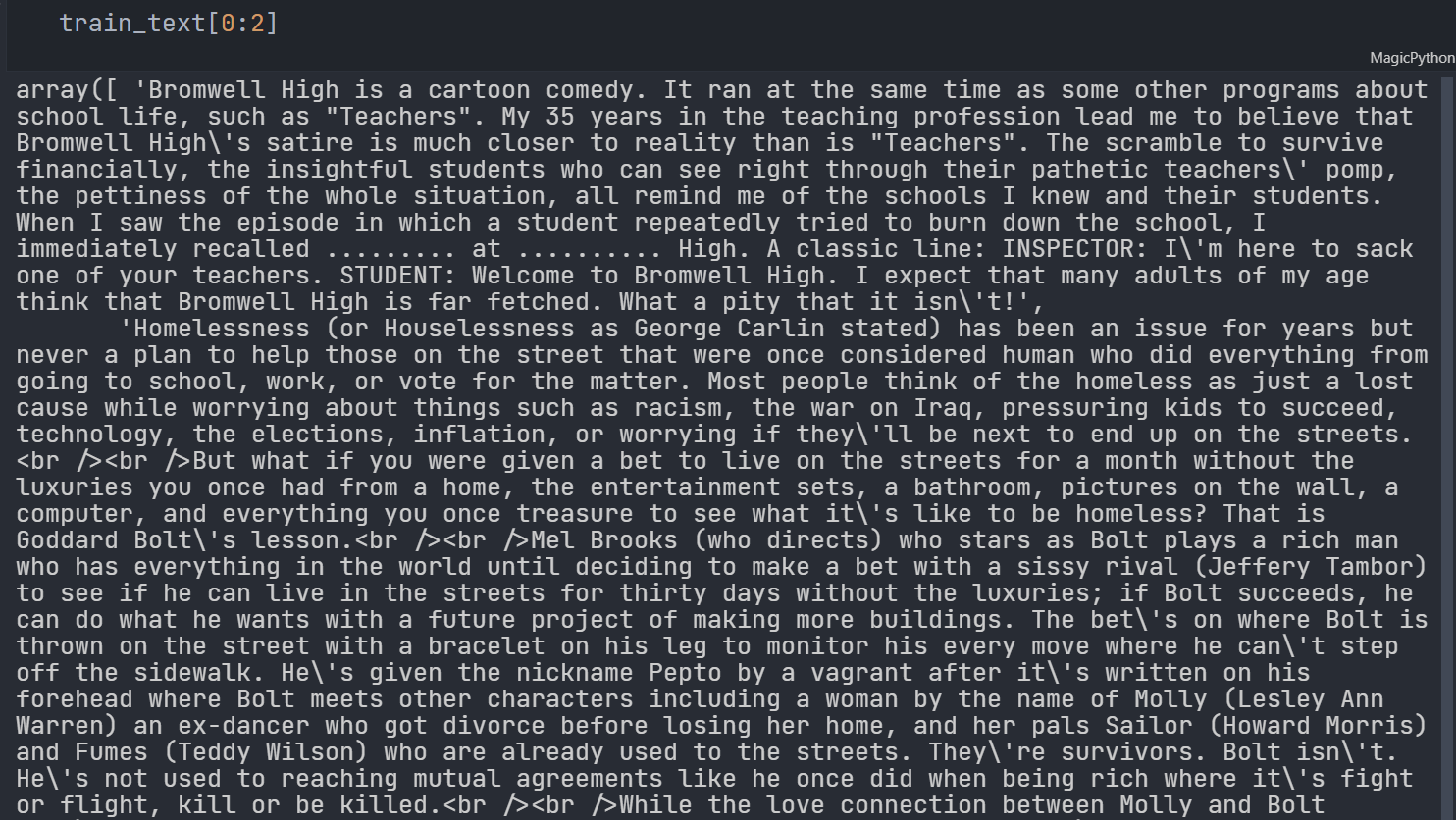
实验4 TextCNN情感分类网络

1. 实验基本思路：

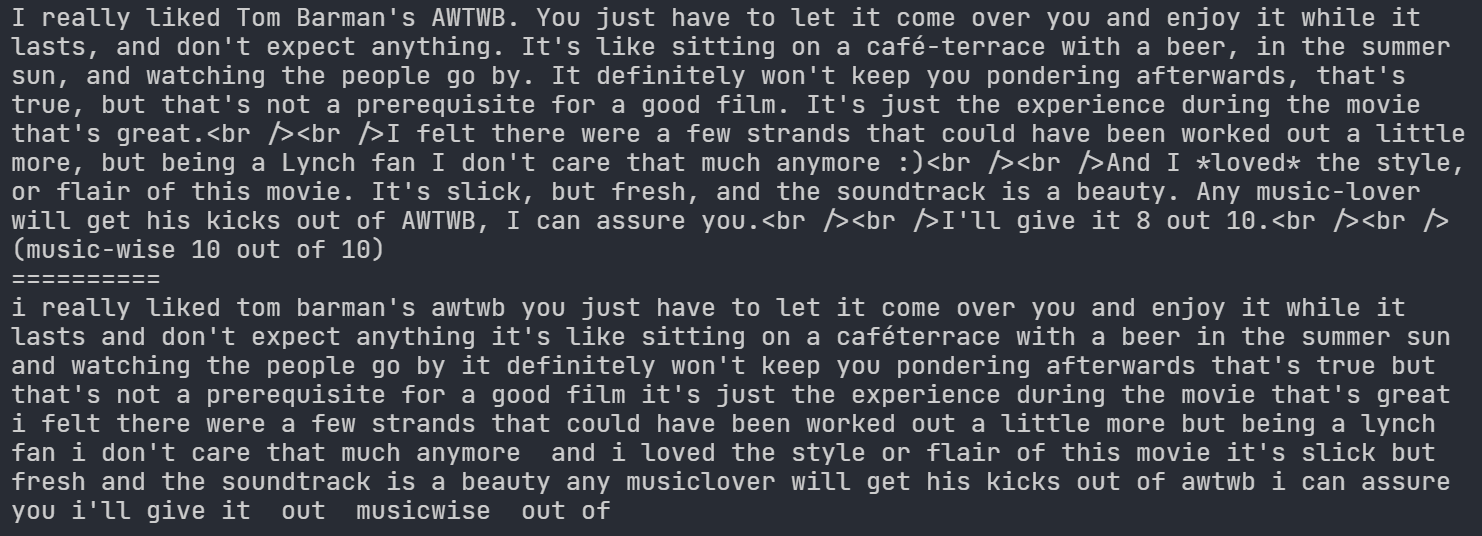
本次实验是使用二维CNN对电影评论数据进行二分类，分为正面评论和负面评论，我们知道二维CNN肯定是要对二维张量进行运算，而文本数据是一维数据且句子长短不同，因此首先要做的事情就是对文本数据进行处理，将其转换为二维的词向量；首先读入所有文本数据，然后清洗掉标点符号等一些不重要的符号，然后产看一下各个评论句子的长度分布，方便后续分析，也绘制出词云，认识一下数据集中那些词汇出现的频率比较高，然后将预处理后的文本数据保存到csv文件中，然后使用torchtext库中的vocab对清洗后的文本数据进行Embedding，也就是词向量嵌入，该过程将文本数据转换为张量表示，还有一种编码方式是one-hot编码，这种编码有一些缺点，就是过于稀疏，对于很长的文本数据来说可能会占用大量的内存空间，且0占得会非常多，会导致编码后的张量维度特别高，影响后续训练，但是Embedding编码是一种分布式的嵌入编码，文本数据平均分散在各个维度，因此不会过于稀疏，而且可以控制编码成指定的维度，当然维度越高自然包含的信息也会多，但是会导致训练困难，因此需要平衡。处理完后搭建网络，在网络输入层进行Embedding，然后就是经过几二维CNN卷积提取特征信息，最后送入全连接层进行二分类。

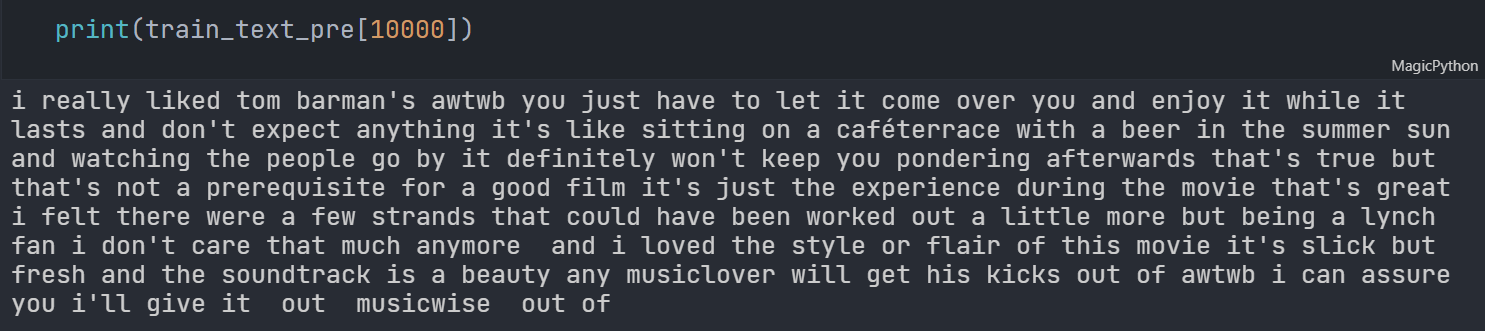
1. 实验结果

部分数据集展示：

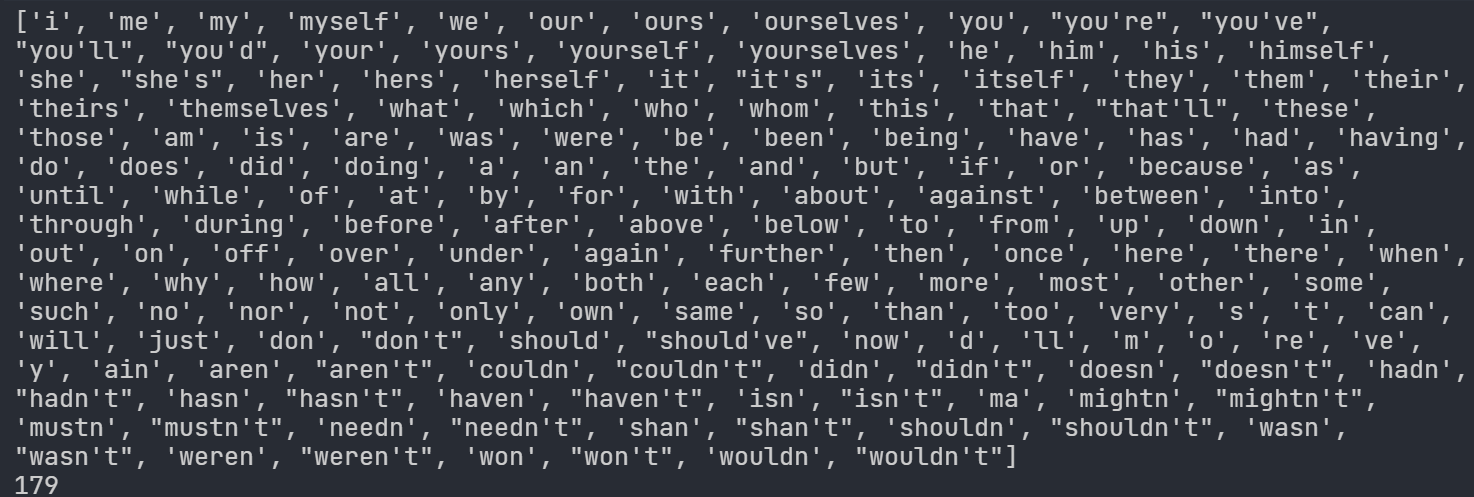


预处理后的数据集:

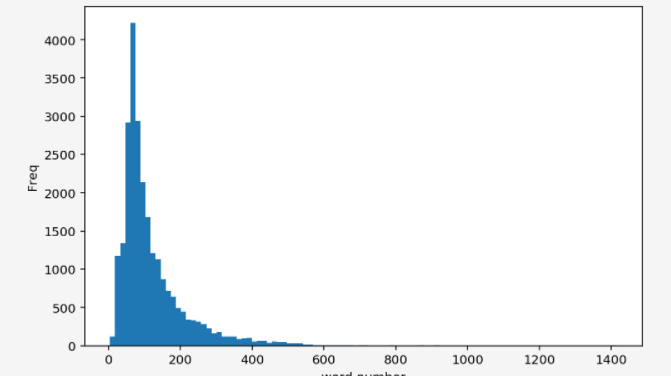




停用词：



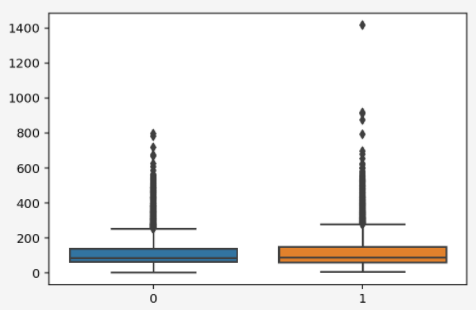
词长度分布直方图：



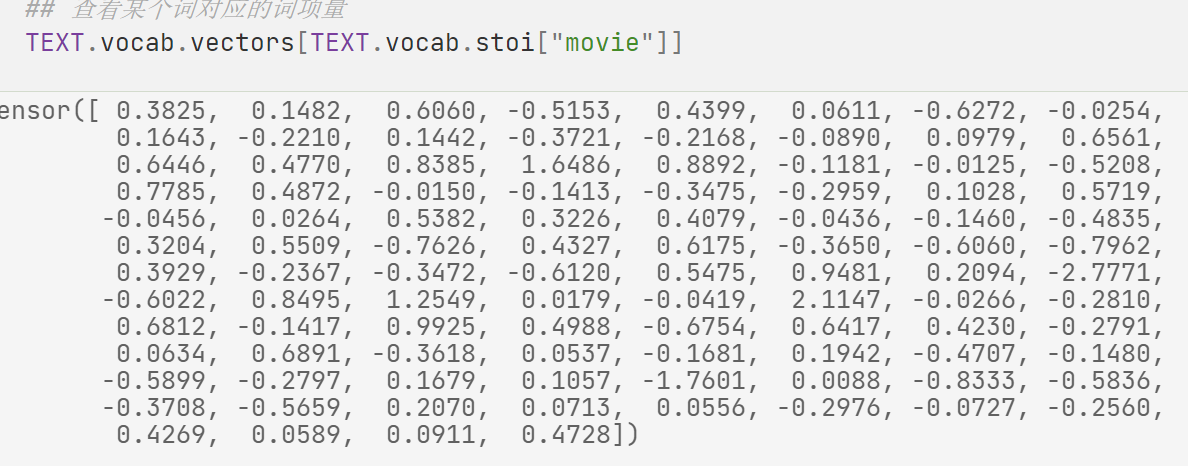
词云:



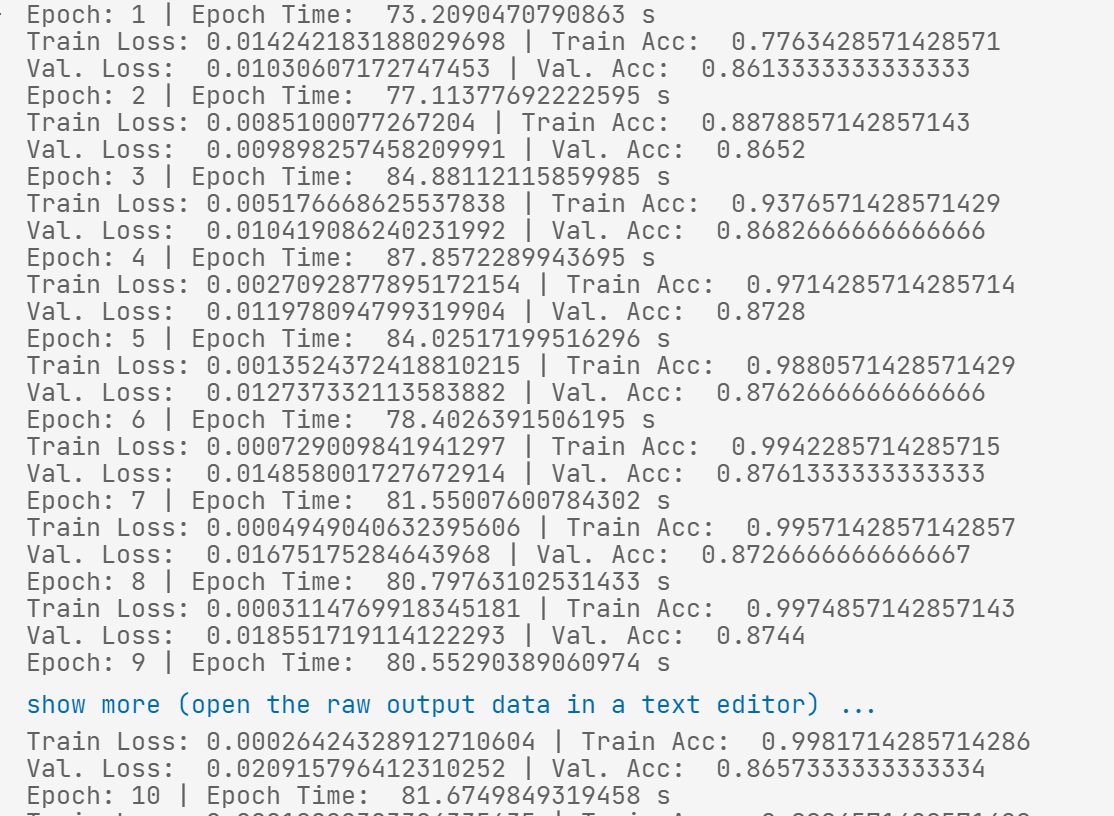
正面词和负面词分布：



查看词向量：



训练过程：



1. 源代码

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import os

import re

import string

import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word\_tokenize

from nltk.stem import PorterStemmer

import seaborn as sns

from wordcloud import WordCloud

import time

import copy

import torch

from torch import nn

import torch.nn.functional as F

import torch.optim as optim

from torchvision import transforms

from torchtext import data

from torchtext.vocab import Vectors, GloVe

## 读取训练数据和测试数据

def load\_text\_data(path):

## 获取文件夹的最后一个字段

text\_data = []

label = []

for dset in ["pos","neg"]:

path\_dset = os.path.join(path,dset)

path\_list = os.listdir(path\_dset)

## 读取文件夹下的pos或neg文件

for fname in path\_list:

if fname.endswith(".txt"):

filename = os.path.join(path\_dset,fname)

with open(filename) as f:

text\_data.append(f.read())

if dset == "pos":

label.append(1)

else:

label.append(0)

## 输出读取的文本和对应的标签

return np.array(text\_data),np.array(label)

## 读取训练集和测试集

train\_path = "./imdb/train"

train\_text,train\_label = load\_text\_data(train\_path)

test\_path = "./imdb/test"

test\_text,test\_label = load\_text\_data(test\_path)

print(len(train\_text),len(train\_label))

print(len(test\_text),len(test\_label))

train\_text[0:2]

string.punctuation[6]

string.punctuation.replace("'","")

## 对文本数据进行预处理

def text\_preprocess(text\_data):

text\_pre = []

for text1 in text\_data:

## 去除指定的字符 <br /><br />

text1 = re.sub("<br /><br />", " ", text1)

## 转化为小写,去除数字,去除标点符号,去除空格

text1 = text1.lower()

text1 = re.sub("\d+", "", text1)

text1 = text1.translate(

str.maketrans("","", string.punctuation.replace("'","")))

text1 = text1.strip()

text\_pre.append(text1)

return np.array(text\_pre)

train\_text\_pre = text\_preprocess(train\_text)

test\_text\_pre = text\_preprocess(test\_text)

print(train\_text[10000])

print("="\*10)

print(train\_text\_pre[10000])

print(train\_text[100])

print("="\*10)

print(train\_text\_pre[100])

print(train\_text\_pre[10000])

## 查看停用词

print(stopwords.words("english"))

print(len(stopwords.words("english")))

## 文本符号化处理,去除停用词，词干化处理

def stop\_stem\_word(datalist,stop\_words,stemer):

datalist\_pre = []

for text in datalist:

text\_words = word\_tokenize(text)

## 去除停用词

text\_words = [word for word in text\_words

if not word in stop\_words]

## 删除带“‘”的词语

text\_words = [word for word in text\_words

if len(re.findall("'",word)) == 0]

## 词干化处理

# text\_words = [stemmer.stem(word) for word in text\_words]

datalist\_pre.append(text\_words)

return np.array(datalist\_pre)

## 文本符号化处理,去除停用词，词干化处理

stop\_words = stopwords.words("english")

stop\_words = set(stop\_words)

stemmer= PorterStemmer()

train\_text\_pre2 = stop\_stem\_word(train\_text\_pre,stop\_words,stemmer)

test\_text\_pre2 = stop\_stem\_word(test\_text\_pre,stop\_words,stemmer)

print(train\_text\_pre[10000])

print("="\*10)

print(train\_text\_pre2[10000])

print(train\_text\_pre2[3])

## 将处理好的文本保存到CSV文件中

texts = [" ".join(words) for words in train\_text\_pre2]

traindatasave = pd.DataFrame({"text":texts,

"label":train\_label})

texts = [" ".join(words) for words in test\_text\_pre2]

testdatasave = pd.DataFrame({"text":texts,

"label":test\_label})

traindatasave.to\_csv("./imdb\_train.csv",index=False)

testdatasave.to\_csv("./imdb\_test.csv",index=False)

print(traindatasave.head())

print(testdatasave.head())

len(texts)

texts[1]

# ### 文本数据可视化

## 将预处理好的文本数据转化为数据表

traindata = pd.DataFrame({"train\_text":train\_text,

"train\_word":train\_text\_pre2,

"train\_label":train\_label})

# testdata = pd.DataFrame({"test\_text":test\_text,

# "test\_word":test\_text\_pre2})

traindata.head()

## 计算每个个影评使用词的数量

train\_word\_num = [len(text) for text in train\_text\_pre2]

traindata["train\_word\_num"] = train\_word\_num

##可视化影评词语长度的分布

plt.figure(figsize=(8,5))

\_ = plt.hist(train\_word\_num,bins=100)

plt.xlabel("word number")

plt.ylabel("Freq")

plt.show()

traindata.head()

## 使用词云可视化两种情感的词频差异

plt.figure(figsize=(16,10))

for ii in np.unique(train\_label):

## 准备每种情感的所有词语

print(ii)

text = np.array(traindata.train\_word[traindata.train\_label == ii])

text = " ".join(np.concatenate(text))

plt.subplot(1,2,ii+1)

## 生成词云

wordcod = WordCloud(margin=5,width=1800, height=1000,

max\_words=500, min\_font\_size=5,

background\_color='white',

max\_font\_size=250)

wordcod.generate\_from\_text(text)

plt.imshow(wordcod)

plt.axis("off")

if ii == 1:

plt.title("Positive")

else:

plt.title("Negative")

plt.subplots\_adjust(wspace=0.05)

plt.show()

## 可视化正面和负面评论用词分布的差异

sns.boxplot(x=train\_label, y=train\_word\_num,)

# #### 数据准备

# Tokenizer：将句子分成单词列表。如果sequential = False，则不应用标记化

#

# Field:存储有关预处理方式的信息的类

## 使用torchtext库进行数据准备

# 定义文件中对文本和标签所要做的操作

"""

sequential=True:表明输入的文本时字符，而不是数值字

tokenize="spacy":使用spacy切分词语

use\_vocab=True: 创建一个词汇表

batch\_first=True: batch悠闲的数据方式

fix\_length=200 :每个句子固定长度为200

"""

## 定义文本切分方法，因为前面已经做过处理，所以直接使用空格切分即可

mytokenize = lambda x: x.split()

TEXT = data.Field(sequential=True, tokenize=mytokenize,

include\_lengths=True, use\_vocab=True,

batch\_first=True, fix\_length=200)

LABEL = data.Field(sequential=False, use\_vocab=False,

pad\_token=None, unk\_token=None)

## 对所要读取的数据集的列进行处理

train\_test\_fields = [

("label", LABEL), # 对标签的操作

("text", TEXT) # 对文本的操作

]

## 读取数据

traindata,testdata = data.TabularDataset.splits(

path="./.", format="csv",

train="imdb\_train.csv", fields=train\_test\_fields,

test = "imdb\_test.csv", skip\_header=True

)

len(traindata),len(testdata)

traindata.fields.items()

## TabularDataset是一个包含Example对象的列表。

ex0 = traindata.examples[0]

print(ex0.label)

print(ex0.text)

## 训练集切分为训练集和验证集

train\_data, val\_data = traindata.split(split\_ratio=0.7)

len(train\_data),len(val\_data)

## 加载预训练的词向量和构建词汇表

## Torchtext使得预训练的词向量的加载变得非常容易。

## 只需提及预训练单词向量的名称（例如glove.6B.50d，fasttext.en.300d等）

vec = Vectors("glove.6B.100d.txt", "./data")

# 将训练集和验证集转化为词项量

## 使用训练集构建单词表，导入预先训练的词嵌入

TEXT.build\_vocab(train\_data,max\_size=20000,vectors = vec)

LABEL.build\_vocab(train\_data)

## 训练集中的前10个高频词

print(TEXT.vocab.freqs.most\_common(n=10))

print("词典的词数:",len(TEXT.vocab.itos))

print("前10个单词:\n",TEXT.vocab.itos[0:10])

## 类别标签的数量和类别

print("类别标签情况:",LABEL.vocab.freqs)

## 查看某个词对应的词项量

TEXT.vocab.vectors[TEXT.vocab.stoi["movie"]]

## 定义一个迭代器，将类似长度的示例一起批处理

BATCH\_SIZE = 32

train\_iter = data.BucketIterator(train\_data,batch\_size = BATCH\_SIZE)

val\_iter = data.BucketIterator(val\_data,batch\_size = BATCH\_SIZE)

test\_iter = data.BucketIterator(testdata,batch\_size = BATCH\_SIZE)

## 获得一个batch的数据，对数据进行内容进行介绍

for step, batch in enumerate(train\_iter):

if step > 0:

break

## 针对一个batch 的数据，可以使用batch.label获得数据的类别标签

print("数据的类别标签:\n",batch.label)

## batch.text[0]是文本对应的标签向量

print("数据的尺寸:",batch.text[0].shape)

## batch.text[1] 对应每个batch使用的原始数据中的索引

print("数据样本数:",len(batch.text[1]))

## 获得一个batch的数据，对数据进行内容进行介绍

for step, batch in enumerate(val\_iter):

if step > 0:

break

## 针对一个batch 的数据，可以使用batch.label获得数据的类别标签

print("数据的类别标签:\n",batch.label)

## batch.text[0]是文本对应的标签向量

print("数据的尺寸:",batch.text[0].shape)

## batch.text[1] 对应每个batch使用的原始数据中的索引

print("数据样本数:",len(batch.text[1]))

# ### 构建网络

class CNN\_Text(nn.Module):

def \_\_init\_\_(self,vocab\_size, embedding\_dim, n\_filters, filter\_sizes, output\_dim,

dropout, pad\_idx):

super().\_\_init\_\_()

"""

vocab\_size:词典大小;embedding\_dim:词向量维度;

n\_filters:卷积核的个数,filter\_sizes:卷积核的尺寸;

output\_dim:输出的维度;pad\_idx:填充的索引

"""

## 对文本进行词项量

self.embedding = nn.Embedding(vocab\_size, embedding\_dim, padding\_idx = pad\_idx)

## 卷积操作

self.convs = nn.ModuleList([

nn.Conv2d(in\_channels = 1, out\_channels = n\_filters,

kernel\_size = (fs, embedding\_dim)) for fs in filter\_sizes

])

## 全连接层和Dropout层

self.fc = nn.Linear(len(filter\_sizes) \* n\_filters, output\_dim)

self.dropout = nn.Dropout(dropout)

def forward(self, text):

#text = [batch size, sent len]

embedded = self.embedding(text)

#embedded = [batch size, sent len, emb dim]

embedded = embedded.unsqueeze(1)

#embedded = [batch size, 1, sent len, emb dim]

conved = [F.relu(conv(embedded)).squeeze(3) for conv in self.convs]

#conved\_n = [batch size, n\_filters, sent len - filter\_sizes[n] + 1]

pooled = [F.max\_pool1d(conv, conv.shape[2]).squeeze(2) for conv in conved]

#pooled\_n = [batch size, n\_filters]

cat = self.dropout(torch.cat(pooled, dim = 1))

#cat = [batch size, n\_filters \* len(filter\_sizes)]

return self.fc(cat)

INPUT\_DIM = len(TEXT.vocab) # 词典的数量

EMBEDDING\_DIM = 100 # 词向量的维度

N\_FILTERS = 100 ## 每个卷积核的个数

FILTER\_SIZES = [3,4,5] ## 卷积和的高度

OUTPUT\_DIM = 1

DROPOUT = 0.5

PAD\_IDX = TEXT.vocab.stoi[TEXT.pad\_token] # 填充词的索引

model = CNN\_Text(INPUT\_DIM, EMBEDDING\_DIM, N\_FILTERS, FILTER\_SIZES, OUTPUT\_DIM, DROPOUT, PAD\_IDX)

model

# ### 网络训练和预测

## 将导入的词项量作为embedding.weight的初始值

pretrained\_embeddings = TEXT.vocab.vectors

model.embedding.weight.data.copy\_(pretrained\_embeddings)

## 将无法识别的词'<unk>', '<pad>'的向量初始化为0

UNK\_IDX = TEXT.vocab.stoi[TEXT.unk\_token]

model.embedding.weight.data[UNK\_IDX] = torch.zeros(EMBEDDING\_DIM)

model.embedding.weight.data[PAD\_IDX] = torch.zeros(EMBEDDING\_DIM)

## Adam优化,二分类交叉熵作为损失函数

optimizer = optim.Adam(model.parameters())

criterion = nn.BCEWithLogitsLoss()

## 定义一个对数据集训练一轮的函数

def train\_epoch(model, iterator, optimizer, criterion):

epoch\_loss = 0;epoch\_acc = 0

train\_corrects = 0;train\_num = 0

model.train()

for batch in iterator:

optimizer.zero\_grad()

pre = model(batch.text[0]).squeeze(1)

loss = criterion(pre, batch.label.type(torch.FloatTensor))

pre\_lab = torch.round(torch.sigmoid(pre))

train\_corrects += torch.sum(pre\_lab.long() == batch.label)

train\_num += len(batch.label) ## 样本数量

loss.backward()

optimizer.step()

epoch\_loss += loss.item()

## 所有样本的平均损失和精度

epoch\_loss = epoch\_loss / train\_num

epoch\_acc = train\_corrects.double().item() / train\_num

return epoch\_loss, epoch\_acc

## 定义一个对数据集验证一轮的函数

def evaluate(model, iterator, criterion):

epoch\_loss = 0;epoch\_acc = 0

train\_corrects = 0;train\_num = 0

model.eval()

with torch.no\_grad(): # 禁止梯度计算

for batch in iterator:

pre = model(batch.text[0]).squeeze(1)

loss = criterion(pre, batch.label.type(torch.FloatTensor))

pre\_lab = torch.round(torch.sigmoid(pre))

train\_corrects += torch.sum(pre\_lab.long() == batch.label)

train\_num += len(batch.label) ## 样本数量

epoch\_loss += loss.item()

## 所有样本的平均损失和精度

epoch\_loss = epoch\_loss / train\_num

epoch\_acc = train\_corrects.double().item() / train\_num

return epoch\_loss, epoch\_acc

## 使用训练集训练模型，验证集测试模型

EPOCHS = 10

best\_val\_loss = float("inf")

best\_acc = float(0)

for epoch in range(EPOCHS):

start\_time = time.time()

train\_loss, train\_acc = train\_epoch(model, train\_iter, optimizer, criterion)

val\_loss, val\_acc = evaluate(model, val\_iter, criterion)

end\_time = time.time()

print("Epoch:" ,epoch+1 ,"|" ,"Epoch Time: ",end\_time - start\_time, "s")

print("Train Loss:", train\_loss, "|" ,"Train Acc: ",train\_acc)

print("Val. Loss: ",val\_loss, "|", "Val. Acc: ",val\_acc)

## 保存效果较好的模型

if (val\_loss < best\_val\_loss) & (val\_acc > best\_acc):

best\_model\_wts = copy.deepcopy(model.state\_dict())

best\_val\_loss = val\_loss

best\_acc = val\_acc

# 将最好模型的参数重新赋值给model

model.load\_state\_dict(best\_model\_wts)

## 模型的保存和导入

torch.save(model,"./textcnnmodel.pkl")

## 导入保存的模型

model = torch.load("./textcnnmodel.pkl")

model

## 使用evaluate函数对测试集进行预测

test\_loss, test\_acc = evaluate(model, test\_iter, criterion)

print("在测试集上的预测精度为:", test\_acc)

# ## 训练好模型的重复使用

import dill

## 保存Field实例

with open("./TEXT.Field","wb")as f:

dill.dump(TEXT,f)

with open("./LABEL.Field","wb")as f:

dill.dump(LABEL,f)

## 导入保存后的Field实例

with open("./TEXT.Field","rb")as f:

TEXT=dill.load(f)

with open("./LABEL.Field","rb")as f:

LABEL=dill.load(f)

print(TEXT.vocab.freqs.most\_common(n=10))

print("词典的词数:",len(TEXT.vocab.itos))

print("前10个单词:\n",TEXT.vocab.itos[0:10])

## 类别标签的数量和类别

print("类别标签情况:",LABEL.vocab.freqs)

## 对所要读取的数据集的列进行处理

train\_test\_fields2 = [

("label", LABEL), # 对标签的操作

("text", TEXT) # 对文本的操作

]

## 读取数据

testdata2 = data.TabularDataset.splits(

path="./.", format="csv",

fields=train\_test\_fields2,

test = "imdb\_test.csv", skip\_header=True

)

test\_iter = data.BucketIterator(testdata,batch\_size = 32)

## 使用evaluate函数对测试集进行预测

test\_loss, test\_acc = evaluate(model, test\_iter, criterion)

print("在测试集上的预测精度为:", test\_acc)