实验五实验报告

1.实验介绍：

本实验使用pytorch完成SST数据集上的情感分类任务，主要网络结构是LSTM

2.代码分析：

- 载入必要的库并载入数据

```
import os
#os.environ["CUDA_VISIBLE_DEVICES"] = "3"
import torch
from torch import nn,functional
from torchtext import data
from torchtext import datasets
from torchtext.vocab import Vectors, GloVe, CharNGram, FastText
import numpy as np  # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import matplotlib.pyplot as plt
```

- 载入测试数据集并且划分训练集，验证集和测试集

```
################################
# DataLoader
################################

# set up fields
TEXT = data.Field()
LABEL = data.Field(sequential=False,dtype=torch.long)

# make splits for data
# DO NOT MODIFY: fine_grained=True, train_subtrees=False
train, val, test = datasets.SST.splits(
    TEXT, LABEL, fine_grained=True, train_subtrees=False)

# print information about the data
print('train.fields', train.fields)
print('len(train)', len(train))
print('vars(train[0])', vars(train[0]))

# build the vocabulary
# you can use other pretrained vectors, refer to
https://github.com/pytorch/text/blob/master/torchtext/vocab.py
TEXT.build_vocab(train, vectors=Vectors(name='vector.txt', cache='./data'))
```

```
LABEL.build_vocab(train)
# We can also see the vocabulary directly using either the stoi (string to int) or itos
(int to string) method.
print(TEXT.vocab.itos[:10],"\n")
print(LABEL.vocab.stoi,"\n")
print(TEXT.vocab.freqs.most_common(20),"\n")

# print vocab information
print('len(TEXT.vocab)', len(TEXT.vocab),"\n")
print('TEXT.vocab.vectors.size()', TEXT.vocab.vectors.size(),"\n")
```

- 载入预先训练好的embedding参数

```
pretrained_embeddings = TEXT.vocab.vectors

print(pretrained_embeddings.shape)
```

- 定义网络 主要结构是LSTM

  主要架构是先经过embedding层进行词嵌入，随后经过LSTM层编码，最后通过一个全连接层解码再通过softmax层归一化输出概率。

```
class SentimentNet(nn.Module):
    def __init__(self,embed_size, num_hiddens, num_layers,
                 bidirectional, labels, **kwargs):
        super(SentimentNet, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.num_layers = num_layers
        self.bidirectional = bidirectional
        self.embedding = nn.Embedding.from_pretrained(pretrained_embeddings)
        self.embedding.weight.requires_grad = False
        self.encoder = nn.LSTM(input_size=embed_size, hidden_size=self.num_hiddens,
                               num_layers=num_layers, bidirectional=self.bidirectional,
                               dropout=0.5)
        if self.bidirectional:
            self.decoder = nn.Linear(num_hiddens * 4, labels)
        else:
            self.decoder = nn.Linear(num_hiddens * 2, labels)
        self.softmax = nn.Softmax(dim=1)
        self.dropout = nn.Dropout(0.5)


    def forward(self, inputs):
        embeddings = self.dropout(self.embedding(inputs))
        states, hidden = self.encoder(embeddings)
        encoding = torch.cat([states[0], states[-1]], dim=1)

        out = self.dropout(self.decoder(encoding))
        out = self.softmax(out)
```

```
        return out
```

- 定义网络参数并实例化model

  batch_size=128，采用了bidirectional结构

  ```
  embed_size = 300
  num_hiddens = 100
  num_layers = 2
  bidirectional = True
  labels = 5
  batch_size=128
  device = torch.device('cuda:7')
  #device = torch.device('cpu')

  model = SentimentNet(embed_size=embed_size,
                     num_hiddens=num_hiddens, num_layers=num_layers,
                     bidirectional=bidirectional,labels=labels)
  model.to(device)
  criterion = nn.CrossEntropyLoss().to(device)
  optimizer = torch.optim.SGD(model.parameters(),lr=0.1,
  momentum=0.9,weight_decay=1e-7)
  ```

- 训练和验证 采用150个epoch

```
train_iter, val_iter, test_iter = data.BucketIterator.splits((train, val, test),

batch_size=batch_size,shuffle=True)
epochs = 150
train_losses, validation_losses,validation_accs = [] ,[],[]
for epoch in range(epochs):
    model.zero_grad()
    model.train()
    running_loss=0
    acc=0
    for batch in train_iter:
        text=batch.text.to(device)
        label=batch.label-1
        #label=label>=3
        #label=label.long()
        label=label.to(device)

        optimizer.zero_grad()
        output = model(text)
        loss = criterion(output,label)
        loss.backward()
        optimizer.step()
```

```
        running_loss += loss.item()
        acc+=torch.sum(torch.argmax(output,1)==label).cpu().item()/128.0

    with torch.no_grad():
        model.eval()
    val_loss=0
    val_acc=0
    for val_batch in val_iter:
        val_text=val_batch.text.to(device)
        val_label=val_batch.label-1
        #val_label=val_label>=3
        #val_label=val_label.long()
        val_label=val_label.to(device)
        val_output = model.forward(val_text)

        val_loss += criterion(val_output,val_label).item()
        val_acc += torch.sum(torch.argmax(val_output,1)==val_label).cpu().item()/128.0


    train_losses.append(running_loss/len(train_iter))
    validation_losses.append(val_loss/len(val_iter))
    validation_accs.append(val_acc/len(val_iter))

    print("Epoch: {}/{}.. ".format(epoch+1, epochs),
          "Train Loss: {:.3f}.. ".format(running_loss/len(train_iter)),
          "Train_Acc: {:.3f}.. ".format(acc/len(train_iter)),
          "Val Loss: {:.3f}.. ".format(val_loss/len(val_iter)),
          "Val_Acc: {:.3f}".format(val_acc/len(val_iter)))
```

- 绘图

```
#plot image
plt.plot(train_losses, label='Training loss')
plt.plot(validation_losses, label='Validation loss')
plt.plot(validation_accs,label='Validation Accuracy')
plt.legend(frameon=False)
```

- 测试模型

```
with torch.no_grad():
    model.eval()
test_loss=0
test_acc=0
for batch in test_iter:
    text=batch.text.to(device)
    label=batch.label-1
    #label=label>=3
    #label=label.long()
    label=label.to(device)
```

```
        output = model.forward(text)
        test_loss += criterion(output,label).item()

        #print(torch.sum(torch.argmax(output,1)==label).cpu().item()/len(label))
        test_acc += torch.sum(torch.argmax(output,1)==label).cpu().item()/128.0

    print("Test Loss: {:.3f}.. ".format(test_loss/len(test_iter)),
            "Test Accuracy: {:.3f}".format(test_acc/len(test_iter)))
```

3.结果分析：

本实验中超参数的选择如下：

batch_size = 128 总epoch次数150次

SGD学习率为 0.1 动量0.9 weight_decay为1e-7

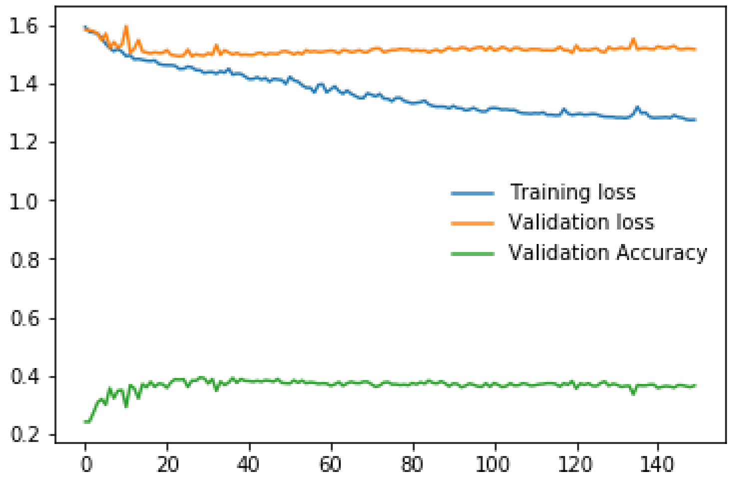
在网络模型的选择中，开始我未对embedding层和fc层采用dropout，发现结果如下：

```
Train:

Epoch: 148/150..  Train Loss: 1.275..  Train_Acc: 0.626..  Val Loss: 1.519..  Val_Acc: 0.364
Epoch: 149/150..  Train Loss: 1.274..  Train_Acc: 0.628..  Val Loss: 1.518..  Val_Acc: 0.361
Epoch: 150/150..  Train Loss: 1.275..  Train_Acc: 0.627..  Val Loss: 1.516..  Val_Acc: 0.366

Test Loss: 1.493..  Test Accuracy: 0.366
```

训练过程如下图：



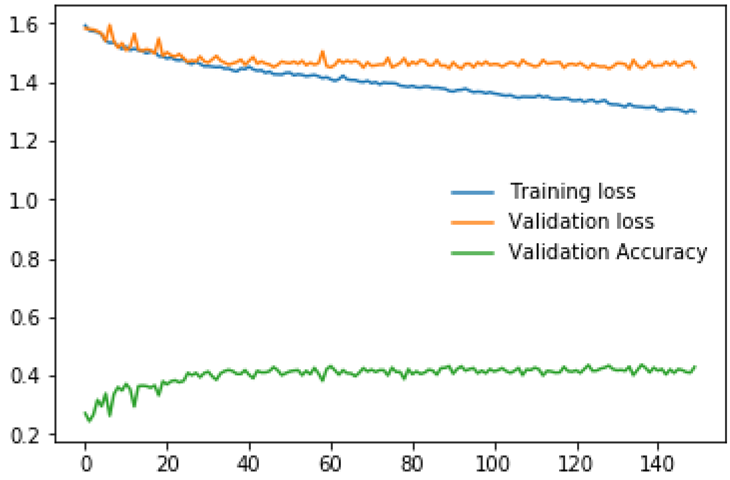
可以发现，训练集准确度能达到63%左右，但验证集和训练集准确度只有36%左右，出现了较严重的过拟合。

因此在此基础上我对embedding层采用了比例为0.5的dropout，结果如下：

```
Epoch: 148/150..  Train Loss: 1.295..  Train_Acc: 0.605..  Val Loss: 1.467..  Val_Acc: 
0.411
Epoch: 149/150..  Train Loss: 1.304..  Train_Acc: 0.594..  Val Loss: 1.469..  Val_Acc: 
0.410
Epoch: 150/150..  Train Loss: 1.298..  Train_Acc: 0.597..  Val Loss: 1.450..  Val_Acc: 
0.429

Test Loss: 1.447..  Test Accuracy: 0.433
```

可以发现,虽然准确度有所提升，但依旧存在较为严重的过拟合现象。

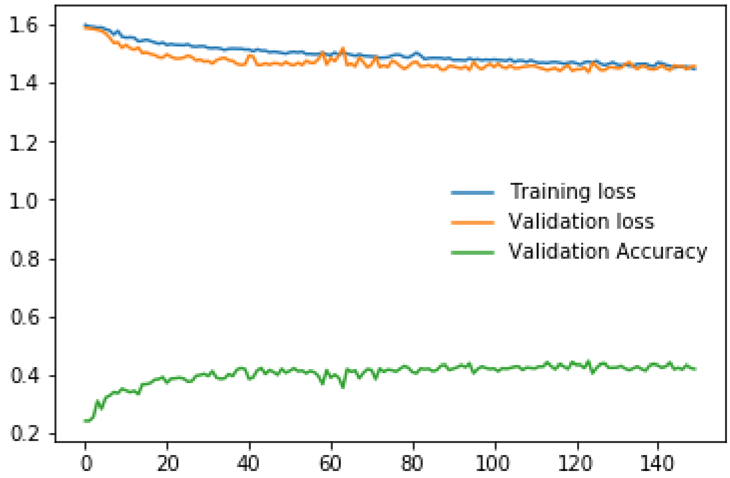


最后我采取了对LSTM,embedding层和fc层都采用dropout的策略，结果如下:

```
Epoch: 148/150..  Train Loss: 1.432..  Train_Acc: 0.467..  Val Loss: 1.440..  Val_Acc: 
0.451
Epoch: 149/150..  Train Loss: 1.431..  Train_Acc: 0.466..  Val Loss: 1.441..  Val_Acc: 
0.450
Epoch: 150/150..  Train Loss: 1.433..  Train_Acc: 0.467..  Val Loss: 1.441..  Val_Acc: 
0.453

Test Loss: 1.443..  Test Accuracy: 0.456
```

可以看到，虽然准确率只能达到0.456这样的水准，但过拟合的问题已经不再显著。考虑提升模型效果，可以采取改变网络结构的方法。



最后，我尝试了简单的二分类问题，即只考虑情感的N/P属性，最后的测试结果达到了0.88左右，可见该模型对于二分类的任务可以较好地完成，但对于五分类的精细任务完成效果不佳。

4.结果分析：

本次试验整体上是在与过拟合作斗争的过程。由于文本训练数据只有几千条，相对较少，因此比较容易过拟合。而最终的结果对于五分类的问题效果也并不理想，只能达到45%这样的准确率。但是可以发现对于简单的积极/消极的情感二分类问题，准确率能够达到88%左右，因此该模型对于细粒度较高的任务不太理想。而在斯坦福的SST官方上我也进行了一些测试，可以发现当输入句子比较简单，情感倾向比较明显时，结果较为理想。但是对于不少表达比较含蓄或者含有转折的句子，效果往往也不甚理想。

由此可见，NLP问题目前还有很大的发展空间，希望能够在这个方向上持续钻研，了解学习更多。