情感分析实验报告

李晨昊 2017011466

2019-5-31

目录

1 运行方法

本项目在 Linux 上编写，文件分隔符等平台相关信息没有做跨平台适配，不建议在 Windows 下运行。

提交的压缩包中没有任何数据，所以是不能直接运行的。我另外传了一个网盘的压缩包，地址附在课程作业中了，这个压缩包中有处理好的输入数据和训练好的网络。如果只想要处理好的输入数据，重新训练一遍网络的话，需要从这个压缩包中的 `word2vec.pk` 和 `data/sinanews\*` 复制到对应的位置，然后执行 `python util.py` 生成训练集/验证集/测试集的词向量表示。

我实现了一个全连接神经网络，采用 C++ 编写，推荐的编译参数是`-Ofast -march=native -fopenmp dnn.cpp -o dnn`。

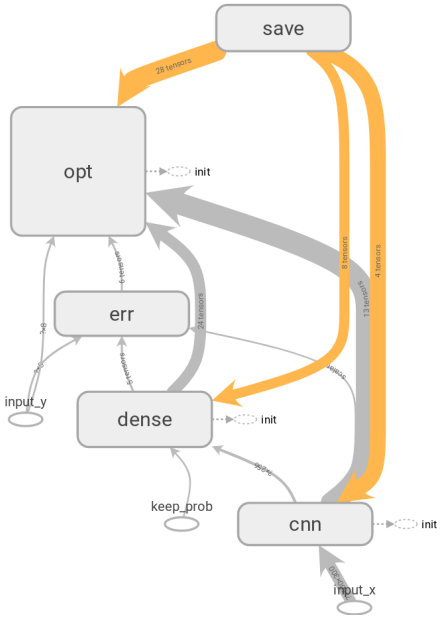
`cnn.py`，`rnn.py`，`dnn.cpp` 均可接受命令行参数，不加参数直接运行均是训练网络，加一个参数 `i`(表示 inference) 运行会读取训练好的网络，在测试集上运行并输出结果。

三者都在在测试集上运行好后，运行 `python evaluate.py` 输出三种网络的准确率，相关系数，f1 score。

2 模型的结构图

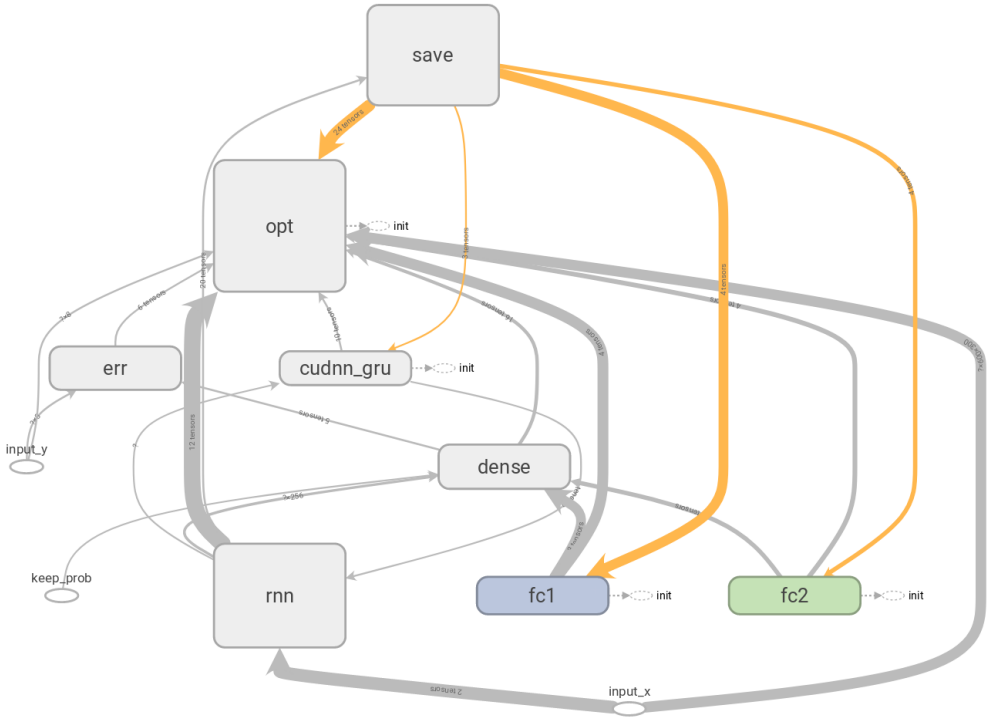
我使用 tensorflow 框架实现了 CNN 和 RNN，此外还用 C++ 手写了一个全连接神经网络。

利用 tensorboard 工具可以画出 CNN 和 RNN 的网络结构图。CNN 网络结构如下：



输入数据维度为 $seq\_len * embed\_dim$，其中 seq_len 的选取会在流程分析部分解释，$embed\_dim$ 为词向量维数，我使用了Chinese-Word-Vectors中预训练好的 300 维的词向量，因此 $embed\_dim = 300$。对输入数据使用 256 个大小为 5 的一维卷积核进行卷积，一个卷积核会使用 $5 * embed\_dim$ 个数字进行计算，所以在卷积结果的维度中就不再含 $embed\_dim$ 了。卷积结果维度为 $(seq\_len - 4) * 256$，对它的第一维做最大池化，得到长度为 256 的向量，该向量在经过一个维度为 256 的全连接层后，输出到维度为 8 的输出层。全连接层采用 relu 激活函数，输出层采用 softmax 激活函数，最后取输出层的 argmax 与结果进行比较。

RNN 网络结构如下：



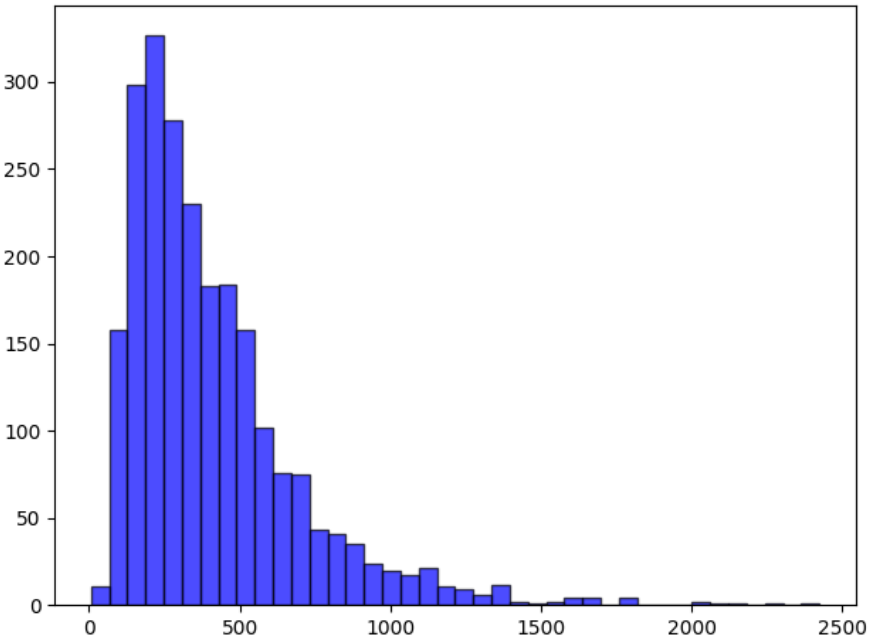
输入数据维度为 $seq\_len * embed\_dim$，输入经过 4 层隐藏层大小为 256 的 GRU 层后，得到长度为 256 的向量，该向量在经过一个维度为 256 的全连接层后，输出到维度为 8 的输出层。

全连接神经网络是 built from scratch 的，所以没有 tensorboard 这样的工具。它的结构很简单：输入层维度为 $200 * embed\_dim$，经过维度为 100 的隐层 1 和维度为 100 的隐层 2 后，到达维度为 8 的输出层。其中隐层 1 和隐层 2 采用 relu 激活函数，输出层采用 softmax 激活函数。这些都需要我去手动实现。之所以输入层维度为 $200 * embed\_dim$ 而不是 $seq\_len * embed\_dim$，主要也是受全连接神经网络参数数量的限制，这里相差的倍数会直接反映到参数数量上，这样大的参数训练起来速度很慢，效果也不好。

3 流程分析

首先需要确定 seq_len，准则是尽量让这个长度覆盖所有的文本，且也不应太长，避免浪费。

统计不同长度的文本数目，结果如下：



结果是长度为 575 及以下的文本占据了约 80% 的比例，取个整，取 $seq\_len = 600$。当然这个 seq_len 只适用于 DNN 和 RNN，而对于全连接神经网络，我选择了一个较小的定值 200，原因在上面已经提到。

为了将输入的词喂给神经网络，我采用的方法是提前将输入的每个词转化成词向量后保存起来，之后直接使用词向量作为输入。之所以没有采用 tensorflow 内置的词嵌入机制 (trainable=False)，有两点考虑：一是预训练好的词向量表本身就很大，通过把预训练好的词向量表 + 词的 index 喂给神经网络，空间其实并没有什么节约；二是我的全连接神经网络没有词嵌入机制这种功能，直接用词向量比较方便喂给它。

我将长度为 2342 的训练集分成了长度为 2048 的训练集和长度为 294 的验证集，为了方便起见，这个划分是静态的，之后不会再调整了。

之后的训练过程中，CNN 采取的训练策略是直接每次都使用整个数据集来训练，这是因为数据集本身很小，这样还能减少 GPU 和 python 间的传输次数，速度完全可以接受。对于 RNN 不能这样，因为 RNN 在训练过程中需要的额外空间更大，直接训练整个数据集会爆显存，因此我采用了大小为 128 的 batch。全连接神经网络采用了大小为 512 的 batch，这个数字其实有一定的随意性，因为我的全连接神经网络所需的空间与 batch 大小无关，所以直接训练整

个数据集是可以的，不过稍微减小些可以加速训练，毕竟 CPU 的算力是非常有限的。

全连接神经网络的训练使用了 openmp 来进行多线程加速，具体方法是每个线程负责一个 batch 中的一部分数据，将计算出来的准确率，损失，梯度信息保存在每个线程私有的内存中，一个 batch 训练完后把这些结果整合起来用于更新网络。

4 实验结果

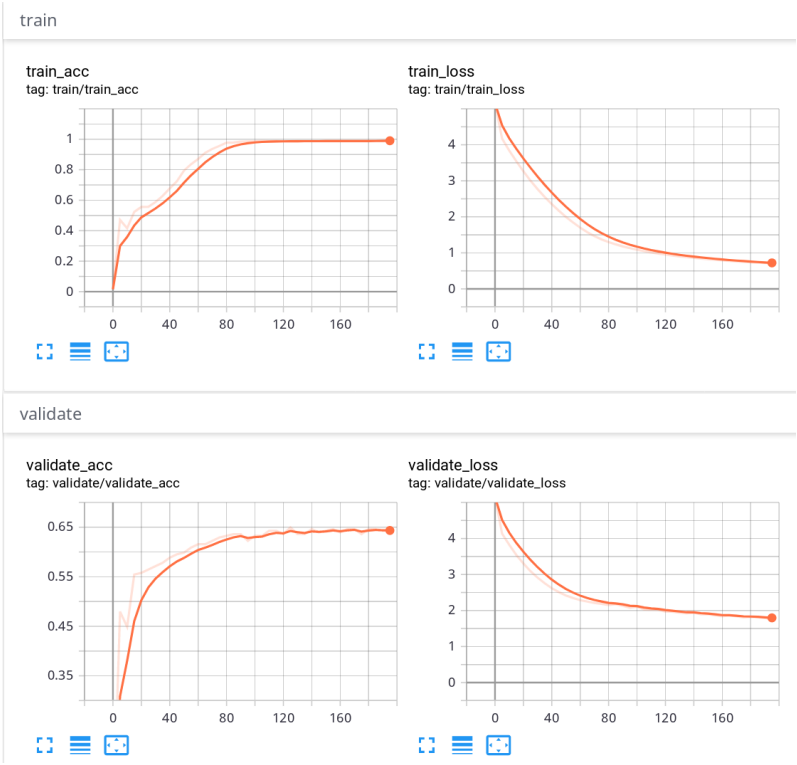
最终结果如下：

	CNN	RNN	DNN
accuracy	0.6005385996409336	0.47755834829443444	0.5659784560143627
correlation	0.6329347427158911	0.45253882971912024	0.569675505102085
f1 score	0.2845291708073689	0.08080194410692589	0.2240019264030131

可以明显看出，实验效果 CNN>DNN>RNN，而且从我往常的观点来看，三者的实验效果都很一般。事实上，RNN 的结果是相当糟糕，观察测试集分布可以发现，直接返回 3 即可达到与 RNN 一样的准确率。也许是我的 RNN 参数选取不得当，也许是数据集自身的问题，以我的能力无法深究下去了。

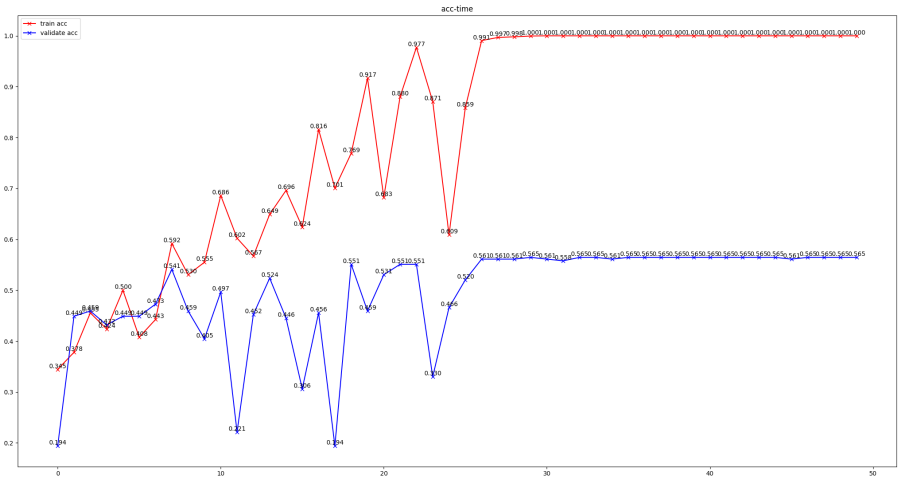
我记录了训练过程中，训练集和验证集的准确率和损失，其中 CNN 和 RNN 采用 tensorboard 来记录，对全连接神经网络的中间输出结果采用 matplotlib 绘图。

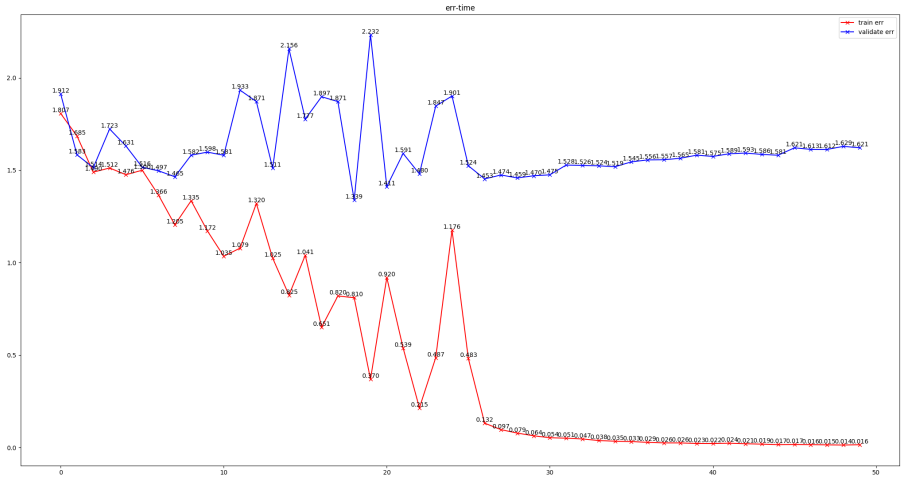
CNN 结果如下：


train
train_acc
tag: train/train_acc
1
0.8
0.6
0.4
0.2
0
0
40
80
120
160
train_loss
tag: train/train_loss
4
3
2
1
0
0
40
80
120
160
validate
validate_acc
tag: validate/validate_acc
0.65
0.55
0.45
0.35
0
40
80
120
160
validate_loss
tag: validate/validate_loss
4
3
2
1
0
0
40
80
120
160

RNN 结果如下：

全连接神经网络效果如下：





从图中大致可以看出，CNN 的训练集/验证集的准确率/损失的上升/下降都很平缓，而其它两个则有一些比较激烈的震荡。这可能与取点的密度有关，也可能与训练方法有关：RNN 选取的 batch 较小，更有可能在训练的时候被干扰，从而变化较为激烈；CNN 和 RNN 都采用了 tensorflow 提供的 AdamOptimizer，而全连接神经网络只使用了一个手写的非常 naive 的梯度下降，因此训练效果可能不够好。

此外，还可以粗略比较一下三者在测试集上运行的速度。为了避免 tensorflow 初始化的干扰，对于 CNN 和 RNN 我都运行了两次，测量后一次的时间结果。三者耗时如下 (CNN/RNN 采用 python 的 time 模块计时，DNN 采用 C++ 的 chrono 模块来计时，所以有效数字不一样)：

CNN	RNN	DNN
0.9153506755828857s	3.0093235969543457s	0.52271823s

由于模型较为简单，CPU 实现的全连接神经网络的速度超过了 GPU 实现的 CNN 和 RNN。

5 不同参数效果

我调整过的参数主要有以下几个，分别讨论如下：

1. 网络的宽度/深度

对于三种网络而言结论基本都是一致的：网络的宽度/深度可以在合理的范围内取的很小，都足够达到现在的效果，例如 RNN 的 GRU 层数目可改为 2，大小可改为 64，结果准确率是一样的。我认为原因是训练集太小，即使有了那么多网络参数，多半也都是用来过拟合训练集

去了，而对于测试集几乎没有什么帮助。尤其是对于全连接神经网络而言，因为训练的算力来自于 CPU，即使采用了多线程优化，算力也非常有限，过大的网络会让训练极其缓慢。

2. L_2 正则项的系数

目前选择的是 0.01，似乎比网上建议的值要小不少，然而经测试如果改成 0.1，训练效果会明显变差，基本没办法训练出来；如果改成 10^{-3}，则在验证集上的效果会变差，当然因为我会把在验证集上的效果纳入保存模型的考虑，所以最终结果也不会差很多。

3. 学习率

目前 CNN 和 RNN 选择的学习率是 10^{-3}，经测试如果改成 10^{-2}，训练效果会明显变差；而如果改成 10^{-4}，则训练速度明显变慢，因此这个速率的选择是比较合适的。

全连接神经网络选择的学习率是 0.05，似乎往大小调整一点都不会有太大的问题。由于它与前两者的权值更新算法不一样，所以二者的学习率并不具有可比性。

4. RNN 的神经元种类

有 LSTM 和 GRU 两种可选，经测试二者的训练效果几乎一样，而后者的训练速度更快一些(尤其是使用了 tensorflow 提供的专为 N 卡优化的 CudnnGRU 后)，所以我选择了 GRU。

6 问题思考

1. 实验训练什么时候停止是最合适的？简要陈述你的实现方式，并试分析固定迭代次数与通过验证集调整等方法的优缺点

我选择的判据很简单：每隔一定更新次数后在验证集上测试一次，如果在验证集上的准确率是当前最好的，则保存模型后继续训练，否则不保存模型直接继续训练。

固定迭代次数：避免了在验证集上测试的开销，训练速度会稍快一些，但是需要合适地选择迭代的停止次数，否则很有可能发生过拟合。

通过验证集调整：这样做可以基本避免过拟合，然而由于我的验证集是选定不变的，且它和测试集的数据分布可能有一定差异，因此在它上面达到的最好准确率对于测试集来讲并不一定适用，所以这样的保存出来的模型也不一定就能在测试集上达到很好的效果。

2. 实验参数的初始化是怎么做的？不同的方法适合哪些地方？

对于 CNN 和 RNN，我直接使用了 tensorflow 提供的 `global_variables_initializer`，查阅文档得知对于 dense 层等结构的 W，这一默认的初始化是 Xaiver 初始化，而对于 b 则是全 0 初始化。Xaiver 初始化的含义如下：

```
It draws samples from a uniform distribution within [-limit, limit],
where limit is sqrt(6 / (fan_in + fan_out))
```

```
where fan_in is the number of input units in the weight tensor
and fan_out is the number of output units in the weight tensor.
```

依据 Xavier Glorot 和 Yoshua Bengio 发表的论文 *Understanding the difficulty of training deep feedforward neural networks*，这种初始化方法可以让每一层输出的方差应该尽量相等，从而使得网络中信息更好的流动。

对于全连接网络，我手动实现了 Xaiver 初始化。

我对于这些初始化方法的特点，优劣，适用场景等并不是很了解，以下内容有些是从网络上摘录得到的。

零均值初始化：这似乎并不是一种确定的初始化的方式，例如 Xaiver 初始化和 $\sigma = 0$ 的正态分布初始化都属于零均值初始化，因此无法评价它的优劣。

正态分布初始化：正态分布的参数有可能与 0 偏离很远，从而导致计算出来的新权值绝对值较大。此时如果采用 sigmoid 或 tanh 激活函数的话，权值处的导数很小，导致训练很慢。

正交初始化：用以解决深度网络下的梯度消失、梯度爆炸问题，在 RNN 中经常使用的参数初始化方法。

分析得，参数梯度正比于参数矩阵特征值 λ_i 的 t 次方。

如果 $|\lambda_i| > 1$，则步数增加时 λ_i^t 超出浮点范围，发生梯度爆炸，优化无法收敛；如果 $|\lambda_i| < 1$，步数增加时 λ_i^t 趋近于 0，发生梯度消失，优化停滞不前。

理想的情况是，特征值绝对值为 1。则无论步数增加多少，梯度都在数值计算的精度内。这样的参数矩阵 W 是单位正交阵。把转移矩阵初始化为单位正交阵，可以避免在训练一开始就发生梯度爆炸/消失现象

3. 过拟合是深度学习常见的问题，有什么方法可以防止训练过程陷入过拟合

我采用了三个方法：

1. 通过验证集来判断当前的模型是否值得保存。如果模型已经过拟合，则不会保存它
2. 在 CNN 和 RNN 的损失函数上，添加了 L_2 正则项
3. 在 CNN 和 RNN 训练时使用了 dropout 机制

此外，诸如适当减小学习率，适当减小网络层数，维数等方法也可以起到一定的避免过拟合的作用。

4. 试分析 CNN，RNN 相对于全连接神经网络的优点

在文本分类这个问题上，CNN 和 RNN 都能一定程度上利用文本自身的性质：CNN 可以将相邻的词使用同一个卷积核来处理，这样可以提取相邻的词之间的关系，同时参数数量不会太大；RNN 可以记住整个序列的历史信息，从而有可能从整个文章的角度来判断文章的类别。

不过我一直认为这种“优点”具有很强的主观性，虽然很形象，但是如果没有严谨的数学论证，是并没有什么说服力的。或许现在这两个基本的模型已经能够用数学很好的解释了，然而很多新兴的模型是缺少这样的解释的，很多时候只能用 it works 来验证这种优点。

7 心得体会

本次实验主要的收获在于我基本熟悉了使用 tensorflow 进行深度学习的基本流程，对于深度学习中的一些重要元素有了基本的认识；同时，通过我手写的全连接神经网络 (如果没写错的话)，对神经网络的底层算法有了更深刻的理解。

本次实验主要的遗憾是没有把 RNN 写好，这样的准确率完全不能接受。也许再调整一下网络参数就可以达到比较理想的效果，但不得不说在这样缺少理论基础的情况下，通过不断的试错来试图达成一个目标，对我的能力提升并没有什么太大价值。如果可能的话，我希望以后能够在以后对 RNN 的理论基础有了更好的理解之后再来尝试修改现在的网络。