**Report2 -CNN Model**

目录

[重要代码 1](#_Toc42530887)

[实验结果 4](#_Toc42530888)

## 重要代码

glove预训练的embedding读取

下载下来的glove文件形式如下



每一行为一个词，以及其所对应的向量。我将词存入words, 将向量存入vectors, 并通过python的pickle库以及numpy.save来保存这些数据结构。（如果每次任务运行时再读取glove embedding, 太浪费时间了）

文件在task2-RNN/Glove-read.py

with open('glove\_wordvec/glove6B200d.txt', 'rb') as f:

for line in f:

line = line.decode().split()

word = line[0]

words.append(word)

Word\_Index[word] = idx

vector = np.array(line[1:])

vectors[idx, :] = vector

idx = idx + 1

vectors = vectors.reshape((400000, 200)).astype('float')

print(vectors[0:4, :])

with open('glove\_wordvec/words200d.pickle', 'wb') as handle:

pickle.dump(words, handle, protocol=pickle.HIGHEST\_PROTOCOL)

with open('glove\_wordvec/Word\_Index200d.pickle', 'wb') as handle:

pickle.dump(Word\_Index, handle, protocol=pickle.HIGHEST\_PROTOCOL)

with open('glove\_wordvec/wordVectors200d.npy', 'wb') as f:

np.save(f, vectors)

根据Glove Embedding创建相应的参数矩阵（每个词所对应的embedding）.同时考虑了out-of-vocabulary word (idx = 0),以及没有对应glove embedding 的词—随机生成一个embedding.

def create\_weights\_matrix(DICT\_SIZE, glove, INDEX\_WORD, embedding\_dim):

weights\_matrix = np.zeros((DICT\_SIZE, embedding\_dim))

# idx = 0 指向out\_of\_vocabulary word

weights\_matrix[0] = np.random.normal(scale=0.6, size=(embedding\_dim, ))

for idx in INDEX\_WORD.keys():

try:

weights\_matrix[idx] = glove[INDEX\_WORD[idx]]

except KeyError:

weights\_matrix[idx]=np.random.normal

(scale=0.6, size=(embedding\_dim, ))

return weights\_matrix

embedding layer

def create\_embedding\_layer(weights\_matrix, non\_trainable=False):

(num\_of\_embeddings, embedding\_dim) = weights\_matrix.shape

emb\_layer = nn.Embedding(num\_of\_embeddings, embedding\_dim)

emb\_layer.load\_state\_dict({'weight': weights\_matrix})

if non\_trainable:

emb\_layer.weight.requires\_grad = False

return emb\_layer

利用了pyTorch提供的Dataset机制，使得mini-batch的划分更加简便

train\_ds = TensorDataset(x\_train, y\_train)

valid\_ds = TensorDataset(x\_val, y\_val)

train\_dl, valid\_dl = getdata(train\_ds, valid\_ds, bs=bs)

def fit(epochs, opt, model, loss\_func, train\_dl, valid\_dl, config):

# 记录training\_set, validation\_set的loss,用于作图

loss\_train = []

loss\_valid = []

for epoch in range(epochs):

model.train()

epoch\_loss = 0

**for xb, yb in train\_dl:**

pred = model(xb)

# loss\_func 似乎不支持Long类型数据，所以进行类型转换

loss = loss\_func(pred.float(), yb)

with torch.no\_grad():

epoch\_loss += loss

loss.backward()

opt.step()

opt.zero\_grad()

model.eval()

loss\_train.append(epoch\_loss/len(train\_dl))

with torch.no\_grad():

val\_loss = sum(loss\_func(model(xb).float(), yb)

for xb, yb in valid\_dl)

loss\_valid.append(val\_loss/(len(valid\_dl)))

print(epoch, val\_loss/(len(valid\_dl)))

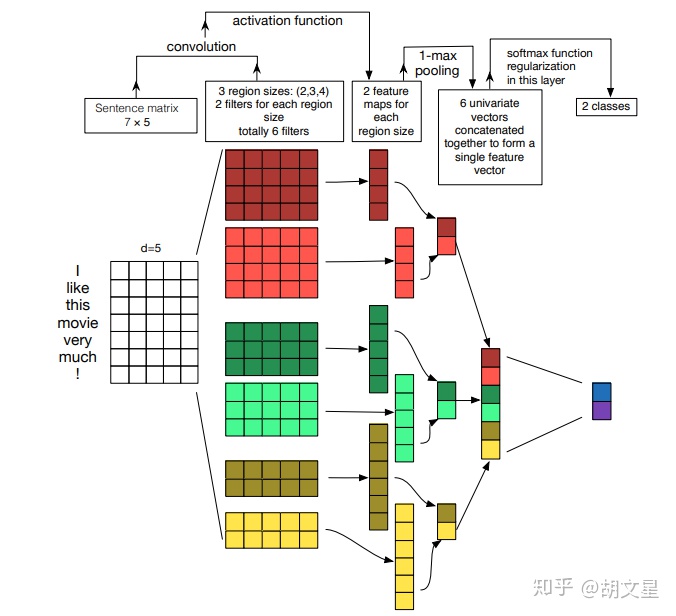
# 保存模型

torch.save(model.state\_dict(), config.savepath + str(epochs))

return loss\_train, loss\_valid

最后是CNN模型， 参考的模型如下，同时也参考了

<https://github.com/649453932/Chinese-Text-Classification-Pytorch/blob/master/models/TextCNN.py>， 但是最后自己还是大体上理解了该算法。



class Min\_CNN(nn.Module):

def \_\_init\_\_(self, config):

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

self.config = config

# num\_embeddings 是文本的词典大小

self.embedding = config.embedding\_layer

# 卷积层， 输入vocab\_size = embedding\_dim \*1 (31\*256\*1)

# 2\*embedding\_dim, 3\*embedding\_dim, 4\*embedding\_dim 的filter各256个

self.convs = nn.ModuleList([

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

self.config.num\_of\_filter, kernel\_size=

(h, self.config.embedding\_dim))

for h in self.config.filter\_size])

self.dropout = nn.Dropout(self.config.dropout\_rate)

# 全连接层 参照图片，输入=filter的总数 输出-- class的数量（5）

self.fc = nn.Linear(self.config.num\_of\_filter

\* len(self.config.filter\_size), config.num\_of\_labels)

def conv\_and\_pool(self, x, conv):

# squeeze: dimension(A\*1\*B) -> (A\*B)

# parameter: dim: 哪个维度的1要删掉

x = F.relu(conv(x)).squeeze(3)

**# 这个max\_pool1d的参数不大明白 x.size(2)是channel?**

x = F.max\_pool1d(x, x.size(2)).squeeze(2)

return x

def forward(self, xb):

out = self.embedding(xb)

out = out.unsqueeze(1)

out = torch.cat([self.conv\_and\_pool(out, conv) for conv in self.convs], 1)

out = self.dropout(out)

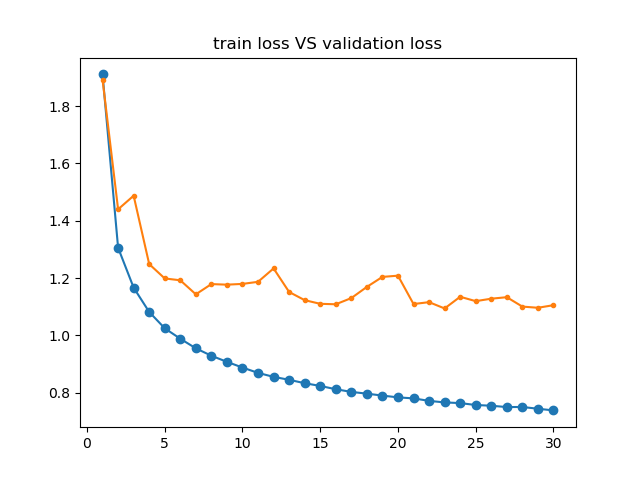
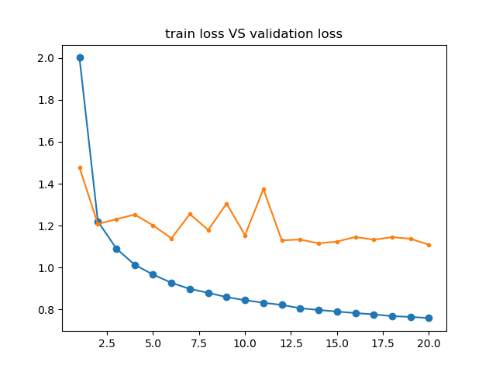
out = self.fc(out)

return out

## 实验结果

使用SGD optimizer

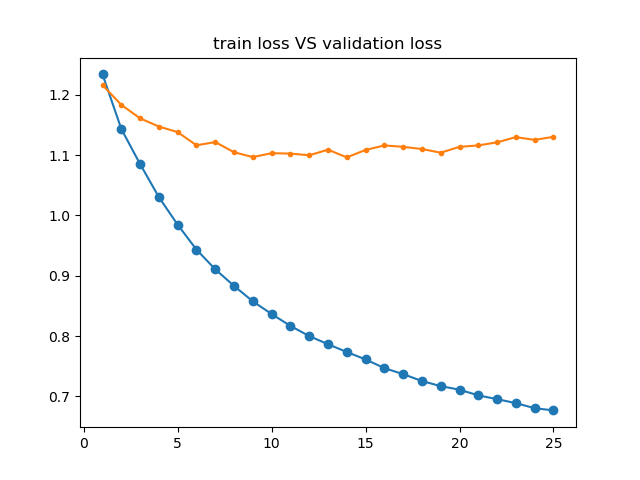
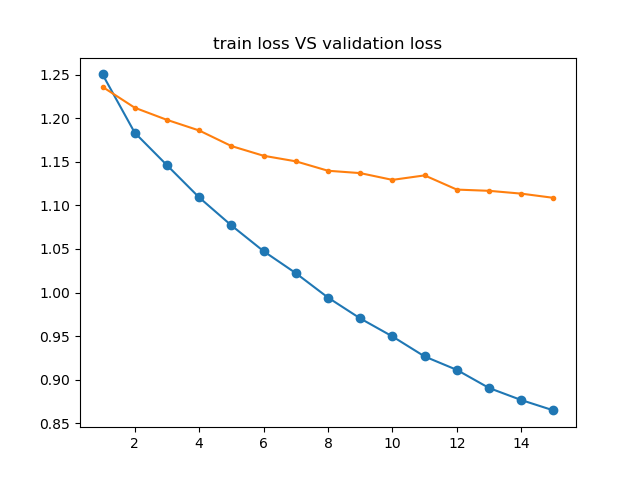
|  |  |  |
| --- | --- | --- |
| condition | train-accuracy | validation-accuracy |
| 1.bs=800, lr=0.2, epochs=20,  dropout\_rate = 0.5 | 74.32% | 56.09% |
| 2.bs=800, lr=0.15, epochs=30,  dropout\_rate = 0.5 | 74.59% | 56.31% |



1. train loss vs validation loss 2. train loss vs validation loss

使用Adam Optimizer, 准确率方面改进不大

|  |  |  |
| --- | --- | --- |
| condition | train-accuracy | validation-accuracy |
| 3. bs=800, lr =1e-4,  epochs=15,  dropout\_rate = 0.5 | 70.36% | 55.66% |
| 4.bs=800, lr=2e-4, epochs=25,  dropout\_rate = 0.5 | 77.21% | 57.71% |

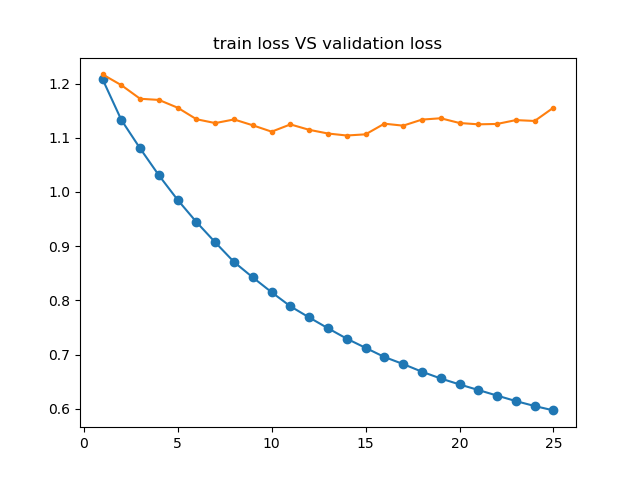


3.train loss vs validation loss 4. train loss vs validation loss

观察图像可知，训练集准确率存在进一步提升的空间，但验证集准确率难以大幅度的提高（57%左右）

由于没出现过拟合，我考虑降低dropout\_rate

|  |  |  |
| --- | --- | --- |
| condition | train-accuracy | validation-accuracy |
| 5.bs=800, lr =1e-4,  epochs=25， dropout\_rate = 0.1 | 78.26% | 56.39% |

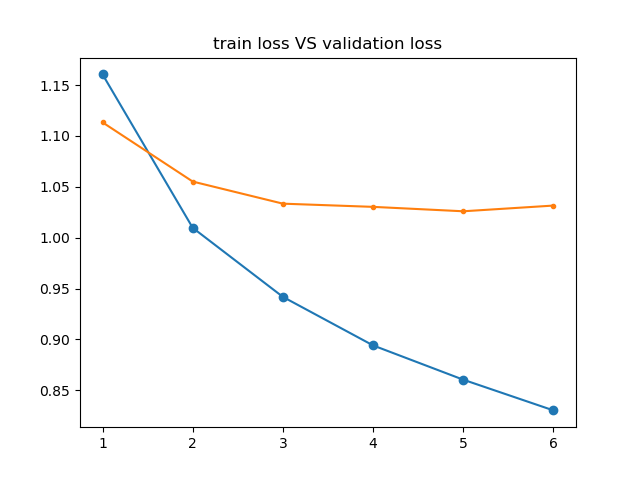


5.train loss vs validation loss

验证集准确率还是没有提高

采用预训练的glove embedding, 测试集准确率提高了2%-3%

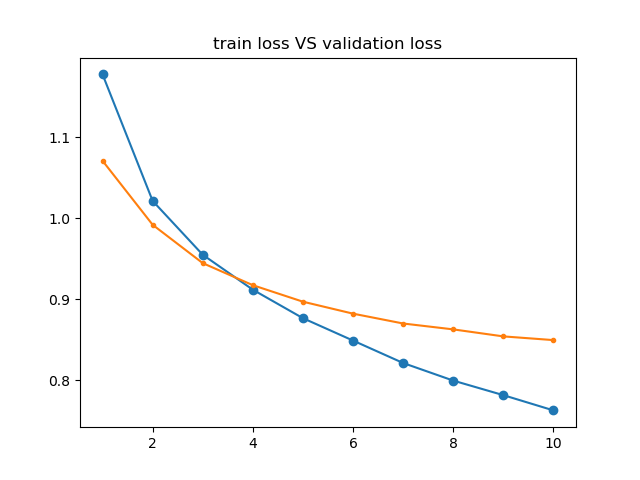
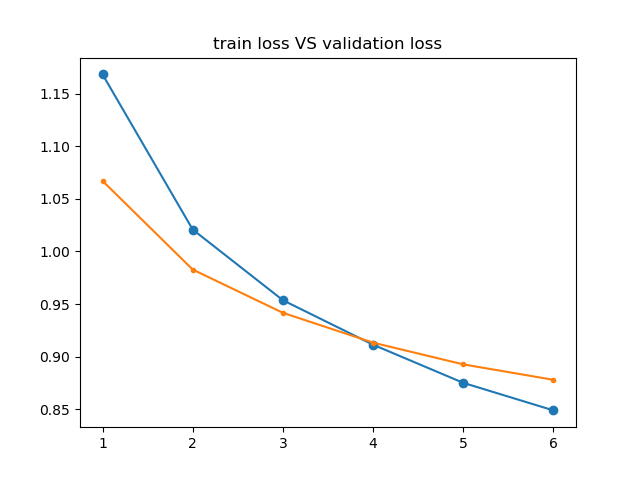
|  |  |  |
| --- | --- | --- |
| condition | train-accuracy | validation-accuracy |
| 6.bs=800, lr =5e-4, epochs=6， dropout\_rate = 0.1 | 67.82% | 59.61% |



6.train loss vs validation loss

**PS: 在写报告重新检查代码时，我发现之前没有对训练数据进行shuffle, 那么看看把样本重新排序后的训练效果**

|  |  |  |
| --- | --- | --- |
| condition | train-accuracy | validation-accuracy |
| 7.bs=800, lr =5e-4, epochs=6， dropout\_rate = 0.1 + shuffle | 66.97% | 64.53% |
| 8.bs=800, lr =5e-4,  epochs=10， dropout\_rate = 0.1 + shuffle | 70.51% | 65.73% |

****

7. train loss vs validation loss 8. train loss vs validation loss

**事实证明，训练数据的随机排列非常非常重要。看似不起眼的一步，能够带来5%左右的准确率提升。**