

情感分析实验报告

计91 刘松铭 2018011960

摘要

本实验实现了CNN、CNN + Attention、CNN + Inception等CNN系列的模型。此外还实现了RNN模型，以及Baseline模型——FastText。Baseline模型在测试集上的准确率达到**0.250**。CNN + Inception模型表现最突出，在测试集上的准确率达到**0.611**，其余模型的准确率在0.6左右。

使用说明

根目录为 `TextClassification`，亦即本实验的项目文件夹。项目文件夹说明和程序运行导引详见 `/README.md`。

注1：本实验的实验环境如下：

```
# 系统版本
Distributor ID: Ubuntu
Description:    Ubuntu 18.04.5 LTS
Release:        18.04
Codename:       bionic

# GPU信息
NVIDIA-SMI 460.73.01
Driver Version: 460.73.01
CUDA Version: 11.2
GeForce RTX 2080

# Python版本
Python 3.7.9 (default, Aug 31 2020, 12:42:55)
[GCC 7.3.0] :: Anaconda, Inc. on linux

# PyTorch版本
1.7.1
```

注2：本实验的测试都是在GPU上完成的，尽管设置相同的随机种子，如果运行脚本时是在**CPU**上运行，也有可能**无法完全复现实验结果**（和本实验报告的结果有些许出入）。如果运行脚本时也在**GPU**上运行，也有可能因为GPU版本等原因**无法完全复现实验结果**，有关这方面的讨论可以参考PyTorch官方解释以及[这篇文章](#)。

数据分析及预处理

本实验采用**V1**数据集。数据分析及预处理是在做分类任务前的必须准备工作。正所谓“好的开始是成功的一半”，这第一步往往对分类结果的好坏有决定性的影响。这部分的工作大致可以分为以下步骤：

- 数据分析
 - 这一步是在处理数据前对数据有一个直观的感受。而这一步的结果往往也会揭示数据处理的重心与方向。
 - 首先是对数据进行宏观分析，了解数据的分布情况。
 - 接着是选取部分数据进行微观分析，了解数据的细节与特点。
- 数据清洗
 - 对数据进行一定的清理，修复异常值，补足缺失值。
- 数据提取

- 提取清理后的文本中的词汇，形成词汇表（词汇表包含训练集、测试集、验证集中出现过的词汇，这是因为测试集和验证集中有训练集没有的词汇，如果仅用训练集进行提取可能会无法量化测试集和验证集）。
- 读取词汇表，量化文本，形成向量集。

数据分析

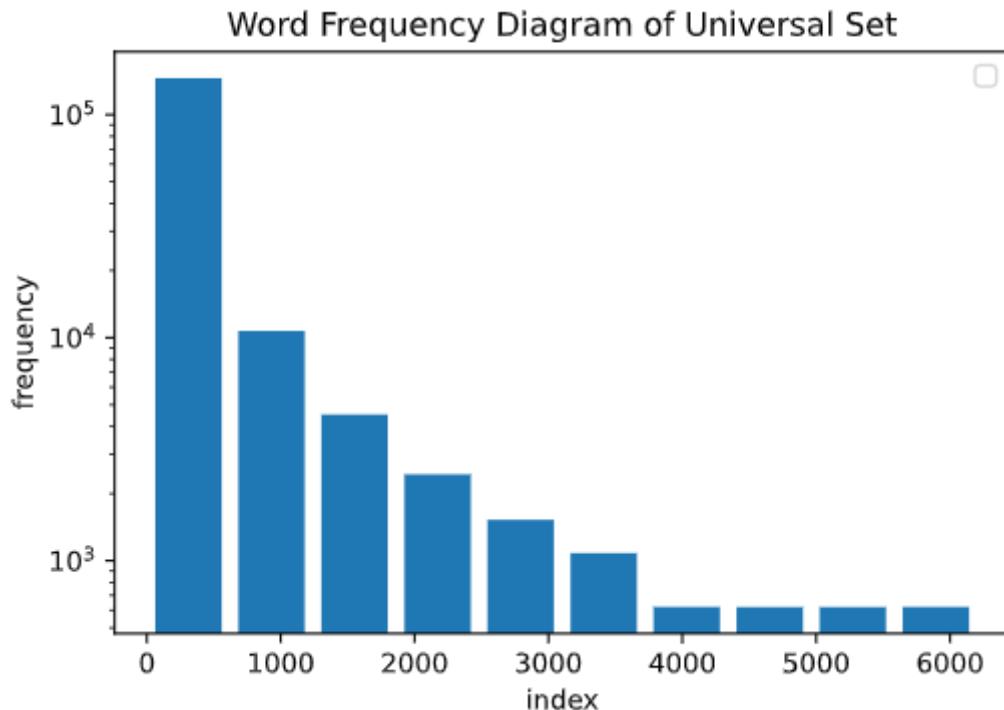
- 宏观分析
 - 数据集词汇总数

	全集	训练集	测试集	验证集
词汇总数	6204	4934	2966	2939

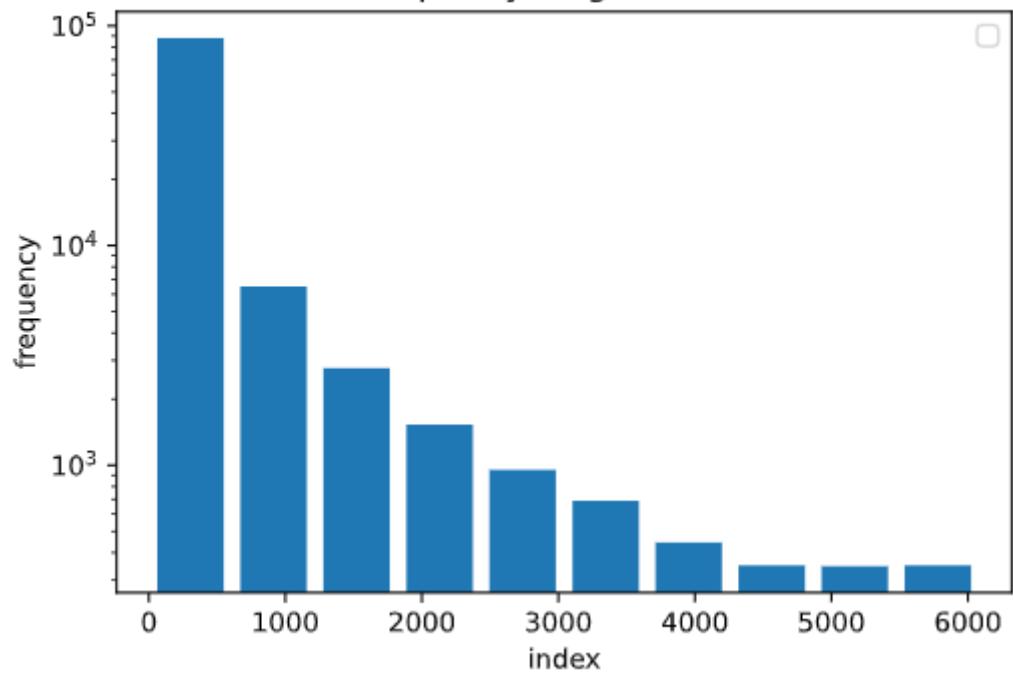
从中可以发现训练集有很多**全集没有**的词汇。

- 词频分布

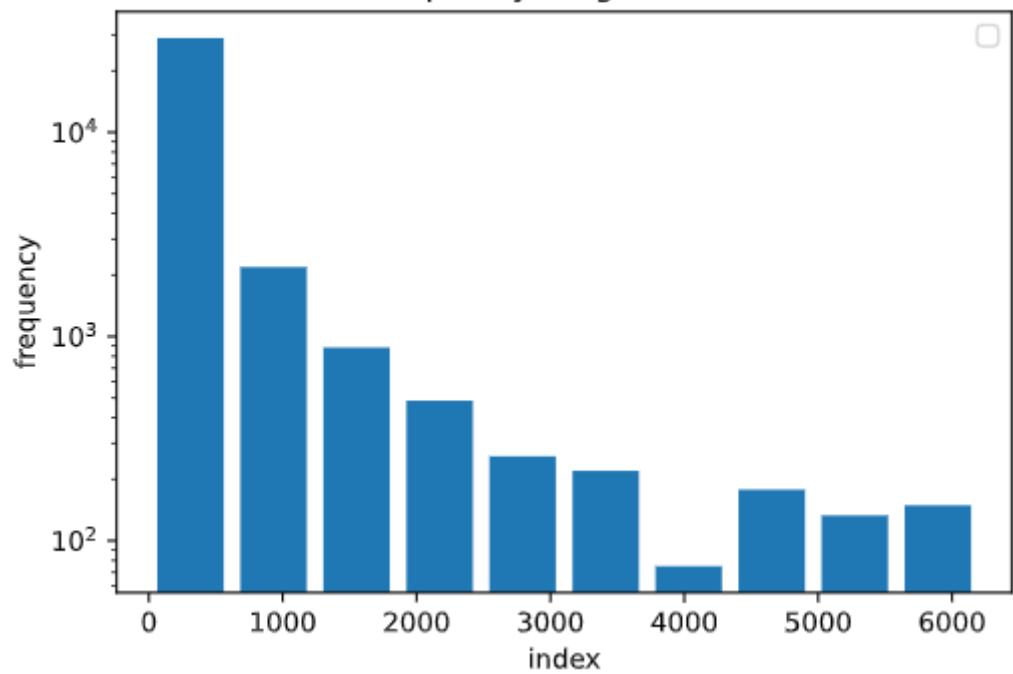
以下是全集，训练集，测试集，验证集的词频分布直方图：



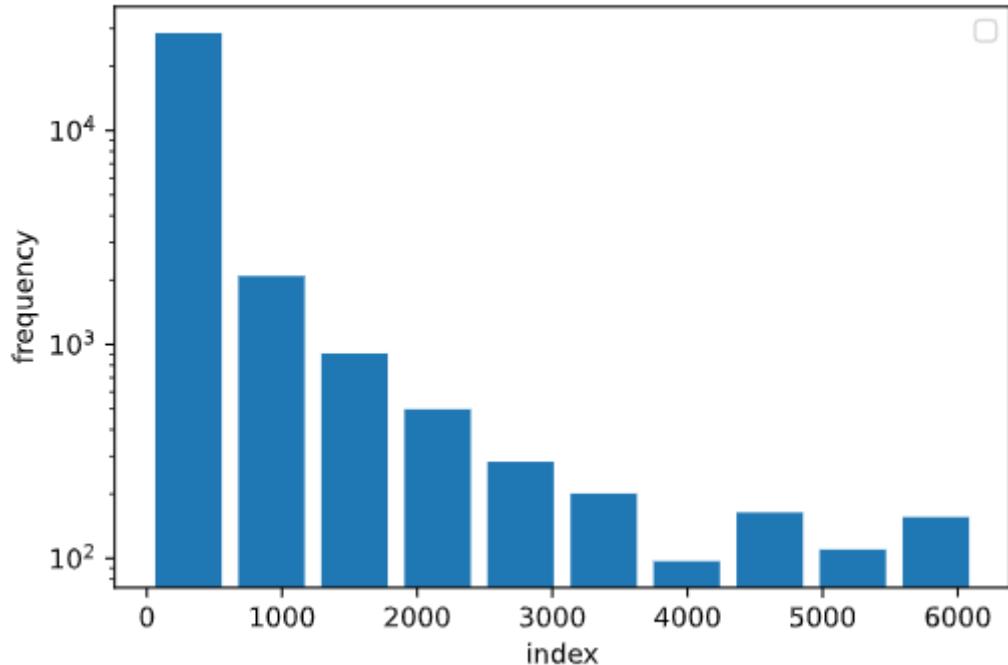
Word Frequency Diagram of Train Set



Word Frequency Diagram of Test Set



Word Frequency Diagram of Validation Set

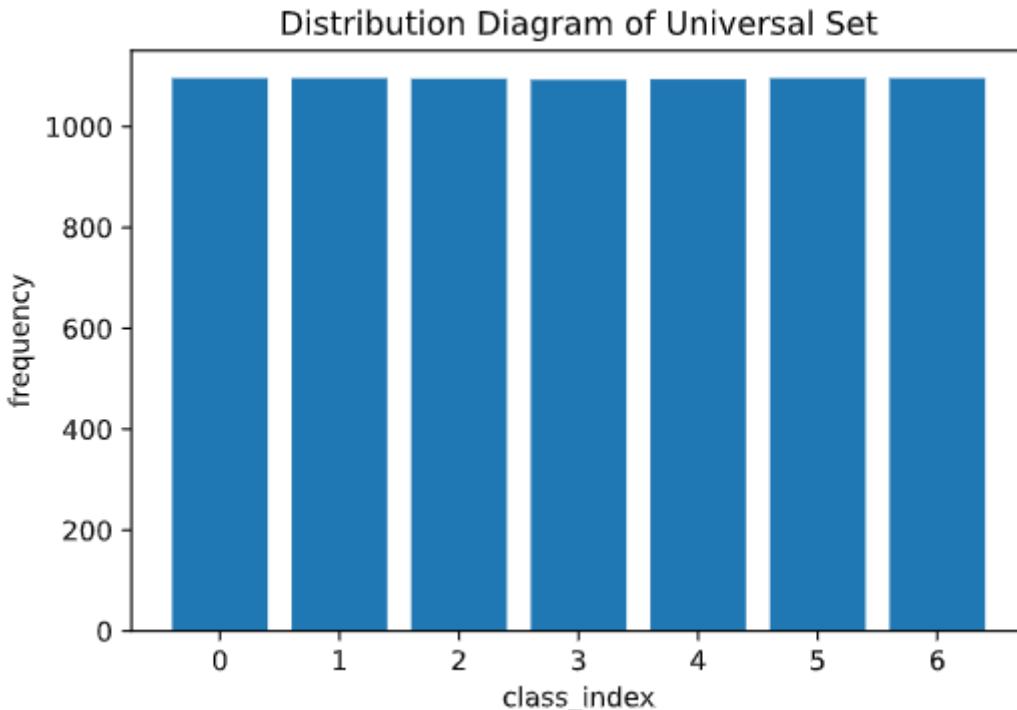


从中可以看出，全集，训练集，测试集，验证集的词频分布在 index 小的部分相似。在 index 大的部分，测试集和验证集相似。此外，从对数坐标轴可以看出，三者的词频分布都不是很均匀。

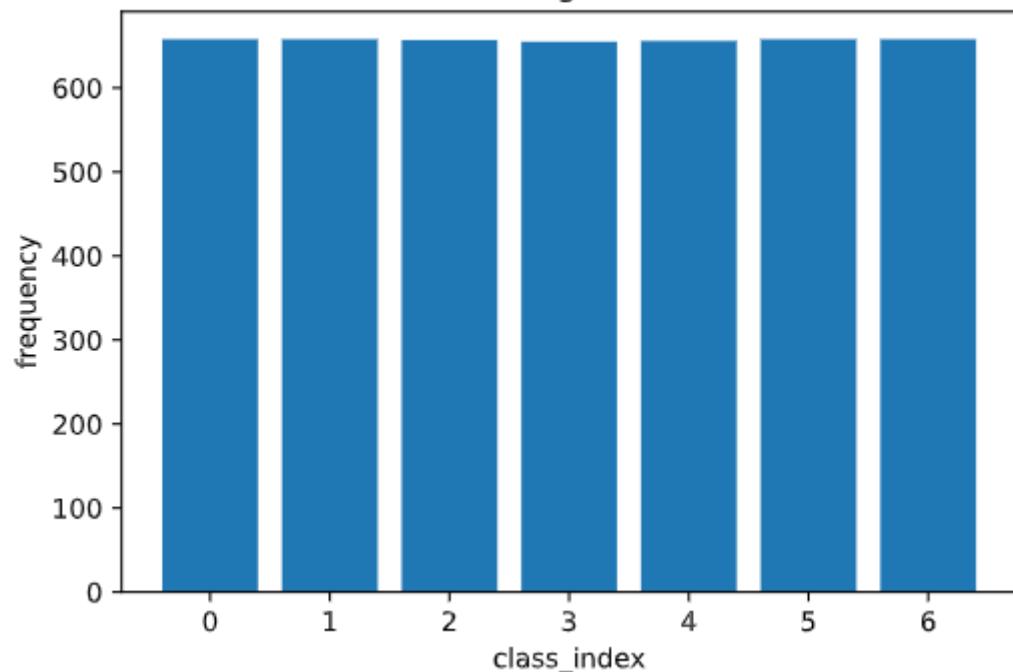
注：frequency 指词汇出现的频数。index 指词汇在词汇表中的编号。词汇表文件是 /dataset/vocab_b.txt。由于编号方式是按照全集中的词频由高到低编号的，因此图表有明显的随 x 轴单调减趋势。

- 句子类别分布

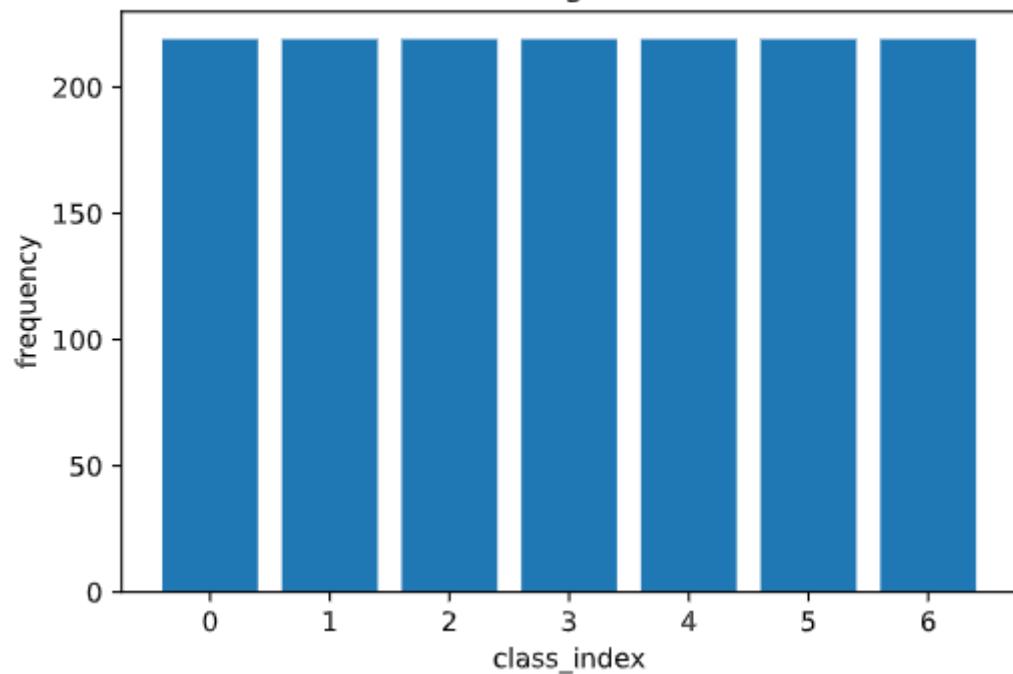
以下是全集，训练集，测试集，验证集的句子类别分布条形图：

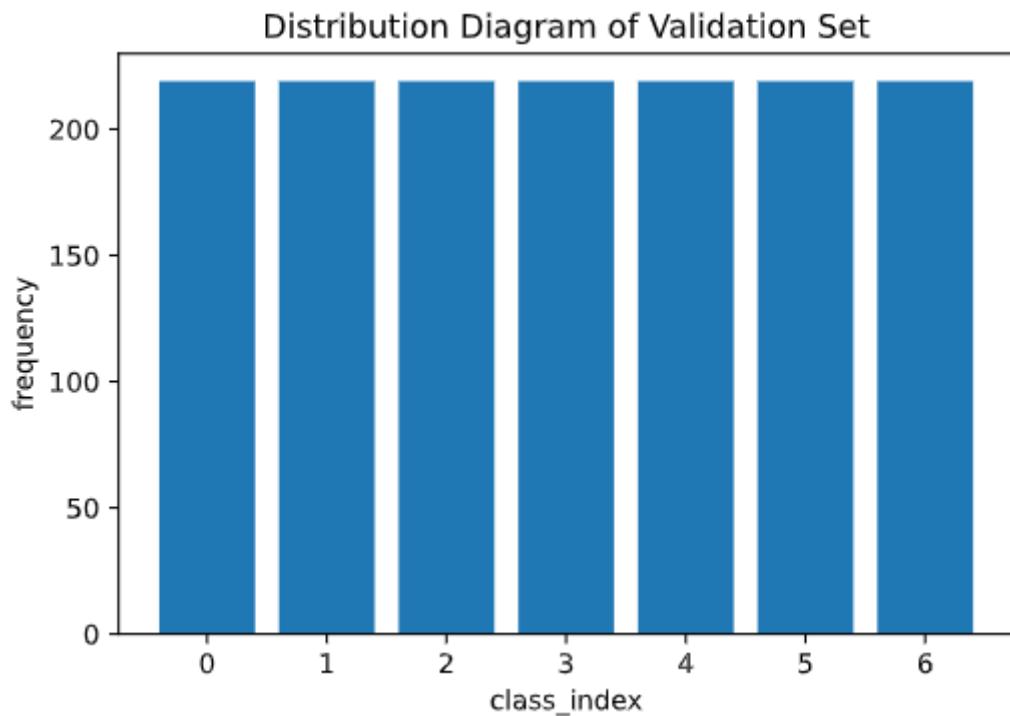


Distribution Diagram of Train Set



Distribution Diagram of Test Set



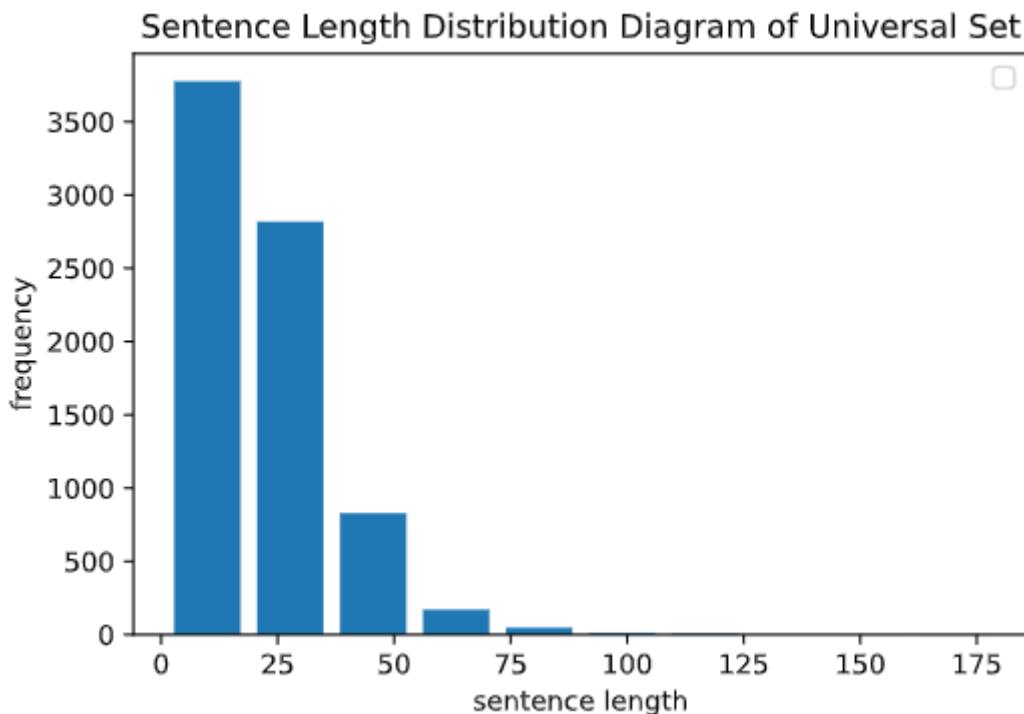


共有**7666**条数据。从中可以看出本实验的数据集类别分布比较均匀，不需要做额外的数据增强。

注：`frequency` 指类别出现的频数。`class_index` 指类别编号，0-6 分别指 `['anger', 'disgust', 'fear', 'guilt', 'joy', 'sadness', 'shame']`。

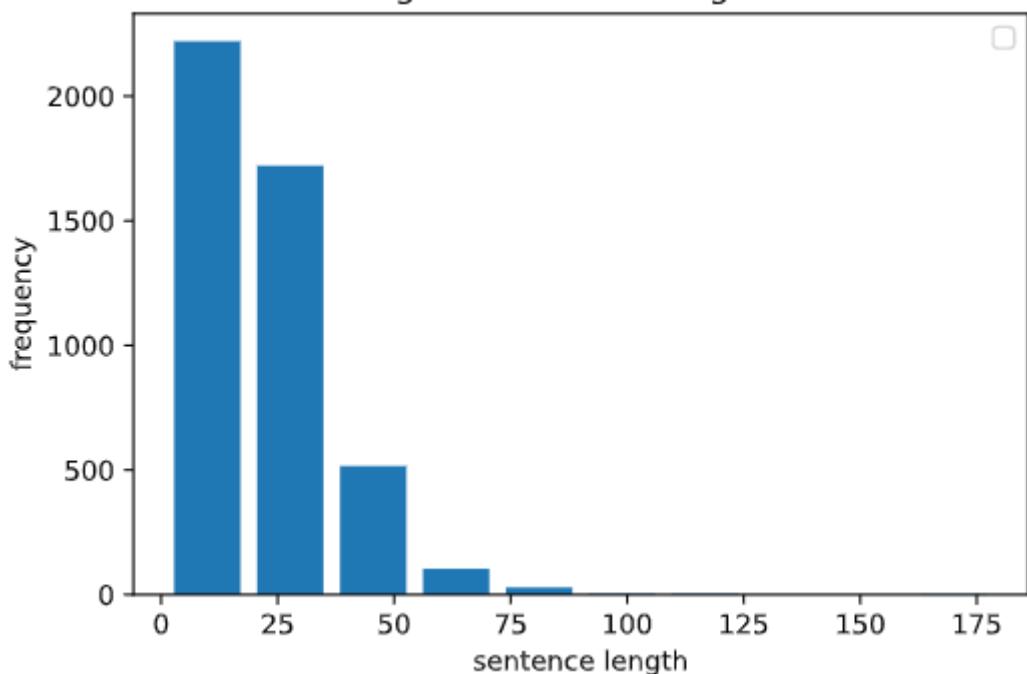
- 句子长度分布

以下是全集，训练集，测试集，验证集的句子长度分布条形图：



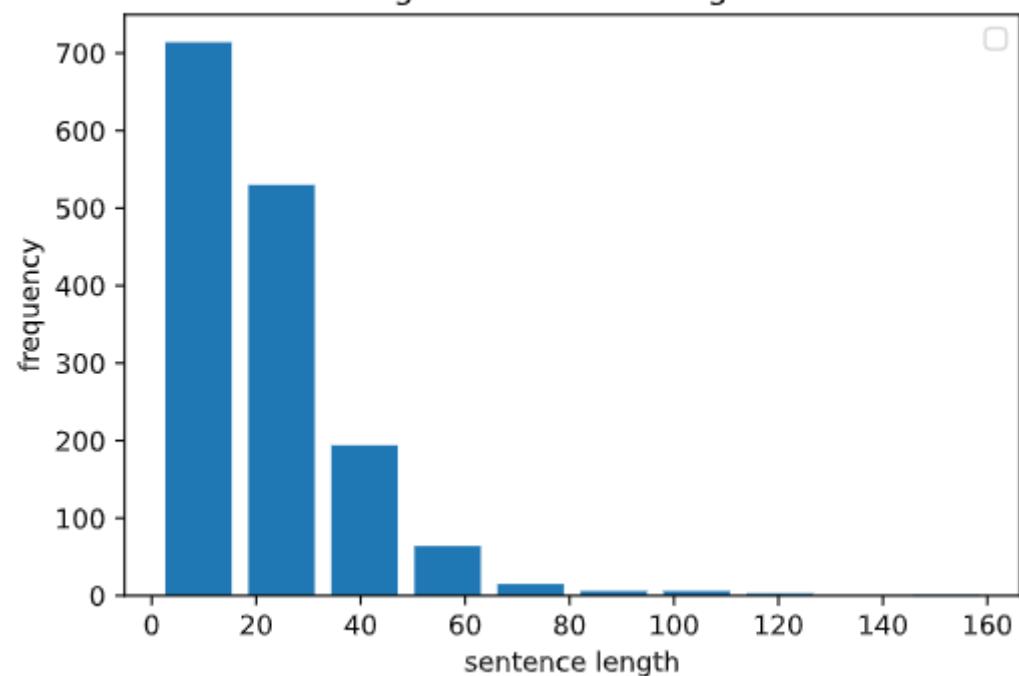
最大句子长度**179**

Sentence Length Distribution Diagram of Train Set



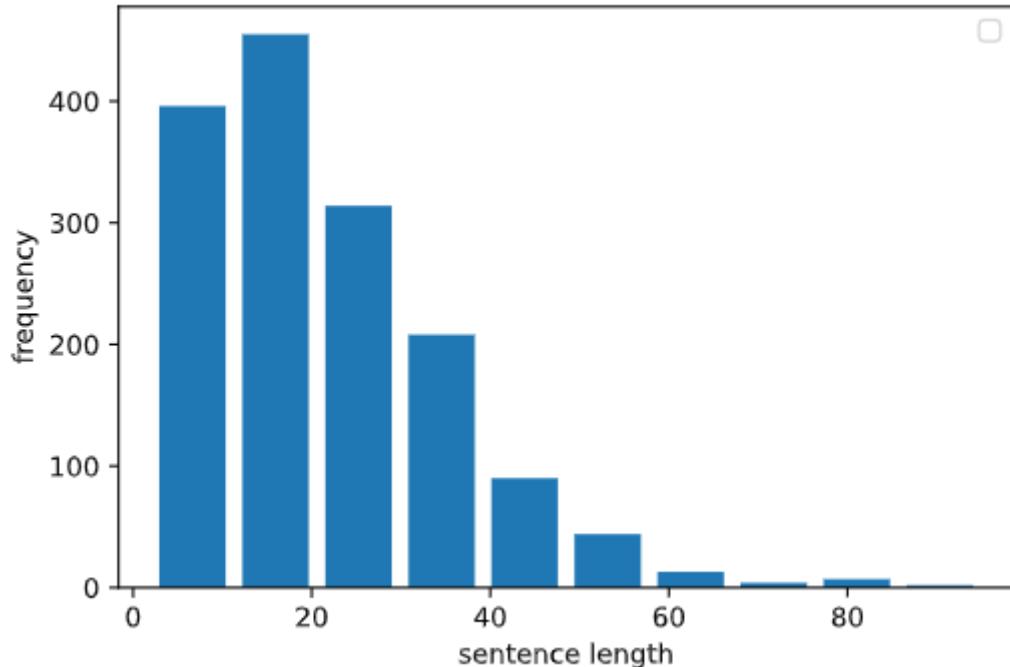
最大句子长度179

Sentence Length Distribution Diagram of Test Set



最大句子长度160

Sentence Length Distribution Diagram of Validation Set



最大句子长度95

从图中可以看出，句子长度的分布很不均匀。比如说全集，绝大部分的句子长度都小于**50**，而最长的句子的长度为**179**，如果直接按照最长的长度进行padding，那么绝大多数的句子向量将会有超过**72%**的内容都是缺失值。

注：`frequency` 指该句子长度出现的频数。

- 微观分析

从数据集中选取部分句子进行分析，列举如下。

1. when i left after the examin to enter the univers and even though i had studi the whole year i made a bad exam label: anger
2. when i fail an exam label: disgust
3. fail an examn label: sadness
4. fail an examn label: shame
5. over an argument label: disgust
6. no respons label: guilt
7. no respons label: joy
8. no respons label: sadness
9. last year when i work dure the summer holidai and studi at the same time for an examin i had to repeat my boyfriend went to greec for a holidai for a month label: sadness
10. do not know label: shame
11. do not know label: shame
12. can t rememb have had thi feel label: disgust
13. be insult in public label: shame

分析以上句子，可以得到数据集存在如下的特点：

- 存在比较多的错误单词

我们发现数据集中相当多的句子都有错误单词，例如 `examin`, `univers`, `holidai` 等。并且许多错误词恰好是关键词，也就是这些错误是有可能影响语义理解的。

- 标签的不确定

第6,7,8个句子的内容**完全一样**，而label却各不相同。

第2,3,4个句子说的是同一件事，而label也各不相同。

类似的情况还有很多，说明label标注本身具有一定的不可靠性。

- 语法错误

第13个句子，跟在 `be` 之后应该是 `insulted` 而不是 `insult`。类似情况还有很多。

- 重复的句子

第10个句子多次出现，属于重复出现的句子。类似情况还有很多。

- 无法正确表示缩写

第12个句子中的 `can t` 应该是 `can't`，但是去掉标点后就变成了不正确的缩写。类似情况还有很多。

数据清洗

- 具体操作

1. 针对错误单词进行修改

这里使用python库[TextBlob](#)进行错误单词的修改。TextBlob不仅可以将错词替换为最接近的词，还可以根据上下文智能选词。

2. 去除重复的文本

重复的文本无论是用来训练还是测试都是不太有裨益的，因此可以去除那些重复的文本。

3. 修复错误的缩写

将文本中的错误缩写替换为正确形式，错误缩写形式列举如下：

```
'i m',
'you re',
'she s',
'he s',
'we re',
'they re',
'it s',
'i ll',
'you ll',
'she ll',
'he ll',
'we ll',
'they ll',
'it ll',
'won t',
'isn t',
'aren t',
'ain t',
'i ve',
'you ve',
'we ve',
'they ve',
'haven t',
'hasn t',
'what s',
'who s',
'where s',
'where re',
'don t',
'doesn t',
```

```
'can t',
'that s',
'let s',
'wasn t',
'weren t',
'couldn t',
'wouldn t',
'shouldn t',
'mustn t',
'mightn t',
'needn t',
'mayn t',
'shan t',
'daren t',
'didn t',
'hadn t'
```

所谓替换，其实是指展开，例如将 `wasn t` 替换为 `was not`。这部分直接手动替换即可。此外，`cannot` 之类的缩写也可展开为 `can not`，这是为了规范统一，减少神经网络的学习量。

4. 关于停用词

一般来说，忽略停用词可以提高模型性能，使模型尽可能聚焦于关键词。但是在该问题上最好不要忽略停用词，一个是该分类问题对性能要求不高，不像信息检索；另一个是文本在去掉停用词后语义会发生改变。

例如：

`not` 在停用词表中，而对于如下的句子

everi time i do not write someth well in english label: shame

如果去掉 `not`，则变为

everi time i do write someth well in english label: shame

这样一来句子的意思就完全改变了，和标签不再匹配。因此最好不要去掉停用词。

由此可以发现：预处理的操作是受任务类型影响的。对于情感分析的任务，关键词固然重要，而上下文有时也同样不可或缺。

- 数据清理后的统计结果

- 数据集词汇总数

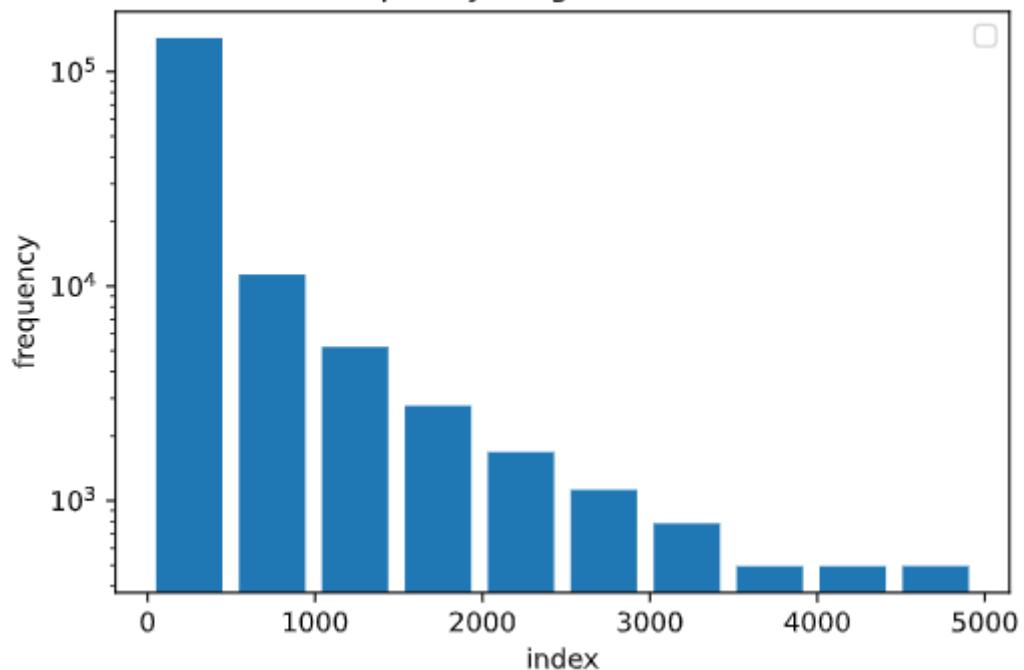
	全集	训练集	测试集	验证集
清洗前词汇总数	6204	4934	2966	2939
清洗后词汇总数	4954	4106	2626	2601
词汇总数变化	-1250	-828	-340	-338

数据清洗后词汇总数下降了，说明有些错词被修复后转化为了出现过的正确词汇。

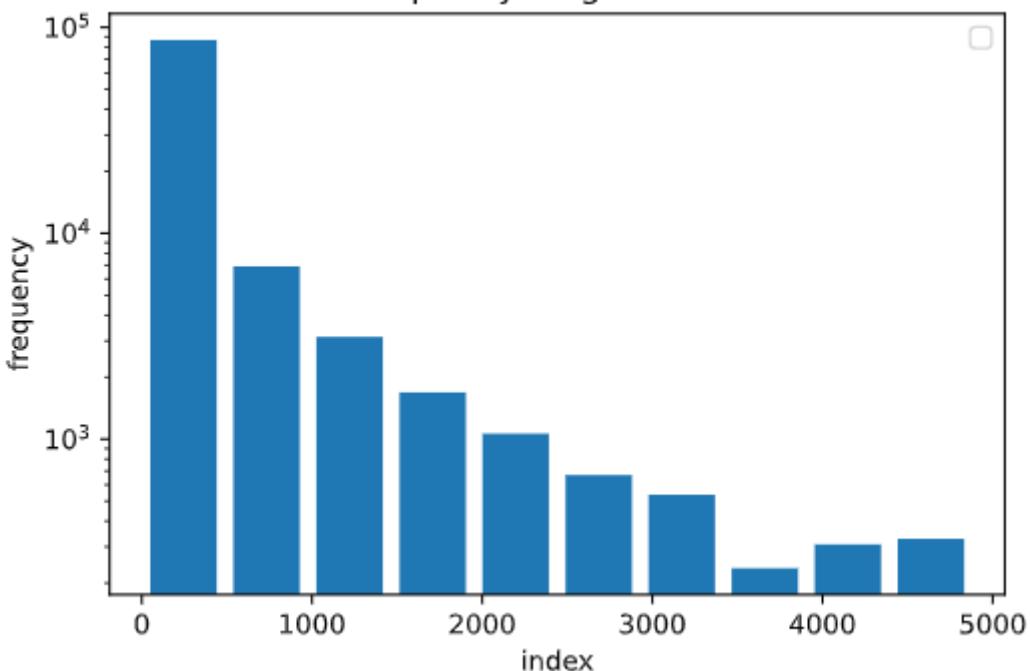
- 词频分布

以下是全集，训练集，测试集，验证集的词频分布直方图：

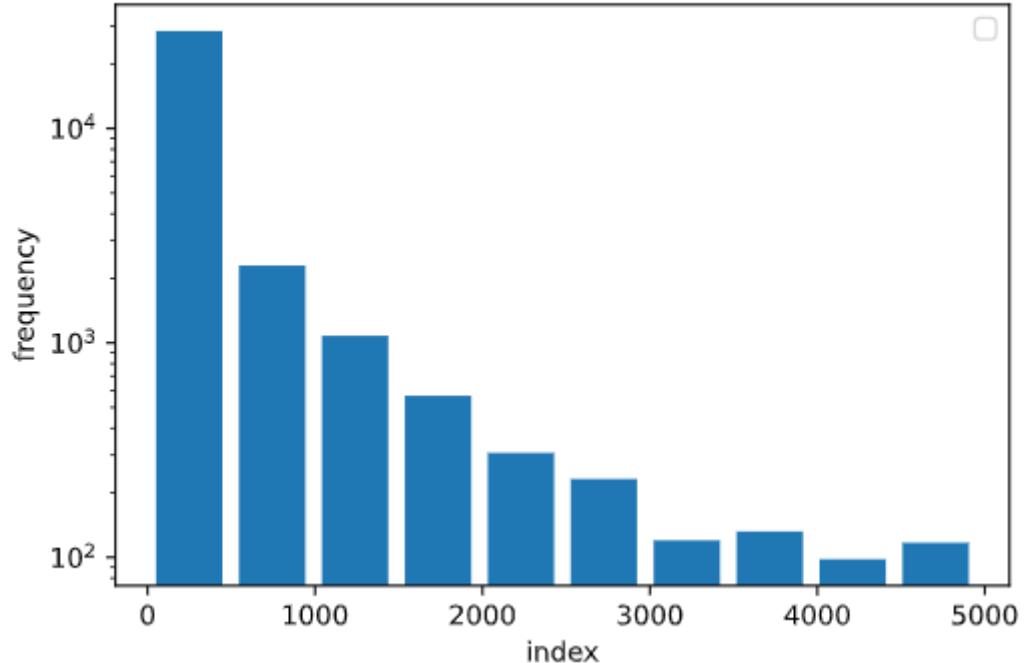
Word Frequency Diagram of Universal Set



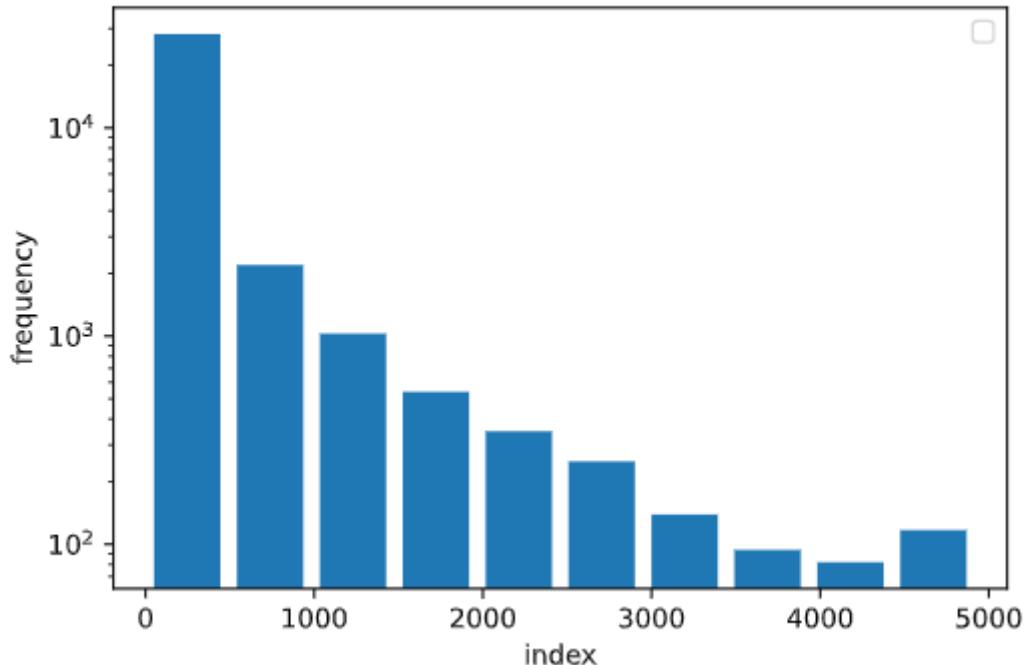
Word Frequency Diagram of Train Set



Word Frequency Diagram of Test Set



Word Frequency Diagram of Validation Set



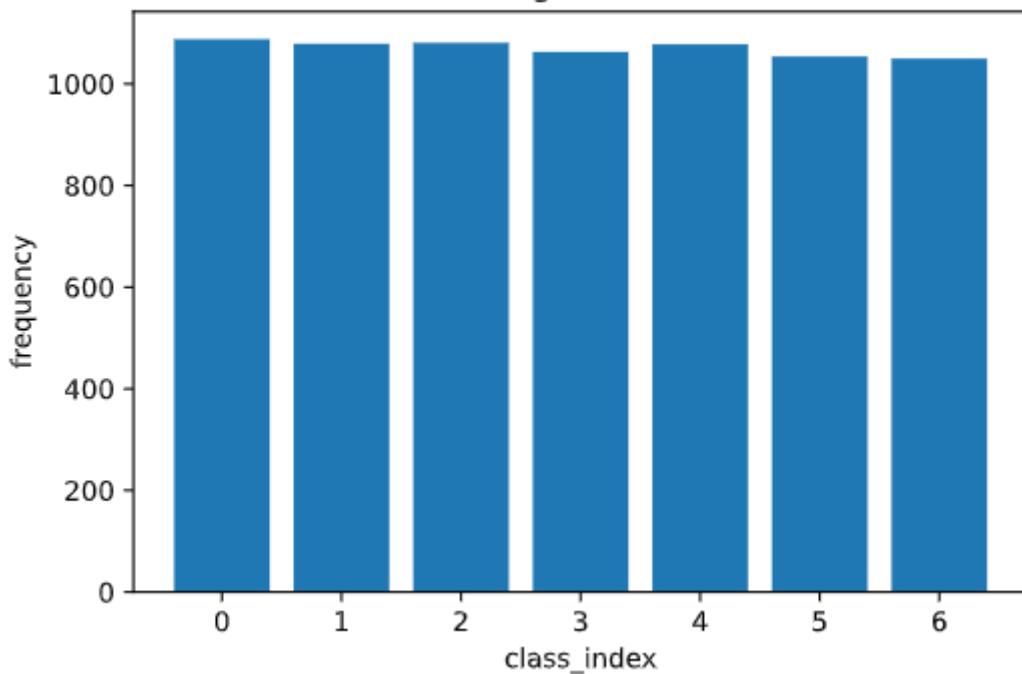
从中可以看出，在数据清洗之后，词频分布变得更加均匀了(指均匀递减)。

注：`frequency` 指词汇出现的频数。`index` 指词汇在词汇表中的编号(清洗后)。词汇表文件是`/dataset/vocab_f.txt`。由于编号方式是按照全集中的词频由高到低编号的，因此图表有明显的随x轴单调减趋势。

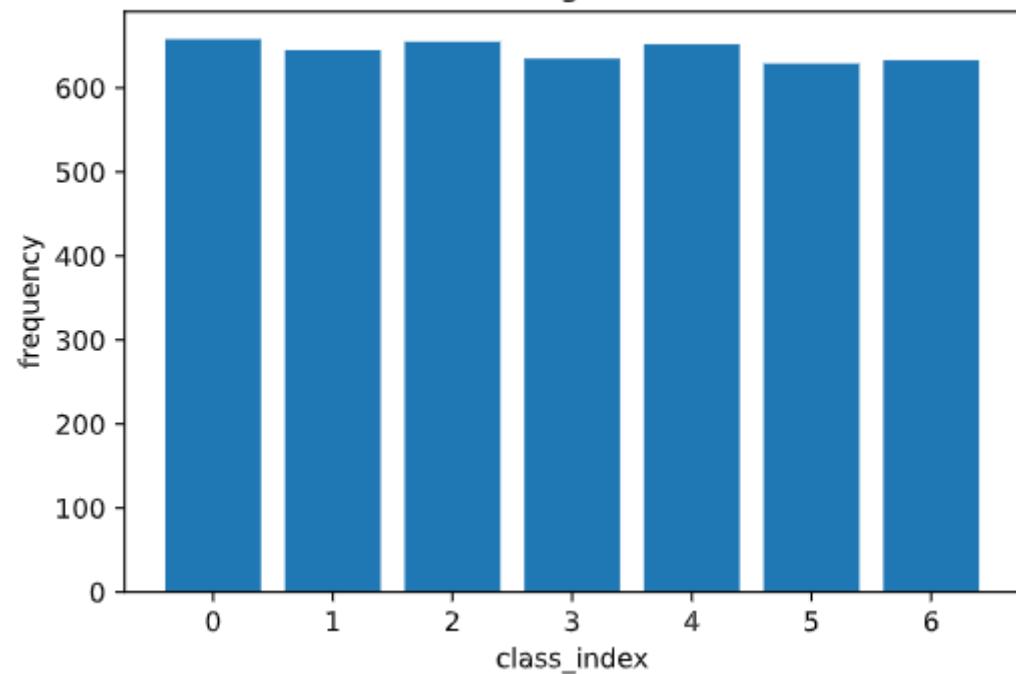
- o 句子类别分布

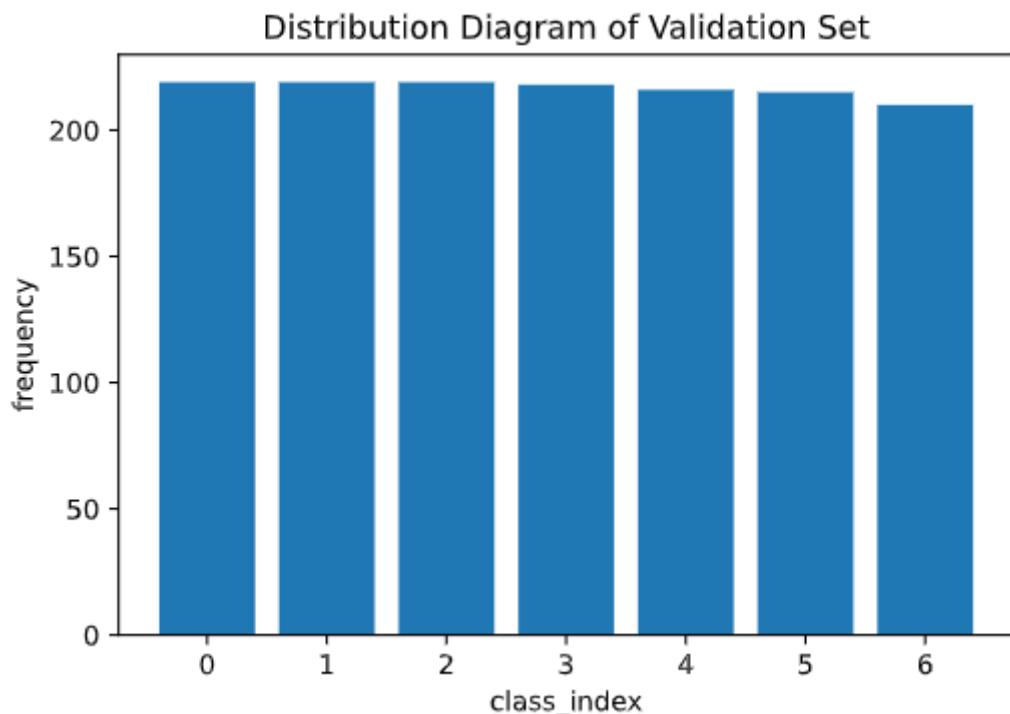
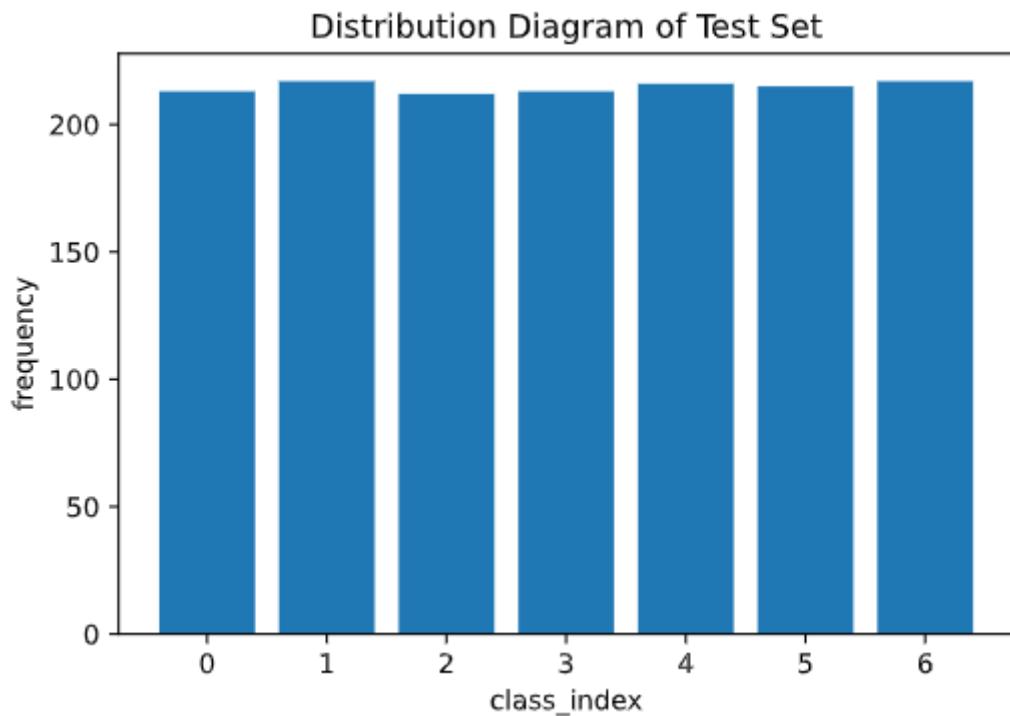
以下是全集，训练集，测试集，验证集的句子类别分布条形图：

Distribution Diagram of Universal Set



Distribution Diagram of Train Set





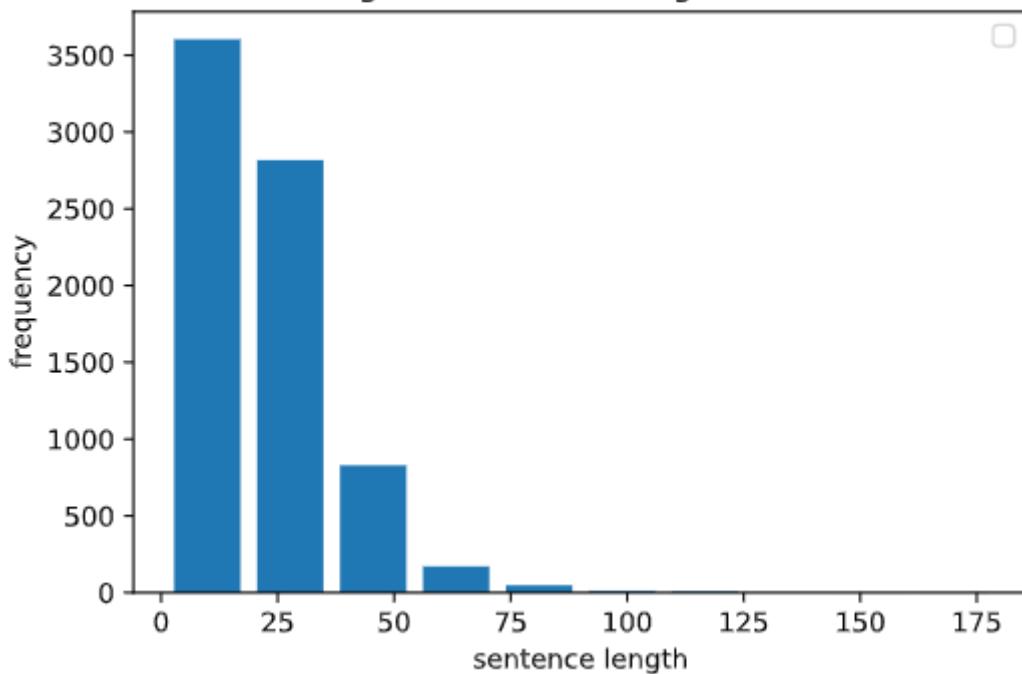
共有**7526**条数据，说明去重去掉了一部分的重复数据。此外清理后类别分布仍比较均匀，不需要做额外的数据增强。

注：`frequency` 指类别出现的频数。`class_index` 指类别编号，0-6 分别指 `['anger', 'disgust', 'fear', 'guilt', 'joy', 'sadness', 'shame']`。

- 句子长度分布

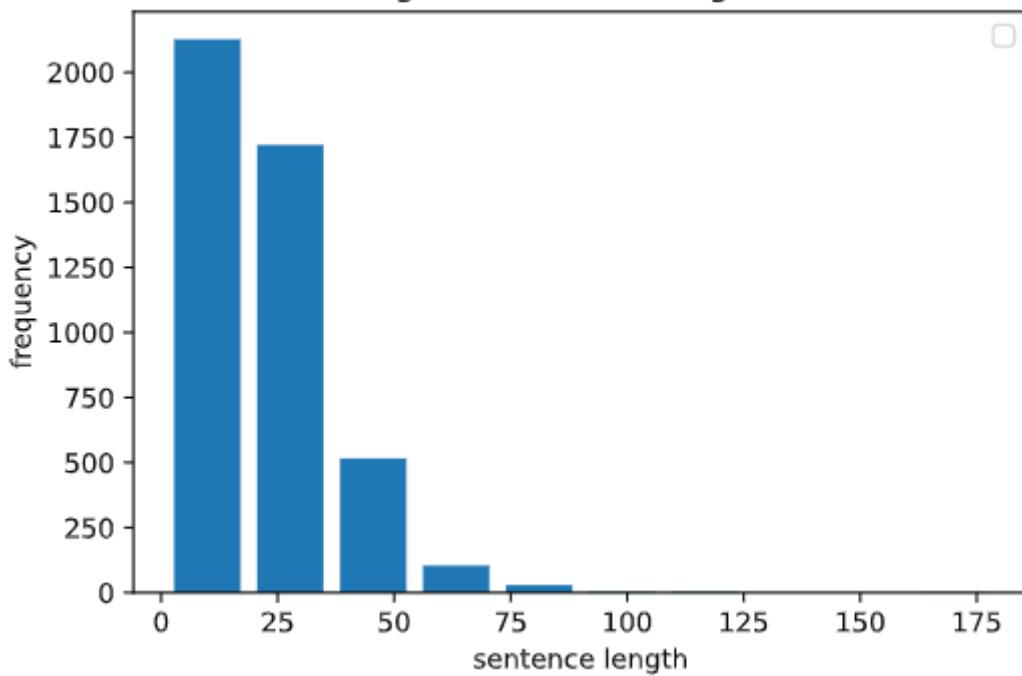
以下是全集，训练集，测试集，验证集的句子长度分布条形图：

Sentence Length Distribution Diagram of Universal Set

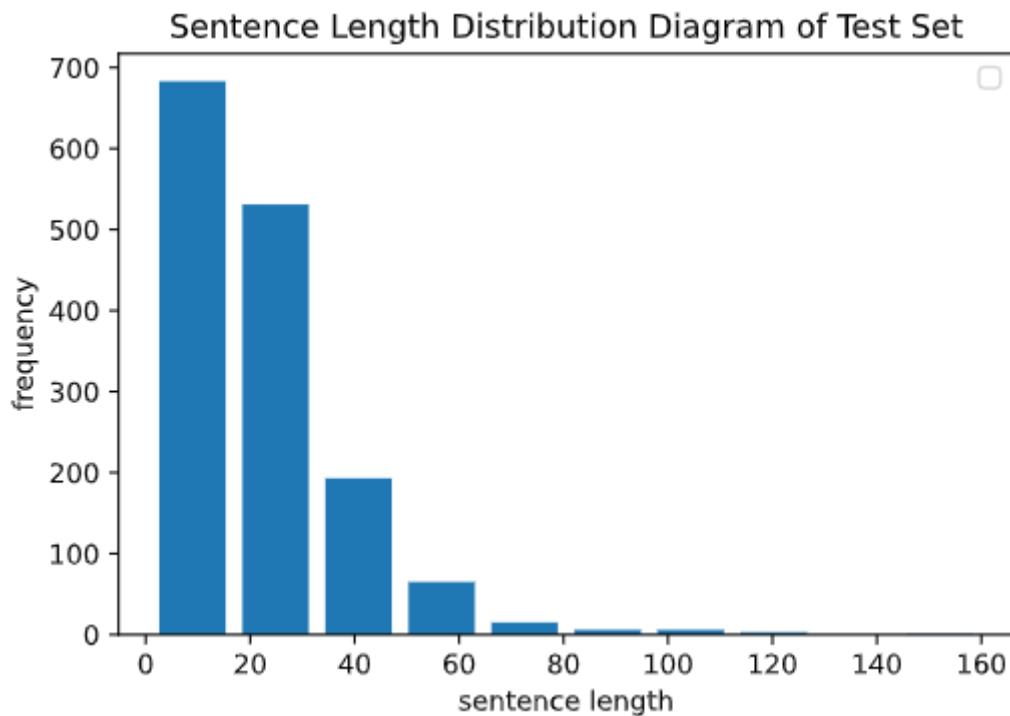


最大句子长度179

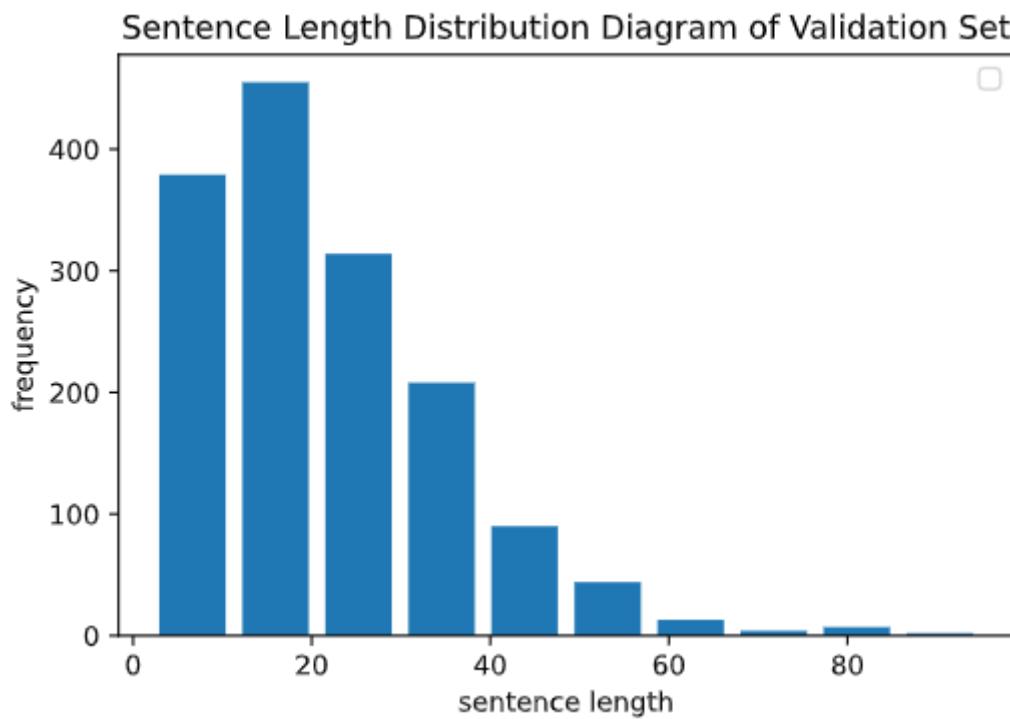
Sentence Length Distribution Diagram of Train Set



最大句子长度179



最大句子长度160



最大句子长度95

从图中可以看出，数据清理后句子长度分布几乎没有变化。

注：frequency 指该句子长度出现的频数。

- 可以发现，在清理完成后数据集变得干净不少，开心！清理完的数据集文件在：

```
/data/ISEAR_ID_f
/data/ISEAR_ID_train_f
/data/ISEAR_ID_test_f
/data/ISEAR_ID_validation_f
```

数据提取

- 从全集中提取词汇表，将词汇以出现频率由高到低进行编号。
- 将label转换成one-hot的形式。
- 将句子转换为词的向量，向量的每个元素是句子中对应词的编号。
- 将所有的词向量pad成与最长句子长度相同的长度。
- 经过预处理后的数据格式如下表

X Data	Shape	Y Data	Shape
train_x	[4507, 179]	train_y	[4507, 7]
test_x	[1503, 179]	test_y	[1503, 7]
validation_x	[1516, 179]	validation_y	[1516, 7]

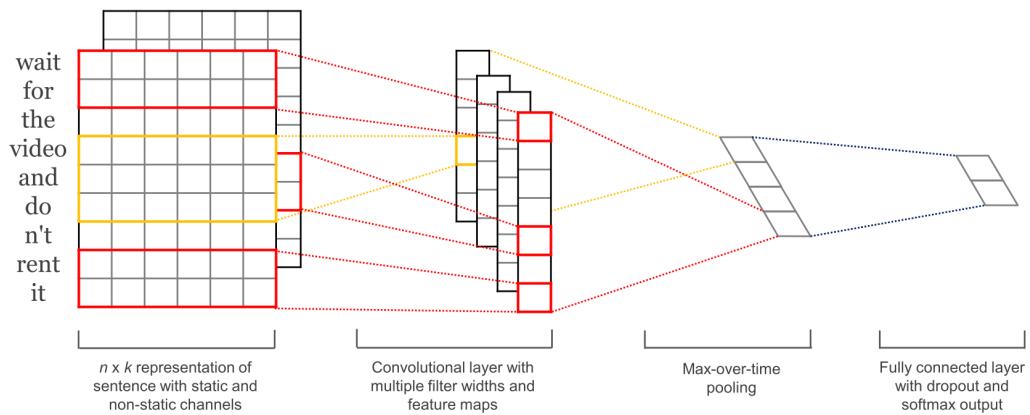
TextCNN

本实验实现了基于卷积层的TextCNN。TextCNN的具体结构可以参考这篇[论文](#)。此外，本实验还实现了带有Attention的CNN，以及增加了Inception层的CNN。

注：本实验部分代码参考了[PyTorch官网教程](#)的代码。

CNN

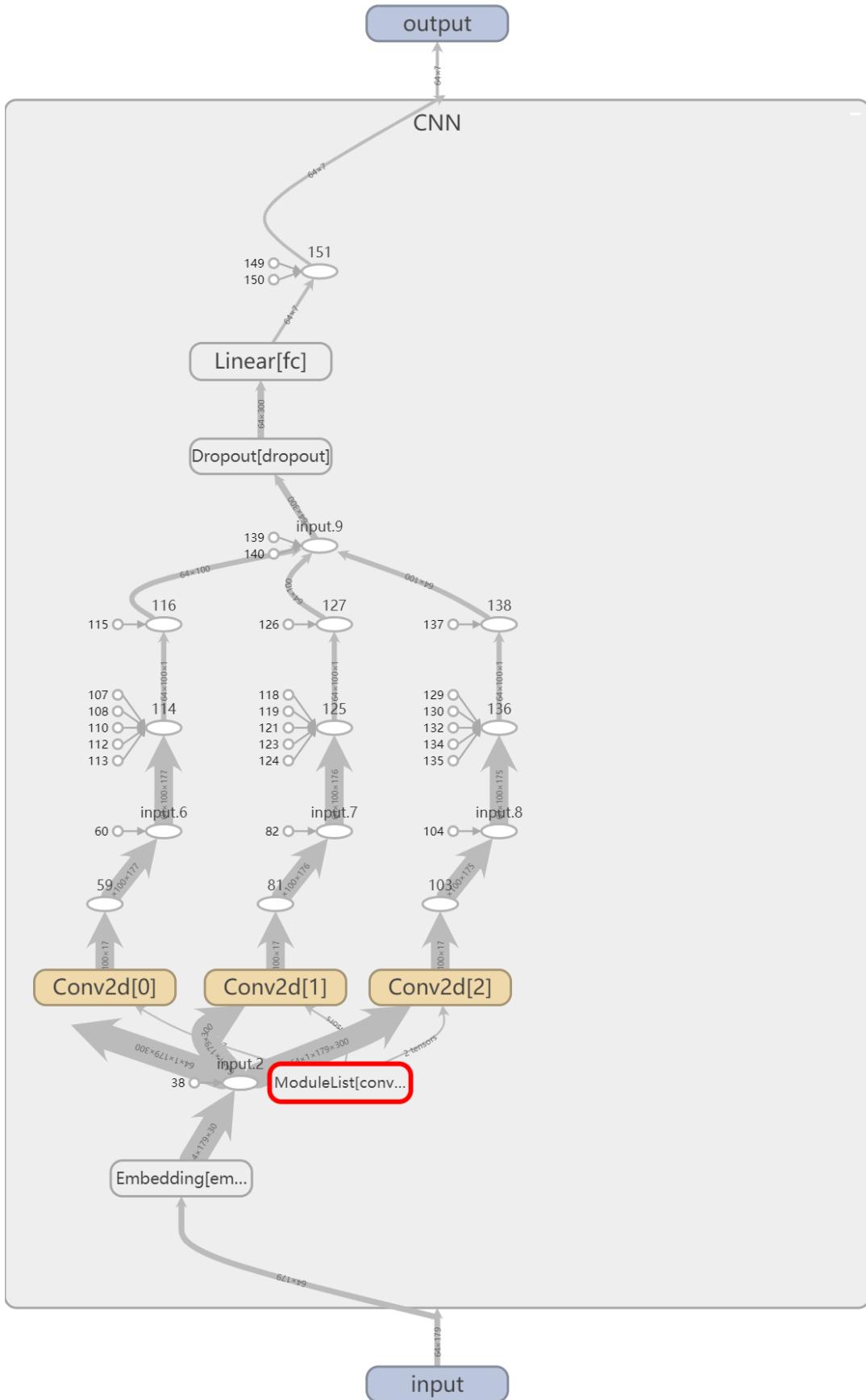
CNN的结构可以参考下图 ([图源文献](#))：



可以看出TextCNN相比传统CNN的区别主要在于其只在句子长度的方向上做卷积，而不在词向量的方向上卷积。

模型结构

本实验使用PyTorch框架实现了CNN模型，下图是利用TensorBoard输出的模型结构图。



计算流程

下面是模型的计算流程：

1. 将句子输入Embedding层，这里的Embedding层采用了[word2vec](#)的预训练权重。
2. 将经过Embedding层之后的向量输入不同尺寸的卷积层进行卷积，模型结构图中选择了三种不同尺寸的卷积核作为示例。实际上卷积核的种类数及大小都是可以调节的超参数。值得注意的是，卷积核的宽度与Embedding Size相等，所以实际上这是一维卷积。
3. 对不同卷积层的输出进行最大池化，筛选特征。然后再将池化后的输出拼接起来。
4. 经过全连接层，最后跟一个log softmax归一化得到模型输出，即不同种类的预测概率。

注：这里使用log softmax的主要原因是它相比softmax而言，能够加快运算速度，提高运算的稳定性，方便CrossEntropyLoss的计算，同时还能比较好地解决溢出的问题。

初步优化

根据[文献](#)中的Optimized Text CNN改进思路，对CNN模型作如下初步改进：

1. 加入dropout减少过拟合。
2. 使用L2正则化减少过拟合。
3. 使用提前停止训练策略减少过拟合。

参数优化

根据[文献](#)《A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification》的建议，结合本实验的特点，设定TextCNN进行参数优化的起点如下：

参数名称	参数取值	备注
dropout	0.5	
filter region size	[3, 4, 5]	即不同的卷积核大小
feature maps	100	即每种卷积核的数量
activation function	ReLU	
weight decay	3	L2正则化参数
learning rate	0.0001	

为了便于比较，训练的Epoch最大数量设置为100。

训练参数

首先进行训练参数的调整。为了便于模型之间的比较，在确定训练参数后，所有模型的实验都保持同样的训练参数不变。

主要调节的参数是Optimizer和Batch Size。

首先尝试不同的Optimizer，测试结果列表如下

Optimizer	Accuracy
Adadelta	0.140
Adam	0.143
AdamW	0.570
ASGD	0.144
RMSprop	0.132
SGD	0.144

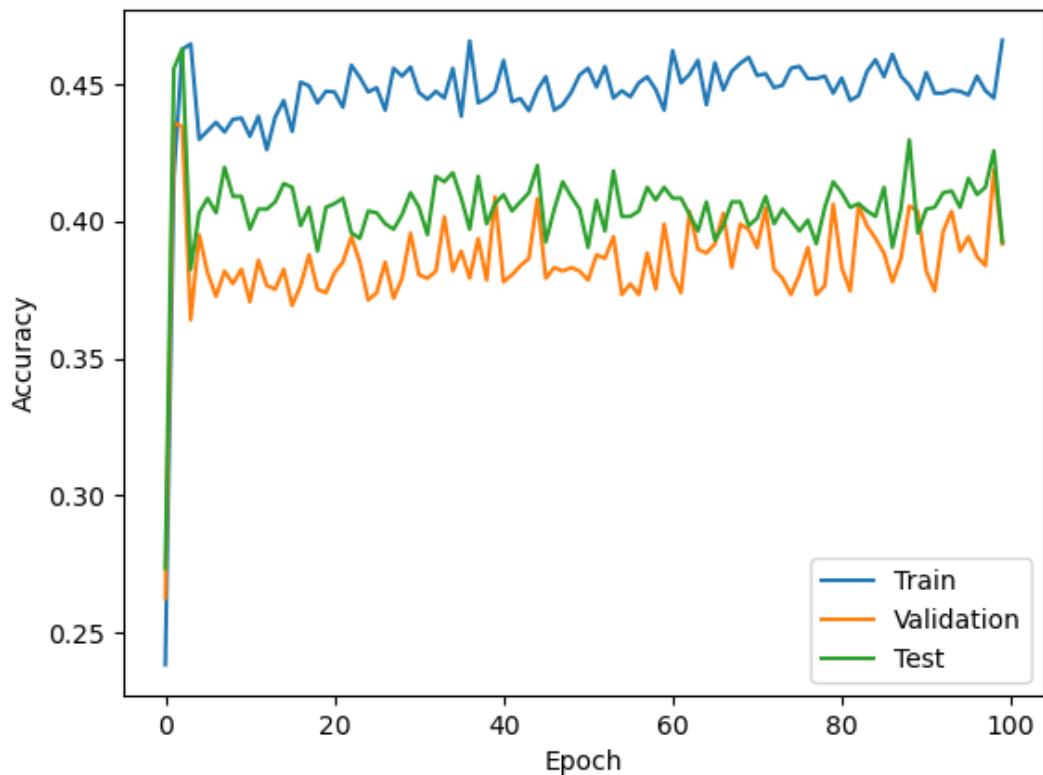
注：这里的Accuracy是由提前停止策略得到的Accuracy。即Validation Accuracy最大时的Test Accuracy。

由以上结果，我们确定使用AdamW作为优化器。

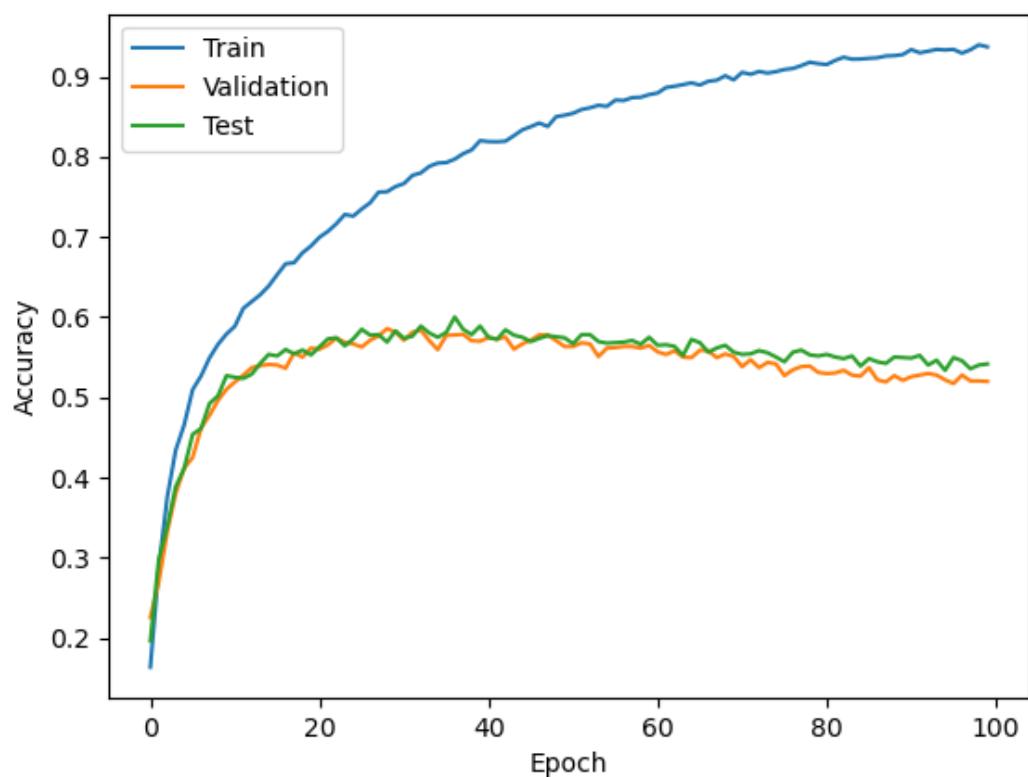
接着尝试不同的Batch Size，结果列表如下

Batch Size	Accuracy
1	0.456
16	0.554
32	0.574
64	0.569
80	0.584
96	0.577
112	0.573
128	0.576
4000	0.475

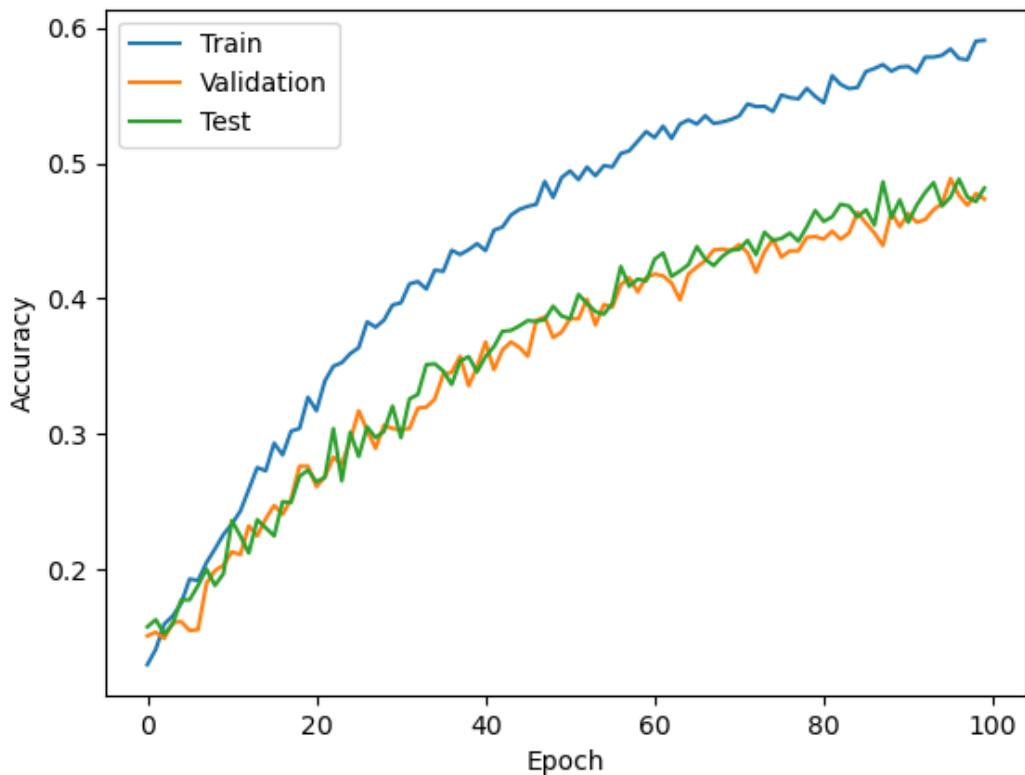
Batch Size为1的训练图：



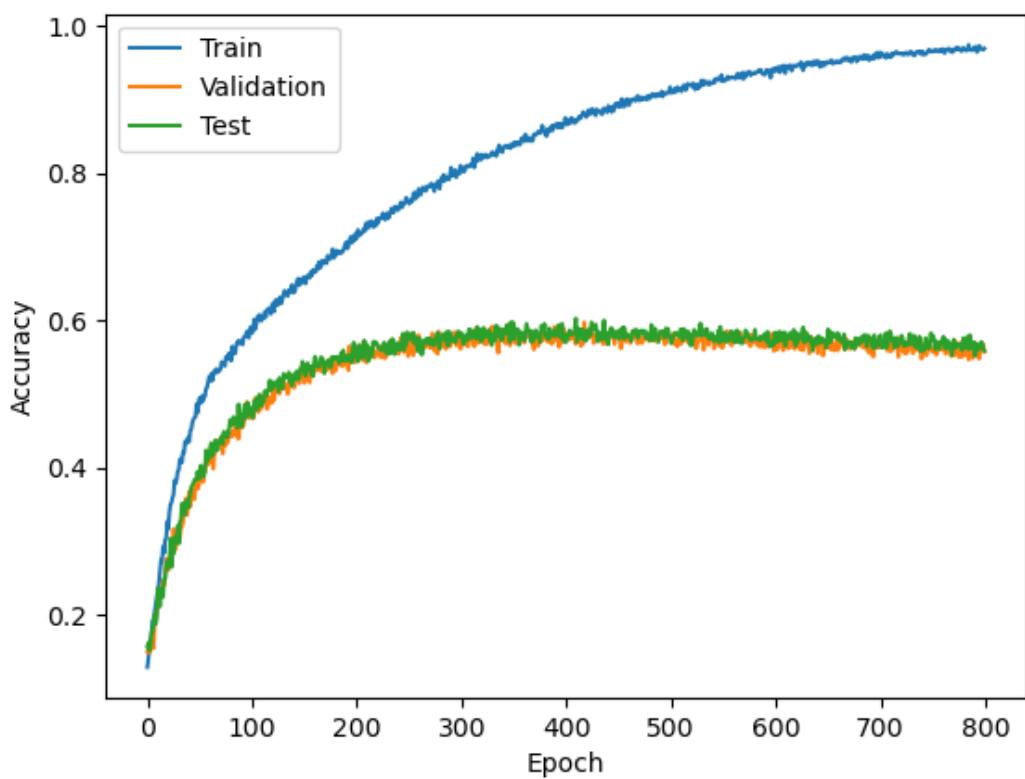
Batch Size为64的训练图：



Batch Size为4000的训练图：



由上图可见训练尚未收敛，将Epoch延长至800再次训练



此时训练收敛，Accuracy达到**0.586**。

由测试结果可以发现，在Batch Size过大或过小都会使训练收敛变慢。Batch Size太小会使模型反复向不同的方向优化，趋向于震荡，而太大也会显著减慢训练速度。虽然Batch Size为4000 (大小与训练集大小几乎一致)时Accuracy最高，但是为了提高训练速度，还是选择Batch Size为**80**。

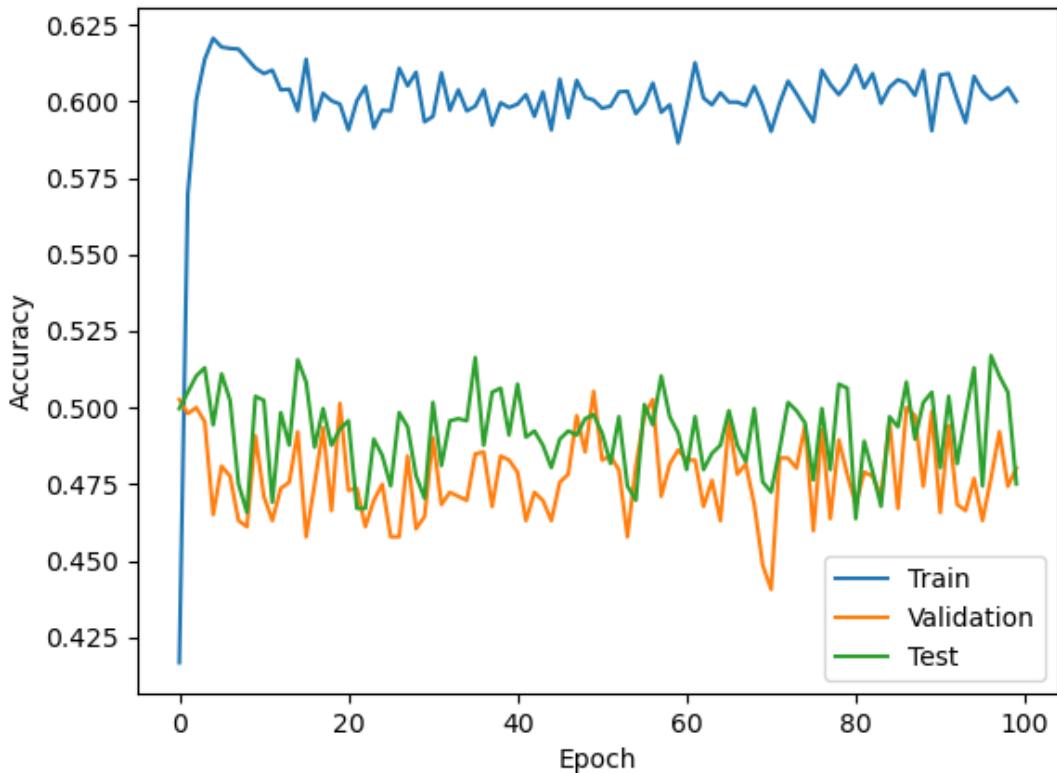
模型参数

按照论文《A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification》的建议，模型参数部分主要调整learning rate, filter region size和feature maps。

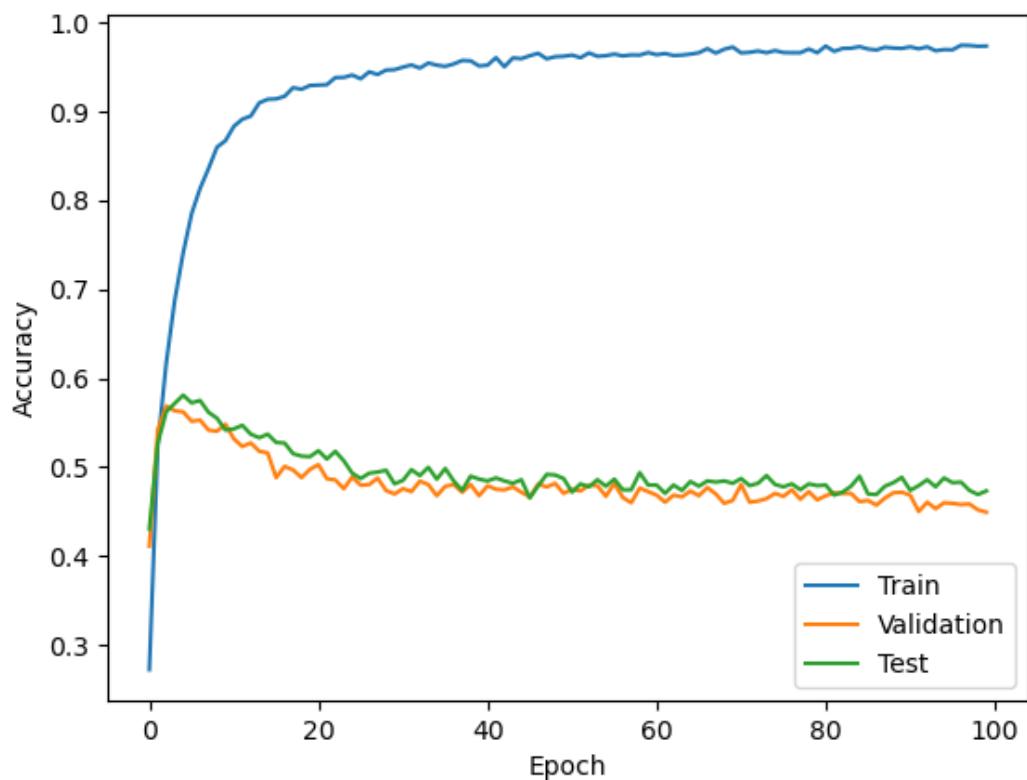
首先调节learning rate，测试结果如下表：

learning rate	Accuracy
0.01	0.498
0.001	0.562
0.0001	0.584
0.00001	0.496

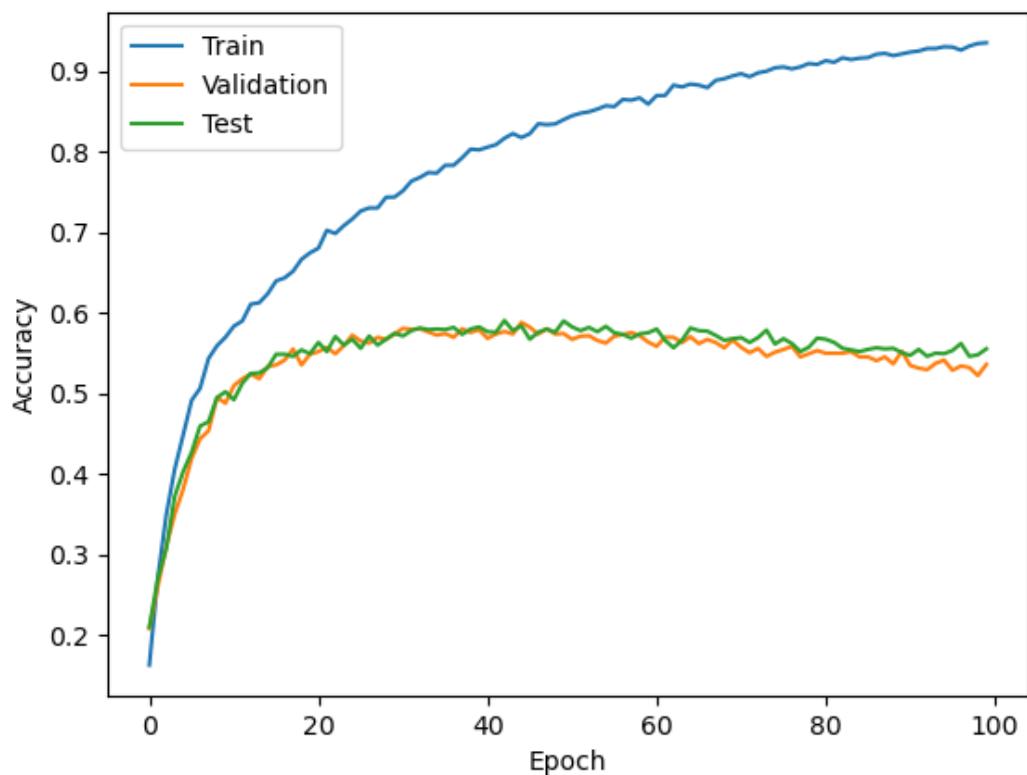
learning rate为0.01的训练图：



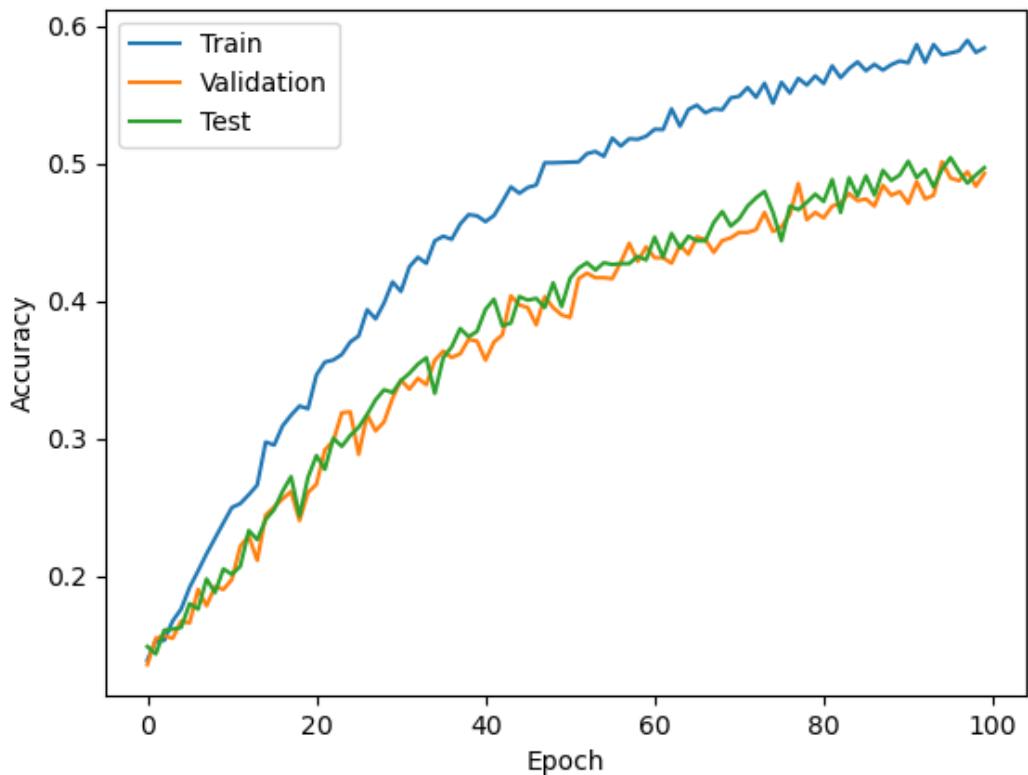
learning rate为0.001的训练图：



learning rate为0.0001的训练图：



learning rate为0.00001的训练图：



可以发现，learning rate越大，模型收敛得越快。这是因为learning rate越大，每次参数改变量就越大。权衡训练时间和准确率，最终选择learning rate为**0.0001**。

接着调节filter region size，按照论文的建议，先使用单一的卷积核进行测试，测试结果如下表(固定feature maps为300)：

filter region size	Accuracy
[1]	0.562
[2]	0.564
[3]	0.568
[4]	0.570
[5]	0.573
[6]	0.590
[7]	0.582
[8]	0.590
[9]	0.592
[10]	0.595
[11]	0.582
[12]	0.584

由上表可知效果最好的filter region size是10，按照论文的建议，下面在10附近探索不同的filter region size组合，测试结果如下表 (固定feature maps为100)：

filter region size	Accuracy
[8, 9, 10]	0.586
[9, 10]	0.588
[6, 8, 9, 10]	0.591
[6, 7, 8, 9, 10]	0.592
[1, 3, 5, 7, 9]	0.594
[3, 6, 9]	0.584
[10, 11, 12]	0.586
[2, 4, 6, 8, 10]	0.600
[2, 4, 6, 8, 9, 10]	0.592

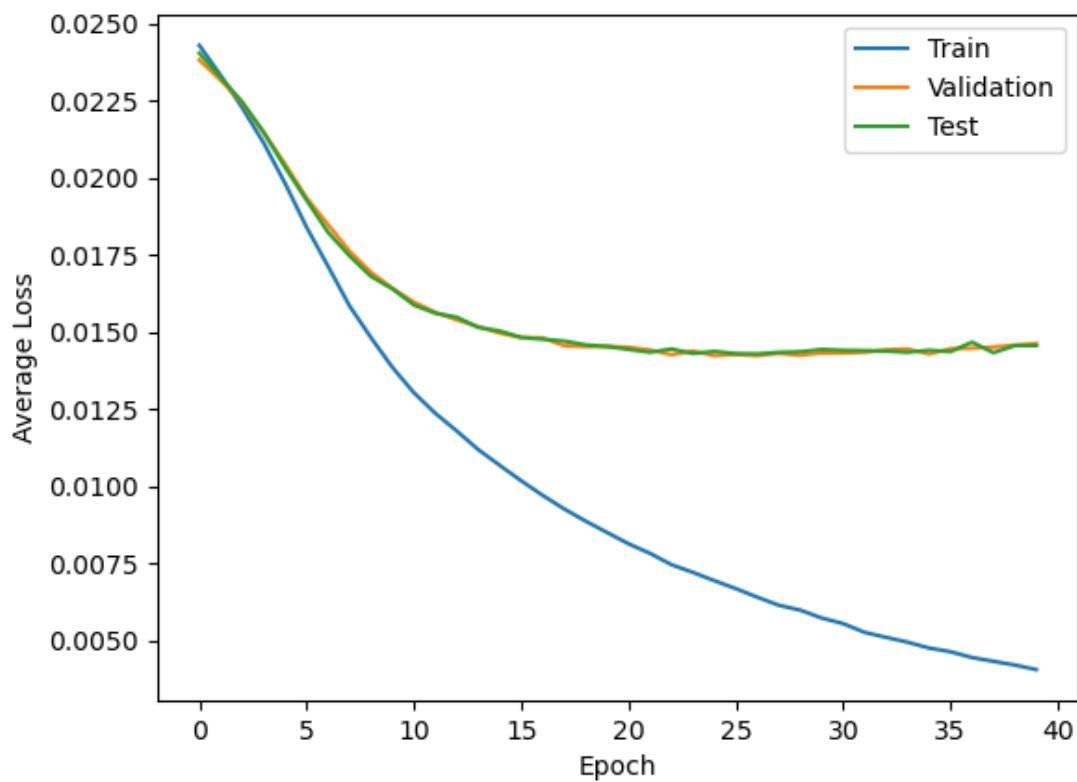
根据上述结果，选择filter region size为**[2, 4, 6, 8, 10]**。下面对feature maps进行调节，测试结果如下表：

feature maps	Accuracy
50	0.569
100	0.600
200	0.594
300	0.602
350	0.603
400	0.600
500	0.601

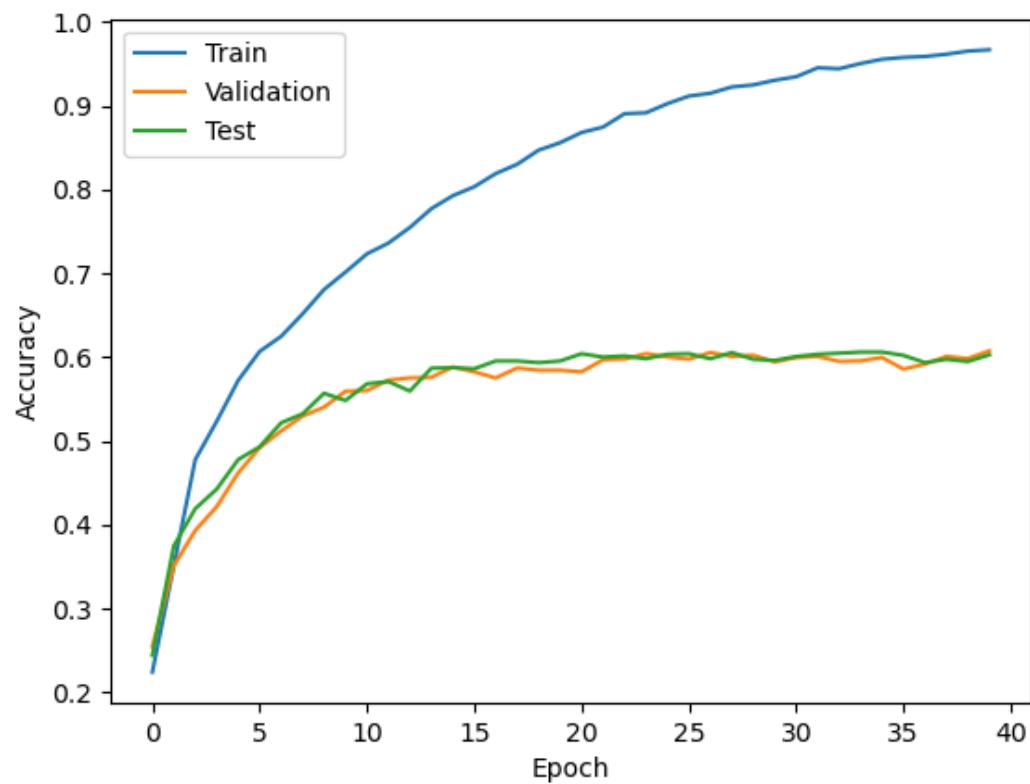
根据上述结果，选择feature maps为**350**。

实验结果

使用以上调节的参数进行训练，训练过程的average loss变化图：



accuracy变化图：

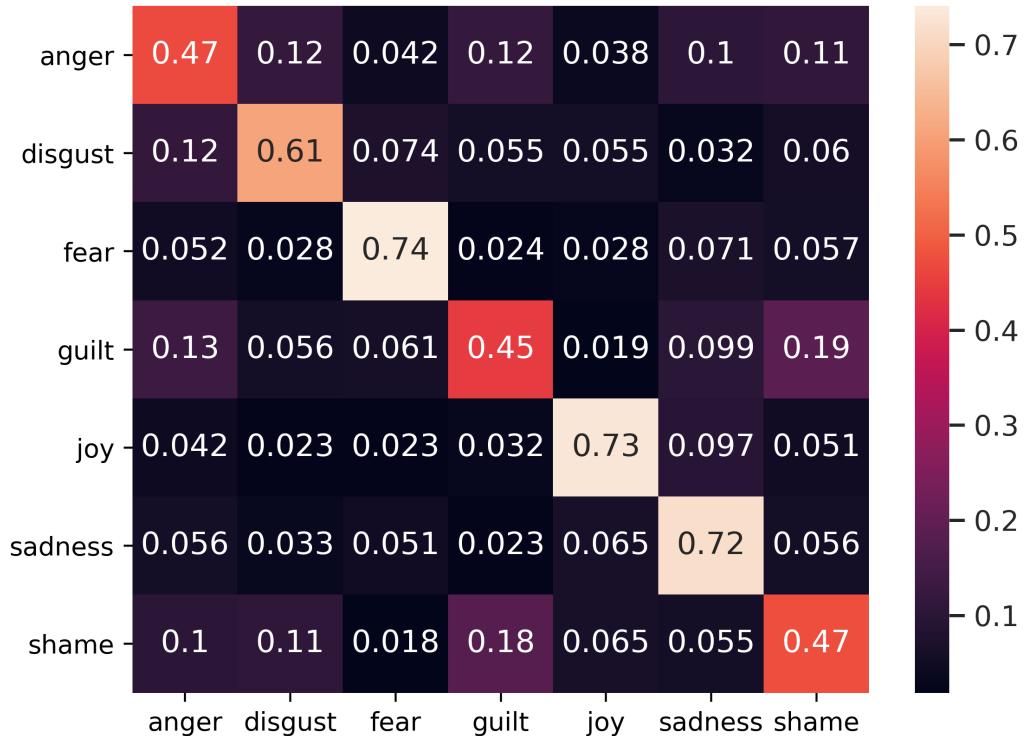


根据提前停止策略，选择在第39个Epoch停止训练，此时Valid Accuracy最大，为**0.608**。下面对训练得到模型进行测试：

原始输出：

	precision	recall	f1-score	support
anger	0.483	0.469	0.476	213
disgust	0.629	0.608	0.618	217
fear	0.730	0.741	0.735	212
guilt	0.505	0.446	0.474	213
joy	0.731	0.731	0.731	216
sadness	0.611	0.716	0.660	215
shame	0.479	0.475	0.477	217
accuracy			0.598	1503
macro avg	0.596	0.598	0.596	1503
weighted avg	0.596	0.598	0.596	1503

混淆矩阵：



从混淆矩阵可以看出 `anger`、`guilt`、`shame` 识别正确率偏低，并且这三者之间误判为彼此的概率也较大。实际上这也是合理的，因为这三种负面情绪确实不太容易分辨。

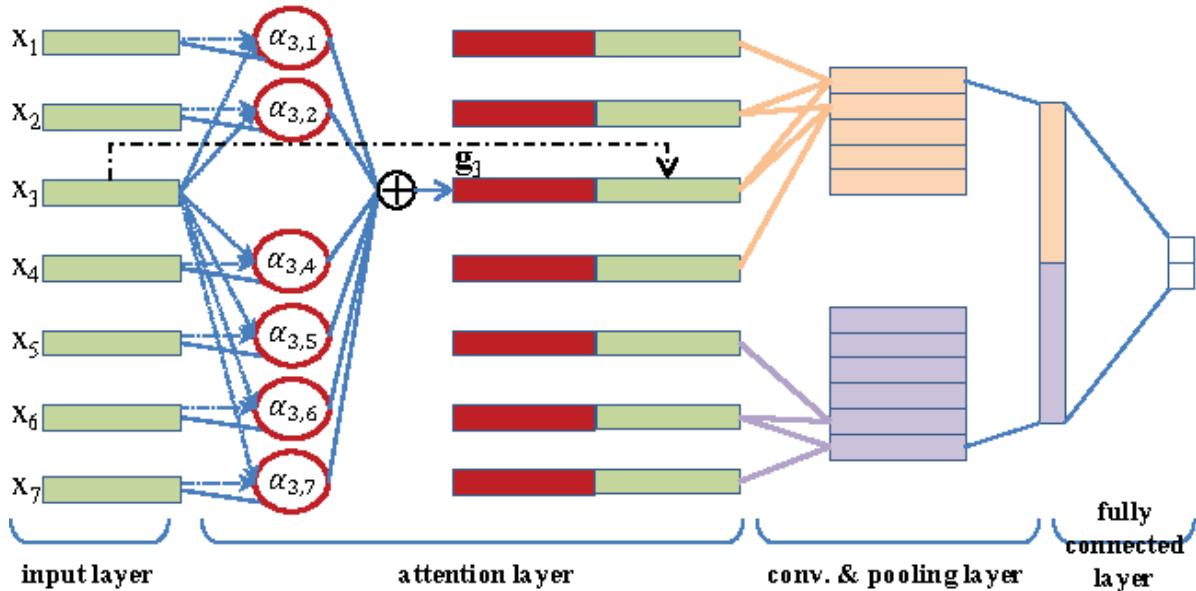
注：绘制混淆矩阵的代码参考了[官网教程](#)。

三项指标：

名称	precision	recall	f1-score
accuracy	\	\	0.598
macro avg	0.596	0.598	0.596
micro avg	0.598	0.598	0.598
weighted avg	0.596	0.598	0.596

CNN + Attention

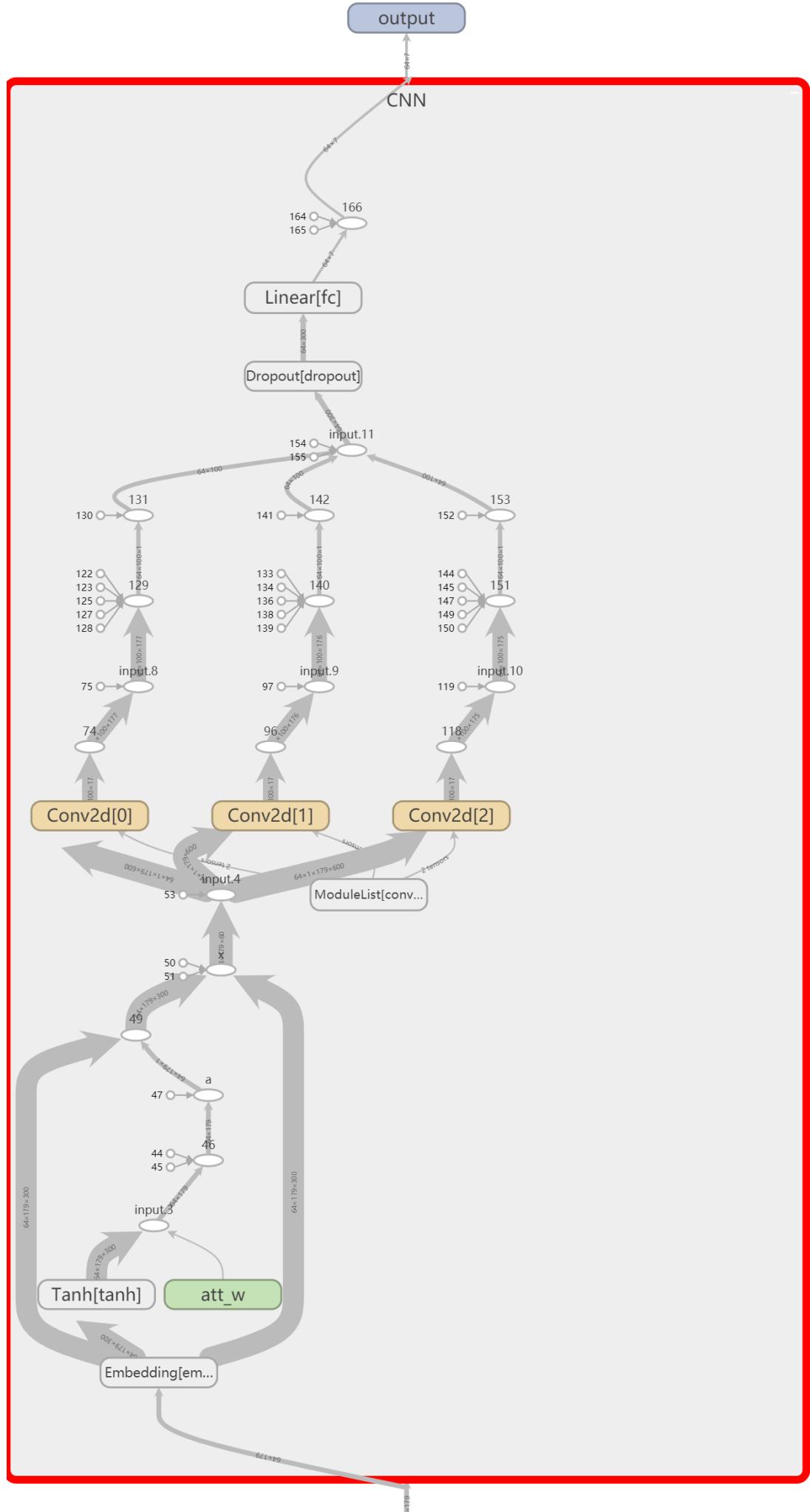
根据[论文](#)的思路，本实验还实现加入Attention机制TextCNN，使TextCNN能够专注于那些情感色彩比较重的词汇。Attention-Based TextCNN的结构简图如下(图源[文献](#)):



从图中可以看出Attention-Based TextCNN与TextCNN的结构几乎一样，只是在Embedding层和卷积层之间加入了Attention层。

模型结构

本实验使用PyTorch框架实现了CNN + Attention模型，下图是利用TensorBoard输出的模型结构图。



计算流程

下面是模型的计算流程：

1. 将句子输入Embedding层，这里的Embedding层采用了[word2vec](#)的预训练权重。
2. **初始化可学习的权重矩阵 W ，将Embedding的输出经过tanh非线性激活后与 W 相乘，再将相乘后的结果与Embedding的输出拼接得到新的张量。**
3. 输入不同尺寸的卷积层进行卷积，模型结构图中选择了三种不同尺寸的卷积核作为示例。实际上卷积核的种类数及大小都是可以调节的超参数。值得注意的是，卷积核的宽度与Embedding Size相等，所以实际上这是一维卷积。
4. 对不同卷积层的输出进行最大池化，筛选特征。然后再将池化后的输出拼接起来。
5. 经过全连接层，最后跟一个log softmax归一化得到模型输出，即不同种类的预测概率。

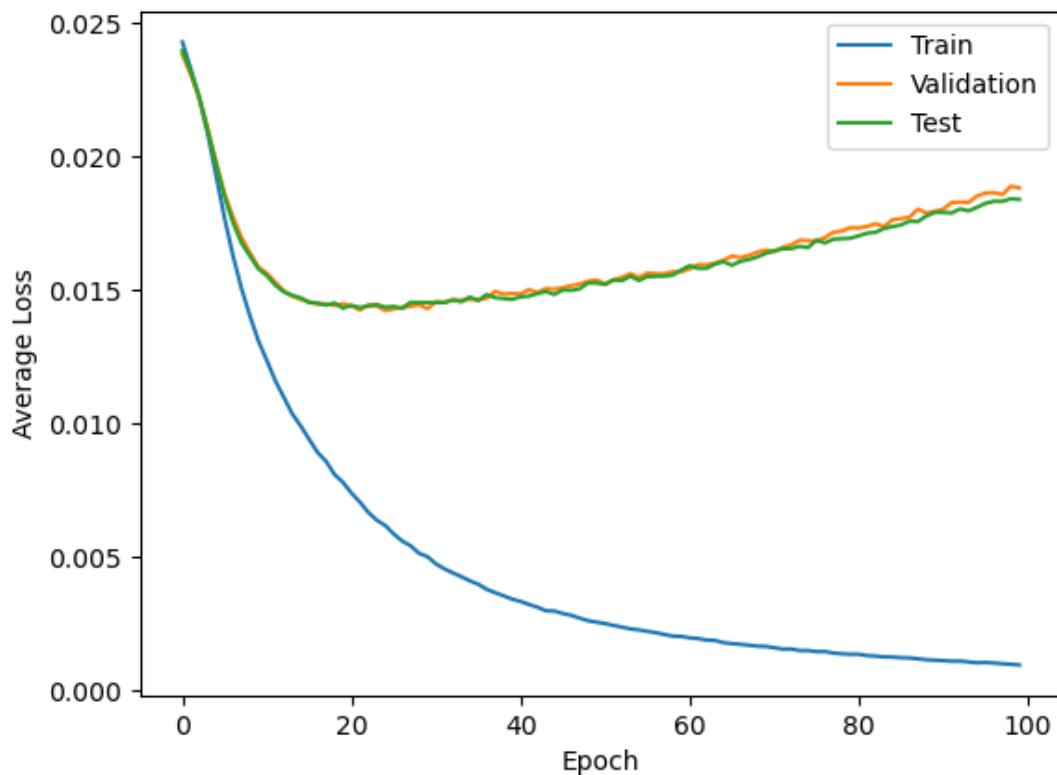
注：黑体字标注的是与TextCNN相比**新增**的部分。

参数选择

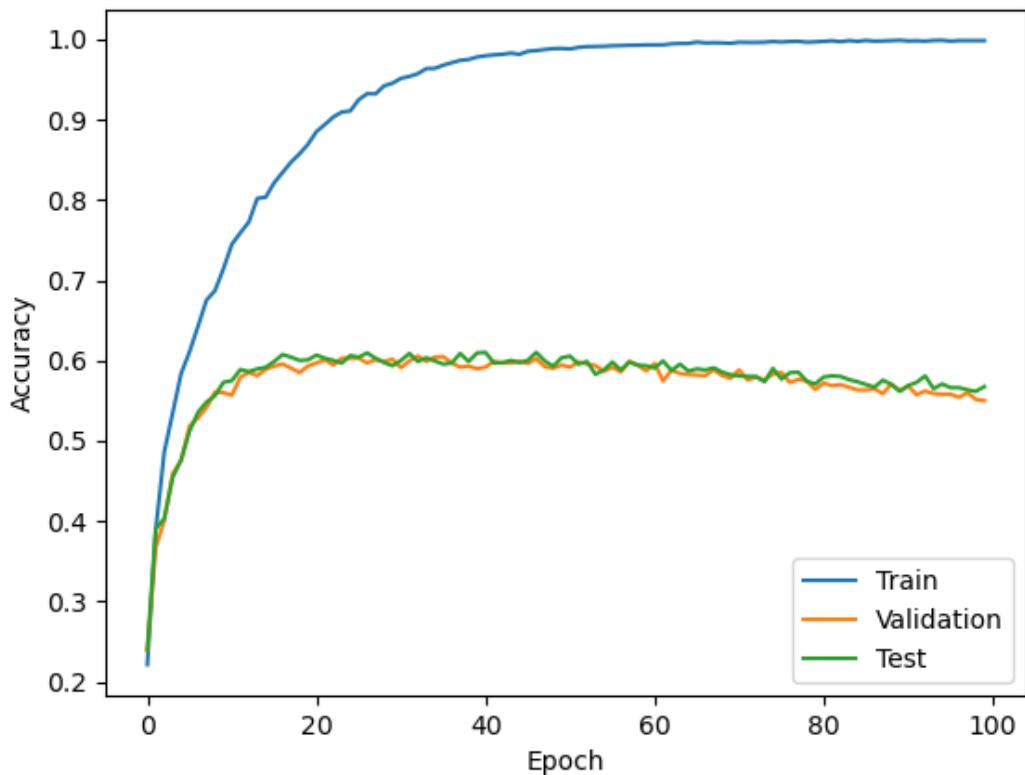
由于Attention-Based CNN相比Pure CNN而言只是加了一层attention层，没有额外的超参数，因此参数选择与Pure CNN参数优化后的结果一致。

实验结果

训练过程的average loss变化图：



accuracy变化图：

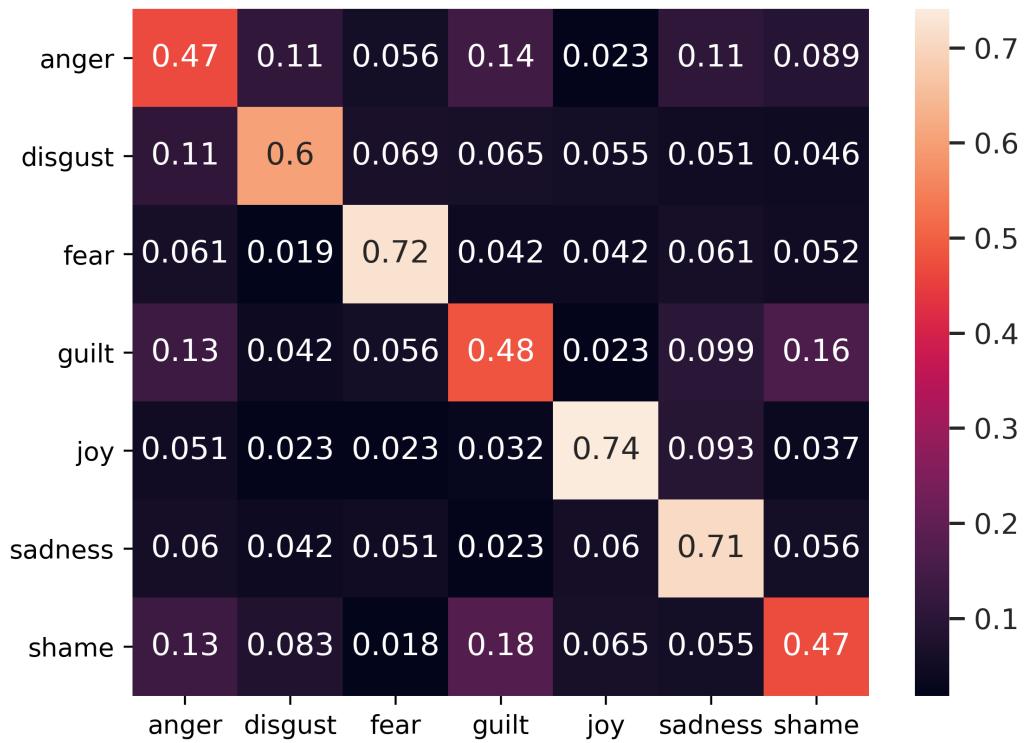


根据提前停止策略，选择在第32个Epoch停止训练，此时Valid Accuracy最大，为**0.607**。下面对训练得到模型进行测试：

原始输出：

	precision	recall	f1-score	support
anger	0.459	0.469	0.464	213
disgust	0.655	0.604	0.628	217
fear	0.722	0.722	0.722	212
guilt	0.500	0.484	0.492	213
joy	0.734	0.741	0.737	216
sadness	0.603	0.707	0.651	215
shame	0.518	0.470	0.493	217
accuracy			0.599	1503
macro avg	0.599	0.599	0.598	1503
weighted avg	0.599	0.599	0.598	1503

混淆矩阵：



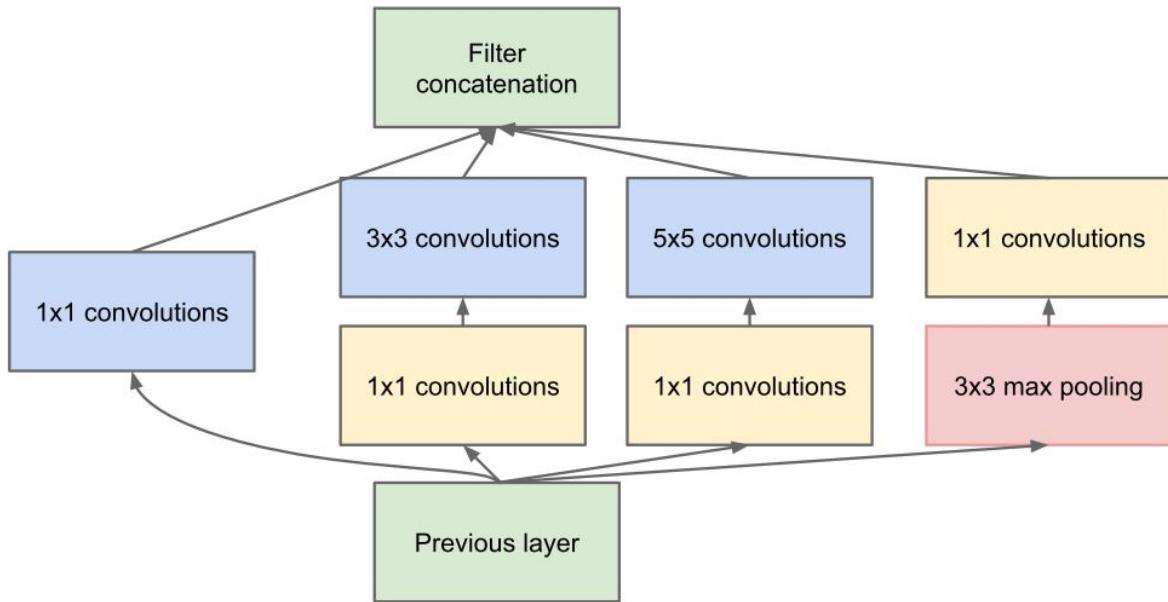
从混淆矩阵可以看出加了Attention之后CNN确实有了一定程度的提升，但仍然是 anger、guilt、shame 的分类正确率较低。

三项指标：

名称	precision	recall	f1-score
accuracy	\	\	0.599
macro avg	0.599	0.599	0.598
micro avg	0.599	0.599	0.599
weighted avg	0.599	0.599	0.598

CNN + Inception

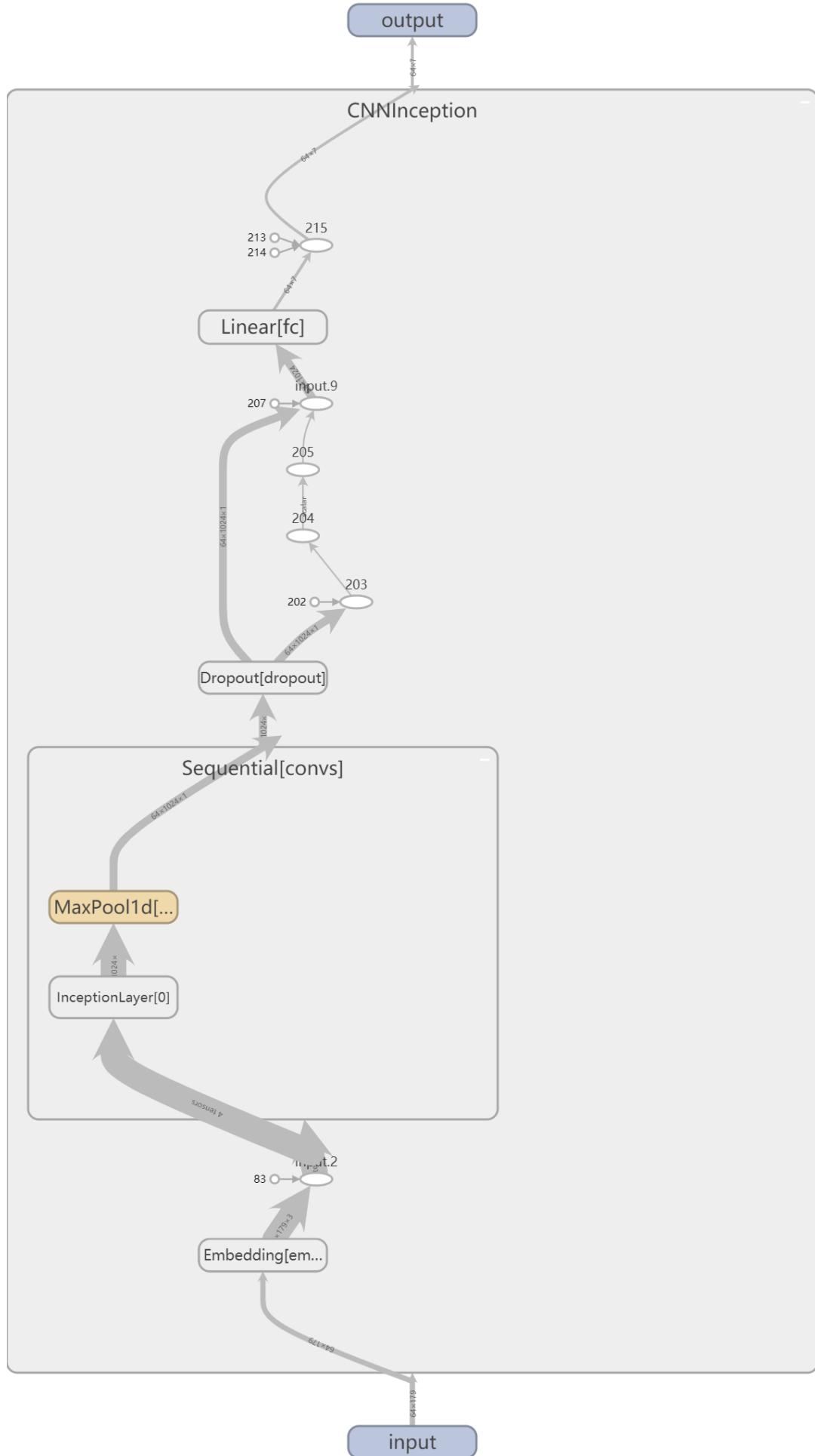
本实验实现了加入Inception层的TextCNN。Inception最早由Google于[论文](#)《Going Deeper with Convolutions》中提出。Inception具有如下的结构：



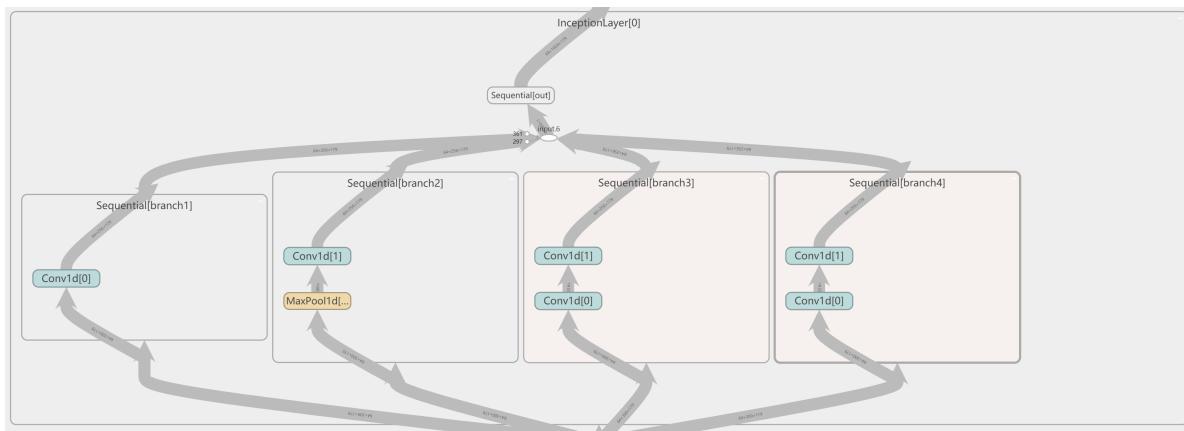
Inception的优势包括降低模型复杂度，减少过拟合的可能以及能更好地聚集不同维度的特征。本实验将Inception这种结构替换掉原来TextCNN中的相应结构，不仅降低了参数量，提高了训练速度，同时也在一定程度上缓解了过拟合，提高了特征筛选聚合的能力。

模型结构

本实验使用PyTorch框架实现了CNN + Inception模型，下图是利用TensorBoard输出的模型结构图。



再来看一下InceptionLayer的具体结构，确实和Google的InceptionLayer完全一致：



计算流程

下面是模型的计算流程：

1. 将句子输入Embedding层，这里的Embedding层采用了[word2vec](#)的预训练权重。
2. **输入一维卷积层的Inception Layer进行卷积操作。Inception Layer的输出是各个branch的拼接。**
3. 进行最大池化，筛选特征。然后再将池化后的输出拼接起来。
4. 经过全连接层，最后跟一个log softmax归一化得到模型输出，即不同种类的预测概率。

注：黑体字标注的是与TextCNN相比改变的部分。

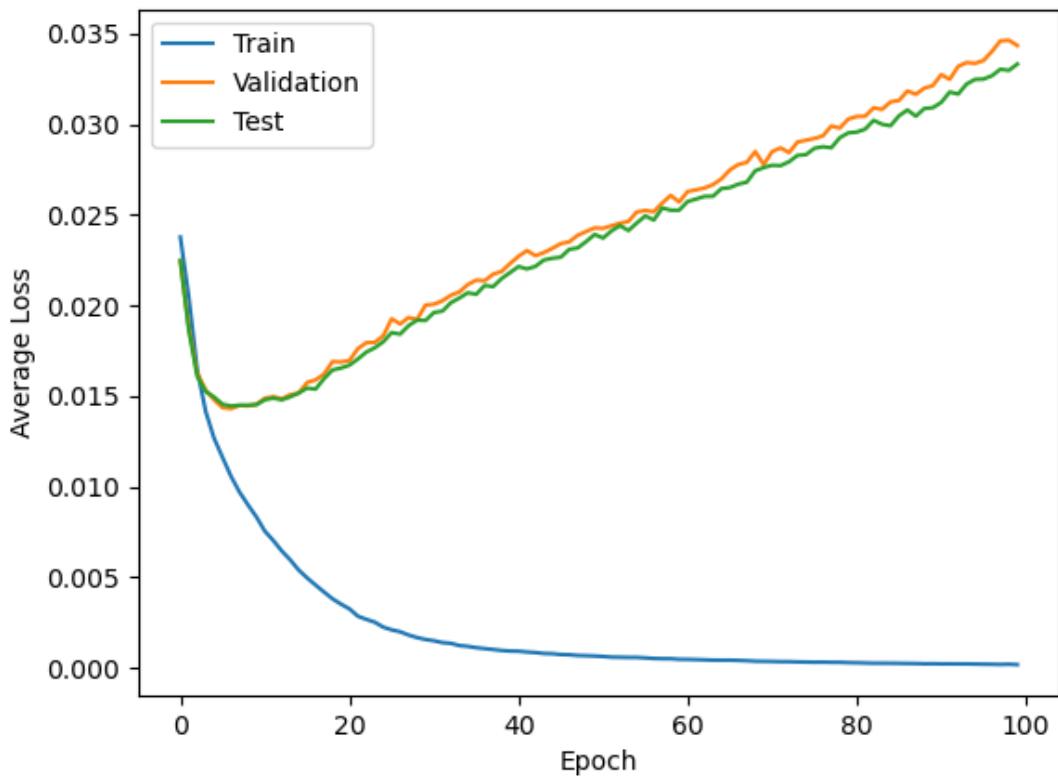
参数选择

参数名称	参数取值	备注
inception size	5000	指Inception Layer的通道数之和

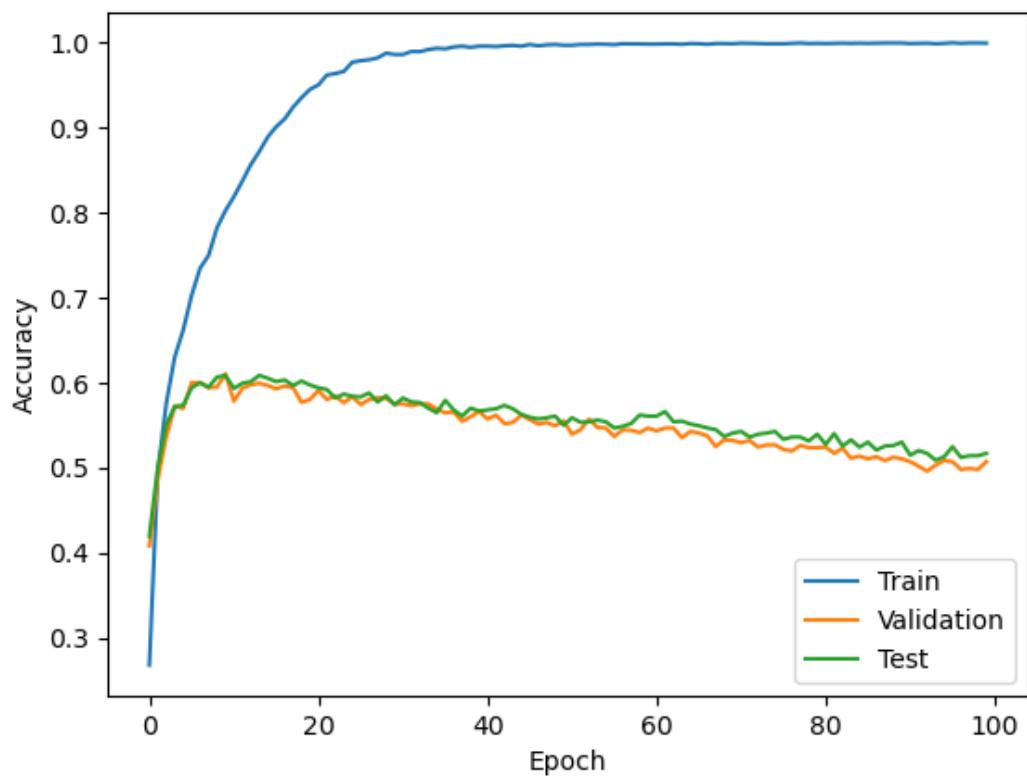
注：上表中没有出现的参数默认与CNN的取值一致。

实验结果

训练过程的average loss变化图：



accuracy变化图：



根据提前停止策略，选择在第9个Epoch停止训练，此时Valid Accuracy最大，为**0.611**。下面对训练得到的模型进行测试：

原始输出：

	precision	recall	f1-score	support
anger	0.559	0.357	0.436	213
disgust	0.684	0.618	0.649	217
fear	0.690	0.736	0.712	212
guilt	0.505	0.526	0.515	213
joy	0.722	0.769	0.744	216
sadness	0.553	0.758	0.639	215
shame	0.561	0.512	0.535	217
accuracy			0.611	1503
macro avg	0.610	0.611	0.604	1503
weighted avg	0.610	0.611	0.605	1503

混淆矩阵：



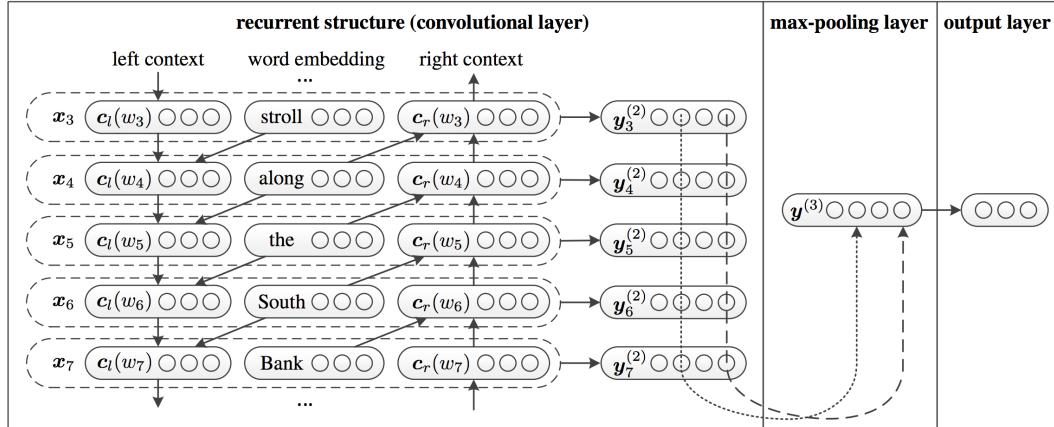
从混淆矩阵可以看出加了CNN + Inception相比CNN + Attention又有了一定程度的提升，但仍然是anger、guilt、shame的分类正确率较低，尤其是anger。

三项指标：

名称	precision	recall	f1-score
accuracy	\	\	0.611
macro avg	0.610	0.611	0.604
micro avg	0.611	0.611	0.611
weighted avg	0.610	0.611	0.605

RNN

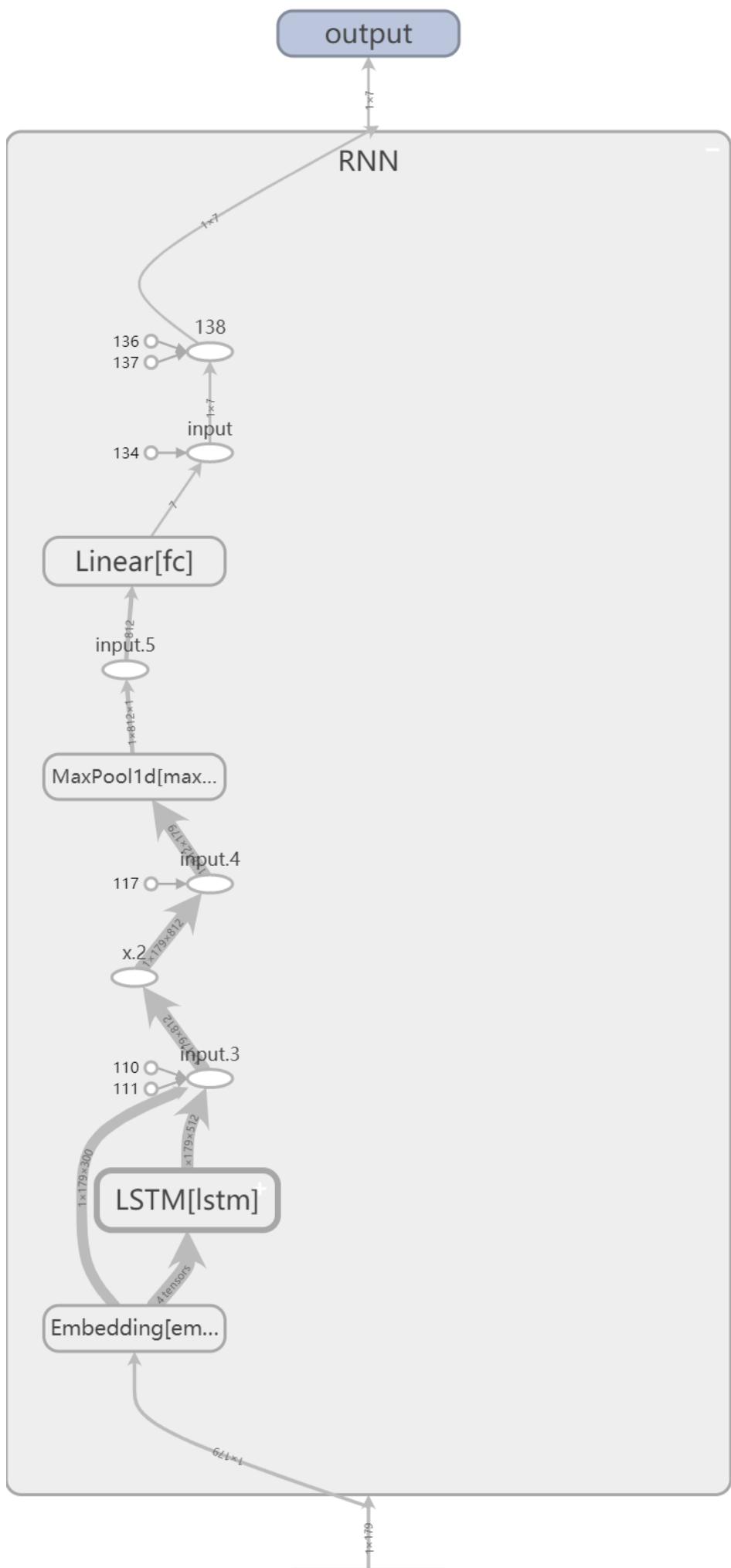
本实验实现了基于双向LSTM的RNN模型。此外，还根据[论文](#)的思路加入了池化层，使RNN在能有效获取上下文同时具备一定的感受野。该模型也被称为RCNN。下面是文献中描述RCNN结构的简图。



可以看到RCNN的重要创新点是在双向LSTM之后加入了一个最大池化的操作，最大池化可以起到特征筛选的作用，同时也给予了RCNN一定的感受野。

模型结构

本实验使用PyTorch框架实现了RNN模型，下图是利用TensorBoard输出的模型结构图。



计算流程

下面是模型的计算流程：

1. 将句子输入Embedding层，这里的Embedding层跟之前一样采用了word2vec预训练的权重。
2. 将经过Embedding层之后的向量输入双向LSTM，得到所有时刻的隐层状态。
3. 将Embedding层的原始输出与双向LSTM的输出拼接，在经过一个ReLU激活。
4. 输入最大池化层，筛选特征。
5. 经过全连接层，最后跟一个log softmax归一化得到模型输出。

参数优化

RNN模型需要调节的主要参数有learning rate, hidden size和layer number(指LSTM层数).

首先调节learning rate，测试结果如下表：

learning rate	Accuracy
0.01	0.531
0.001	0.584
0.0001	0.530
0.00001	0.395

根据上表结果，选择**0.001**作为学习率。

接着调节hidden size，测试结果如下表：

hidden size	Accuracy
128	0.566
256	0.593
384	0.576
448	0.572
512	0.589
768	0.581
1024	0.570

值得注意的是，尽管设置了随机种子，但是RNN的结果复现性仍比较差，甚至有一定的波动，猜测这是由LSTM的底层实现导致的。综合上述结果，我们选择hidden size为**256**。

最后调节layer number (指LSTM的层数)，测试结果如下表：

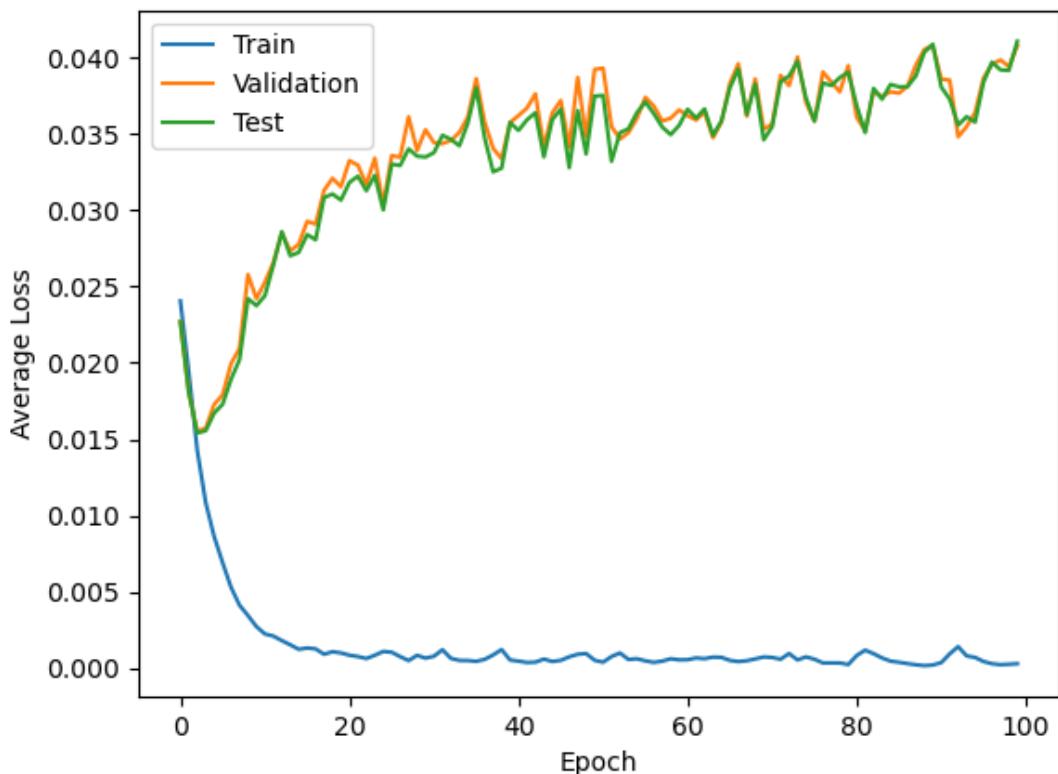
layer number	Accuracy
1	0.591
2	0.593
3	0.518
4	0.504

综合上述结果，选择layer number为2。

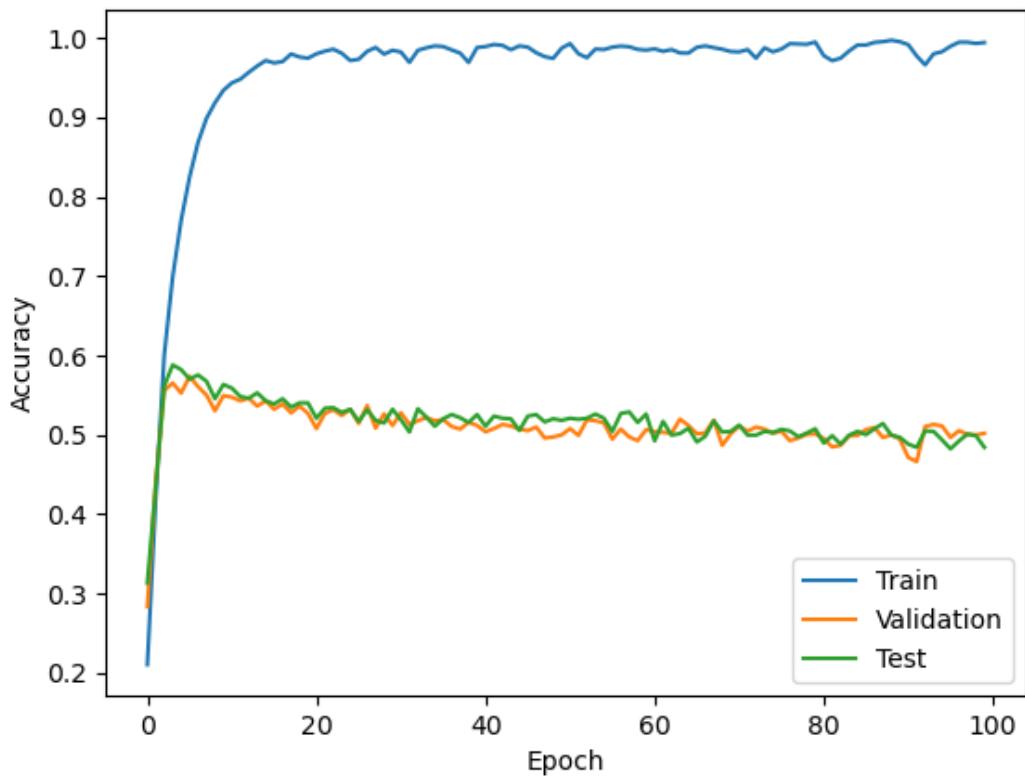
注：没有调优的参数默认与上一次取值一致。

实验结果

训练过程的average loss变化图：



accuracy变化图：



根据提前停止策略，选择在第5个Epoch停止训练，此时Valid Accuracy最大，为**0.574**。下面对训练得到模型进行测试：

原始输出：

	precision	recall	f1-score	support
anger	0.441	0.653	0.527	213
disgust	0.639	0.604	0.621	217
fear	0.656	0.745	0.698	212
guilt	0.593	0.315	0.411	213
joy	0.787	0.718	0.751	216
sadness	0.542	0.693	0.608	215
shame	0.618	0.447	0.519	217
accuracy			0.596	1503
macro avg	0.611	0.596	0.590	1503
weighted avg	0.611	0.596	0.591	1503

混淆矩阵：



从中可以看出 `guilt`、`shame` 的分类正确率较低，且RNN的效果与CNN相近，没有特别明显的优势。

三项指标：

名称	precision	recall	f1-score
accuracy	\	\	0.596
macro avg	0.611	0.596	0.590
micro avg	0.596	0.596	0.596
weighted avg	0.611	0.596	0.591

Baseline

本实验实现了FastText模型作为Baseline进行比较。FastText模型是2016年由Facebook提出的迅速文本分类方法。模型的具体设计思路详见[论文](#)，下面仅简要介绍其结构。

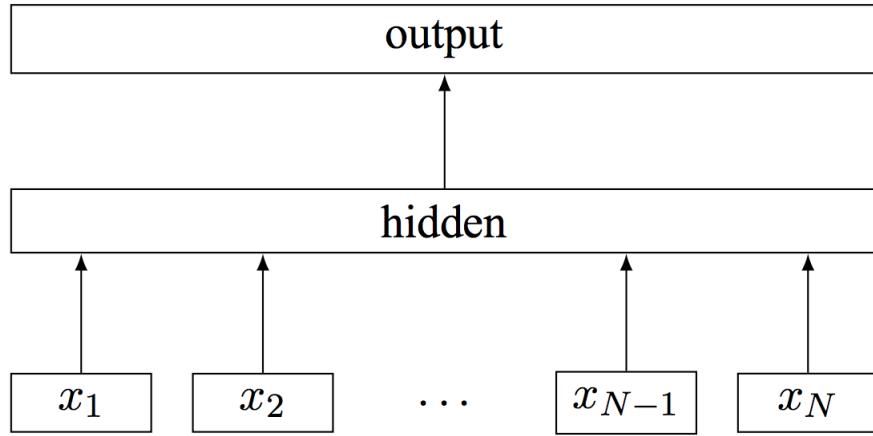
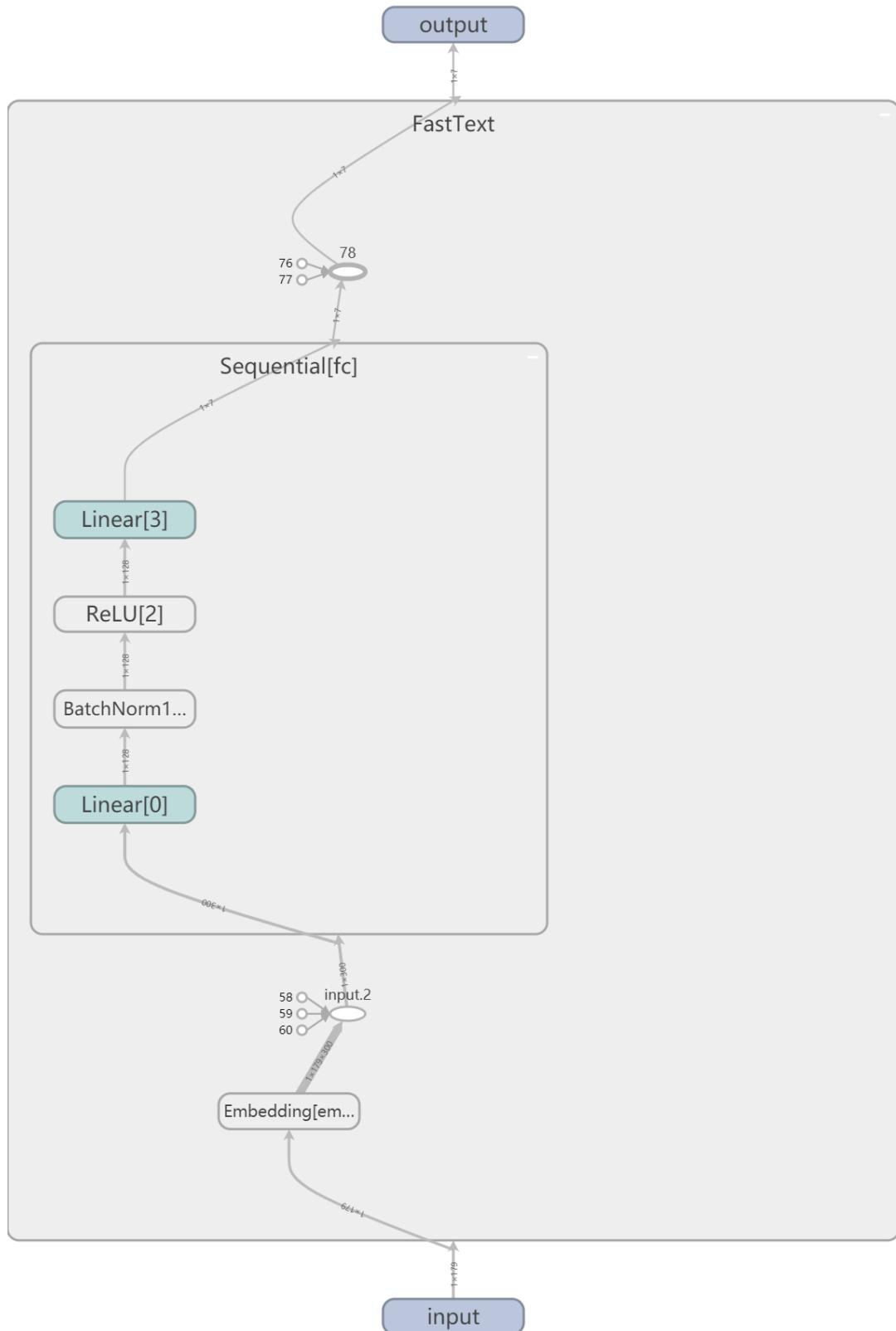


Figure 1: Model architecture of fastText for a sentence with N ngram features x_1, \dots, x_N . The features are embedded and averaged to form the hidden variable.

上图是FastText模型的主要结构。FastText模型仅含一个隐藏层，参数量小，训练速度快。

模型结构

本实验使用PyTorch框架实现了FastText模型，下图是利用TensorBoard输出的模型结构图。



计算流程

下面是模型的计算流程：

1. 用Hash对句子的N-Gram信息进行编码，得到N个句子向量。
2. 将N个句子向量输入Embedding层。

3. 拼接Embedding层的输出向量，并在句子长度维度上求平均(如果这一维度的大小为100，亦即句子长度为100，求平均后的输出在这一维度的大小变为1)。
4. 将求平均后的向量输入全连接层(即隐藏层)，再跟一个非线性激活。
5. 跟一个全连接层，最后跟一个log softmax归一化得到模型输出。

参数选择

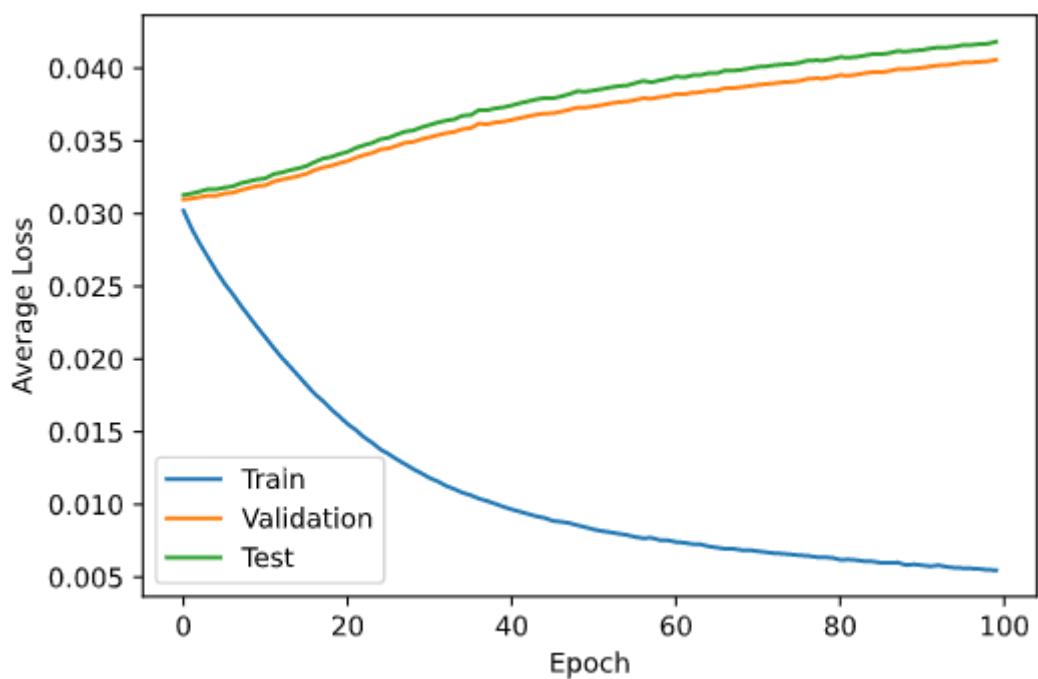
模型的主要参数选择如下：

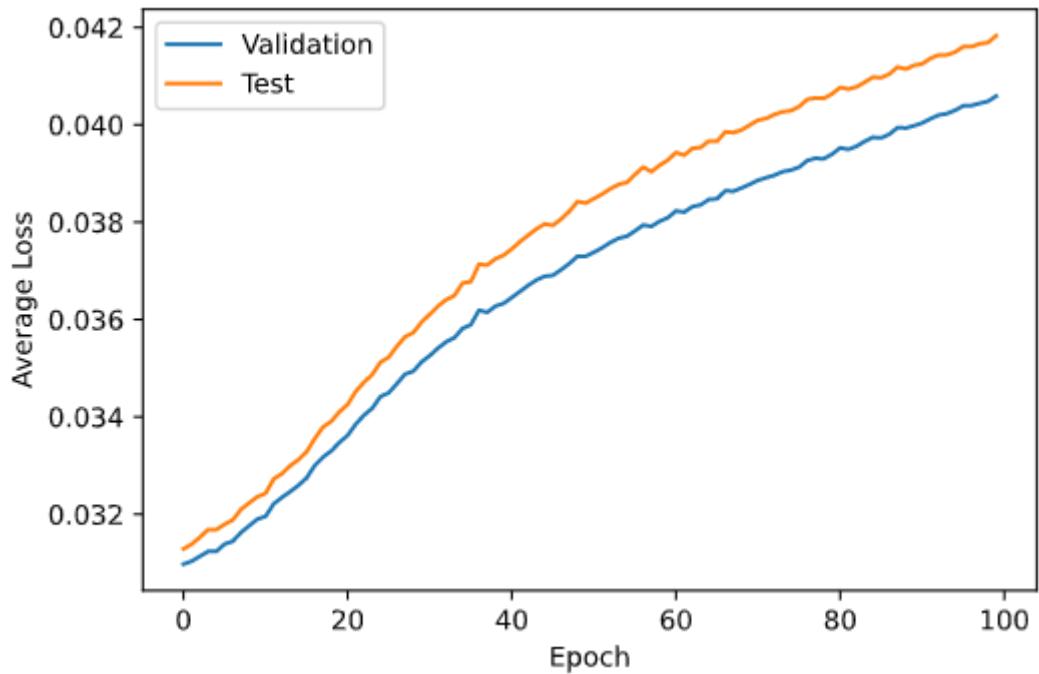
参数名称	参数取值	备注
dropout	0.5	
hidden size	128	隐藏层大小
N-Gram哈希表大小	250007	即对N-Gram进行编码所用的哈希表的长度。
N-Gram	3	虽然模型实现了3-Gram，但baseline只用了1-Gram原因是1-Gram的表现更好。
learning rate	0.0001	

注：上表中没有出现的参数默认与上一次取值一致。

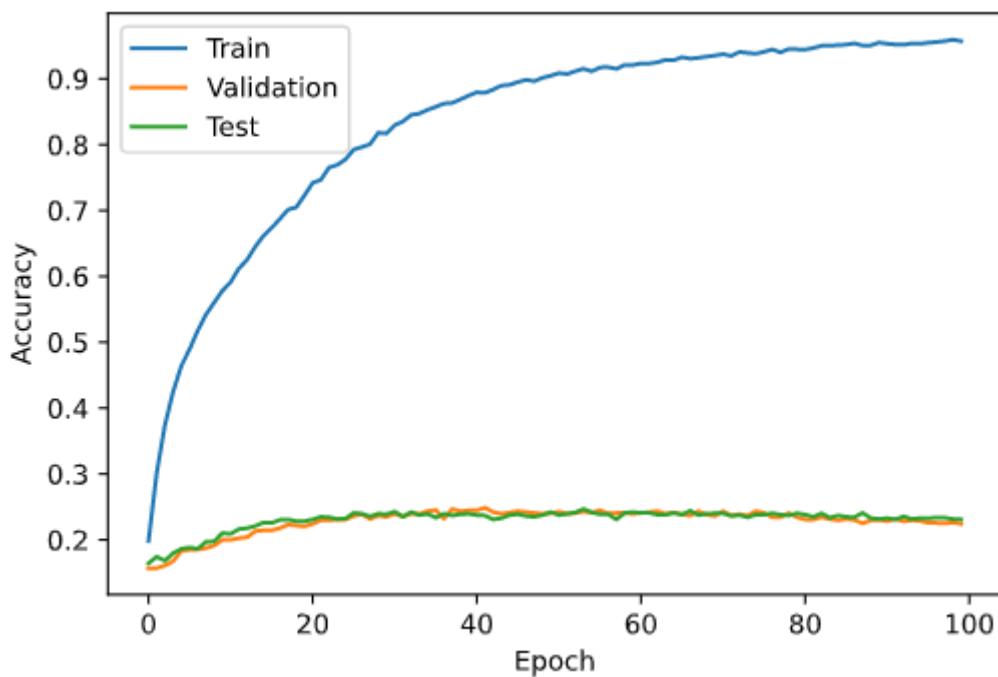
实验结果

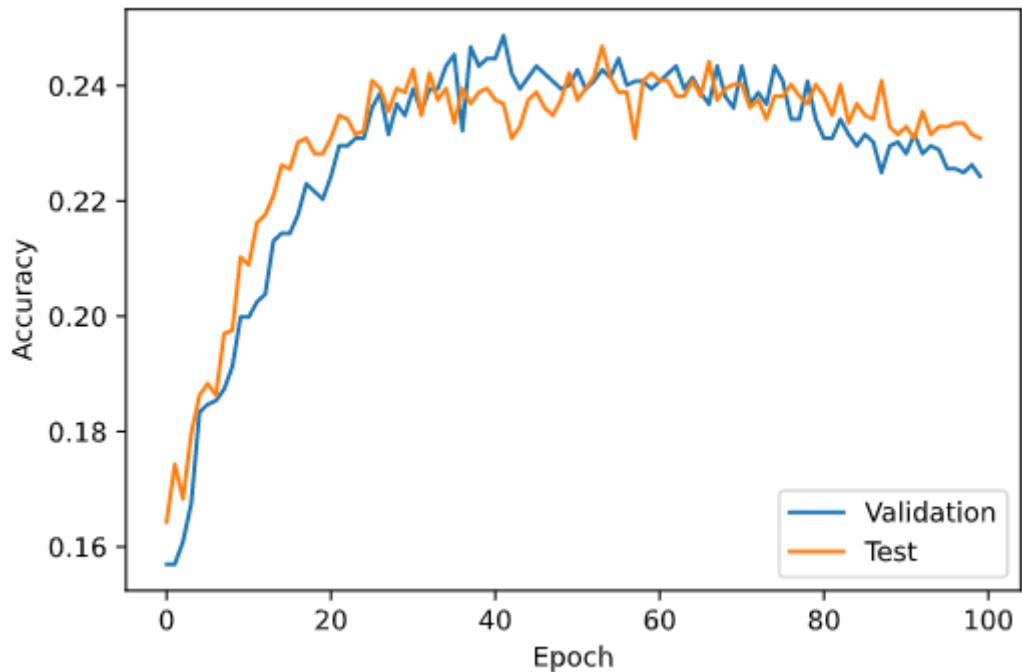
训练过程的average loss变化图：





accuracy变化图：



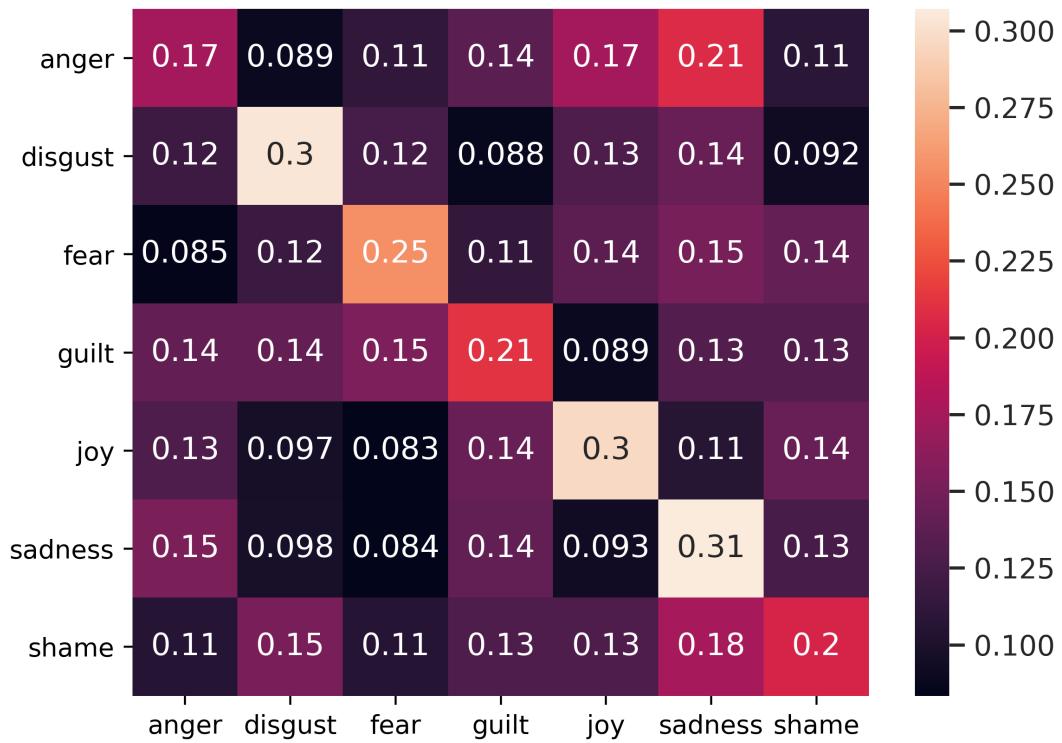


根据提前停止策略，选择在第41个Epoch停止训练，此时Valid Accuracy最大，为0.249。下面对训练得到模型进行测试：

原始输出：

	precision	recall	f1-score	support
anger	0.190	0.174	0.181	213
disgust	0.307	0.304	0.306	217
fear	0.274	0.255	0.264	212
guilt	0.218	0.211	0.215	213
joy	0.284	0.296	0.290	216
sadness	0.252	0.307	0.277	215
shame	0.217	0.203	0.210	217
accuracy			0.250	1503
macro avg	0.249	0.250	0.249	1503
weighted avg	0.249	0.250	0.249	1503

混淆矩阵：



从混淆矩阵可以看出 anger 识别正确率偏低，disgust 的正确率较高。此外，负面情感之间容易混淆，例如 sadness 和 shame，而正面情感和负面情感之间相对不容易混淆，这是符合常识判断的。

注：绘制混淆矩阵的代码参考了[sklearn官网教程](#)。

三项指标：

名称	precision	recall	f1-score
accuracy	\	\	0.250
macro avg	0.249	0.250	0.249
micro avg	0.250	0.250	0.250
weighted avg	0.249	0.250	0.249

模型比较

各个模型经过参数优化后的结果总结于下表

模型	Accuracy
CNN	0.598
CNN + Attention	0.599
CNN + Inception	0.611
RNN	0.596
Baseline	0.250

- 整体来看，CNN + Inception模型比较突出，准确率达到了0.611，Baseline模型达到了0.250，其余模型的准确率都在0.6左右。
- Baseline模型虽然性能一般，但是训练速度极快，这也是FastText最吸引人的地方之一。
- RNN模型的训练时间最长，但只需要较少的Epoch就能收敛。
- CNN系列模型的训练时间介于前二者之间，但是随着feature maps的增加训练时间也会迅速增加。值得注意的是加了Attention之后CNN的提升不明显，但是换成Inception之后却有明显的提高。这有可能是因为在参数个数大体相同的情况下Inception层的特征抽取能力更强。

问题思考与心得体会

1. 如何控制实验训练的停止时间？

为了防止过拟合，本实验的训练停止采用的early stopping的方法。即在Validation Error达到最小时的Epoch (也就是Validation表现不在优化时)停止，用此时的模型作为训练结果。

- early stopping
 - 优点：可以在一定程度上防止过拟合。
 - 缺点：1. 验证集需要数据做支撑，这一定程度上浪费了数据。2. 这种方法对于数据集比较敏感，以至于验证集和测试集不同的数据分布都能显著影响停止的时机，尤其是准确率在很长的一段时间内都不变的情况下(例如本次实验验证集的准确率在20epoch之后基本都在58-59%之间)。3. 这种方法也对优化器比较敏感。不同优化器可能会带来不同的结果。
- 固定迭代次数
 - 优点：能有效控制变量，排除迭代次数变化的干扰。此外，原本用于验证集的数据可以用来训练，更大程度利用数据。
 - 缺点：最大的问题在于如何确定迭代次数。迭代次数少了容易欠拟合，大了容易过拟合。此外，迭代次数还和模型的复杂度，学习率，优化器等诸多方法有关，不容易加以确定。

2. 解决过拟合与欠拟合的常见方法

- 过拟合
 - 交叉验证(主要用于确定是否存在过拟合)
 - 增加训练数据，这里既包括收集更多训练样本，也包括数据增强。即从现有的数据中产生新数据，产生方法包括加入噪声、旋转裁剪(CV)、同义词替换(NLP)、利用对抗生成网络生成数据。
 - 增加权重约束，例如L2正则化。
 - 加入Dropout。
 - 降低模型复杂度，减少参数数量。
- 欠拟合
 - 获取更多的训练数据，主要指收集更多的训练样本。
 - 提高模型的参数数量，或者增加模型的复杂度。
 - 增加训练的迭代次数。

3. 梯度爆炸与梯度消失

- 成因：

梯度爆炸与梯度消失其实来自于同一个问题，即偏导数的累积。

- 从网络深度的角度来看。反向求导时每经过一层就要乘上一个偏导数 $\partial f_n / \partial f_{n-1}$ 。如果层数很多，并且 $\partial f_n / \partial f_{n-1} > 1$ 那么最终结果(即 Δw)就会指数增加，造成梯度爆炸。反之，如果 $\partial f_n / \partial f_{n-1} < 1$ 那么最终结果(即 Δw)就会指数衰减，造成梯度消失。
- 从激活函数的角度看，某些激活函数，尤其是sigmoid的梯度通常很小，那么经过链式求导后就很容易产生梯度消失。

- 解决方案：

- 使用预训练，一定程度缓解参数更新过慢的问题。

- 针对梯度爆炸，可以采用梯度剪切或权重正则化，强制限制梯度的大小。
- 更换成ReLU等激活函数，缓解梯度爆炸与消失。
- 使用BatchNorm进行规范化。
- 改变网络结构，使用残差网络和LSTM等结构。

4. CNN、RNN和MLP的优缺点与适用场景

◦ CNN

- 优点：1. 权值共享策略减少了参数量，相同的权值可以让filter不受相同信号的不同位置的影响，提高泛化性。2. 卷积操作提升了特征抽取能力，对数据预处理的依赖性降低。3. 池化层可以降低网络的空间分辨率，对信号的微小偏移和扭曲不敏感。
- 缺点：1. 随着网络深度增加会产生梯度消失的问题。2. 通过卷积层抽取的特征具有一定的不可解释性。3. 需要依赖大量的数据进行训练才能学习到特征。4. 容易发生过拟合的问题。
- 适用场景：由于CNN的卷积操作天然地适用于提取图像特征，因此CNN主要用于图像处理中，例如目标检测、分类。此外，CNN的卷积具有不变性，也可以应用于序列的处理，例如语音识别、文本分析等。

◦ RNN

- 优点：具有时间维度上的深度模型，能对输入的时序信息进行建模。能够提取复杂的时序关联特征。
- 缺点：1. 训练参数量大，训练速度慢，也容易产生梯度爆炸或梯度消失的问题。2. 缺乏对空间特征的提取能力。
- 适用场景：由于RNN天然地具备时序结构，因此主要用于处理时间序列数据，例如自然语言、语音识别等。

◦ MLP

- 优点：1. 具有比较强的非线性表达能力。2. 具备一定的处理高维度数据的能力。3. 层数加深时可以提取更高级的语义特征。
- 缺点：1. 训练参数量更大，容易过拟合，陷入局部最优。2. 如果减少参数量，模型拟合能力不足，增加参数又会明显降低训练速度。根本问题在于全连接特性——参数随着层数和每层节点数的增加迅速增加。3. 随着深度加深也容易产生梯度爆炸或消失的问题。
- 适用场景：全连接层是最基本的结构，MLP的应用范围比较广阔，可以是语音、图像的处理，也可以是文本的处理。而也正因为MLP结构比较单一，在实际应用中往往会基于MLP加上一些其他的结构再应用。

通过这次实验，我尝试并实践了课上学习的模型，增加了对知识的理解。当我初步实现基本的CNN模型后，其实结果并不理想，而当我将课上学到的结构(Inception Layer)加入到模型中后，结果变得惊人的好！这让我深深感受到知识就是力量！除了利用课内的所学知识，我还查阅了大量文献去尝试不同的模型，包括RCNN等。这不仅开阔了我的视野，让我了解更多这个领域的知识，也锻炼了我的文献阅读能力。我认为这种以实践促学习的方式是非常有益的。此外我通过自己的实践，包括**数据清洗，模型设计，参数调优，模型测试评估，模型改进优化**等，体会到训练出一个好的网络是多么的不容易。这些工作让我的实践能力，理论能力和创新能力都有了一定提高。

参考文献

1. 关于PyTorch实验可复现性的讨论：<https://zhuanlan.zhihu.com/p/109166845>
2. TextBlob库：<https://textblob.readthedocs.io/en/dev/index.html>
3. Kim, Y. (2014). Convolutional Neural Networks for Sentence Classification. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL* (p./pp. 1746-1751), .
4. PyTorch官网教程：https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html
5. word2vec预训练模型仓库：<https://github.com/nishankmahore/word2vec-flask-api>
6. Yadav, R. (2020). Light-Weighted CNN for Text Classification. *arXiv preprint arXiv:2004.07922*.

7. Zhang, Y., & Wallace, B. (2015). A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. *arXiv preprint arXiv:1510.03820*.
8. Zhao, Z., & Wu, Y. (2016). Attention-Based Convolutional Neural Networks for Sentence Classification. In *INTERSPEECH* (pp. 705-709).
9. Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 1-9).
10. Lai, S., Xu, L., Liu, K., & Zhao, J. (2015, February). Recurrent convolutional neural networks for text classification. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 29, No. 1).
11. Joulin, A., Grave, E., Bojanowski, P., & Mikolov, T. (2016). Bag of tricks for efficient text classification. *arXiv preprint arXiv:1607.01759*.
12. sklearn官网教程: https://scikit-learn.org/dev/auto_examples/model_selection/plot_confusion_matrix.html#sphx-glr-auto-examples-model-selection-plot-confusion-matrix-py