情感分析任务实验报告

计算机科学与技术 75 班 赵成钢 2017011362

2019 年 5 月

1 摘要

本次实验，我使用 Word-Embedding 的方法表示助教提供的数据集，分别使用 CNN、RNN 和 MLP（全连接神经网络）的各种模型用 PyTorch 实现了代码进行研究，对其中的一些具体的问题进行了探究，下面我会具体阐述。

2 代码实现

2.1 环境要求

1. Python3
2. PyTorch
3. TorchText
4. SkLearn
5. SciPy
6. TensorBoardX（可选项）

2.2 运行方法

code 文件夹中的 main.py 为程序入口，输入'python3 main.py –config path_to_config'（config 前是两个横线）即可，后面接的是配置文件的路径。配置文件为标准的 JSON 格式，在 configs 文件夹下有很多样例，下面对其

中的 cnn_with-static-w2v.json（带有固定的预训练词向量的 CNN）进行举例说明：

```
{
    "sgns_model": "sgns.sogou.word",
    "dataset": "data",
    "cuda": true,
    "comment": "cnn_with-static-w2v",
    "tensorboard": true,
    "train": "sinanews.train",
    "test": "sinanews.test",
    "preload_w2v": true,
    "epoch": 50,
    "dropout": 0.5,
    "freeze": true,
    "learning_rate": 0.001,
    "train_batch_size": 128,
    "test_batch_size": 128,
    "model": "CNN",
    "loss": "cel",
    "rnn_type": "gru",
    "bidirectional": false,
    "vector_dim": 300,
    "filter_num": 256,
    "class_num": 8,
    "kernel_size": 5,
    "hidden_dim": 256,
    "rnn_layers": 2,
    "fix_length": 2500
}
```

cnn_with-static-w2v.json

1. sgns_model 参数为预训练的 Word-To-Vector 模型，如果你想调用，需要在 code 文件夹建立名为 sgns 的目录，并把模型放进去，把模型的文件名作为该参数的设置，如果不使用预训练模型可以忽略该选项
2. dataset 参数为数据集所在位置
3. cuda 参数为是否启用 CUDA 加速计算，程序会根据 GPU 设备是否可用进行进一步判断
4. comment 参数为此次运行的备注，主要用来 TensorBoardX 可视化
5. train 为训练数据的文件名，格式同助教所给

6. test 为测试数据的文件名，格式同助教所给
7. preload_w2v 为是否使用预训练的模型
8. epoch 为训练轮数
9. dropout，在每个模型中均加入了 nn.Dropout 层，该参数为 dropout 的比例，0 为不启用
10. freeze 为是否固定预训练的模型，如果不使用预训练模型可以忽略该选项
11. learning_rate 为训练的学习率设置
12. train_batch_size 为训练时 MiniBatch 的大小
13. test_batch_size 为测试时 MiniBatch 的大小
14. model 为选用的模型，可以写 MLP、CNN 或者 RNN
15. loss 为损失函数，cel 为交叉熵，mse 为 MSE
16. rnn_type 为 RNN 的类型，可以选择 lstm 或者 gru，如果 model 选项不是 RNN，可以忽略这一项
17. bidirectional 为 RNN 是否双向，如果 model 选项不是 RNN，可以忽略这一项
18. vector_dim 为 Embedding 层向量的维数，即词向量的维数，注意如果使用预训练的词向量这里需要一致
19. filter_num 为 CNN 中 filter 的个数，如果 model 选项不是 CNN，可以忽略这一项
20. class_num 为最后的类别数，助教的训练集为 8
21. kernel_size 为卷积核的大小，如果 model 选项不是 CNN，可以忽略这一项
22. hidden_dim 为 RNN 的内部的 hidden_dim 或为 MLP 中间层的节点数，如果 model 选项是 CNN，可以忽略这一项

23. rnn_layers 为 RNN 中 LSTM Cell 或 GRU Cell 的个数，如果 model 选项不是 RNN，可以忽略这一项

24. fix_length 为数据 Padding 到的固定长度，如果 model 选项不是 MLP，可以忽略这一项

2.3 代码框架

代码已上传到：https://github.com/LyricZhao/EmotionAnalyzer，其中 code 文件夹问具体实现代码，其中大框架用 PyTorch 实现，用了 TorchText 处理数据，TensorBoardX 对数据可视化，还有一些科学计算的库来辅助计算。

2.4 main.py

程序的入口，主要来加载文件，调用数据集、模型和训练的接口，在流程中传递数据。

2.5 dataset.py

加载数据集的接口，助教给的数据集格式用 Tab 符号分隔，也就是原生的 TSV 格式，我用 TorchText 库把文件分成 Index、Label、Text 三个 Field，通过其前后处理的接口把数据去字母和数字，把出现次数最多的 Label 最为最后的 Label，同时还会进行混洗等操作。

2.6 model.py

定义了 CNN、RNN 和 MLP 的模型结构，具体结构见后面的章节。

2.7 trainer.py

进行训练和测试的代码，同时还会用 TensorBoardX 的接口对数据进行记录和可视化。

2.8 本地环境

1. Intel(R) Xeon(R) CPU E5-2670 v3

2. NVIDIA Telsa P100
3. Ubuntu 16.04
4. PyTorch 1.10
5. Python 3.6.7

3 网络结构

3.1 CNN

CNN 的网络结构如图 1所示，下面我对每一层会进行详细的解释：

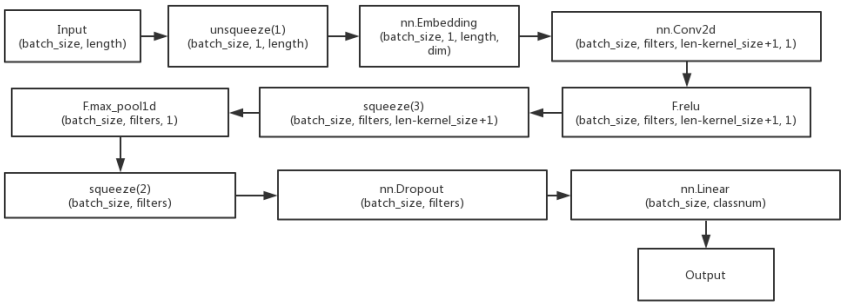

图 1: CNN 网络结构

1. 首先输入为一个 MiniBatch，包含 batch_size 个长度相同的已经数字化的文章，如果长度不相同已经做过 Padding 处理
2. 后经过 unsqueeze 操作增加一维 Channel，再通过 Embedding 层转换为词向量，这时又多了一个词向量的维度
3. 之后进行卷积操作，卷积核的大小为 (kernel_size, vector_dim)，这样相当于同时对 kernel_size 个词进行操作，并把最后一个词向量的维度乘为长度为 1
4. 之后在 squeeze 一次把最后一个维度去掉，这时再经过一个 max_pool 层把在最后一维上取 max

5. 这时再 squeeze 一次把最后的 max 维度去掉，再经过一个全连接层得到 8 个分类的值，最后的概率体现在 loss 中，这点后面会提到

对应的代码如下：

```
class CNN(nn.Module):
    def __init__(self, config):
        super(CNN, self).__init__()
        self.vector_dim = config['vector_dim']
        self.class_num = config['class_num']
        self.kernel_size = config['kernel_size']
        self.filter_num = config['filter_num']
        self.vocabulary_size = config['vocabulary_size']

        self.embedding = nn.Embedding(self.vocabulary_size, self.vector_dim)
        if config['preload_w2v']:
            self.embedding = self.embedding.from_pretrained(config['vectors'
                ], freeze=config['freeze'])
        self.conv = nn.Conv2d(in_channels=1, out_channels=self.filter_num,
            kernel_size=(self.kernel_size, self.vector_dim))
        self.dropout = nn.Dropout(config['dropout'])
        self.fc = nn.Linear(self.filter_num, self.class_num)

    def forward(self, x, lengths):
        x = x.unsqueeze(1) # x: (batch_size, 1, len)
        x = self.embedding(x) # x: (batch_size, 1, len, wv_dim)
        x = self.conv(x) # x: (batch_size, filter_num, len - kernel_size +
            1, 1)
        x = F.relu(x).squeeze(3) # x: (batch_size, filter_num, len -
            kernel_size + 1)
        x = F.max_pool1d(x, x.size(2)) # x: (batch_size, filter_num, 1)
        x = x.squeeze(2) # x: (batch_size, filter_num, 1)
        x = self.dropout(x)
        x = self.fc(x) # x: (batch_size, class_num)
        return x
```

model.py:cnn

3.2 RNN

RNN 的网络结构如图 1所示，下面我对每一层会进行详细的解释：

1. 首先输入为一个 MiniBatch，包含 batch_size 个长度相同的已经数字化的文章，如果长度不相同已经做过 Padding 处理

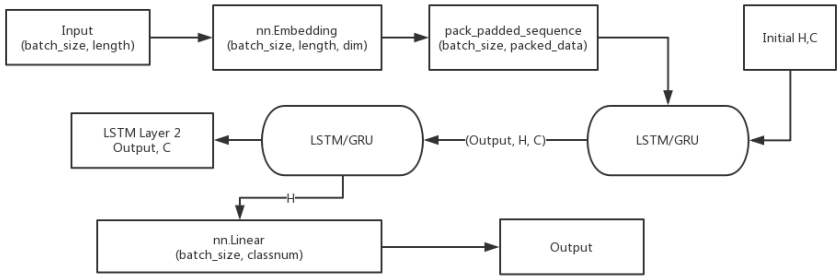

图 2: RNN 网络结构

2. 后经过通过 Embedding 层转换为词向量，这时又多了一个词向量的维度

3. 这时注意并不能直接传入 RNN 中，经测试如果 Padding 到相同的长度，会导致序列敏感的 RNN 无法收敛（或者说效果比较差），这里需要 pack_padded_sequence 把等长的文章重新按照长度连续放在内存里，这样的话在经过后面的 RNN Cell 时就不会把大量无用的填充字符喂入网络，造成很大的负面影响

4. 后面用不同的方法产生 (H,C)（如果是 LSTM 的话，GRU 为 H）的初始状态，和输入一同喂入 RNN Cell 中，经过 2 层 Cell 得到输入和最终状态

5. 最后把最后一次的状态 H 通过全连接层得到对于 8 个分类的输出

6. 对于 RNN 的 Dropout，已经体现在 LSTM/GRU Cell 中，作为一个参数在代码中体现

对应的代码如下：

```
class RNN(nn.Module):
    def __init__(self, config):
        super(RNN, self).__init__()
        self.vector_dim = config['vector_dim']
        self.class_num = config['class_num']
        self.vocabulary_size = config['vocabulary_size']
        self.hidden_dim = config['hidden_dim']
        self.layers = config['rnn_layers']
```

```
        self.use_cuda = config['cuda']
        self.batch_size = config['train_batch_size']
        self.rnn_type = config['rnn_type']

        self.embedding = nn.Embedding(self.vocabulary_size, self.vector_dim)
        if config['preload_w2v']:
            self.embedding = self.embedding.from_pretrained(config['vectors'
                ], freeze=config['freeze'])
        if self.rnn_type == 'lstm':
            self.rnn = nn.LSTM(self.vector_dim, self.hidden_dim, batch_first
                =True, dropout=config['dropout'], num_layers=self.layers,
                bidirectional=config['bidirectional'])
        else:
            self.rnn = nn.GRU(self.vector_dim, self.hidden_dim, batch_first=
                True, dropout=config['dropout'], num_layers=self.layers,
                bidirectional=config['bidirectional'])
        self.fc = nn.Linear(self.hidden_dim, self.class_num)

    def forward(self, x, lengths):
        # x: (batch_size, len)
        x = self.embedding(x) # x: (batch_size, len, wv_dim)
        x = pack_padded_sequence(x, lengths, batch_first=True)
        if self.rnn_type == 'lstm':
            x, (h, c) = self.rnn(x) # x: (batch_size, len, hidden_dim)
            x = h[-1, :, :] # x: (batch_size, hidden_dim)
        else:
            x, h = self.rnn(x) # x: (batch_size, len, hidden_dim)
            x = h[-1, :, :] # x: (batch_size, hidden_dim)
        x = self.fc(x) # x: (batch_size, class_num)
        return x
```

model.py:rnn

3.3 MLP

这里 MLP 作为 Baseline 和前二者做对比，只是通过把文章 Padding 到一个固定的比较大的长度也取得了较好的效果，具体的网络结构如图 1所示，下面我对每一层会进行详细的解释：

1. 首先输入为一个 MiniBatch，包含 batch_size 个长度完全相同（所有的 MiniBatch 都相同）的已经数字化的文章
2. 后经过通过 view，把最后两维合并成一维并通过第一个全连接层，映射到中间层并 Dropout

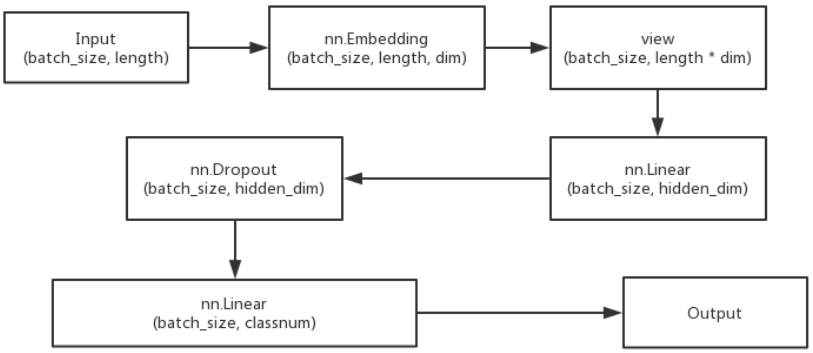

图 3: MLP 网络结构

3. 最后再经过一次全连接层得到 8 个分类的输出

对应的代码如下:

```
class MLP(nn.Module):
    def __init__(self, config):
        super(MLP, self).__init__()
        self.vector_dim = config['vector_dim']
        self.class_num = config['class_num']
        self.vocabulary_size = config['vocabulary_size']
        self.fix_length = config['fix_length']
        self.hidden_dim = config['hidden_dim']

        self.embedding = nn.Embedding(self.vocabulary_size, self.vector_dim)
        if config['preload_w2v']:
            self.embedding = self.embedding.from_pretrained(config['vectors'
                ], freeze=config['freeze'])
        self.dropout = nn.Dropout(config['dropout'])
        self.fc1 = nn.Linear(self.fix_length * self.vector_dim, self.
            hidden_dim)
        self.fc2 = nn.Linear(self.hidden_dim, self.class_num)

    def forward(self, x, lengths):
        x = self.embedding(x) # x: (batch_size, len, wv_dim)
        x = x.view(x.size(0), -1) # x: (batch_size, len * wv_dim)
        x = self.dropout(self.fc1(x))
        x = self.fc2(x) # x: (batch_size, class_num)
        return x
```

model.py:mlp

3.4 Loss 函数

代码为 Loss 函数提供了交叉熵(Cross Entropy Loss)或均方误差(Mean Square Error)，可以在选项中调整。

3.4.1 交叉熵

Pytorch 中的 nn.CrossEntropyLoss 已经顺序包含了 nn.Softmax、nn.Log 和 nn.NLLLoss，将会把网络的最后一层输出映射到 $[0,1]$ 区间的概率，之后取取对数计算交叉熵作为 loss 函数。

3.4.2 均方误差

第二种 loss 就是在数据处理时保留情感分布而非最大值，这样在得到最后一层后再经过一次 Softmax 得到概率分布后直接和原始数据分布计算 MSE 作为 loss，具体的结果后面会讨论。

4 实验

4.1 CNN

4.1.1 预训练模型的影响

首先，我参照助教提供的 CNN 文本分类论文实现了代码，并实现了论文中：A、没有预训练的 Embedding 层；B、固定预训练好的 Embedding 层；C、不固定预训练好的 Embedding 层。下面是我具体测试出的结果（loss 是交叉熵函数，50 个 Epoch，Acc 表示准确率，FScore 即为对类平均的 FScore，Pearson 代表平均分布相关系数，0 为训练集上的表现，1 为测试集上的表现）：可以看出，三者在训练集上的表现没有太大差异，不过在测试集上的表现差异悬殊。其中加了预训练模型的 Embedding 层对词的语义有更好的理解，故表现更佳。而不固定词向量的方法相比固定，在准确率和相关系数上差别不大，而可能因为对训练集有更好的亲和性在 FScore 上表现较好。

Embedding	Acc0	Acc1	FScore0	FScore1	Pearson0	Pearson1
Non	0.9985	0.5745	0.9975	0.2	0.8657	0.5671
Static	0.9995	0.6045	0.9991	0.2892	0.8664	0.6235
Non-Static	0.9995	0.6061	0.9992	0.3056	0.8660	0.6222

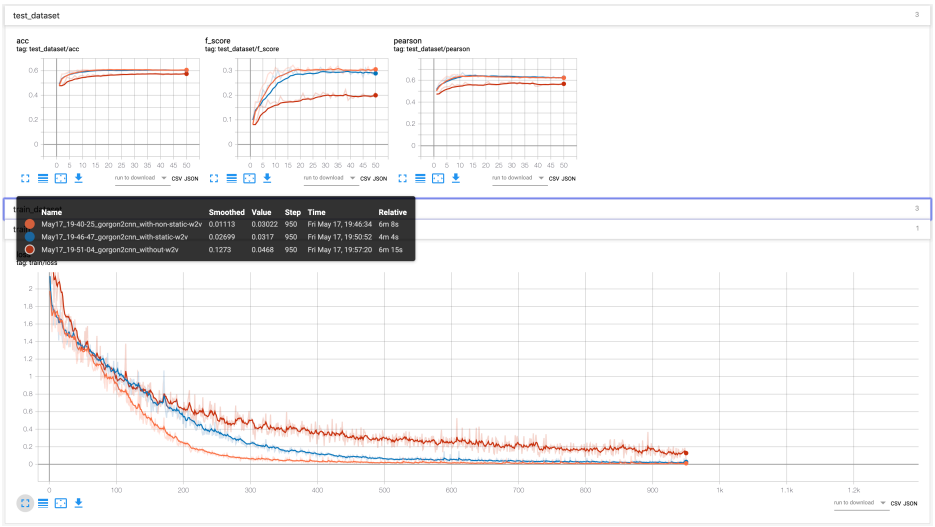

图 4: CNN 训练过程

4.1.2 走向

除了最后的结果，过程也比较重要，下面给出 TensorBoard 记录的数据如图 4所示。

可以看到，从训练时间上来看，固定 Embedding 层收敛，因为这里的参数不需要调整；而相对迭代次数来说，不固定有预训练的 Embedding 层最快，因为其对数据亲和性更好。

4.2 RNN

4.2.1 预训练模型的影响

同 CNN 一样，首先我对 RNN 的预训练模型的影响进行了实验，需要说明的是在训练过程中出现了一定程度的过拟合，这里的结果为训练过程中的最好的结果。同样的，三者在训练集上的表现没有太大差异，不过在测

Embedding	Acc0	Acc1	FScore0	FScore1	Pearson0	Pearson1
Non	0.9983	0.4778	0.997	0.1978	0.8658	0.3447
Static	0.9947	0.5582	0.9917	0.2402	0.8633	0.586
Non-Static	0.9967	0.5694	0.9941	0.2402	0.8658	0.5604

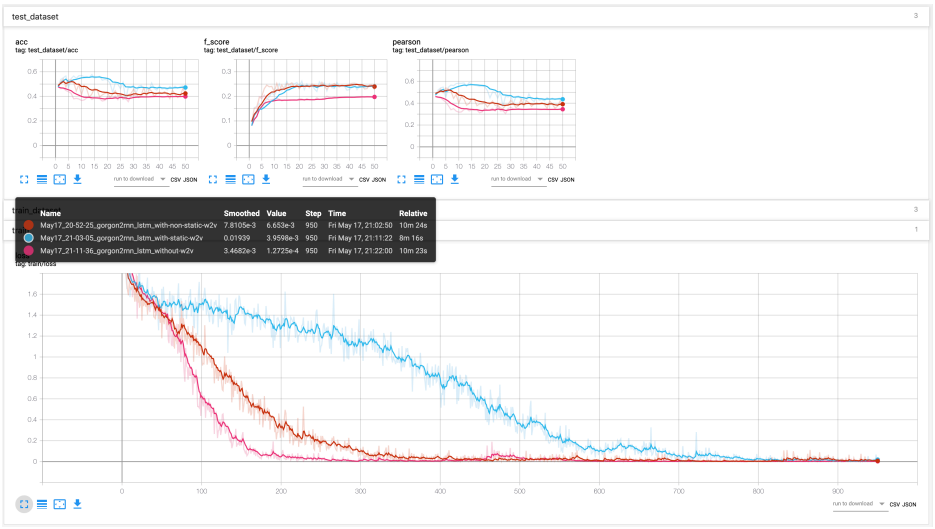

图 5: RNN 训练过程

试集上的表现差异悬殊。其中加了预训练模型的 Embedding 层对词的语义有更好的理解，故表现更佳。而不固定词向量的方法相比固定，在准确率和相关系数上差别不大，而可能因为对训练集有更好的亲和性在 FScore 上表现较好。

4.2.2 走向

同样下面给出 TensorBoard 记录的数据如图 5所示。

可以看到在准确率为尺度看，RNN 出现了一定程度的过拟合，准确率几乎在第 2 个 Epoch 上表现最好，后面一直在波动，而 FScore 则一直在增加，相关系数也出现了一定程度的过拟合，后面在我做的大量实验中，我推测是因为数据集分布不好而且规模较小造成的。而不同于 CNN 的是，在 loss 下降时，没有用预训练模型的 RNN 收敛的最快，我这里的推测是 RNN 真的过拟合的；相比之下，固定词向量的 RNN 收敛最慢，过拟合的程度也

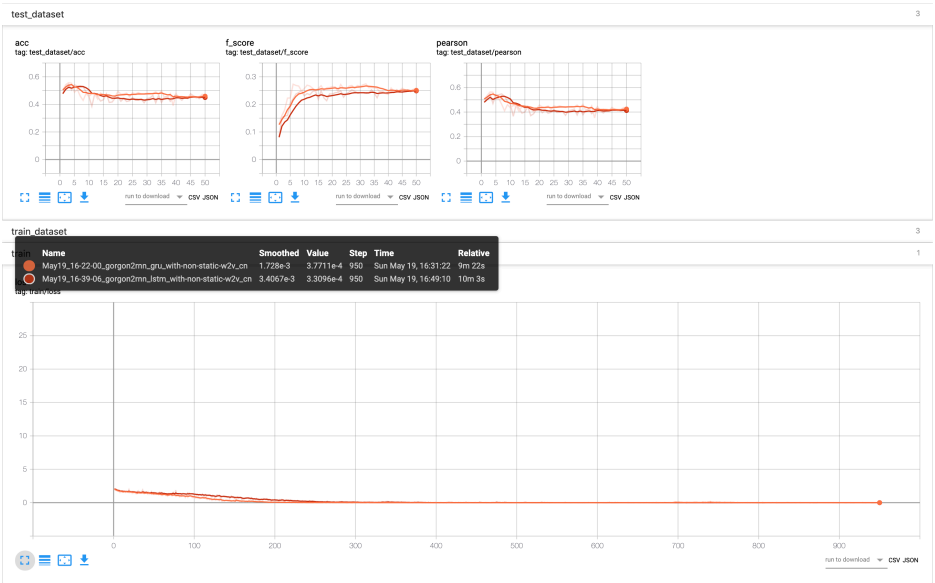

图 6: GRU 和 LSTM 对比

比较低（这在三个指标上表现也最好），不过收敛很慢。但是令人欣喜的是，RNN 学习速度非常快，在几乎第 2 个 Epoch 就得到了最高的准确率，虽然没有 CNN 高，不过作为业界一个久经考验的模型，相信问题更多在我这里和训练集上。

进一步我对 RNN 和 CNN 的看法将在后面的思考中给出。

4.2.3 GRU 相比于 LSTM

在实现的过程中，我也尝试了 GRU，为了方便比较这里比较的都是有预训练模型的非静态 Embedding 层的模型，图 6是二者的对比，可以发现在三个指标上和 loss 的收敛速度上 GRU 均比 LSTM 略好，我的推测是 GRU 的结构组合了 LSTM 的遗忘门和输入门到一个单独的更新门，参数更少更加简单。

4.2.4 Bidirectional LSTM

单向的 RNN 只能对输入的历史状态有记忆功能，但是不能对未来的上下文信息作出一定的预判，尤其在复杂的中文语义环境下，所以我对双向结构的 LSTM 也进行了实验，结果取得了非常好的效果，和普通 LSTM 的对

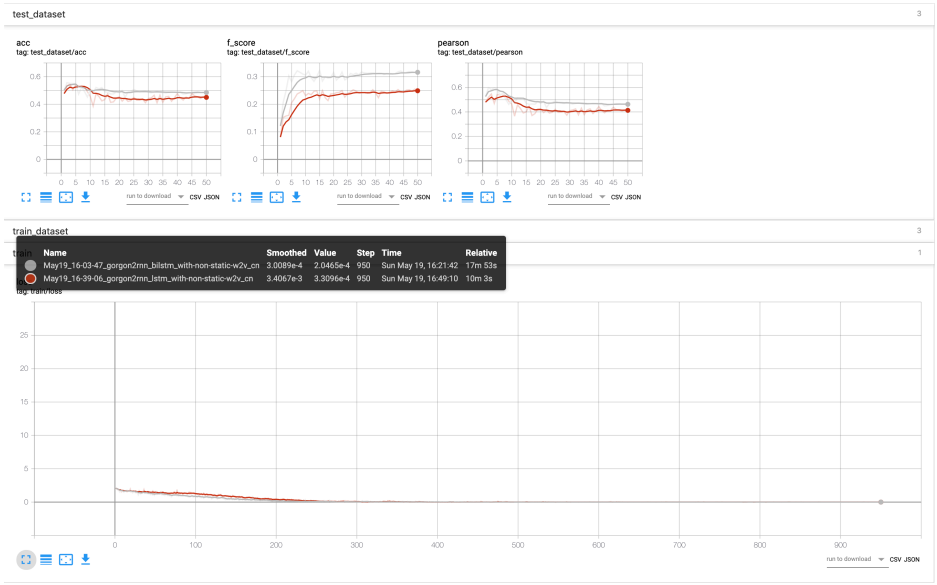

图 7: 双向 LSTM 和 LSTM 的对比

比如图 7，相对于时间上，双向 LSTM 比普通的因为参数多近乎一倍所以速度也响应慢一倍，不过在三个指标上完胜普通的 LSTM，得到了 0.55 的正确率，0.3174 的 FScore，0.46 的相关系数，在 FScore 上的得分甚至超过了之前最好的 CNN。

4.2.5 Padding 到相同长度

上述的结果均是使用了 nn.utils.rnn.pack_padded_sequence 会把 TorchText 已经 Padding 好的数据重新按照长度连续放在一起，之前我对 MiniBatch 中长度固定的方式也进行过实验，只是完全没有收敛，下面连在训练集上都没有收敛。经验性的，我认为文本的长度不一，过多 Padding 的填充字符对输入顺序敏感的 RNN 有很大影响，去掉 Padding 的填充字符即可收敛，效果相对较好。

4.3 MLP

4.3.1 预训练模型的影响

同 CNN 一样，首先我对 MLP 的预训练模型的影响进行了实验，并得到了下面的结果。同样的，在 MLP 这里，三种方法的差距在测试集上表现

Embedding	Acc0	Acc1	FScore0	FScore1	Pearson0	Pearson1
Non	0.9952	0.4821	0.9945	0.1435	0.8619	0.4288
Static	0.9998	0.5563	0.9996	0.2317	0.8658	0.5287
Non-Static	1	0.576	1	0.2463	0.8659	0.5605

悬殊，明显是不固定比固定比没有预训练的好。

4.3.2 走向

据如图 8所示为 MLP 的训练过程，和 CNN、RNN 都不太相同，过程并没有出现过拟合，相比另外两者因为网络结构更加简单所以训练速度非常快。但是从 loss 曲线中可以看到，没有加预训练模型的 MLP 训练过程非常抖动 loss 变化很大，这和 MLP 本身只是两层线性结构相对应，在 Embedding 层的巨大变化很可能会导致对结果的影响非常大，而用预训练好的方法相比之下效果就会很好。

可以看到，从训练时间上来看，固定 Embedding 层收敛，因为这里的参数不需要调整；而相对迭代次数来说，不固定有预训练的 Embedding 层最快，因为其对数据亲和性更好。

4.4 Dropout 层对过拟合的作用

为了防止过拟合，我在实验中的三个模型都加了 Dropout 层，Dropout 层会在每次把一定比例的参数设置为不更新参数，这样的随机性在一定程度上也就避免了过拟合，为了突出 Dropout 层的效果，我对每个模型的带有不固定预训练模型的版本和原来的版本进行了对比，下图 9是实验的结果（在三个评价指标中相同颜色线的上面是带有 Dropout 的，在 loss 曲线中因为下降的更慢所以也在上面），不过可以看到的是没有 Dropout 层收敛更快，而且在训练集上会保持很长一段时间的 1 的准确率。

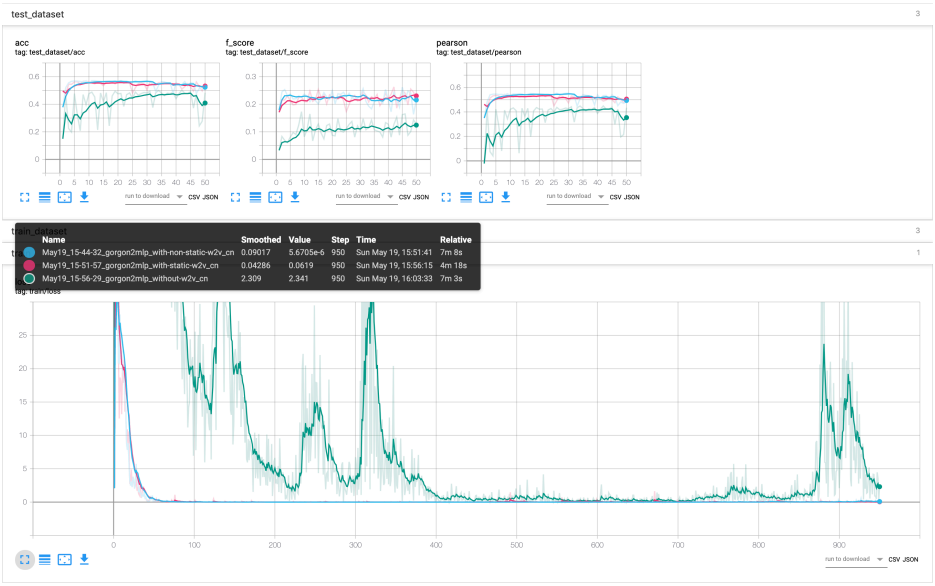

图 8: MLP 训练过程

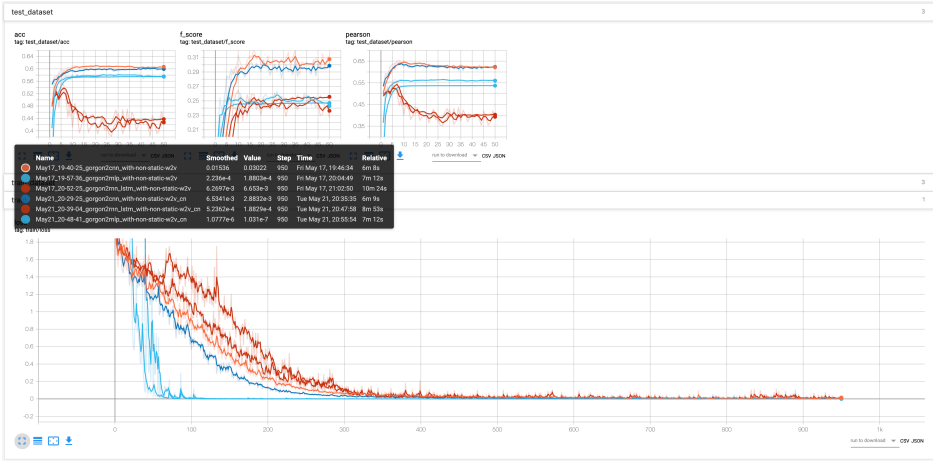

图 9: 有无 Dropout 的对比

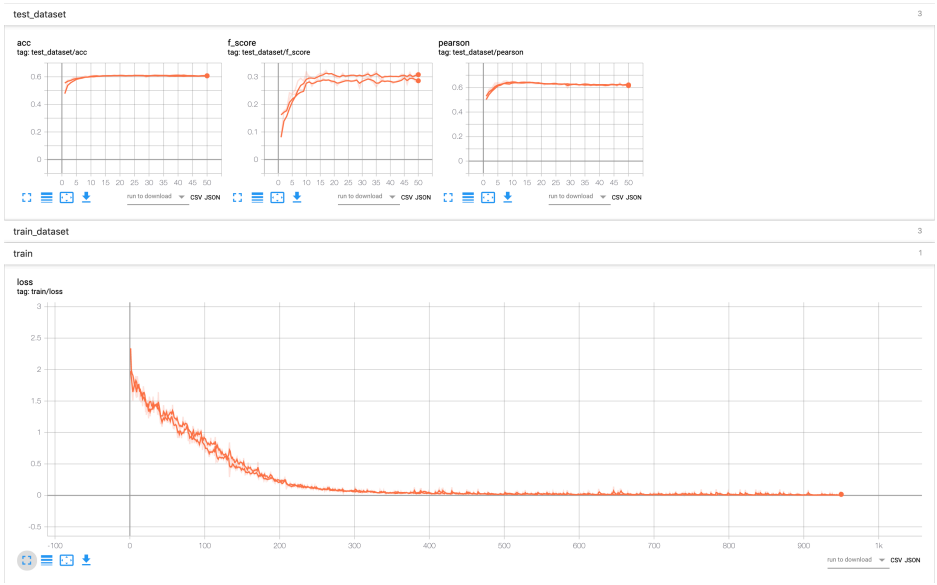

图 10: sgns.sogou.bigram-char

4.5 其他预训练模型

在 https://github.com/Embedding/Chinese-Word-Vectors 中，还有用 Word+Character+Ngram 等更多的 Context Features 训练出来的预训练模型，词向量的质量更高，图 10是一个对比，由于计算资源和时间有限，我只进行了 CNN 的实验，用新的 Word+Character+Ngram 训练好的模型在三个指标和 loss 下降速度中均取得了更好的效果。

4.6 MSE 作为 loss

我也尝试过保留情感的分布，并把 MSE 作为 loss，很不幸的是 MSE 并没有收敛，如图 11是把交叉熵和 MSE 作为 loss 的 CNN 模型的对比。我们可以从图中看出 MSE 作为 loss 根本没有收敛，这里我的猜测是用户们对一篇文章有很大的随机性，分布规律可能随着时间不同而不同，而强调了标签的交叉熵则忽略了这种随机的规律使得效果反而更好。我可以想到的一个更好的策略是把数量最大但相同的多个标签或数量近似的次大标签也放入网络中，即把文章复制几份带有不同的可能的标签，由于时间有限，我没有做这个尝试。

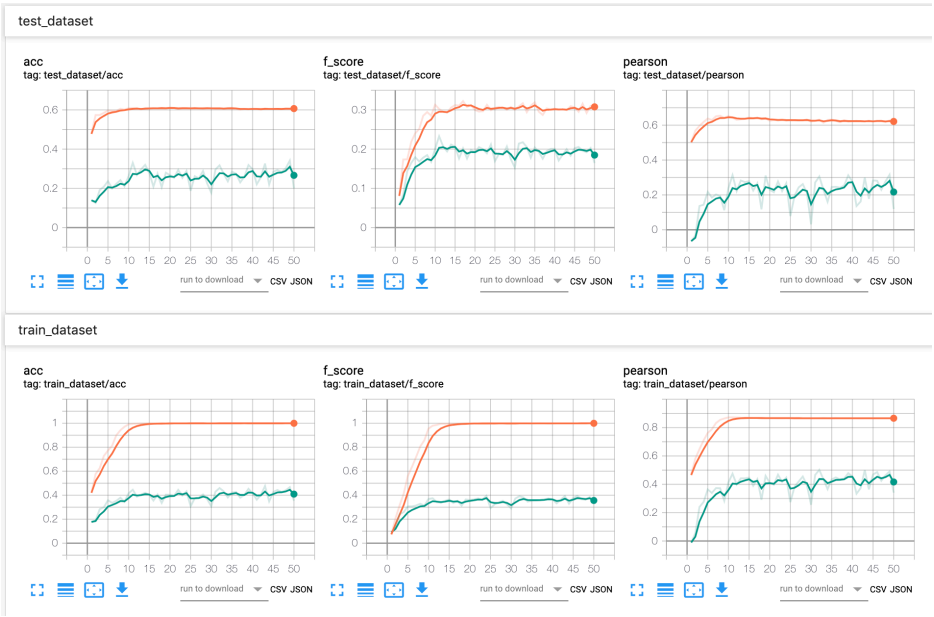

图 11: MSE 和交叉熵对比

4.7 Best Collection

为了方便比较和助教的判断，我把我三种模型的最好结果和对应配置给出。

	Model	Value
BestAcc	CNN_Non-Static-Pretrained_Sogou.Bigram.Char	0.61
BestFScore	CNN_Non-Static-Pretrained_Sogou.Word	0.3123
BestPearson	CNN_Static-Pretrained_Sogou.Word	0.6343

CNN's Best

	Model	Value
BestAcc	RNN_BiDir_Non-Static-Pretrained_Sogou.Word	0.5644
BestFScore	RNN_BiDir_Non-Static-Pretrained_Sogou.Word	0.315
BestPearson	RNN_BiDir_Non-Static-Pretrained_Sogou.Word	0.6028

RNN's Best

	Model	Value
BestAcc	MLP_Non-Static-Pretrained_Sogou.Word	0.5761
BestFScore	MLP_Non-Static-Pretrained_Sogou.Word	0.2651
BestPearson	MLP_Non-Static-Pretrained_Sogou.Word	0.5619

MLP's Bests

5 思考

5.1 思考题

5.1.1 实验训练什么时候停止最合适

对于这个问题的回答，我们需要注意考虑的有几点：一是训练有没有充分；二是有没有出现过拟合；三是过多对于计算资源的浪费。

为了验证第一个问题，首先需要做出判断的就是 loss 还有没有在下降，在测试集上的表现如何，这两个可以直接从数值上看出，进一步的，经验性的，我的做法是把学习率降低到大约原来的十分之一，这样可以避免学习率过大导致参数无法收敛到一个精细的解上。

第二个问题是过多的训练会出现过拟合的效果，为了一定程度上避免这个问题，我的做法是在第一个的基础上观察 loss 有没有下降，如果出现严重的下降，则需要 Early Stopping。

（第三个问题是为了省些电费）

我觉得这个问题非常开放，我们需要对于不同模型、不同参数等等不同情况作出自己的判断。

5.1.2 实验参数的初始化的影响

对于这个问题，我现在还没有一个比较好的答案，响应作出的尝试也比较少，我代码中采用的方法针对网络中的不同层有不同的策略，这一段代码直接调用了网络上一段开源的代码，比如：对于 RNN 中的参数采用了正交

初始化、对卷积层进行了 Xavier 均匀分布初始化等等，具体的原因推导网络上有很多，这里不再复读。

我也尝试了给 CNN 进行 Gauss 分布初始化，最后导致开始的 loss 非常大，训练了完整的 50 个 Epoch 却也下降不到 Xavier 分布初始化的结果；而给 RNN 进行 Gauss 分布初始化，也出现了类似的情况；而给 RNN 进行 Xavier 初始化，效果和正交初始化相当。

网络的参数初始化在一定程度上是为了避免训练中的梯度消失或者梯度爆炸，虽然不能完全避免这个问题，但是好的初始化会有很大的改善，没有依据的初始化会导致上述的问题，我在实验中也几次遇到了这种情况，不过没有进行更加深入的研究。

5.1.3 避免过拟合

第一个可以想到的方法就是在上一个小节中提到的 Dropout 层，在上面的实验结果中也体现了较好的效果，在此不再赘述。

另外一些经验性的，可以想到的是：

1. 增大数据集（提个小建议是这个数据集规模过小，而且测试集和训练集比例太大了，个人见解）
2. 从数据集的分布产生更多数据，比如这个可以把次大的标签也作为最大重复出现一次，见上一节
3. 网络结构适中，过复杂会导致网络的拟合能力过强
4. Early Stopping，对于网络中的激活函数，训练时间正合适会让激活函数处在线性区，拟合能力弱
5. 输入中增加噪声，这也是我为什么没有除了数据中少量乱码的原因
6. 网络参数初始化

5.2 网络的比较

5.2.1 CNN

可以看出的优点是速度非常快，而且最后效果相对比较好，我对于 CNN 的理解是它通过卷积来对每 kernel_size 个词语进行特征的提取，同时这些特征也通过 max_pool 和 relu 来激活最后把特征通过来映射到分类的结果

上。不过缺点是，在一定程度上“记忆”非常短暂，如果在非常复杂的语义环境下可能会出现断章取义的结果。

5.2.2 RNN

RNN 是 3 个模型中相对迭代次数而言学习最快的，在第二个 Epoch 就达到了非常高的准确率，其中的长期记忆的设计真正的在一定程度上避免了断章取义的问题，实现了信息的持久化，不过就结果上看，RNN 比较容易出现过拟合的问题，我的猜测是 RNN 的 Cell 中的 tanh 导致的。

5.2.3 MLP

在三种模型中，MLP 是结构最简单的，训练最快的做法，不容易过拟合，但是就我的理解来说，MLP 不会记住特征，但是却一次性充分利用了所有的信息。不足的是，结构过于简单，虽然在本次的实验中取得了较好的结果，但是我认为也只是看到了文章中的某些词语而断章取义，比如看到了“跳楼”等词，或者看到了词向量中某个本身蕴含情感的维度就会把最后的文章定性，在一定程度上也是一种断章取义。我甚至觉得，如果把所有的词之间加权平均得到一个文章向量，再连接到网络中都能取得类似的效果，而实际网上也是这样说的，我没有做进一步的尝试。但是不可否定的是，从结果上看，MLP 不失为一个好的方法和思想。

5.3 有关评价指标

5.3.1 准确率

准确率是最为直接的评价指标，但是可以看到的是一篇文章用户可能有多种情感，准确率也一定程度上忽略了这一点，不过最为直观和直接，但是没有体现出来网络学习出来的具体分布。

5.3.2 FScore

我们考虑为什么准确不能完全作为一个好的指标，假如我的模型一直输出“同情”的情感分类，准确率也能达到约 0.1 到 0.2，这非常可观，我们应该同时注意到测试集中真的为”同情”的有多少，我们又猜对了多少；我们预测了多少个“同情”，而这里面又对了多少。这两个指标一个是 Recall，一个是 Precision，而 F-Score 是二者的一个综合，对于多分类问题，对于所

有的类做平均就得到了宏平均的 FScore，相对准确率较好，我们也可以看到 CNN 和 RNN 的 FScore 近似，取得了较好的水平，不能因为 RNN 的准确率低就完全否定它，实际上还是一个非常好的做法的。

5.3.3 相关系数

相关系数相比前两者更加侧重分别，即考虑到了用户对待一篇文章的多种情感，进而得到一个分布。不过需要注意的是对于一篇文章，不同人对待其情感有所差别，有考虑到人的随机性，所以分布略有随机性和迷惑性，而不同文章可能蕴含的情感的分布也是不同的。不过好处是考虑到了更多种情感，更符合作业的题目：情感分析任务。

5.4 进一步改进

由于时间和计算资源有限，我还想到了一些优化，因为时间有限在此只是列举：

1. 数据根据次大的标签产生更多文章，使得数据集规模更大
2. 数据过滤得更加干净，去掉更多噪声，其实 Dropout 的过拟合效果已经很好了
3. 进一步调整网络中各层的大小和参数
4. 对于 MLP 可以使用 Bags of Words 的模型，或者我还看到过用 TF-IDF 作为权重，之后把词向量加权平均，可以作为实验参考
5. 可以尝试 CNN 论文中的多 Channel 的方法
6. 进一步比较朴素 RNN Cell 和 LSTM、GRU 的区别

5.5 实验感受

这次实验是我第二次完整接触神经网络的项目，前一次是用 PyTorch 做人脸超分辨率，当时就觉得很困难，这次实验有了上次的些许经验相对顺利，但也是遇到了很多困难，毕竟一次是 CV 一次是 NLP，感受到了很大不同，也更是有很多相同的地方，收获了很多，更是结合老师上课的讲解加深了我对神经网络的理解。

但是，毕竟没有系统学习过机器学习、神经网络，我认为这次的作业更是一次实验，而非普通的作业，作业是基于自己的完整的体系理解做到这次最好的效果，而面对这样的课题，我觉得更多的尝试和对模型的改进也只是解释性不强的调参性尝试，没有太大意义，应该将更多的时间放在基础内容的学习上而实验为辅。

其次，我认为这次作业任务量较大，对大部分同学可能比较困难，尤其是大家设备悬殊，在我的机器上 GPU 的速度至少是 CPU 的 30 倍，因此可以做更多的实验。但是这可能在一定程度上造成不公平，在此提出这样一个建议。

感谢马老师的耐心讲解和助教的帮助。