NLP实验报告

实验报告

数据集读取

在 `loadData.py` 中对数据进行加载

```python
import tarfile
import re


def load_imdb(is_training):
    text_set = []
    label_set = []

    # aclImdb_v1.tar.gz解压后是一个目录
    # 我们可以使用python的rarfile库进行解压
    # 训练数据和测试数据已经经过切分，其中训练数据的地址为：
    # ./aclImdb/train/pos/ 和 ./aclImdb/train/neg/，分别存储着正向情感的数据和负向情感的数据
    # 我们把数据依次读取出来，并放到data_set里

    for label in ["pos", "neg"]:
        with tarfile.open("./aclImdb_v1.tar.gz") as tarf:  # 打开压缩包
            path_pattern = "aclImdb/train/" + label + "/.*\.txt$" if is_training \
                else "aclImdb/test/" + label + "/.*\.txt$"
            path_pattern = re.compile(path_pattern)  # 生成正则对象
            tf = tarf.next()  # 显示下一个文件信息
            while tf != None:
                if bool(path_pattern.match(tf.name)):
                    sentence = tarf.extractfile(tf).read().decode()  # 从tf文件中提取文件，读取内容并解码
                    sentence_label = 0 if label == 'neg' else 1
                    text_set.append(sentence)
                    label_set.append(sentence_label)
                tf = tarf.next()

    return text_set, label_set


train_text, train_label = load_imdb(True)
test_text, test_label = load_imdb(False)

```

加载完后，保存到对应的 `json` 文件中去，方便读取而不需要重新生成。在 `spiltData.py` 中完成对数据集的划分和保存

```python
from sklearn.utils import shuffle
from loadData import *
import json

# 合并训练集和测试集的数据和标签
all_text = train_text + test_text
all_label = train_label + test_label

# 随机洗牌，确保数据集的样本分布随机且均匀
all_text, all_label = shuffle(all_text, all_label, random_state=42)

# 计算总样本数量
total_samples = len(all_text)

# 划分比例
train_size = int(0.7 * total_samples)
val_size = int(0.1 * total_samples)

# 划分训练集、验证集和测试集
train_text_split = all_text[:train_size]
train_label_split = all_label[:train_size]
val_text = all_text[train_size:train_size + val_size]
val_label = all_label[train_size:train_size + val_size]
test_text_split = all_text[train_size + val_size:]
test_label_split = all_label[train_size + val_size:]


# 计算评论的平均长度、最大长度和最小长度
def calculate_lengths(text_data):
    lengths = [len(sentence.split()) for sentence in text_data]
    avg_length = sum(lengths) / len(lengths)
    max_length = max(lengths)
    min_length = min(lengths)
    return avg_length, max_length, min_length


avg_length_train, max_length_train, min_length_train = calculate_lengths(train_text_split)
avg_length_val, max_length_val, min_length_val = calculate_lengths(val_text)
avg_length_test, max_length_test, min_length_test = calculate_lengths(test_text_split)

# 保存数据到文件
data = {
```

```
    "train": {"text": train_text_split, "label": train_label_split},
    "validation": {"text": val_text, "label": val_label},
    "test": {"text": test_text_split, "label": test_label_split}
}

with open("data.json", "w") as outfile:
    json.dump(data, outfile)
```

在 `CSVTest.py` 文件中保存到对应的 `csv` 文件中去

```python
import pandas as pd
from readData import *
from gensim.models import Word2Vec


def convert_comments_to_vectors(model, text, labels, output_file):
    # 创建一个空的列表来存储评论向量和标签
    vectors = []
    labelList = []

    # 遍历每条评论，并将其转换为向量
    for comment, label in zip(text, labels):
        words = comment.split()
        sentence_vector = [model.wv[word] for word in words if word in model.wv]
        if sentence_vector:
            sentence_vector = sum(sentence_vector) / len(sentence_vector)
            vectors.append(sentence_vector)
            labelList.append(label)

    # 创建包含向量和标签的DataFrame
    df = pd.DataFrame({"vector": vectors, "label": labelList})

    # 保存DataFrame为CSV文件
    df.to_csv(output_file, index=False)


# 构建Word2Vec模型
model = Word2Vec.load("word2vec.model")
convert_comments_to_vectors(model, train_text, train_label, 'train.csv')
convert_comments_to_vectors(model, test_text, test_label, 'test.csv')
convert_comments_to_vectors(model, val_text, val_label, 'validate.csv')

```

这样子就完成了对数据的处理和准备

word2vec数据构造

在完成了对数据的处理之后，就需要使用 `word2vec` 对数据进行处理，转化为对应的向量，并将模型的相关设置进行保存

```python
from gensim.models import Word2Vec
from loadData import train_text
train_text = [sentence.split() for sentence in train_text]
# train_text: 预处理后的训练文本数据。
# vector_size: 词向量的维度，这里设置为20。
# window: 上下文窗口大小，表示在预测当前单词时考虑前后的上下文词汇数，这里设置为2。
# min_count: 考虑计算的单词的最低词频阈值，出现次数低于此阈值的单词将被忽略，这里设置为3。
# epochs: 训练的迭代次数，这里设置为5。
# negative: 负采样的数量，用于优化模型训练，这里设置为10。
# sg: 训练算法的选择，sg=1表示使用Skip-gram算法，sg=0表示使用CBOW算法
# 调用Word2Vec训练 参数：size: 词向量维度；window: 上下文的宽度，min_count为考虑计算的单词的最低词频阈值
model = Word2Vec(train_text, vector_size=20, window=2, min_count=3, epochs=5, negative=10, sg=1)
# print("has的词向量：\n", model.wv.get_vector('has'))
# print("\n和has相关性最高的前20个词语：")
# print(model.wv.most_similar('has', topn=20))  # 与Nation最相关的前20个词语
# 保存模型
model.save("word2vec.model")
```

数据集构造

在构造数据集的时候，遇到了许多难点，主要的问题在于对张量的维度的处理和匹配

1. 由于在写入 `vector` 的时候，是以字符串的形式进行存储，例如以下形式

```
"[-0.02869788  0.00125298  0.6470118  -0.10332304  0.21247803 -0.12369961
  0.18099312  0.68768823 -0.46420577 -0.08278476  0.6634453   0.12011047
  0.34487027 -0.17034408  0.28443986  0.16248481  1.1805248  -0.28720433
 -0.4377342  -0.5942211 ]"
```

那么就需要对该字符进行处理，最开始的想法是转化为数组，也就是list

`li = self.data.loc[idx, 'vector'].strip('[]').split()` 中 `strip` 能够去掉首尾的方括号， `split` 中会根据若干个空格作为分隔符，转化为list

2. 在转化为 list 后还是不够的，因为需要的不是数组，而是张量 tensor 。因此，就需要对张量进行转化

`vector = torch.from_numpy(np.array(li, dtype=np.float64))`

通过 `np.array(li, dtype=np.float64)` 将 list 转化为 numpy 数组，并统一了数据类型

在通过 `torch.from_numpy` 将相应的 numpy 数组转化为 pytorch 的张量

3. 需要对维度进行统一：将 vector 的维度和 lstm 中的隐藏向量（3维）保持一致，因此需要额外补充一个维度

`vector = vector.unsqueeze(0).to(torch.float32)` 扩展了一个维度，并规定了数据类型

将

```Python
tensor([-0.0878, -0.0270,  0.6165,  0.0309,  0.1692, -0.1830,  0.3833,  0.6098,
        -0.5347, -0.1297,  0.8146, -0.0496,  0.3281, -0.2284,  0.2903,  0.2218,
         1.3887, -0.1735, -0.4927, -0.6715], dtype=torch.float64)
```

转化为了

```Python
tensor([[-0.0878, -0.0270,  0.6165,  0.0309,  0.1692, -0.1830,  0.3833,  0.6098,
         -0.5347, -0.1297,  0.8146, -0.0496,  0.3281, -0.2284,  0.2903,  0.2218,
          1.3887, -0.1735, -0.4927, -0.6715]])
```

```python
import pandas as pd
import numpy as np
import torch
from torch.utils.data.dataset import Dataset


# 注意这里是继承了一个 Dataset 基础类，很多功能是这个基础类就有的，
# 我们只需要按照这个示例实现__init__和__getitem__就行了
class CustomDataset(Dataset):
    def __init__(self, annotations_file, transform=None, target_transform=None):
        # 这里使用 pandas 读取 csv 文件，参数应该是一个文件路径，这个文件里面放的是每一张图片的类型，图片是另外存放的
        # 我们的数据不同，我们是评论的向量和标签放在一起的
        self.data = pd.read_csv(annotations_file)
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.data)  # 会计算CSV文件中的行数，从而确定数据集中样本的数量。

    # 这个方法是关键，需要我们根据 idx 返回每一条对应序号的数据，包括 x 和 y，这个例子中 x 是 image,y 是 label
    # 我们的评论数据中，x 是转换之后的 word vector 数组，y 是极性标签
    # 我们的数据集比这个示例要简单，可以直接在__init__中将所有数据以数组读出来，然后在__getitem__中用 idx 获取就行了
    def __getitem__(self, idx):
        li = self.data.loc[idx, 'vector'].strip('[]').split()
        vector = torch.from_numpy(np.array(li, dtype=np.float64))
        vector = vector.unsqueeze(0).to(torch.float32)
        # unsqueeze() 方法可以在指定的维度上插入一个大小为 1 的维度，从而将一维张量变为二维张量。
        # 解析 CSV 中的向量字符串为 NumPy 数组
        # vector = vector_str.strip('[]').split()
        # print(vector_str,vector)
        label = torch.tensor(self.data.loc[idx, 'label'])
        # label = label.unsqueeze(0)
        # label = torch.nn.functional.one_hot(label, num_classes=2)

        # if label == 1:
        #     label = torch.tensor([1, 0])
        # else:
        #     label = torch.tensor([0, 1])
        label = label.to(torch.float32)  # 和pred = model(x) 进行单位的统一
```

```
        if self.transform:
            vector = self.transform(vector)
        if self.target_transform:
            label = self.target_transform(label)

        return vector, label
```

LSTM模型构建

```python
import torch
from torch import nn
from torch.autograd import Variable


# 按照阈值进行分类
def getBinaryTensor(tensor, boundary=0.5):
    one = torch.ones_like(tensor)
    zero = torch.zeros_like(tensor)
    binary_tensor = torch.where(tensor > boundary, one, zero)
    return binary_tensor


class MyLSTM(nn.Module):
    # input_size:这是输入数据的特征维度。每个句子的词向量表示是输入数据的特征
    # hidden_dim:这是 LSTM 中隐藏状态的维度，也称为 LSTM 单元中的单元数。较大的隐藏状态维度可以捕捉更复杂的模式，但也可能导致更多的计算开销。
    def __init__(self, input_size, hidden_dim, num_layers, output_size):
        super(MyLSTM, self).__init__()
        self.num_layers = num_layers
        self.input_dim = input_size
        self.hidden_dim = hidden_dim
        # 此处 input_size 是我们 word2vec 的词向量的维度;
        # 这里设置了输入的第一个维度为 batchsize，那么在后面构造输入的时候，需要保证第一个维度是 batch size 数量
        self.lstm = nn.LSTM(input_size, hidden_dim, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_size)  # 创建一个线性层对象，将LSTM的输出映射到二分类的结果上.hidden_dim表示输入维度,1表示输出维度（二分类）
        self.softmax = nn.Softmax(dim=output_size)  # 创建一个softmax层对象，用于将输出进行归一化，得到概率分布.dim=1表示对第一维进行softmax操作（即每个样本的输出）。

    def init_hidden(self, batch_size):  # 初始化两个隐藏向量 h0 和 c0
        return (Variable(torch.zeros(self.num_layers, batch_size, self.hidden_dim)),
                Variable(torch.zeros(self.num_layers, batch_size, self.hidden_dim)))

    def forward(self, x):  # 不可以自己显式调用，pytorch 内部自带调用机制
        # input 是传递给 lstm 的输入，它的 shape 应该是（每一个文本的词语数量，batch size，词向量维度）
        # 输入的时候需要将 input 构造成
        self.hidden = self.init_hidden(x.size(0))  # input.size(0)得到 batch_size
```

```
36          # x = x.view(len(x), 1, -1).to(torch.float32)  # 通过使用view直接修改维度，并修改精度
37          # https://blog.csdn.net/weixin_35757704/article/details/118384899
38          # https://blog.csdn.net/qq_19841133/article/details/127824863
39          lstm_out, _ = self.lstm(x, self.hidden)
40          lstm_out = self.fc(lstm_out[:, -1, :])
41          lstm_out = torch.sigmoid(lstm_out)  # Sigmoid激活函数处理后的输出值（0到1之间的概率）代表了模型对样本属于正类的置信度。
42          lstm_out = getBinaryTensor(lstm_out, torch.mean(lstm_out).item())  # 使用平均值作为阈值，将概率映射到0或者1上
43          # lstm_out = self.softmax(lstm_out)
44          lstm_out = lstm_out.squeeze()  # 使用squeeze对维数进行压缩
45          # print(lstm_out, lstm_out.shape)  #
46          return lstm_out  # 查看文档，了解 lstm_out 到底是什么
47
48  # "batch size"（批次大小）是深度学习中一个重要的超参数，它定义了在训练过程中一次迭代所使用的样本数量。
49  # 具体来说，每个迭代步骤中，模型会根据给定的批次大小从训练数据中随机选择一组样本进行前向传播、计算损失和反向传播。
50  # 批次大小的选择会影响训练过程的速度和内存消耗。
51  # 较大的批次大小可以加快训练速度，因为可以并行处理更多的数据，但也可能导致内存占用较大。
52  # 较小的批次大小可能会让训练过程更稳定，但可能会增加每个迭代步骤之间的计算负担。
53
```

这里需要注意的是： `forward` 的返回结果就是预测值，因此需要与 `label` 的维度保持一致，因此使用了 `squeeze` 对维度进行压缩

实验结果

数据集的特征统计

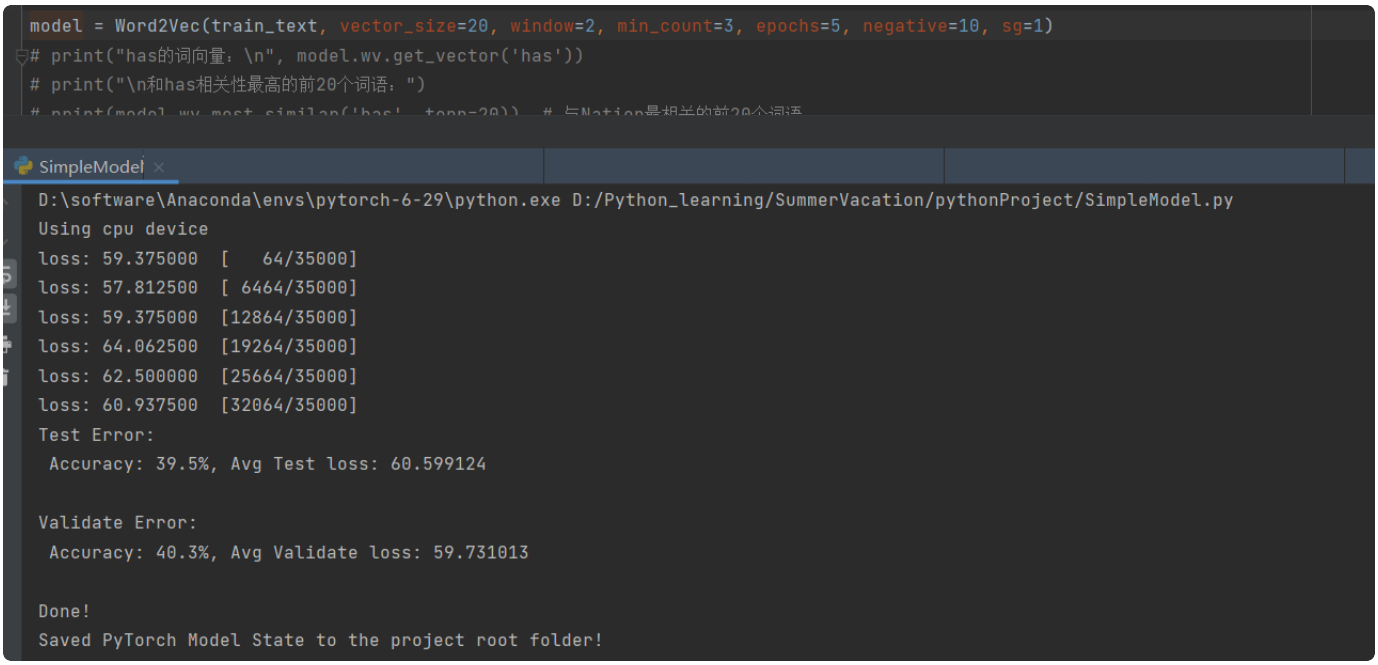
划分方式：训练集 70% 验证集 10% 测试集 20%

```latex
Training Set:
Positive samples: 17493 Negative samples: 17507
Average length: 230.75337142857143 Max length: 2470 Min length: 6

Validation Set:
Positive samples: 2480 Negative samples: 2520
Average length: 235.7036 Max length: 2108 Min length: 4

Test Set:
Positive samples: 5027 Negative samples: 4973
Average length: 230.2961 Max length: 1723 Min length: 10
```

训练结果

window的影响

`window = 2`



`window = 20`

```
#调用Word2Vec训练 参数：size: 词向量维度；window: 上下文的宽度，min_count为考虑计算的单词的最低词频阈值
model = Word2Vec(train_text, vector_size=20, window=20, min_count=3, epochs=5, negative=10, sg=1)
# print("has的词向量：\n", model.wv.get_vector('has'))
# print("\n和has相关性最高的前20个词语：")
```

```
SimpleModel
Using cpu device
loss: 48.437500  [   64/35000]
loss: 35.937500  [ 6464/35000]
loss: 45.312500  [12864/35000]
loss: 43.750000  [19264/35000]
loss: 51.562500  [25664/35000]
loss: 43.750000  [32064/35000]
Test Error:
 Accuracy: 51.0%, Avg Test loss: 49.044586

Validate Error:
 Accuracy: 51.2%, Avg Validate loss: 49.050633

Done!
Saved PyTorch Model State to the project root folder!
```

理论上：`window` 代表上下文窗口大小，表示在预测当前单词时考虑前后的上下文词汇数，当 `window` 越大的时候，准确率也就会越高

epochs的影响

`epochs = 10`

```
model = Word2Vec(train_text, vector_size=20, window=50, min_count=3, epochs=10, negative=10, sg=1
# print("has的词向量：\n", model.wv.get_vector('has'))
# print("\n和has相关性最高的前20个词语：")
# print(model.wv.most_similar('has', topn=20))  # 与Nation最相关的前20个词语
# 保存模型
model.save("word2vec.model")
```

```
SimpleModel
D:\software\Anaconda\envs\pytorch-6-29\python.exe
 D:/Python_learning/SummerVacation/pythonProject/SimpleModel.py
Using cpu device
loss: 50.000000  [   64/35000]
loss: 51.562500  [ 6464/35000]
loss: 51.562500  [12864/35000]
loss: 57.812500  [19264/35000]
loss: 46.875000  [25664/35000]
loss: 51.562500  [32064/35000]
Test Error:
 Accuracy: 50.0%, Avg Test loss: 50.029857

Validate Error:
 Accuracy: 50.8%, Avg Validate loss: 49.208861
```

`epochs = 50`

```
model = Word2Vec(train_text, vector_size=20, window=50, min_count=3, epochs=50, negative=10, sg=1)
# print("has的词向量：\n", model.wv.get_vector('has'))
# print("\n和has相关性最高的前20个词语：")
# print(model.wv.most_similar('has', topn=20))  # 与Nation最相关的前20个词语
# 保存模型
model.save("word2vec.model")
```

```
SimpleModel
D:\software\Anaconda\envs\pytorch-6-29\python.exe
 D:/Python_learning/SummerVacation/pythonProject/SimpleModel.py
Using cpu device
loss: 56.250000  [   64/35000]
loss: 56.250000  [ 6464/35000]
loss: 65.625000  [12864/35000]
loss: 67.187500  [19264/35000]
loss: 70.312500  [25664/35000]
loss: 54.687500  [32064/35000]
Test Error:
 Accuracy: 43.1%, Avg Test loss: 56.876990

Validate Error:
 Accuracy: 44.2%, Avg Validate loss: 55.992880
```

可以看到当 `epochs` 增加的时候，准确度并没有提升，甚至开始下降了，这也说明了本次实验的结果不太稳定

vector_size的影响

`vector_size = 10`

```
# 调用Word2Vec训练 参数：size: 词向量维度；window: 上下文的宽度，min_count为考虑计算的单词的最低词频阈值
model = Word2Vec(train_text, vector_size=10, window=50, min_count=3, epochs=50, negative=10, sg=1)
# print("has的词向量：\n", model.wv.get_vector('has'))
# print("\n和has相关性最高的前20个词语：")
# print(model.wv.most_similar('has', topn=20))  # 与Nation最相关的前20个词语
# 保存模型
```

```
SimpleModel
loss: 57.812500  [12864/35000]
loss: 48.437500  [19264/35000]
loss: 65.625000  [25664/35000]
loss: 48.437500  [32064/35000]
Test Error:
 Accuracy: 45.9%, Avg Test loss: 53.951035

Validate Error:
 Accuracy: 46.6%, Avg Validate loss: 53.678797

Done!
Saved PyTorch Model State to the project root folder!
```

`vector_size = 200`

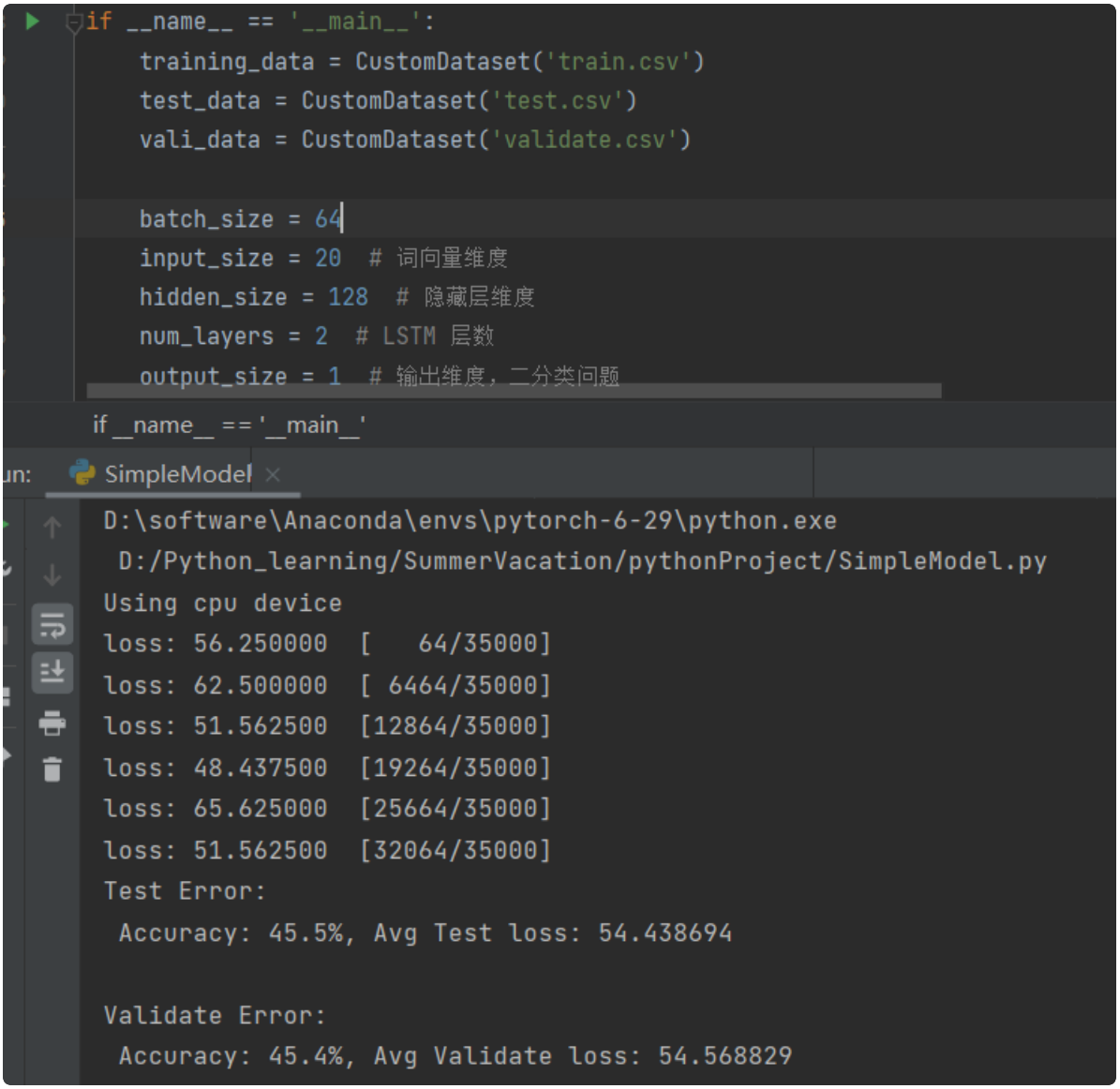
```
model = Word2Vec(train_text, vector_size=200, window=50, min_count=3, epochs=50, negative=10, sg=1)
# print("has的词向量：\n", model.wv.get_vector('has'))
# print("\n和has相关性最高的前20个词语：")
# print(model.wv.most_similar('has', topn=20))  # 与Nation最相关的前20个词语
# 保存模型
```

```
SimpleModel
loss: 56.250000  [12864/35000]
loss: 51.562500  [19264/35000]
loss: 51.562500  [25664/35000]
loss: 42.187500  [32064/35000]
Test Error:
 Accuracy: 43.9%, Avg Test loss: 56.090764

Validate Error:
 Accuracy: 44.6%, Avg Validate loss: 55.636867

Done!
Saved PyTorch Model State to the project root folder!
```

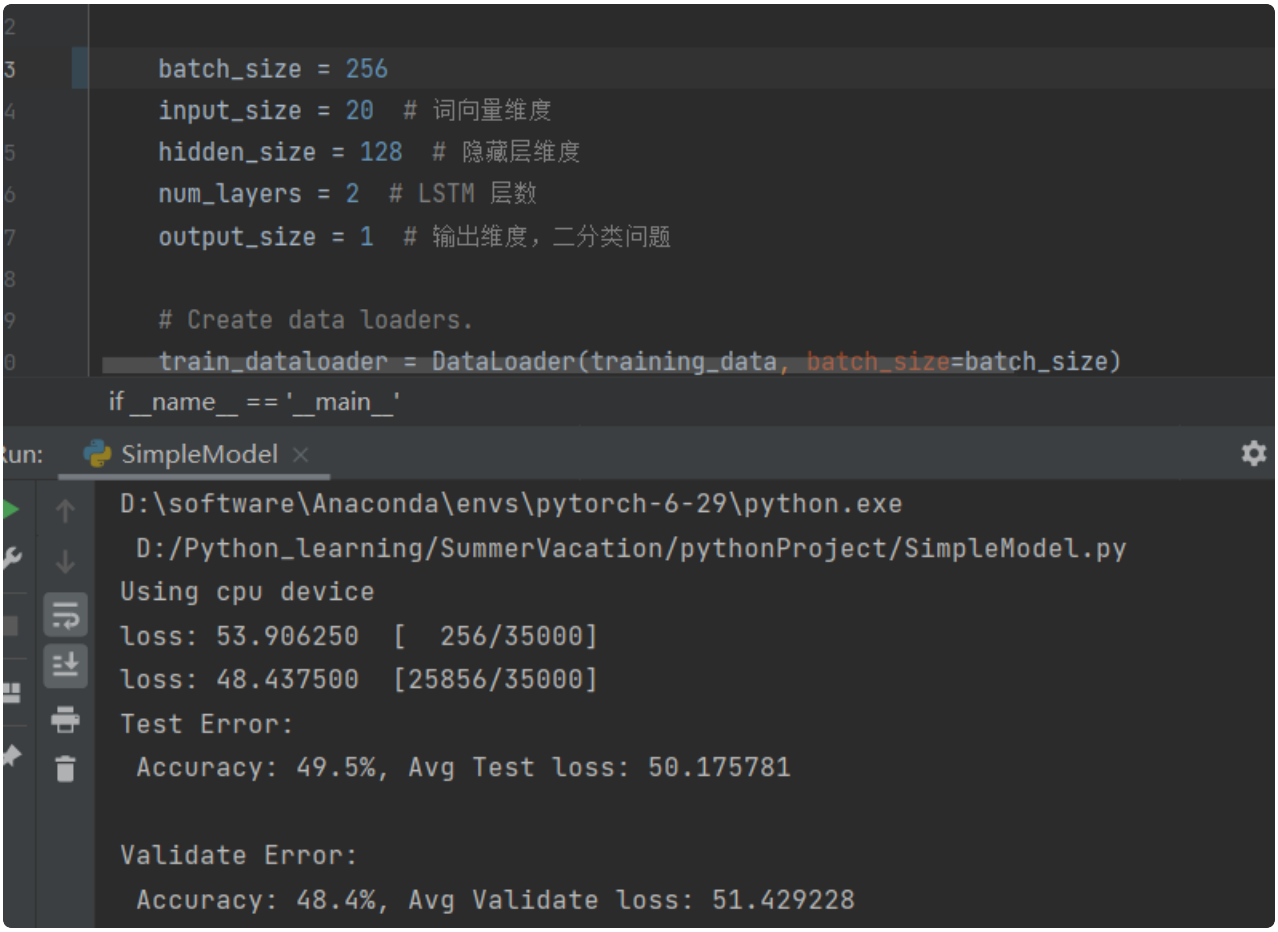
可以看到 `vector_size` 的影响也并不明显

batch_size的影响

`batch_size = 64`



`batch_size = 256`



可以看到由于一次性读入的样本数量的增加，提高了训练的速度，同时也一定程度上提高了准确度

num_layers的影响

`num_layers = 16`

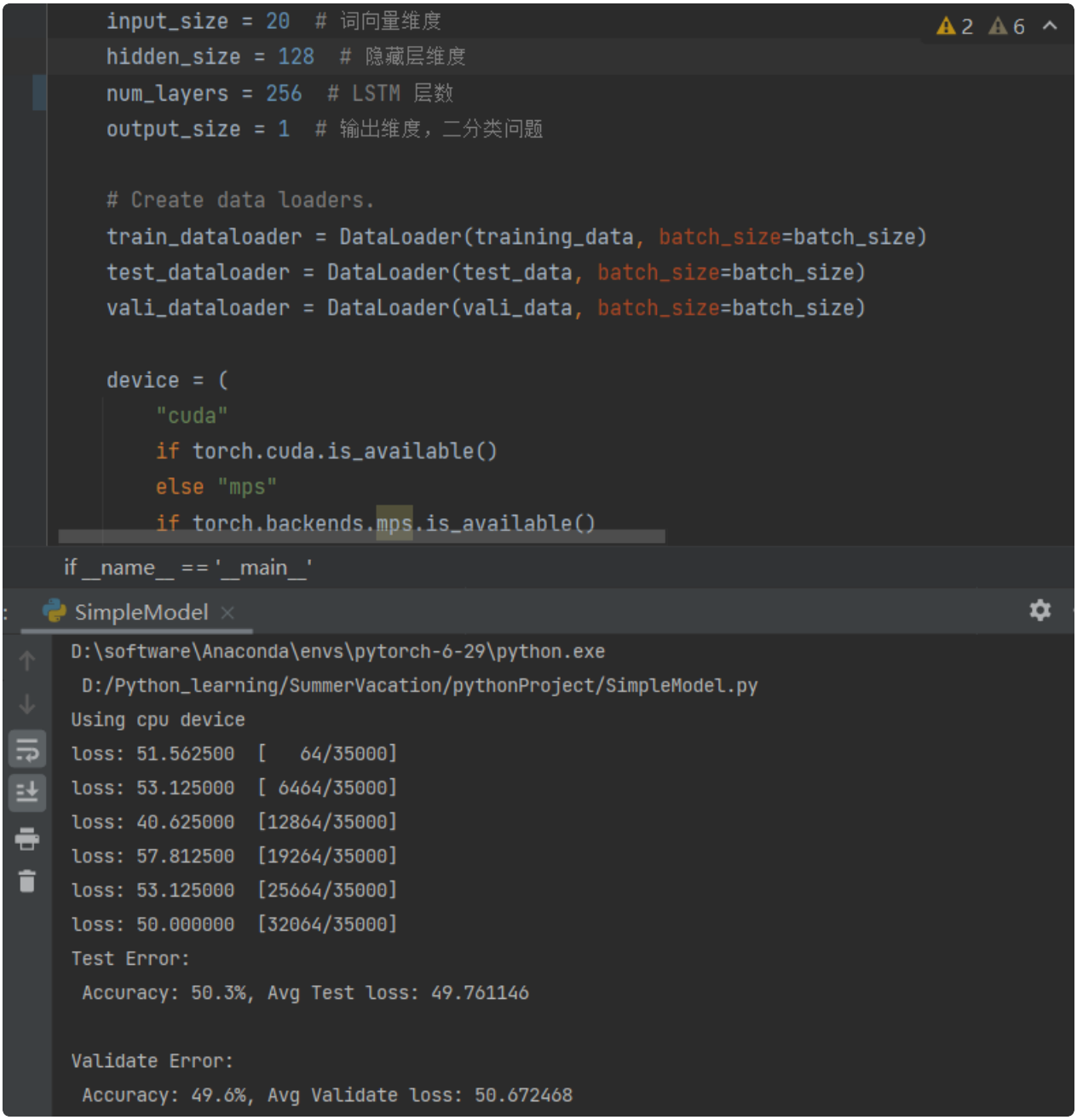
```python
    num_layers = 16  # LSTM 层数
    output_size = 1  # 输出维度，二分类问题

    # Create data loaders.
    train_dataloader = DataLoader(training_data, batch_size=batch_size)
    test_dataloader = DataLoader(test_data, batch_size=batch_size)
    vali_dataloader = DataLoader(vali_data, batch_size=batch_size)

    device = (
        "cuda"
        if torch.cuda.is_available()
        else "mps"
        if torch.backends.mps.is_available()
```

```
if __name__ == '__main__'
```

SimpleModel

```
D:\software\Anaconda\envs\pytorch-6-29\python.exe
 D:/Python_learning/SummerVacation/pythonProject/SimpleModel.py
Using cpu device
loss: 48.437500  [   64/35000]
loss: 46.875000  [ 6464/35000]
loss: 59.375000  [12864/35000]
loss: 42.187500  [19264/35000]
loss: 46.875000  [25664/35000]
loss: 50.000000  [32064/35000]
Test Error:
 Accuracy: 49.7%, Avg Test loss: 50.238854

Validate Error:
 Accuracy: 50.4%, Avg Validate loss: 49.327532
```

`num_layers = 256`



由于LSTM层数的增加，一个明显的变化是训练的时间延长，但是准确度没有明显的变化

总结

1. 本次实验无法选到合适的超参数，来使得准确度最高。因为没有实现对同一个参数进行多次评估，取平均值的情况。因此，在改变参数的过程中，无法确定准确度的提高是由于参数的影响还是受到本身随机性的影响
2. 本次实验的主要难点，或者说花费时间最多的点在于对 `tensor` 的维度上的理解，如何对维度进行扩展或是压缩，选择什么维度对我来说都是比较困难的点
3. 本次实验的一个缺陷是在 `forward` 函数中，对产生的 `tensor` 映射到0或者1上时，是以平均值

作为分类的阈值，这种方式是否恰到还需要更进一步的研究