Assignment4 实验报告

Task1

- **数据处理**

  如同assignment2一样，本次实验所采用的数据集依旧是通过sklearn中的fetch_20news获得的，为了更好的检测模型的分类能力，本次实验使用了库中所有标签下的数据集，一共20个类别。其中，train_data一共有11314篇文章，test_data和dev_data各有3766篇文章。

  对于文章的处理与assignment2相似，通过正则表达式以及strip函数去除文章中多余的空格、换行符以及标点符号、无意义的数字等，并将其全转化为小写：

  ```
  re.sub("\d+|\s+|/", " ", data[i] )
  word.strip(string.punctuation).lower()
  ```

  然后运用fastNLP中的Instance模块，将其储存于fastNLP的DataSet实例中:

  ```
  train_data.append(Instance(raw = data[i], label = int(target[i]), words = temp))
  ```

  其中 'raw' 键值对应的区域储存了数据的原文(方便查看)，'label' 对应为文章的类别，而此处的 'words' 对应的为分词处理后的原文。

  之后，**定义fastNLP中Vocabulary类**的实例：

  ```
  vocab = Vocabulary(min_freq=10)
  train_data.apply(lambda x: [vocab.add(word) for word in x['words']])
  vocab.build_vocab()
  ```

  根据train_data中的词语，建立字典。同时规定min_freq滤去出现频率极低的词，提升效率。这其中，fastNLP的Vocabulary在建立的时候，会自动补充定义一些我们在nlp项目中常用到的字符例如 "" 以及 ""，分别用于文章向量的padding以及测试集中为录入单词的查找，其对应的index分别为0和1。这之后，根据Vocabulary对dataset中words的词语数组进行转化，在每一个instance中得到一个int向量储存于 'seq' 域中。

  这之后还尝试了一些教程中说明的instance处理方式，在instance中添加了multi_hot、one_hot以及padding后的表示形式，不过由于模型的原因之后并没有用到。

  根据新版fastNLP教程，为了方便处理，会将模型的输入输出以及长度通过内置的Const模块重命名域名:

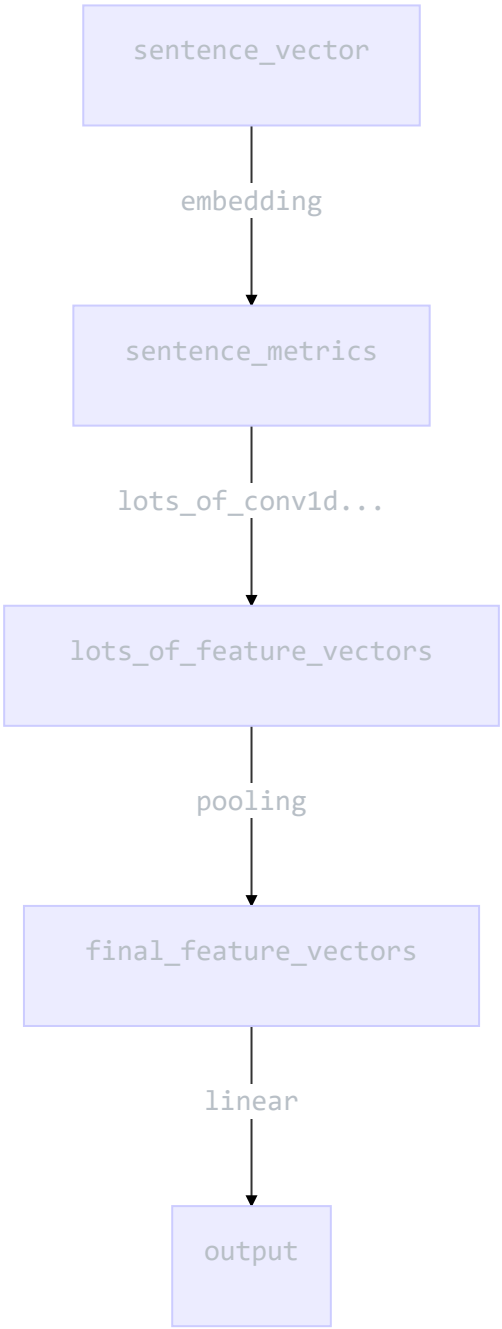
```
train_data.rename_field('seq', Const.INPUT)#->'words'
train_data.rename_field('len', Const.INPUT_LEN)#->'seq_len'
train_data.rename_field('label', Const.TARGET)#->'target'
```

- # CNN 模型

1.模型搭建

对于CNN文本分类器，本次实验根据文档中提供的论文实现了最基本的textCNN classification。而为了体现fastNLP的高效性，我在模型中充分利用了其封装的embedding、conv_pool(卷积加池化)，但后来发现其实现已经在源码里写得很明白了，而CNN模型又较为简单所以实现起来可以说是跟fastNLP中基本一样。为了避免产生直接抄袭的误会，这里就简单谈谈对CNN文本分类的理解。

CNN分类的基本思想就是通过一个个卷积核对embedding后的文章矩阵进行特征提取，经过池化操作后最后得到一个特征向量，最后便可利用这个特征向量进行线性分类。其基本步骤如下：



其中，核心部分就是其卷积和池化操作。为了尽可能多的提取文本特征，CNN中需要用不同大小的不同数值的多个卷积核对文本矩阵进行卷积，其中的主要联系参数为**"kernel_sizes" 和 "out_channels"。** "kernel_sizes" 顾名思义为 kernel_size 的集合，集合中每一个数代表了一种卷积核大小（因为是一维卷积所以就只有一个数），而对于每一种大小的卷积核，对应了一个 out_channel, 代表了有多少不同的该尺寸的卷积核，由于每个卷积核会扫过一次生成一个长度为x的特征向量，再对其进行池化得到一个数（**此处池化根据论文内容选择最大池化效果最佳，但根据测试感觉平均池化与最大池化的准确率效果相似**），最后每一种大小的卷积核都能得到长度为 out_channel 的特征向量，再把所有大小的卷积核得到的结果拼接起来，便得到了最终向量。

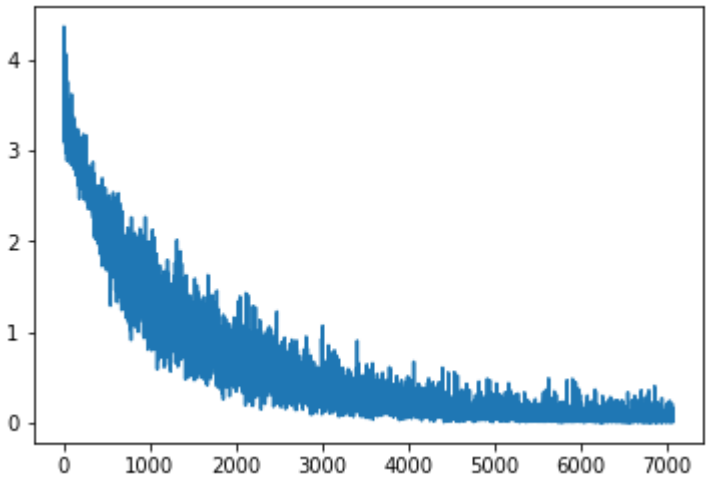
2.训练参数及结果（20个epoch为例）——20分类任务

损失函数	学习率	dropout(防止过拟合)	embedding size	优化器	padding	kernel size	kernel nums	Accuracy	loss
交叉熵损失函数	4e-3	0.1	128	Adam	2	(3,4,5)	(20,15,10)	0.713	0.16

当提高参数数量之后：

损失函数	学习率	dropout(防止过拟合)	embedding size	优化器	padding	kernel size	kernel nums	Accuracy	loss
交叉熵损失函数	4e-3	0.2	256	Adam	2	(3,4,5,6)	(20,25,30,30)	0.718	0.18

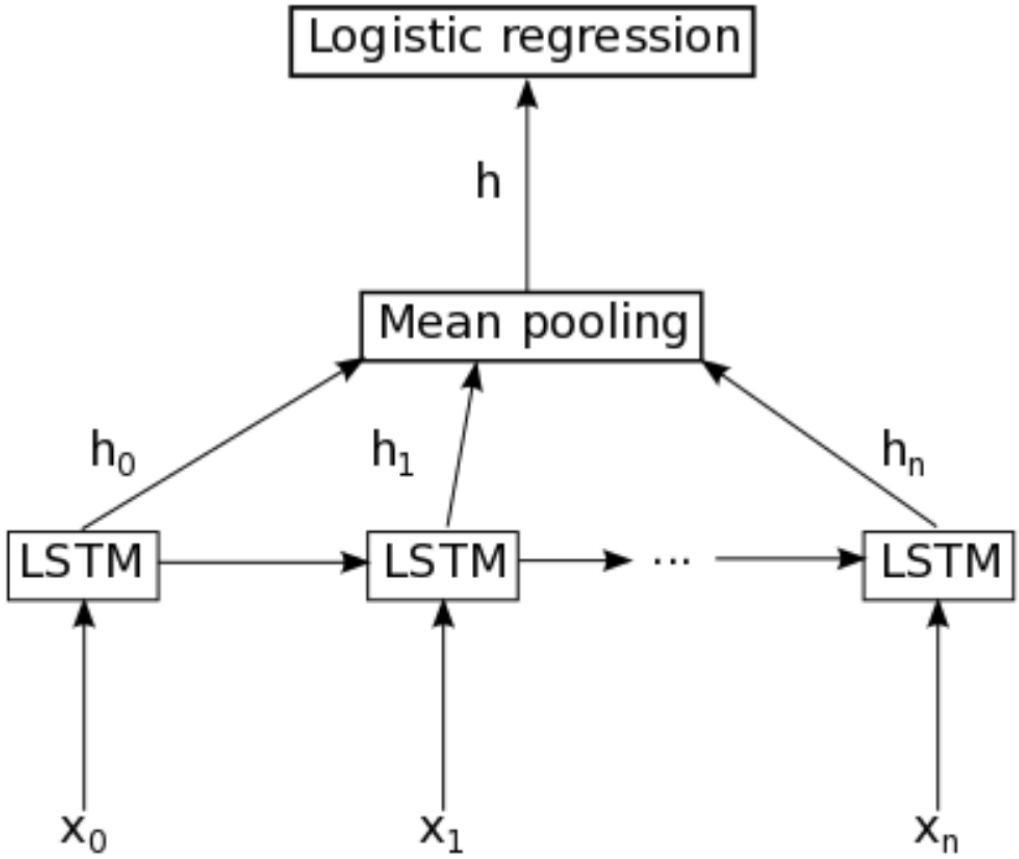
可见简单的加大维度似乎无法明显地提高分类效果。其loss图像如下:



- # RNN 模型

1.模型搭建

对于基于RNN的文本分类模型，由于assignment3的缘故，顺其自然地便想利用LSTM来实现。与生成模型类似，将文本向量输入lstm网络中，可对最后隐藏层的输出进行linear操作，达到分类的效果。但是经过测试，这种只考虑lstm网络最后几个时间点的输出效果不够优秀，在较多分类的问题上正确率很不理想。因此比较正确且自然的想法是，根据整个序列的输出来进行预测分类。对此，输出就将是[seq_len, hidden_dim]的一个矩阵，为了便于线性，可对0维度上进行均值池化，再进行linear操作，这样便充分保留了整个文本的输出信息(模型如下图):

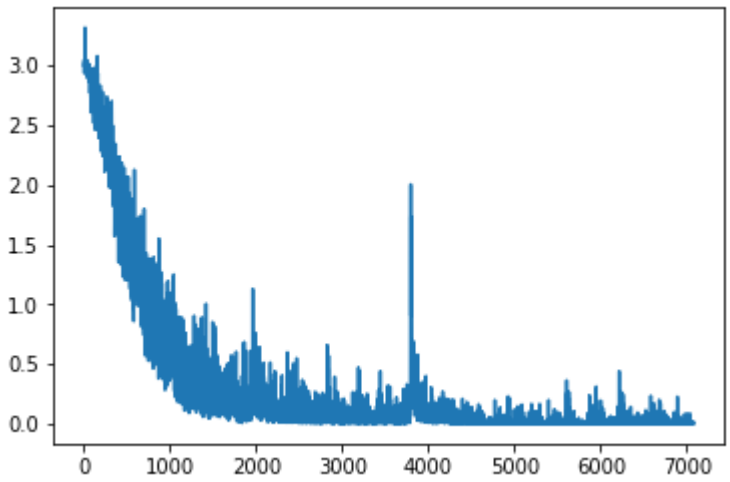


（图片来源于网络）

2.训练参数和结果 （20个epoch为例）——20分类任务

损失函数	学习率	embedding size	优化器	hidden size	num layer	Accuracy	loss
交叉熵	4e-3	256	Adam	256	1	0.776	0.01

其效果要较优于CNN模型，loss曲线如下:



Task2

- ## fastNLP使用体会

  简单来说，我觉得fastNLP这个框架的使用很大程度上提高了我的编程效率。因为其一方面封装好了自然语言处理任务中很多基本但有些麻烦的步骤，为我省去了很多工夫，代码的实现也显得简洁明了，另一方面，其实现过程是流程化的，结构化的，一环套一环，令我的代码书写更为规范、逻辑清晰。

  比如数据处理方面，其Dataset的构建因为是基于Instance的，因此很好地体现了封装性。在以前，我经常就是一个数组一个数组地去储存数据，对于每一个数据，其属性地关联仅仅依赖于数组索引，很容易搞乱掉。而fastNLP中地Instance，可以将一个数据地所有属性包含在内，方便使用也方便处理。

  还有其训练方式，通过set_input、set_target设置dataset的输入输出，然后通过trainner类直接进行训练，简单明了，不仅便于书写，更便于其他人理解你的代码。

- ## fastNLP改进建议

  总体来说体验不错，不过还是有一些地方觉得不够方便。一是training时的loss储存，需要基于fitlog来使用。不过fitlog似乎再windows上无法使用，而由于其显示形式是基于网页的，因此在服务器上运行无法显示出来，所以当时我用的时候感觉有点窒息。

  还有一点是报错反馈系统，在刚开始使用时，难免有一些地方打错或是处理错误，但我感觉fastNLP对错误的反馈略微简单，没有细致地对每一种类型地错误进行不同的反馈，或是在非常深层、底层的代码才会进行报错，有时会让人不知道到底是哪里出现错误。