化层（average-pooling）。

改进部分代码如下:

```
    def forward(self, words):
        embedded = self.dropout(self.embedding(words))
        output, (hidden, cell) = self.lstm(embedded)
        # output: (batch_size, seq_len, hidden_dim * 2)
        out = output.permute(0, 2, 1)
        # max-pooling or average-pooling
        if self.pool_name == 'max':
            out = nn.functional.max_pool1d(input=out, kernel_size=out.size(2)).squeeze(2)
        elif self.pool_name == 'avg':
            out = nn.functional.avg_pool1d(input=out, kernel_size=out.size(2)).squeeze(2)
        else:
            raise Exception("Undefined pooling function: choose from: max, avg")

        out = self.dropout(out)
        # out: (batch_size, hidden_dim * 2)
        pred = self.fc(out)
        # pred: (batch_size, output_dim)
        return {Const.OUTPUT:pred}
```

超参数，训练、测试结果如下:

4分类小数据集上:

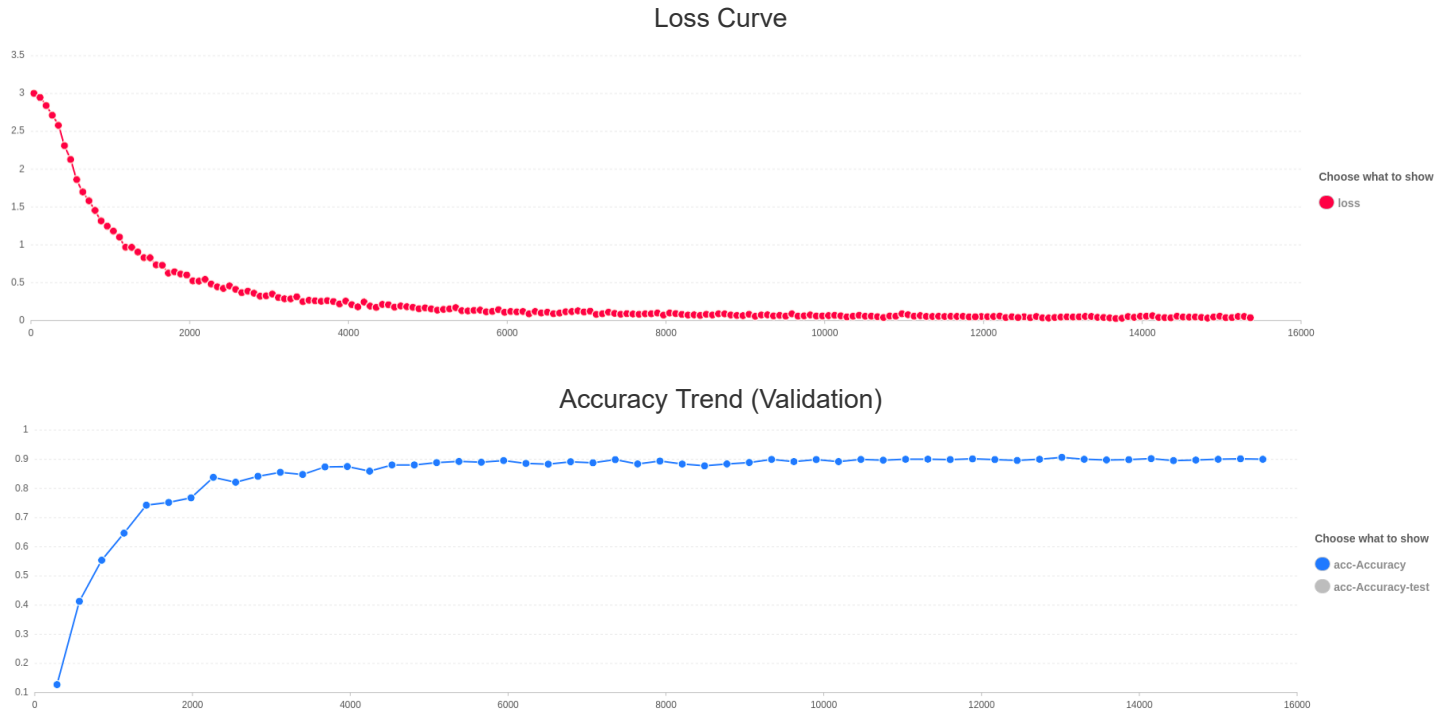
model	Embedding dim	Hidden dim	batch size	Epochs	Accuracy(Validation)	Accuracy(Test)
BiLSTM-maxpooling	128	128	32	30	0.96882	0.92246
BiLSTM-avgpooling	128	128	32	30	0.959911	0.919118

可见，在小数据集上，使用2种池化方式都能使模型在测试集上的准确率有一定提升。

20分类大数据集上：

model	Embedding dim	Hidden dim	batch size	Epochs	Accuracy(Validation)	Accuracy(Test)
BiLSTM-maxpooling	256	256	32	46	0.90672	0.804965
BiLSTM-avgpooling	256	256	32	36	0.876216	0.780802

BiLSTM-maxpooling在大数据集上的训练过程如下：



可见，在大数据集上，最大池化能使模型在测试集上准确率有少许提升，而使用平均池化的模型表现稍差，但它们都能使模型的收敛速度加快。

综合来说，通过池化处理LSTM所有序列输出来整合特征的方法，能提升模型的性能，且最大池化（max-pooling）的性能更好。这与使用CNN的一般经验一致。

2.5 其他的一些的尝试

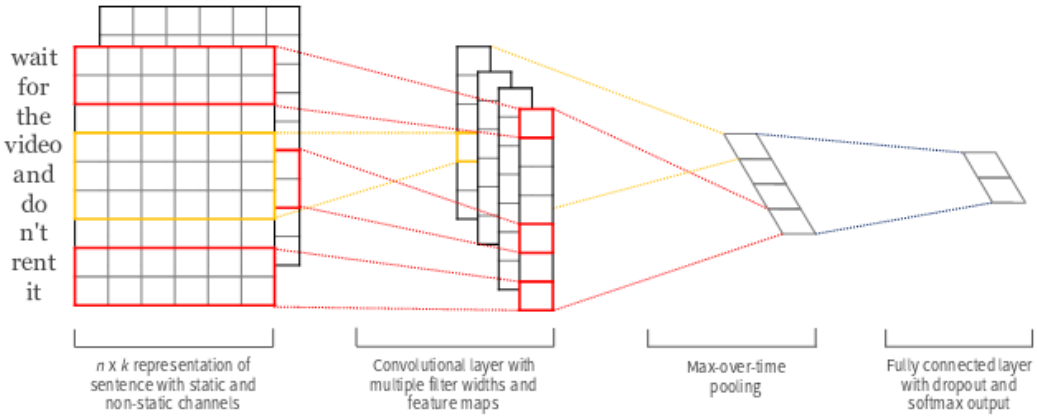
1. 在BiLSTM之后加入一个含有激活函数的全连接层。激活函数选用ReLU。对线性层输出做层归一化。
2. 使用预训练的GloVe。有几种方法：GloVe作为静态嵌入层，GloVe初始化动态嵌入层，2个GloVe在同一模型中分别作为静态嵌入层和动态嵌入层。

以上尝试未得到更加好的效果。代码均在model.py和model_glove.py中。

3. 文本分类：使用卷积神经网络（textCNN）

3.1 TextCNN

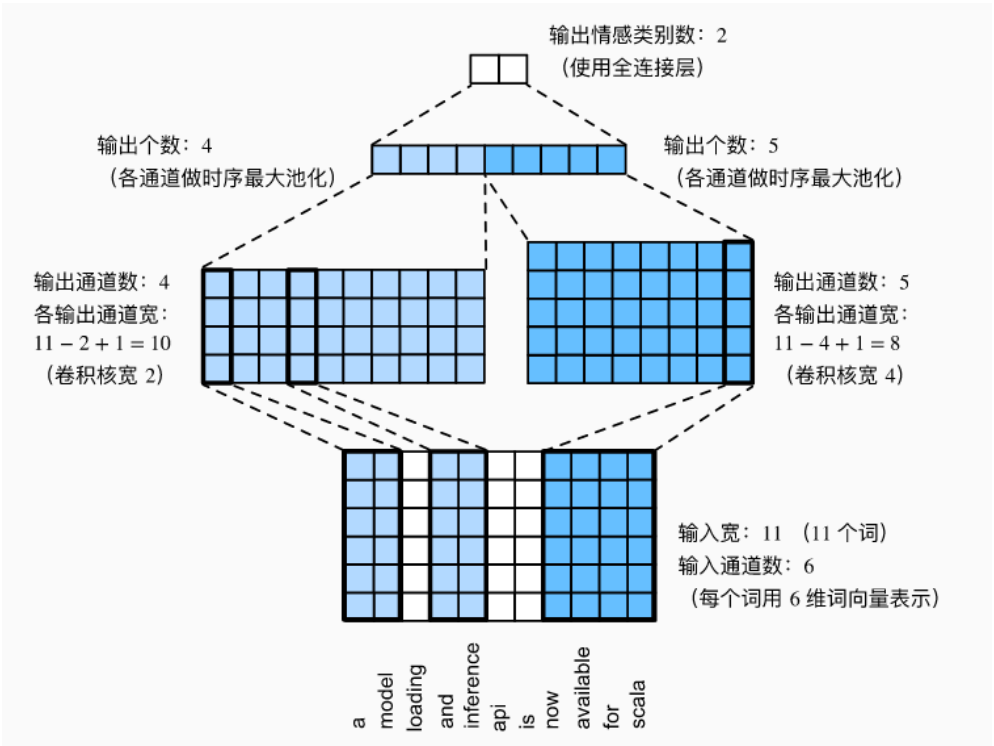
本次实验选用了2014年的文章“Convolutional Neural Networks for Sentence Classification”中提出的TextCNN模型。



TextCNN模型主要使用了一维卷积层和时序最大池化层。假设输入的文本序列由n个词组成，每个词用d维的词向量表示。那么输入样本的宽为n，高为1，输入通道数为d。TextCNN的计算主要分为以下几步:

> 1. 定义多个一维卷积核，并使用这些卷积核对输入分别做卷积计算。宽度不同的卷积核可能会捕捉到不同个数的相邻词的相关性。
> 2. 对输出的所有通道分别做时序最大池化，再将这些通道的池化输出值连结为向量。
> 3. 通过全连接层将连结后的向量变换为有关各类别的输出。这一步可以使用丢弃层(Dropout)应对过拟合。

下图是TextCNN模型应用于情感分类问题的一个样例:



TextCNN模型应用于情感分类问题

本实验中使用的TextCNN模型(参考了fastNLP中的CNN_Text):
（由于使用了一维最大池化层，所以TextCNN可以处理变长的序列输入）

```
class TextCNN(nn.Module):
    def __init__(self, vocab_size, embedding_dim, kernel_sizes, num_channels, num_classes, activation='relu',
                 dropout=0.5, bias=True, stride=1, padding=0, dilation=1, groups=1):
        super(TextCNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.dropout = nn.Dropout(dropout)
        # 创建多个一维卷积层
        self.convs = nn.ModuleList([nn.Conv1d(
                in_channels=embedding_dim,
                out_channels=oc,
                kernel_size=ks,
                stride=stride,
                padding=padding,
                dilation=dilation,
                groups=groups,
                bias=bias)
                for oc, ks in zip(num_channels, kernel_sizes)])
        # activation function
        if activation == 'relu':
            self.activation = nn.functional.relu
        elif activation == 'sigmoid':
            self.activation = nn.functional.sigmoid
        elif activation == 'tanh':
            self.activation = nn.functional.tanh
        else:
            raise Exception(
                "Undefined activation function: choose from: relu, tanh, sigmoid")

        self.fc = nn.Linear(sum(num_channels), num_classes)

    def forward(self, words):
        # (input) words : (batch_size, seq_len)
        embedded = self.dropout(self.embedding(words))
        embedded = embedded.permute(0, 2, 1)
        # embedded : (batch_size, embedding_dim, seq_len)

        # convolution
        xs = [self.activation(conv(embedded)) for conv in self.convs]
        # max-pooling
        xs = [nn.functional.max_pool1d(input=x, kernel_size=x.size(2)).squeeze(2)
              for x in xs]
        # x : (batch_size, num_channels)

        encoding = torch.cat(xs, dim=-1)

        # Dropout
        encoding = self.dropout(encoding)
        pred = self.fc(encoding)
        return {Const.OUTPUT:pred}
```

3.2 在小数据集上训练

数据集规模信息:

> 单词表: 4604条
> 训练集: 1798条
> 验证集: 449条
> 测试集: 1496条
> 类别数： 4类

初步确定部分超参数：

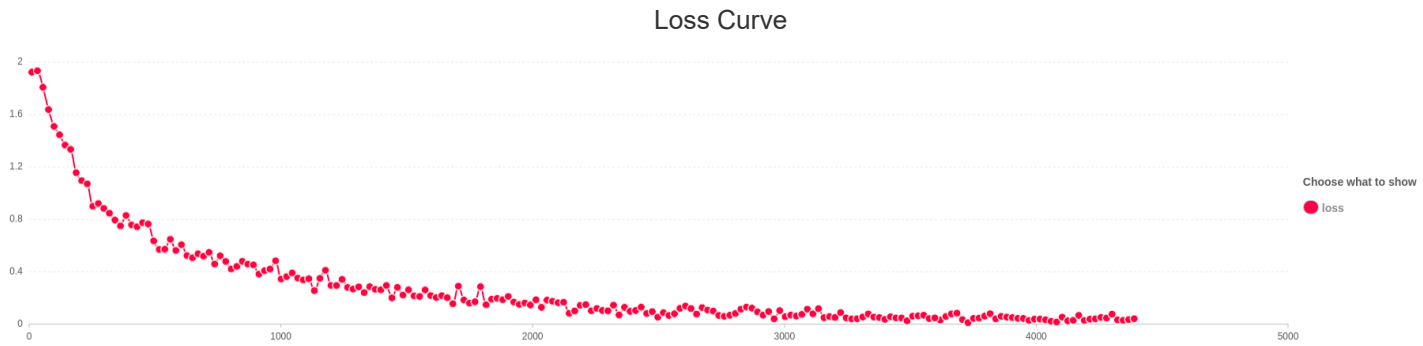
- kernel sizes = (3, 4, 5)
- batch size = 16

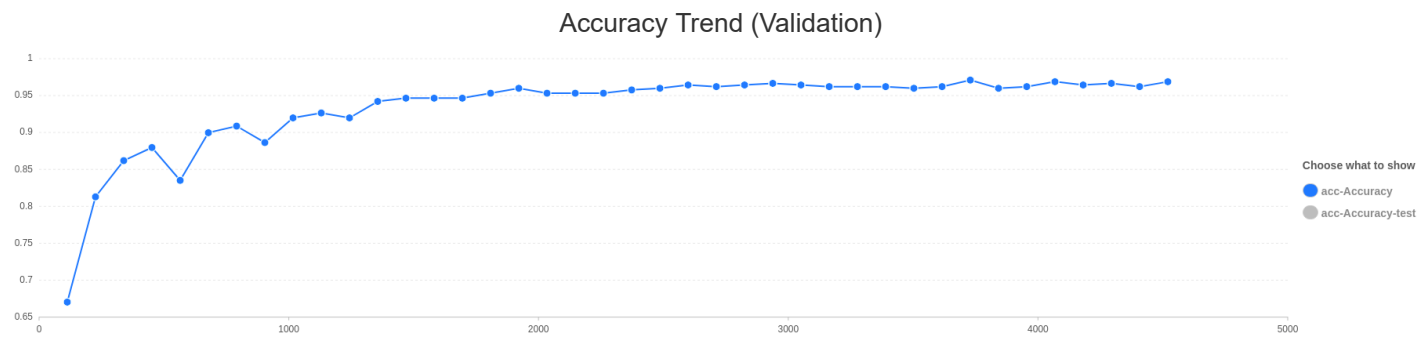
对剩下的超参数——词向量维度、输出通道数(卷积核数目)，进行调参。通过验证集准确率，确定最合适的超参数：

No.	Embedding_dim	Out channels	Activation Function	Epochs	Accuracy(Validation)
1	128	(32, 32, 32)	ReLU	46	0.964365
2	128	(48, 48, 48)	ReLU	35	0.971047
3	128	(64, 64, 64)	ReLU	24	0.962138
4	128	(100, 100, 100)	ReLU	33	0.96882
5	128	(48, 48, 48)	Sigmoid	35	0.959911
6	128	(48, 48, 48)	tanh	41	0.948775
7	256	(48, 48, 48)	ReLU	33	0.971047
8	512	(48, 48, 48)	ReLU	16	0.96882

选取第7组超参数，使用ReLU激活函数,测试结果如下

model	Embedding dim	Kernel sizes	Out channels	batch size	Epochs	Accuracy(Validation)	Accuracy(Test)
TextCNN	256	(3, 4, 5)	(48, 48, 48)	16	33	0.971047	0.940508





3.3 在更大的数据集上训练

数据集规模信息：

> 单词表: 13908条
> 训练集: 9052条
> 验证集: 2262条
> 测试集: 7532条
> 类别数: 20类

初步确定部分超参数：

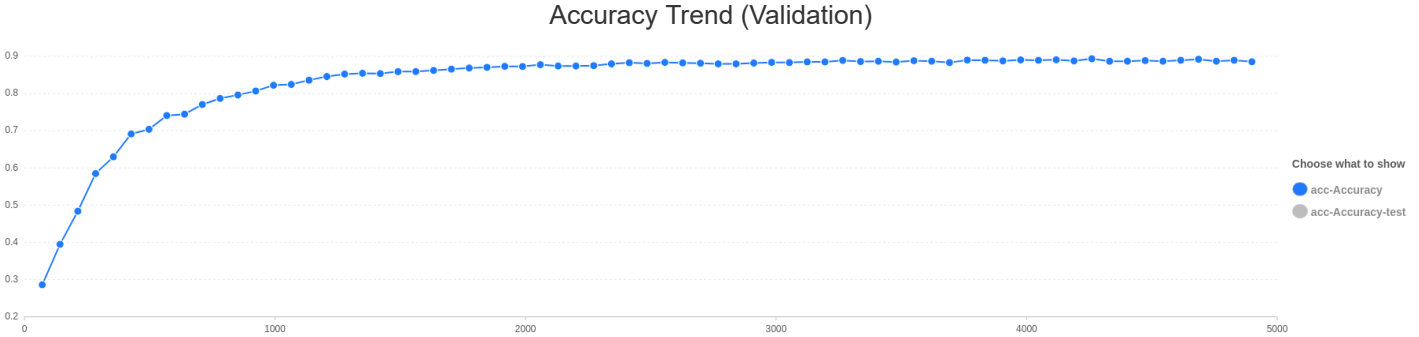
- embedding dim = 256
- kernel sizes = (3, 4, 5)
- batch size = 128

对输出通道数(卷积核数目)进行调参。通过验证集准确率，确定最合适的超参数：

No.	Out_channels	Epochs	Accuracy(Validation)
1	(48, 48, 48)	32	0.856322
2	(100, 100, 100)	34	0.861627
3	(128, 128, 128)	33	0.873563
4	(160, 160, 160)	38	0.885057
5	(120, 160, 200)	60	0.893457

选取第5组超参数，测试结果如下：

model	Embedding dim	Kernel sizes	Out channels	batch size	Epochs	Accuracy(Validation)	Accuracy(Test)
TextCNN	256	(3, 4, 5)	(120, 160, 200)	128	60	0.893457	0.788635



3.4 使用预训练GloVe词向量

在20分类的数据集上训练时，我们发现在训练结束时模型的交叉熵损失值已经很低（0.1以下），且在验证集上的准确率和测试集上的准确率相差较大。因此我们推测模型在一定程度上发生了过拟合。此外，模型的收敛速度也比较慢。

为防止过拟合并加快训练过程，我们使用在更大规模语料上预训练的词向量。本实验中使用的词向量是在2014年的英文维基百科上用GloVe训练得到的。(https://nlp.stanford.edu/projects/glove/)

为合理利用预训练词向量，在TextCNN模型中，我们将设置2个词嵌入层：embedding layer和constant embedding layer。它们都由根据GloVe词向量生成的词表Embedding初始化，但前者参与训练，而后者权重固定。

这样做的优点是：词嵌入层既保留了预训练词向量的信息，又有一部分能根据序列输入（真实文本信息），基于权重固定的词向量部分，对嵌入层进行微调（Fine-tuning）。

预训练词向量借助fastNLP.io的EmbedLoader读入，load_with_vocab可以在预训练词向量中抽取单词表中词的embedding。

```
embedding_file_path = "../glove.6B.100d.txt"
glove_embedding = EmbedLoader.load_with_vocab(embedding_file_path, vocab)
embedding_dim = glove_embedding.shape[1]

text_cnn_model = TextCNN_glove(vocab_size=len(vocab), embedding_dim=embedding_dim,
                        kernel_sizes=kernel_sizes, num_channels=num_channels, num_classes=cate_num)
text_cnn_model.embedding.load_state_dict({"weight":torch.from_numpy(glove_embedding)})
text_cnn_model.constant_embedding.load_state_dict({"weight":torch.from_numpy(glove_embedding)})
text_cnn_model.constant_embedding.weight.requires_grad = False
text_cnn_model.embedding.weight.requires_grad = True
```

预训练词向量选用glove.6B.100d.txt，超参数，训练、测试结果：

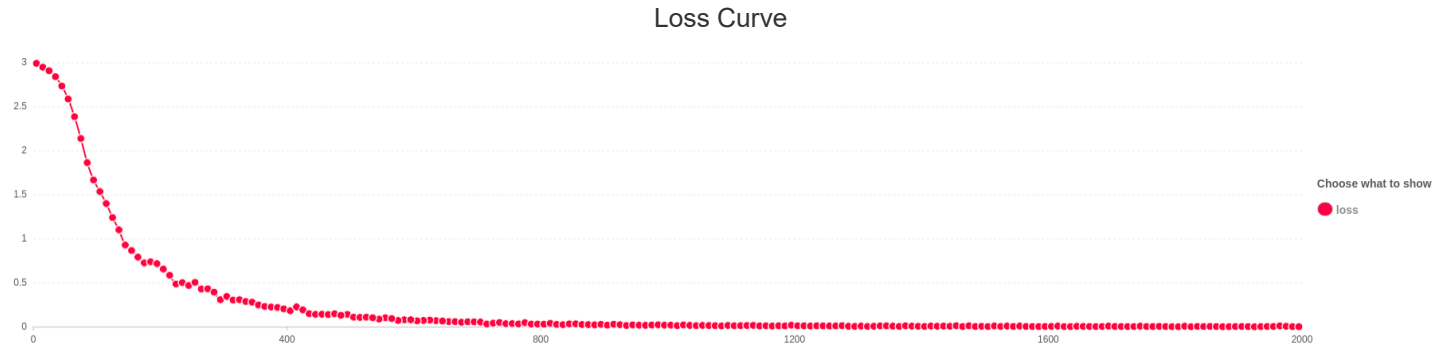
在4分类小数据集上：

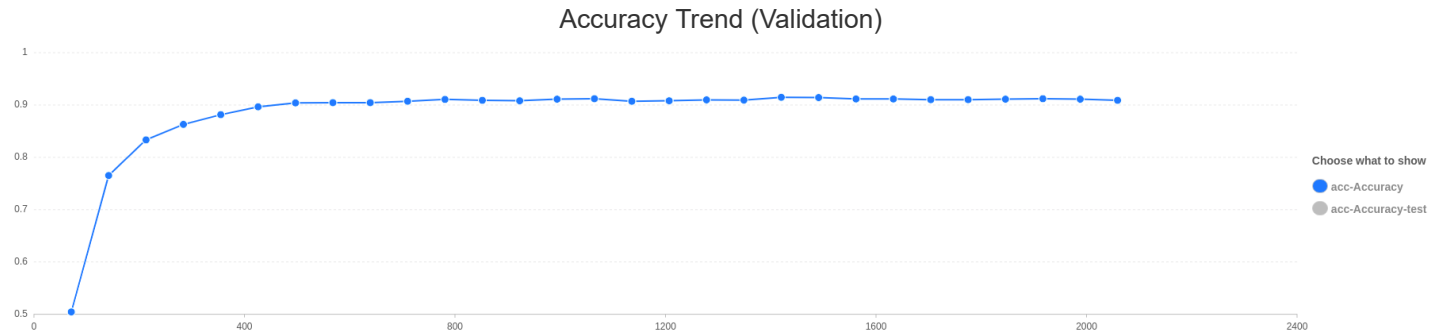
model	Embedding dim	Kernel sizes	Out channels	batch size	Epochs	Accuracy(Validation)	Accuracy(Test)
TextCNN-GloVe	100	(3, 4, 5)	(48, 48, 48)	16	8	0.986637	0.965241

在20分类大数据集上：

model	Embedding dim	Kernel sizes	Out channels	batch size	Epochs	Accuracy(Validation)	Accuracy(Test)
TextCNN-GloVe	100	(3, 4, 5)	(120, 160, 200)	128	20	0.914677	0.842539

在20分类大数据集上的训练过程:





可以发现，使用GloVe后，模型的收敛速度更快，且在测试集上准确率有明显提高。

4. 模型性能汇总与比较

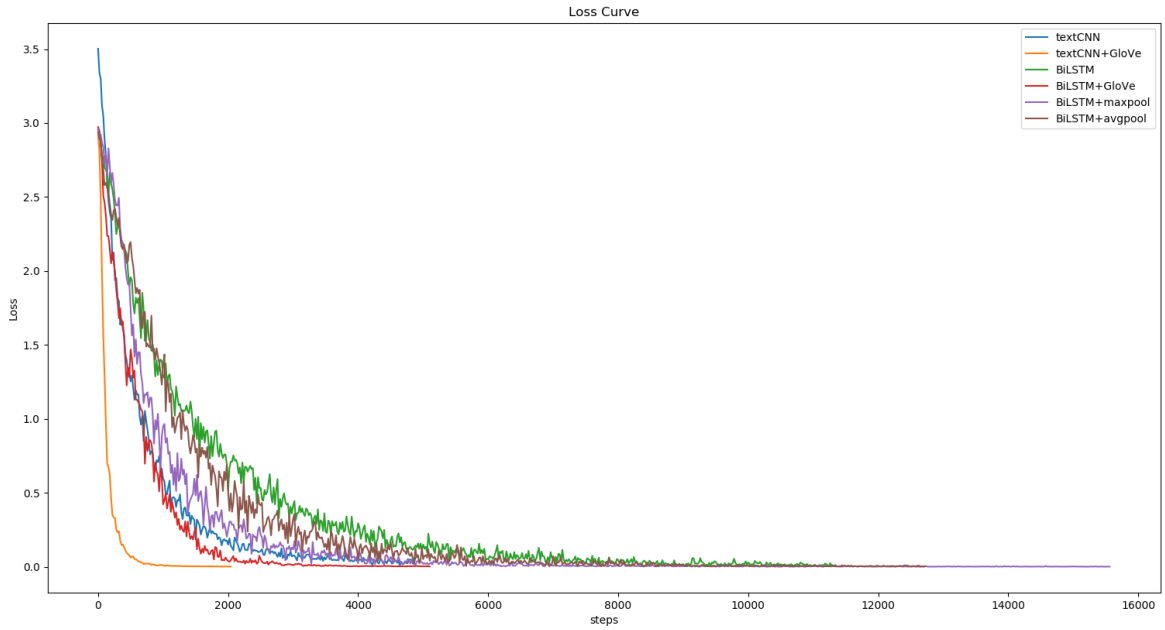
在fetch_20newsgroups的4分类小数据集上：

model	Epochs	Accuracy(Validation)	Accuracy(Test)
BiLSTM	36	0.944321	0.904412
BiLSTM-maxpooling	30	0.96882	0.92246
BiLSTM-avgpooling	30	0.959911	0.919118
TextCNN	33	0.971047	0.940508
TextCNN-GloVe	8	0.986637	0.965241

在fetch_20newsgroups的20分类大数据集上：

model	Epochs	Accuracy(Validation)	Accuracy(Test)
BiLSTM	71	0.904951	0.797531
BiLSTM-GloVe	27	0.875774	0.743229
BiLSTM-maxpooling	46	0.90672	0.804965
BiLSTM-avgpooling	36	0.876216	0.780802
TextCNN	60	0.893457	0.788635
TextCNN-GloVe	20	0.914677	0.842539

在20分类大数据集上，训练过程中各模型的损失下降曲线如下：



综合来看，使用GloVe预训练词向量的TextCNN模型在测试集上的准确率最高，且收敛速度明显快于实验中的其他模型。

总结与思考：

1. TextCNN的核心是利用卷积核捕捉文本序列的局部相关性，类似于N-Gram模型，然后使用池化整合特征信息。相比与RNN模型，CNN处理一批序列的计算速度更快，效率更高。但TextCNN的超参数数量更加多，如果要调节所有超参数，调参工作量还是很大。
   此外，TextCNN的卷积核尺寸(kernel size)有限，如果要提取长序列信息，只靠卷积层后的池化可能还不够。
2. RNN能更好地表达上下文信息，而Bi-directional LSTM通过加入双向的信息传递，加强了上下文信息前后关联的表达。实验证明，对LSTM在每个时间步上的输出进行整合，能提高LSTM模型的性能。若要在此基础上更进一步，则可引入Attention机制。
   LSTM模型在较小的数据集中容易过拟合，在调参时要注意控制隐藏层的规模(hidden dimension)。如果过拟合现象仍很严重，可以尝试将LSTM单元替换为GRU单元。
   由于要逐时间步处理序列输入，且参数数量很多，Bi-LSTM模型的训练速度很慢，所以不要轻易尝试较大的隐藏层(hidden dimension)，2倍的hidden dimension一般意味着2倍的训练时间。
3. 预训练词向量一般能加快模型的收敛速度，但不一定能提升模型在测试集上的表现。如果要使用预训练词向量来提升模型性能，我们还要关注数据集的处理方式。我们要根据预训练中对数据集的处理方式，对我们任务中的数据集进行处理，这样才能使训练数据集的语料接近预训练数据集的语料。
   在使用预训练词向量的时候，一般要在训练模型的过程中对词嵌入层（embedding）做微调。由于训练数据集与预训练数据集词信息分布不同，如果直接使用静态（固定）的词嵌入层，效果一般不好。

Part 2 fastNLP的体验与建议

在实验3中我便尝试使用了fastNLP 0.3.3。那时候fastNLP的文档内容很少，教程内容比较单薄，所以我使用Trainer训练模型时，为了弥补Trainer及其他核心模块功能的不足，只能通过阅读源代码来了解如何写各个接口的派生类及重载函数。

在这次实验中使用的fastNLP 0.4.1的文档内容更加丰富，各个核心工具也增加和改进了许多接口，用户和开发者使用体验大幅上升。

总结一下fastNLP的使用经验：

1. DataSet和Vocabulary的接口简洁易用。我们能简洁地将Batch和Pad应用于Dataset，且在Trainer中Batch和Pad功能是高度自动化的(如果不追求特殊形式的pad)。

在DataSet中，我们可以将某个field同时设置为input和target,因此如果我们需要自行计算loss，我们可以在模型类的forward函数中由input获取到target的信息，从而计算loss，并且可以将其通过模型的字典输出传给Trainer。这就可以在不用定义losses派生类的情况下设计自己的loss计算方式。（对metrics同理，可在predict函数中计算，可惜在API文档的fastNLP.core.const说明中没看见metrics对应的字段是什么）

2. fastNLP的Trainer模块集成了其他核心模块的功能，实现了模型训练的快捷接口，可以在训练结束时重新载入在验证集上表现最好的模型。更好的是，我们还可以通过Callback类增强Trainer的功能，而且在Trainer的callbacks参数可以传入Callback类的列表，这为用户和开发者都提供了极大的便利。
   fastNLP本身已经提供了EarlyStop，GradientClip等Callback派生类。我们可以定义自己的派生类，在训练过程中特定阶段加入一些操作。
   此外，Fitlog是一个功能简洁的训练结果记录和训练过程可视化工具。借助Fitlog，用户可查看动态的loss图和metrics图。训练结果（最佳模型的验证集准确率和测试集准确率）会被自动记录在表格中，表格的memo可以写入备注（比如记录调参信息）。这种数据可视化功能和模型性能对比功能，能节省调参过程的工作量。
3. fastNLP封装了很多NLP中常用的工具。比如NLP任务中经常需要用到外部的预训练词向量，而fastNLP.io的EmbedLoader可以从外部文件中读入预训练词向量，且拥有2种读入方式——基于现有词表抽取词向量，读取词向量并生成一个对应的单词表。EmbedLoader还可以自动判断外部文件是word2vec格式还是glove格式的数据。

对fastNLP的建议：

1. 希望为Vocabulary添加快捷的保存功能。在fastNLP中，DataSet有保存功能，但Vocabulary没有保存功能。在重新运行程序，加载之前保存的模型继续训练时，我们肯定要加载之前训练时使用的数据集和词表（重新在数据集上生成的词表的键值映射会与之前的不一致）。虽然自己写一个词表保存和载入功能很快（在提交的代码中有实现），但是还是希望在框架中加入这种必需的功能。
2. 在使用Trainer训练时，loss可视化只能通过Fitlog快速实现。但是我认为Fitlog的定位是训练数据记录工具和调参助手，有时为了制作用于报告和论文的图表，我们还是需要用不同的方式对训练过程的loss进行数据处理和可视化分析。然而我们缺乏在训练后直接获取训练loss list的手段，只能在Callback中的on_backward_begin函数中得到loss，或者在Fitlog的loss.log文件中找到训练中每一步的信息。希望能在Trainer的初始化参数中加入返回loss的选项并提供方便的获取loss list的接口。
   （同理，对metrics等其他训练数据，也有相同的需求）
3. pad是fastNLP的一项重要功能，主要关系到训练时的批量输入（Batch）。但在fastNLP的自动化训练机制中，在计算loss和计算metrics时，批量输入中的padding与别的普通字符“别无二致”——例如在实验3唐诗生成模型的训练过程中，如果使用Trainer进行训练，模型预测错了target中的padding，也是要算入loss中以及算入Perplexity中的。这会影响模型参数更新的方向（梯度下降的方向）和对模型性能量化评价的正确性。
   在目前版本下，除了之前提到过的在模型的predict函数中自行计算loss的方法，我自己想到的补救方法：创建Callback派生类，在on_loss_begin(batch_y, predict_y)函数中将predict_y中的padding位置对应的batch_y中的位置的值修改为padding，使得模型不会因为预测错padding而造成loss增大。
   但Callback的on_valid_begin函数没有提供batch_y和predict_y的参数，因此比较难校准验证集metrics的计算（除非我们直接在模型的predict函数中直接计算metrics并通过字典输出传给trainer）
   所以我希望fastNLP能补充训练时对padding的处理（主要是计算loss和metrics），这可以通过加强fastNLP.core的losses模块和metrics模块的功能来实现。
4. 对第3点建议的补充：在批量序列输入中，padding使长短不一的序列变为长度一致。在用RNN模型处理批量序列输入时，若序列输入中含有大量的padding，则会使RNN模型进行很多不必要的计算。我希望fastNLP中padding能更好地处理不等长序列输入，以提高RNN模型的性能。

附录

文件说明

1. model.py: 本次实验使用的模型
2. model_glove.py: 使用预训练词向量的模型
3. handle_data.py: 处理fetch_20newsgroups数据集，在当前路径下生成词表，训练集，验证集，测试集文件

4. main.py: 选择数据集，模型，超参数，进行训练和测试（上交的版本里去除了Fitlog相关的部分；使用加载GloVe的模型时，请将对应的预训练词向量文件放在当前路径下）

```
usage: main.py [-h] [--method {cnn,cnn_glove,rnn,rnn_maxpool,rnn_avgpool}] [--dataset {1,2}]

optional arguments:
  --method {cnn,cnn_glove,rnn,rnn_maxpool,rnn_avgpool}      train model and test it
  --dataset {1,2}       1: small dataset; 2: big dataset
```

参考文献

1. fastNLP 中文文档
2. Pennington, Jeffrey, Richard Socher, and Christopher Manning. "Glove: Global vectors for word representation." Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014.
3. Kim, Yoon. "Convolutional neural networks for sentence classification." arXiv preprint arXiv:1408.5882 (2014).
4. Liu, Pengfei, Xipeng Qiu, and Xuanjing Huang. "Recurrent neural network for text classification with multi-task learning." arXiv preprint arXiv:1605.05101 (2016).
5. "Dive into Deep Learning": https://d2l.ai/index.html
6. "GloVe": https://nlp.stanford.edu/projects/glove/