情感分析语义识别

――项目报告

注：本次报告既是最后一次学期作业,又是实验项目,所得实际更加贴近于经验.所以组织报告的时候会更加松散,也希望记录一点实用性大于可解释性的东西,更加类似于实验笔记之类的产物.

**1.模型是如何搭建的? ---前瞻或回顾**

基于pytorch的深度学习框架,已经可以解决三个部分的问题----神经网络怎么搭建,模型如何训练,训练数据的数据结构是怎样的.按照搭建时序的正确考虑顺序其实是＂训练数据的数据结构是怎样的＂,＂神经网络怎么搭建＂，＂模型如何训练＂，不过从构思的视角来看，容易的问题总是更先被解决的．

神经网络的搭建取决于具体想要采用的方法,一定程度上这里有很大的自由，可以在我搭建的RNN中也看到CNN层，混合所带来的效果提升，是很符合直觉的．但大体上却**不能把RNN中的CNN layer看作一套卷积核与感受野的对应操作,它没有pooling层,仅仅是用于数据的降维**,而很多时候这种降维实际比一层NN神经网络(linear layer + activation)更加合理且有效.

模型的训练主要难点在于rnn加入新的数据结构,这个新的类即是**pack\_padded\_sequence**,**这个类可以允许我们训练rnn网络时不必对句子做padding**,当然这个类同已经padding的tensor也可以做到很轻便的转换效果,具体可参见: [pytorch对这个类的解释文档](https://pytorch.org/docs/stable/generated/torch.nn.utils.rnn.pack_padded_sequence.html#torch.nn.utils.rnn.pack_padded_sequence),一些应用实例展示可参见:[来自知乎的应用实例](https://zhuanlan.zhihu.com/p/601705984).在本次作业中也给出了实例展示, 见”*Pack Padded Data ---- The Class Used latter*”一节.加入这个类带来一个问题,初始化这个类需要每个句子的长度信息,当padding完再获取这些信息是不容易的,因此需要提前获得,而且在训练时,每个batch都需要对应的句长信息,这意味着train函数所需的参数需要包含句长信息.

实际上,一边考虑训练数据时可能遇到的麻烦,一边会注意到:之前的模型训练总是依赖于dataset数据结构的,而**dataset实际上比dataframe或者tensor迭代起来会更加麻烦**[[1]](#footnote-1),这在时间复杂度上就可以证明.在本程序最初,本来并没有dataset转dataset的过程,但是在计算最长句子所包含的单词时,以dataset输入会非常缓慢.dataset很方便于shuffle和batch的操作,但是本次实验中,仅仅依赖于tensor就可以以不慢的速度训练67349行的数据,这个训练数据集的内存占用在0.72GB左右,对于CPU的运行内存来说实际是一不小的考验.

综上,我们可以得到这样一些环节的考量,无论数据最初是什么样子,都会以tensor的形式被投入训练,而训练用的函数主要有两种,一种是对CNN的普通训练函数,还有一种是针对RNN开发的基于pack\_padded\_sequence数据结构的训练函数.

**2.大体的框架是? ---整套流程**

其中前两步骤起步难度相对较大,这里面主要有两个难题:

1. 如何从句子得到分割开的词典内词汇.
2. 向量化的词典从哪里来.

其实,由于语料繁杂,始终有部分单词很难被囊括在词典中,这里面包括单纯合成词,比如＂tensor-flow＂,一个好的词典包括了＂tensor＂,也包括＂flow＂,但是绝不会有这个合成词的影子,不过这种合成词说到底还是好分离的,因为只需要匹配＂-＂即可分开这种合成词,但是对于芜杂的语料这仍然是不够的.

本次任务中使用了许多外源的可以借用的tokenizer或词向量词典,如下:

1. **vocab.GloVe**词向量词典,它包含多个版本,包括50维度,100维度,200维度和300维度的.
2. **BertTokenizer**的"bert-base-uncased"版本，它能够处理上述这种＂－＂连接的合成词以及一些特殊的合成词．但实际效果不能算很好．
3. **nltk.stem.porter.PorterStemmer**,这是自然语言库的一种提取词干的函数,但是从主观上讲使用这个函数应当是无奈的选择,因为选出词干的同时也会丢失一些语义信息,比如前缀＂dis＂,比如后缀＂ing＂，这些信息丢失的影响到底多大仍未可知.

当然,纵使采取了上述的方法,仍然有单词能够不被囊括在词典中,这就是棘手的合成词＂unentertaining＂,由于只存在这样一个特例,所以我手动把它添加进了词典.

实际上,这是代码中函数**sentence\_picture()**所实现的功能,它能够转化大部分的英文句子,而输出一个m\*n的numpy型数据,m表示本句的句长或规定的最大句长,n表示词向量的维度.我在设计这个函数时并没有规定m要大于单词数最多的句子长度,因此如果觉得某些句子过长,也可以采用在转化的时候就经过一道pca降维处理,但后面会指出,这种做法的问题不是一星半点的.

我设计了dataframe到训练数据集(不含标签)的函数,所返回的是一个tensor,实践证明,这个函数具有很好的复用性,因为一开始我并没有注意到老师提供的数据集只有10000条数据,而且也没有提前预留好的验证集数据.但是由于提供的数据集是CSV格式,很容易读取为dataframe,因此新的训练数据集生成并不会花费更多的工作量.

上面流程中的另一个trick来自训练时采取的batch并不是提前划分好的,而借由简单的循环语句和列表切片来完成,batch的最后一尾由于不等于batch\_size,看似造成了处理的困难,但可以通过切片**batch\_start:min(batch\_start + batch\_size, train\_tensor.shape[0])**完成.这样每次训练的次序不会改变,也并没有增大多少的代码量,实际上对于含attention的RNN, train函数甚至可以实现复用.

**3.调参过程的启示(细调)**

调参过程可以分为三部分,关于CNN的,关于RNN的,以及关于attention RNN的.其中关于CNN的可调参数实际更多,因为CNN的卷积核长宽都是便于解释的,而RNN层调参时容易改变词向量的维度,但这并不好解释为什么改变了会更好.

在CNN调参过程里有这样一些事件是重要的,有助于我们理解为什么可以使用CNN来处理自然语言,我把它作为警示给出.

1. **方形的卷积核可能是不太好的**,最好的建议其实是一开始不要改变词向量的维度.但是实际过程中,如果词向量被压缩得厉害,那么也可以通过增大输出通道数去获取更多信息.
2. **使用等于词向量长度的卷积核也许是有道理的**,因为在大部分语句中,一个单词的含义都能以一个确定性的意指标示出来,这意味着在一句话里,某个单词有50个维度也许太多了,也许只需要一个维度就能在这句话里理解这个单词的含义.
3. 还有一种卷积方式是只在单词纵向毗邻的维度里进行卷积,**这种卷积核长成长条状**,我们每次都只是关注相邻单词在某一维度上的联系,它也能取得不错的效果,但是和(2)放在一起则需要注意池化层的操作方法.
4. **池化层最好和卷积核的宽扁性相互对照**,也就是说,卷积核是矮宽的,池化层就应当是宽的,卷积核是扁长的,池化层就也应当采取扁长的窗口.但是这个原则最好也考虑到信息的损失率,把池化层窗口尽量设计得小一点.
5. **池化方法采取average\_pooling比max\_pooling要好.**这同样是为了减少信息损失.
6. 如果存在一种卷积核,能做隔行的空洞卷积,那么应当也使用含这种卷积核的卷积层,因为单词与单词并不总是毗邻而联系的,也有很多跨度很长的联系,这集中在一些含有指示代词的句子里,CNN很难处理这种联系,但是更大的空洞卷积核说不定可以.

关于RNN的调参我遵循了下面一些简单的原则或后验的规律:

1. **数据降维可以用一维的CNN卷积核—--采取一维卷积核是由于RNN的输入数据维度和CNN不一样**,这也允许我们把seq\_len看成是一维图片的通道数,这无疑给之前的CNN网络设计了一些新的启示—--但是从逻辑上讲,这种卷积核无法发现不同通道之间的语义联系,因为卷积核之间缺乏直接的相关性,而且最终的Channels实际是多个通道图像的堆叠.另外可以稍提一下的是,Google的transform架构中也包括1\*1的卷积核做数据降维的神经网络层,我未必不是从那里得到的一些启发.
2. RNN的输出维度,即hidden\_size设置得与原始词向量维度一致,设置成其他的维度或许会使得情况更优,但缺乏依据,也很难解释.
3. Flatten后的全连接层总是两层,并且dropout只在两层之间发生.其中,两层的全连接层也许是并不必要的,attention RNN就只使用了一层全连接层.
4. **使用了nn.GRU层代替一般的RNN层**,参见:[门控循环神经网络](https://blog.csdn.net/Michale_L/article/details/122778270#:~:text=GRU%EF%BC%88Gate,Recurrent%20Unit%EF%BC%89%E6%98%AF%E5%BE%AA%E7%8E%AF%E7%A5%9E%E7%BB%8F%E7%BD%91%E7%BB%9C%EF%BC%88RNN%EF%BC%89%E7%9A%84%E4%B8%80%E7%A7%8D%EF%BC%8C%E5%8F%AF%E4%BB%A5%E8%A7%A3%E5%86%B3RNN%E4%B8%AD%E4%B8%8D%E8%83%BD%E9%95%BF%E6%9C%9F%E8%AE%B0%E5%BF%86%E5%92%8C%E5%8F%8D%E5%90%91%E4%BC%A0%E6%92%AD%E4%B8%AD%E7%9A%84%E6%A2%AF%E5%BA%A6%E7%AD%89%E9%97%AE%E9%A2%98%EF%BC%8C%E4%B8%8ELSTM%E7%9A%84%E4%BD%9C%E7%94%A8%E7%B1%BB%E4%BC%BC%EF%BC%8C%E4%B8%8D%E8%BF%87%E6%AF%94LSTM%E7%AE%80%E5%8D%95%EF%BC%8C%E5%AE%B9%E6%98%93%E8%BF%9B%E8%A1%8C%E8%AE%AD%E7%BB%83%E3%80%82)&[GRU官方文档](https://pytorch.org/docs/stable/generated/torch.nn.GRU.html#torch.nn.GRU).这种循环神经网络具有更强的长期记忆能力,比之LSTM更易训练,其主要的区别在于加入了重置门与更新门,重置门的主要作用是如何将输入信息与前面的信息相结合,生成候选隐藏状态,而更新门则负责在候选隐藏状态和隐藏状态之间做选择.
5. 学习率调整为了0.005,这是由于RNN内在的收敛速度较快的缘故.当学习率为0.01时,前两个epoch内验证集上准确率就会达到峰值,而且RNN的验证集交叉熵有一些不良的增减性(变化幅度是最大的),这和学习率的内部关系未知,但是却警告我们要使得RNN收敛得慢一点.

到了attention RNN这里,需要强调的已经不多:

1. 学习率被调整为了0.001,比前两个model都小很多,这实际上说明attention RNN的loss function有更高的收敛速度.在这种训练率下,attention RNN的泛化性能也是最稳定的,这一点将在后面指出.
2. 采用的attention计算方式是自注意力,Q,K,V都来自同一个矩阵,而且采取了pytorch已经实现的多头注意力方法----nn.MultiheadAttention函数.
3. **注意力的头数选取适合在6-16之间**,这参考了[多头注意力的heads数如何设置](https://m.thepaper.cn/baijiahao_19955038)文章中的一些说法.我在调参中使得num\_heads严格地同hidden\_size2相等.

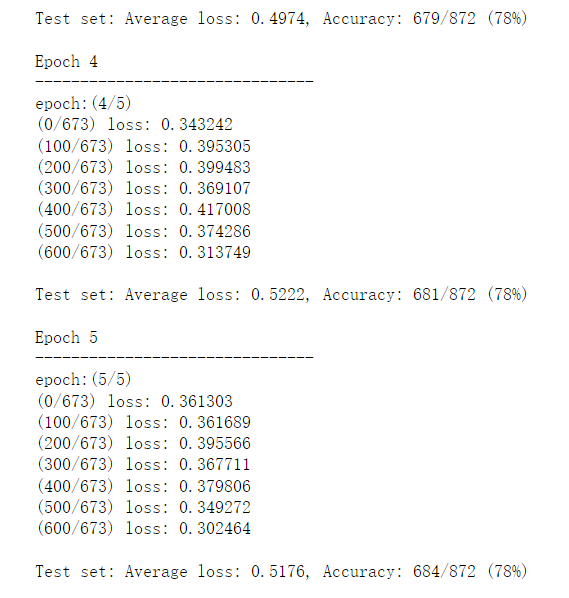
**4.超参数的调整与模型宏观对比的结论**

实验结果所带来的启示是方面的,一方面来自数据,另一方面来自方法.

数据处理对训练结果的影响启示是很惊人的.这里主要是说数据进入模型的sequence length与word vector dimension的影响.在最初,我使用的glove字典其实是100d的,数据类型是float64,sequence length为了方便则同样取100的长度,这种训练数据集的大小是现在的8倍,也就是0.72\*8 = 5.76GB,这么大的数据提取到虚拟内存都异常艰难,显然必须要降维操维操作.因此,我选用了50d的词向量,根据句长的分布,把sequence length设置为了25,在此种情形下,有6%(4684个)的句子都会遭到压缩,最终的效率或结果并不理想,在CNN初次训练时,25\*50的sentence picture的验证准确率是从70%起步的,而改换成50\*50的句图则很快就有75%以上的验证准确率.当然,这时**影响泛化性能的却不只有PCA主成分分析中的讯息丢失,还有rand padding的方法**,这种方法是最初嵌入**zip\_st\_pics()**函数的,它负责把长度为sequence length或max sequence length的句图投入到一个固定长宽的空矩阵的任意位置,使得padding的0元素既可能出现在句子的前面,也可能出现在句子的后面.这样做有一种直觉性的好处,就是训练得到的神经网络一定要学会忽略0,并且忽略句子位置或长短所带来的附加信息,这些信息在人脑看来基本是和情感判断无关的.但是,实际考虑到CNN卷积核的权重共享,这种做法后来被我遗弃了,因为让卷积核从一开始卷积就能得到重要信息(posterior padding)看起来更有效,而且无伤大雅.

除此以外,是否对数据做shuffle()也能影响训练的均衡性,实际上,单纯的随机shuffle()不一定是好的,如果shuffle()操作能够使任何batch内的句长分布同总体分布大致一致,则很可能会提升训练的速度,但是为了调参过程更加稳定,最终也没有采用这种方式.

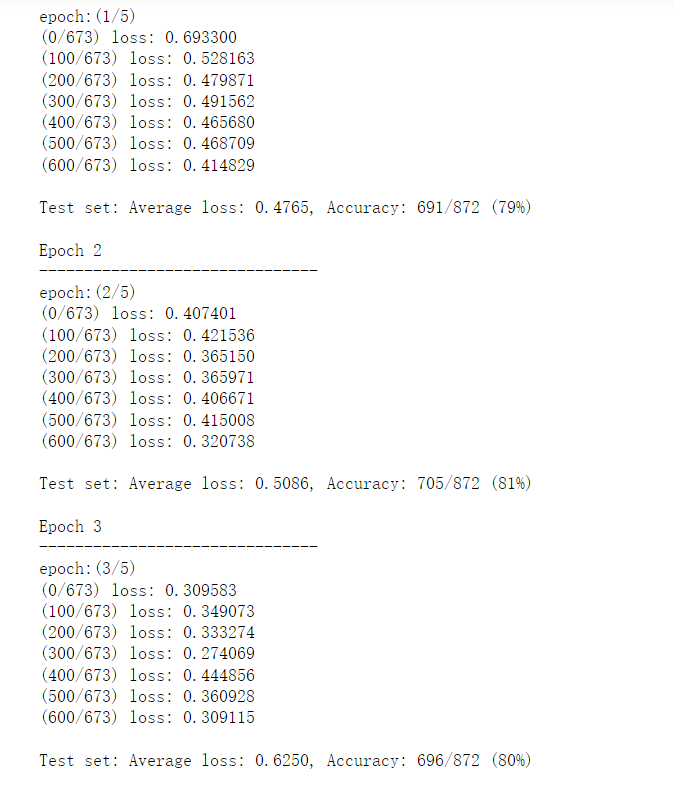
在梯度下降的过程中,一些现象是很有趣的,之前并没有注意过这样的现象.比如说下图:

出现了test loss上升,而验证集上的准确率也上升的情况,这一点在jupyter notebook文件里已经给出了说明,但是在这里要说明的是,在三种神经网络的训练中,交叉熵的上升下降的起伏某种程度上是有规律的,**首先是随着训练集交叉熵的下降,验证集的交叉熵也会缓慢的下降,但总是维持在0.4以上的水平,交叉熵的下降不一定会带来准确率的提高(但多数情况可以带来),但是到了epoch >= 15之后,验证集的交叉熵也不再下降,而是开始波浪形上升,最终也会反映到准确率的下降**.这个表现也决定了训练迭代的数目不能设置得太大,我经常使用的max epoch范围是 56.

**CNN训练输出截图**

尽管上述情况是很普遍的,但是仍然要强调这个过程在CNN,RNN和attention RNN的表现细节非常不同.但要强调的是,这些表现的不同与学习率也有关系,它们的学习率分别是[0.01,0.005,0.001]:

在CNN中,训练集和验证集的交叉熵下降都是一个相对缓慢的过程,甚至迭代5代后,可以看到平均的交叉熵估计仍然是大于0.3的,但是CNN的验证交叉熵并不会像GRU的RNN那样跳到0.6以上,甚至验证交叉熵较大时,准确率还能上升----epoch4 to epoch 5,从681个查准到684个查准.

右图是RNN训练的前3个epoch,可以发现,尽管验证集准确率似乎是上升了,但验证集上的交叉熵却上升得更快,甚至快得有点不合理,0.005的学习率经过每一轮训练后,验证集上的交叉熵都提高了0.12,这和CNN训练时的时升时降其实是不同的.

而attention RNN训练的情形则更加给人深刻的印象,首先是attention RNN的所需的训练时长是最长的.与此同时,attention RNN的验证集交叉熵始终没有从0.4跳变过,虽然从epoch4以后,验证集loss也逐渐地稳步上升.但是最高也并不超过0.47,数值相对稳定.

**RNN训练输出截图**

我们也回过头来注意一下模型在训练集上的表现,记准确率函数为A,损失函数为F,则训练结束后三个模型有如下大小关系:

其中train的交叉熵损失由于都是在batch上计算的,尚没有准确的数据比较,但大致上两个RNN的表现都远好于CNN.

最终转化为CSV的结果是三个模型做简单投票,多数同意的结果. SA\_test\_result是给出检验数据集的预测,有1000行, sst2\_original\_test\_result是从datasets数据集中下载的test数据集预测,有1821行.

1. [参考链接](https://blog.csdn.net/Twilightzsj/article/details/123941780) [↑](#footnote-ref-1)