Report of task 2: 基于深度学习的文本分类

1. 实验介绍

使用pytorch构建CNN和RNN深度神经网络，重新完成task1中的任务，实现基于深度学习的文本分类。使用pytorch等深度学习框架，一方面使用autograd简化了求导过程，另一方面支持GPU的计算加速。

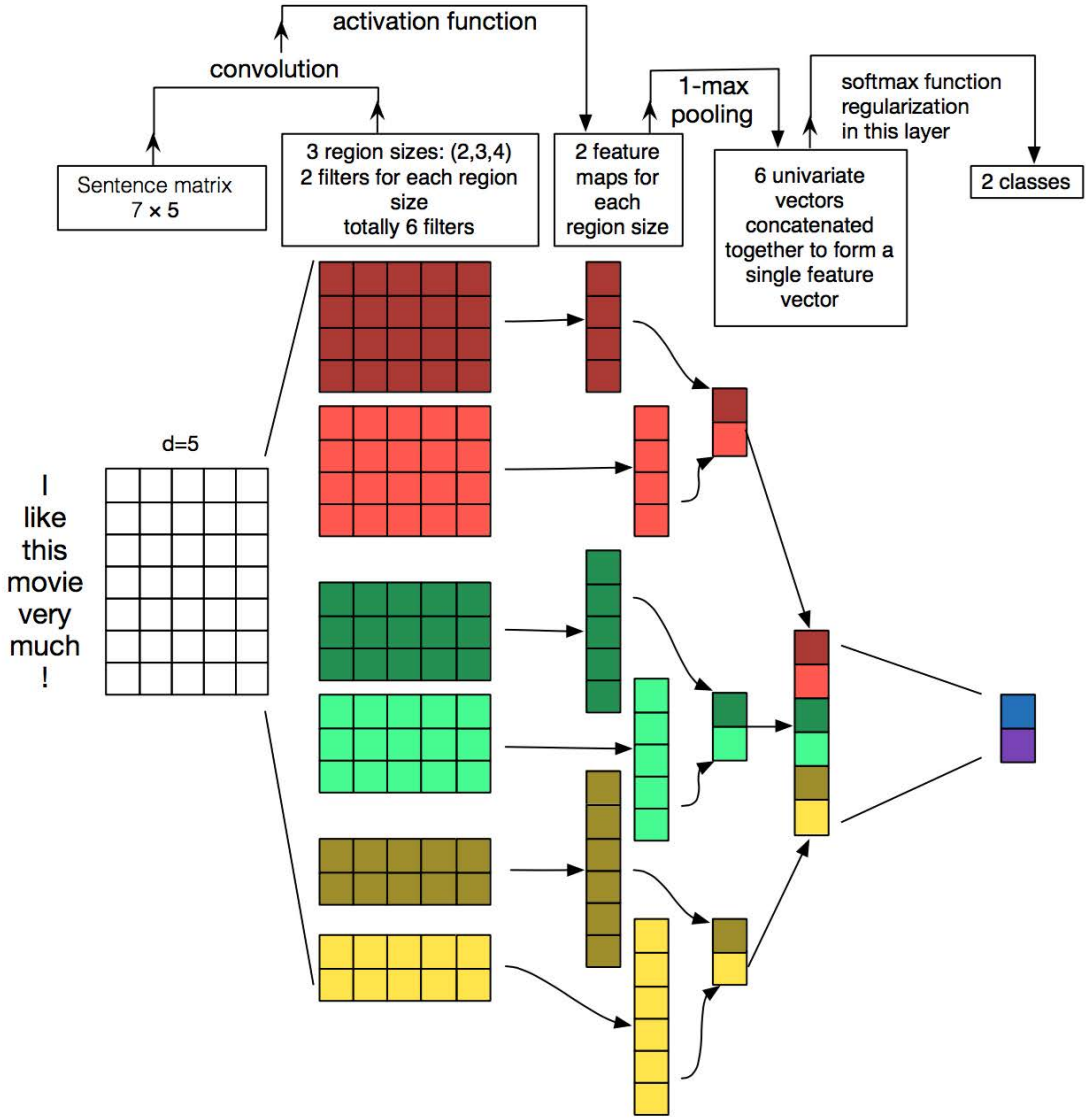
Pytorch作为一种基于动态计算图的深度学习框架，相较于Tensorflow等基于静态计算图的深度学习框架，拥有更强的灵活性，但是相对的，计算速率和分布式计算效率相对较低。

2. 实验原理

2.1 CNN文本分类

这里使用的是最简单的TextCNN。使用fastNLP将每一条文本从字符文本转变为index list，作为文本的字符级特征，经过embedding层将每一个index转变为一个低维向量，这样就将每一条文本转变为了一个二维矩阵。使用卷积网络提取特征，实际上就是2-gram的另一种形式，但是通过卷积核的使用实现了参数共享，大大加快了特征提取的速度。将CNN的激活层经过maxpooling层转变为相同尺寸后就可以输入到FC层实现softmax分类器。网络具体结构见下图，选自A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification。

按照要求，实验中主要使用了两种embedding方法：随机embedding和Glove。Word embedding是一种将文本映射为低维向量的方法，类似的方法也使用在了network data mining当中（network embedding）。Glove和word2vec是两种常用的unsupervised pre-trained word embedding，相比之下Glove更好的捕捉到了词根和派生词之间的关系，同时可以选择较低维的嵌入向量，移动端算力友好。



2.2 RNN文本分类

这里使用的是简单的sequence-to-one模型，基于双向LSTM。实验原理较为简单这里不多赘述，值得一提的是一些pytorch coding上面的tips：

- pytorch的双向LSTM正向结果在ht，而反向结果在h0
- 与vanilla RNN不同，LSTM和GRU设置了forget gate，所以padding的0不能作为LSTM的输入。为了保持batch的同维度，在pytorch中我们使用 pack_padded_sequence(), pad_packed_sequence()两个API进行处理。

3. 实验结果

3.1 Hyperparameter

Hyperparameter	Value
num_epoch	5
batch_size	64
learning_rate	0.0001(CNN) & 0.0005(RNN)
embedding_size	300
dropout (if use)	0.5
hidden_size (LSTM)	300
num_kernel (CNN)	100

实验中所使用的一些超参数如上图所示，具体实验当中我们取10%的训练数据作为验证集进行超参选择，这里直接给出最终超参选择，省略烦琐的调参过程。

3.2 CNN

实验中使用的模型具体结构如下：

```
<bound method Module.named_modules of CNN(
  (embedding): Embedding(400002, 300)
  (conv1): Conv2d(1, 100, kernel_size=(2, 300), stride=(1, 1))
  (conv2): Conv2d(1, 100, kernel_size=(3, 300), stride=(1, 1))
  (conv3): Conv2d(1, 100, kernel_size=(4, 300), stride=(1, 1))
  (pooling): AdaptiveMaxPool2d(output_size=1)
  (dropout): Dropout(p=0.5)
  (classifier): Linear(in_features=300, out_features=5, bias=True)
)>
```

实验结果如下：

Model	test accuracy / %	avg training time (s/epoch on Nvidia Titan X)
random embedding	59.598	9
glove + fine tuning	**65.137**	69
glove + freeze embedding	63.404	**7**
glove + freeze embedding + dropout = 0.5	62.535	21

3.3 RNN

实验中使用的模型结构如下：

```
<bound method Module.named_modules of RNN(
  (embedding): Embedding(400002, 300)
  (LSTM): LSTM(300, 300, batch_first=True, dropout=0.5, bidirectional=True)
  (classifier): Linear(in_features=600, out_features=5, bias=True)
)>
```

实验结果如下：

Model	test accuracy / %	avg training time (s/epoch on Nvidia Titan X)
random embedding	61.998	26
glove + fine tuning	**65.105**	30
glove + freeze embedding	64.992	**26**
glove + freeze embedding + dropout = 0.5	64.942	26

3.4 结论

- 使用预训练的词向量可以有效的提高模型的性能，其中CNN的提升幅度比RNN要来得大
- 对pre-trained词向量进行fine-tuning可能提高模型的性能，但是对于小数据集来说对词向量进行fine-tuning的话有可能造成相关词语在磁向量空间内的距离变大
- 由于实验中训练迭代次数较少，dropout的效果并不明显
- CNN实验中使用fine-tuning会显著影响训练时间，而RNN并不明显。猜想可能是因为RNN的并行能力并不如CNN那样强，GPU可以同时将更多的算力放在词向量的更新上；另外也有可能是因为RNN的参数量较少。

4. 结语

- fastNLP真香真好用啊！
- 数据的结构化对于模型的拟合效果影响很大
- RNN + Attention或许是解决这个问题的更好的方式
- 实验中遇到一个问题：如果将batch事先按照句子的长度进行对x和y均进行排序后训练，最终模型的效果明显变差（40%左右）。讲道理SGD的方法下，一个batch内的样本顺序应该是无关的才对。如果有哪位同学有想到原因希望可以告诉我！