人工智能导论——情感分析作业报告

张晨，计 71，2017011307

1 网络结构

我实现了 CNN，RNN，MLP 这 3 类模型，每一类都尝试了若干种不同的网络结构。

1.1 CNN

SimpleCNN 采用 Yoon Kim[1] 提出的模型.

```
def forward(self, x, _):
    x = x.unsqueeze(1)
    x = [conv(x) for conv in self.conv]
    x = [F.relu(i.squeeze(3)) for i in x]
    x = [F.max_pool1d(i, i.size(2)).squeeze(2) for i in x]
    x = torch.cat(x, 1)
    x = self.dropout(x)
    x = self.fc(x)
    return x
```

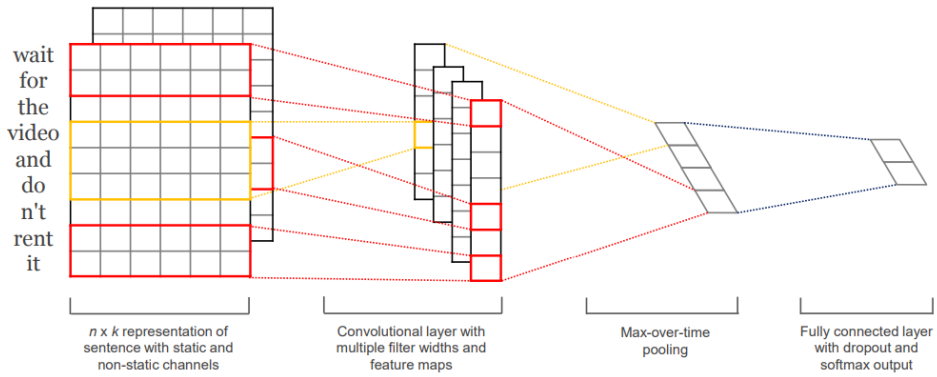

图 1: SimpleCNN 网络结构

DeepCNN 与传统的 CNN 类似，在网络中多次堆叠卷积层、激活函数、缩小一倍的池化层。最后用一个全连接层进行分类。

```
def forward(self, x, _):
    x = self.dropout(x)
```

```
    x = x.unsqueeze(1)
    x = self.conv0(x)
    x = x.squeeze(3)
    x = self.pool0(x)
    x = self.relu0(x)
    x = self.conv1(x)
    # ...
    x = self.conv5(x)
    x = self.fc(x)
    return x
```

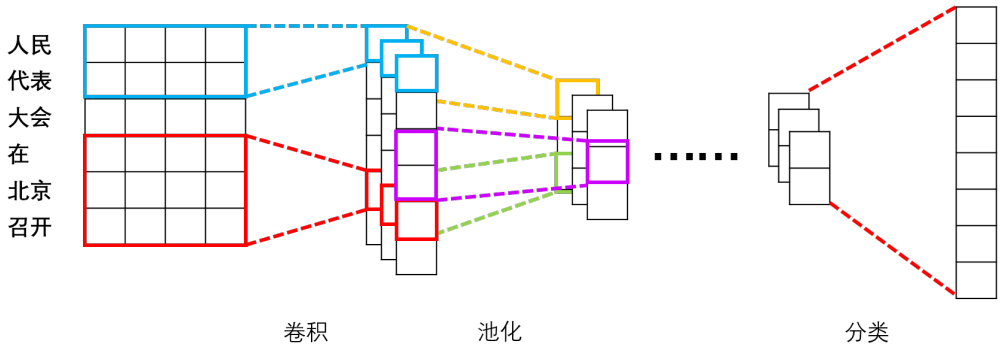

图 2: DeepCNN 网络结构

1.2 RNN

BasicRNN 将词向量依次喂入循环神经网络，把循环神经网络最后一个时刻的输出喂入分类器。

```
def forward(self, x, x_len):
    # ...
    x_padded = nn.utils.rnn.pack_padded_sequence(x, lengths=list(x_len[idx_sort]), batch_first=True)
    _, (ht, _) = self.lstm(x_padded)
    ht = ht.permute([1, 0, 2])
    ht = ht.index_select(0, Variable(idx_unsort))
    ht = ht.view([ht.shape[0], -1])
    output = self.fc(ht)
    return output
```

MPRNN 将词向量依次喂入循环神经网络，取每个时刻的输出的最大值过分类器。

```
def forward(self, x, x_len):
    # ...
    x_padded = nn.utils.rnn.pack_padded_sequence(x, lengths=list(x_len[idx_sort]), batch_first=True)
    states, (ht, _) = self.lstm(x_padded)
    states = nn.utils.rnn.pad_packed_sequence(states, batch_first=True)[0]
    states = states.index_select(0, Variable(idx_unsort))
    states = torch.max(states, dim = 1)[0]
    output = self.fc(states)
    return output
```

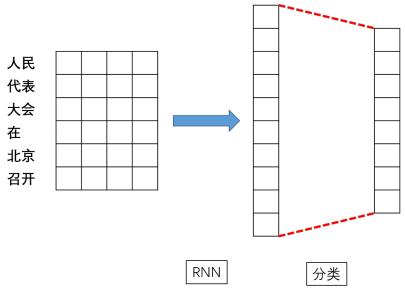

图 3: BasicRNN 网络结构

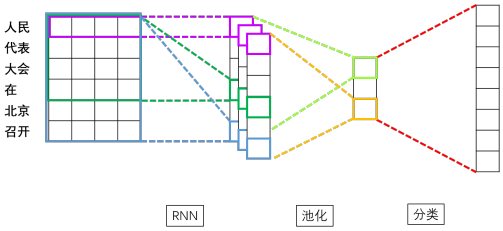

图 4: MPRNN 网络结构

1.3 MLP

FC 将词向量一维化，过分类器。

```
def forward(self, x, _):
    x = x.view(x.shape[0], -1)
    x = self.dropout(x)
    x = self.fc(x)
    return x
```

BasicMLP 将词向量一维化，过全连接层、激活函数，最后进入分类器。

```
def forward(self, x, _):
    x = x.view(x.shape[0], -1)
    x = self.fc1(x)
    x = self.sigmoid(x)
    x = self.dropout(x)
    x = self.fc2(x)
    return x
```

MPMLP(min-max pooling MLP) 对词向量的每一维，分别取输入的词的最大值、最小值。将这个合成向量过全连接层、激活函数、分类器。

```
def forward(self, x, _):
    x = torch.cat([torch.max(x, dim=1)[0], torch.min(x, dim=1)[0]], dim=1)
    x = self.fc1(x)
    x = self.dropout(x)
    x = self.sigmoid(x)
    x = self.fc2(x)
    return x
```

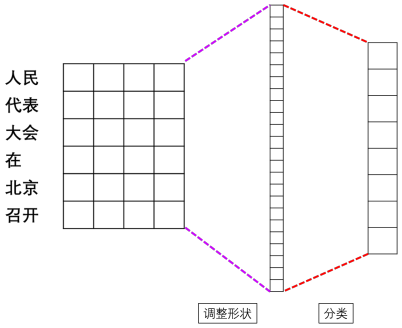

图 5: FC 网络结构

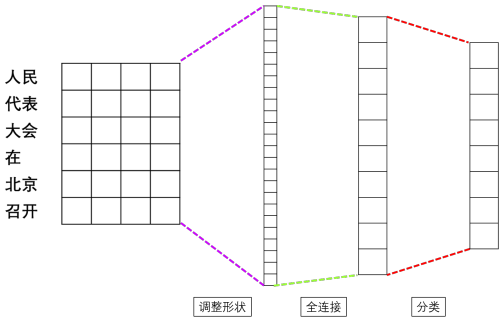

图 6: BasicMLP 网络结构

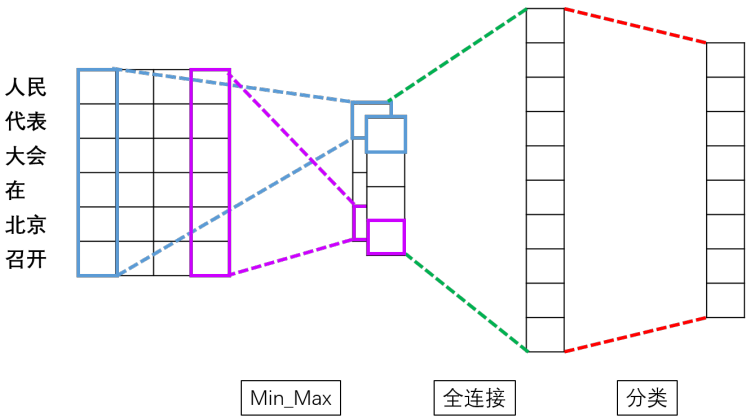

图 7: MPMLP 网络结构

2 实现细节

2.1 输入数据

pytorch 的 dataloader 要求数据的维数严格一致，因此需要截取较长的文章，将较短的文章用 0 补全。图 8中展示了训练集的文章长度分布，为了尽可能保留文章的全部信息，又不至于带来太大的计算负担，我选择将所有文章调成到 1000 个词。此外，文章中有一小部分的词没有出现在预训练的 word to vector 中，我选择将这些位置填上 0。

数据的类别分布极不均衡，为了让模型“愿意”输出出现次数较小的类别，我在 loss function 中加入了类别权重，某一类的权重为它出现次数的倒数。

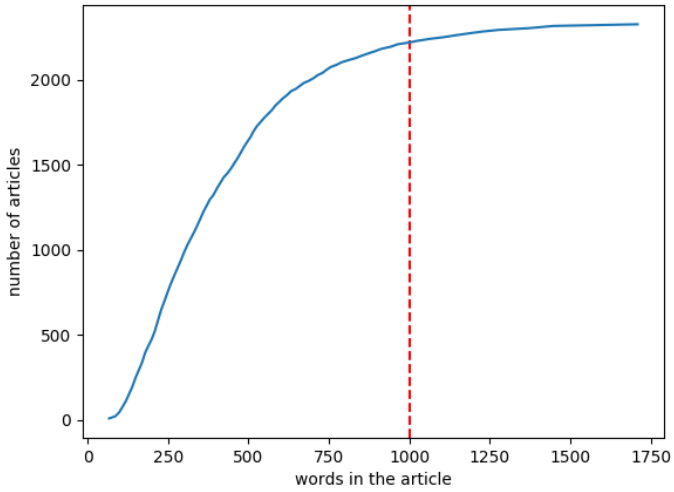

图 8: 训练集文章长度统计

2.2 RNN

在 RNN 中，我使用了 pack_padded_sequence，这一编程接口可以去掉填补的 0。

2.3 脚本化

为了方便调参，我使用 argparse 来读取命令行参数。模型内部可调参数较多，将它们同时暴露给 argparse 不太优雅，因此我在 argparse 中定义了 hyperint 与 hyperfloat 两个变量，分别用来调整模型中的整数参数和浮点型参数。

2.4 结果保存

我保存了训练过程中每一轮的正确率、loss 来辅助我分析训练过程。另外，我还保存了训练中间过程的模型。由于模型数量及文件大小都很大，我仅在精度达到新高或轮数为 100 的整数倍的时候进行

保存。保存的模型命名为 “模型名 + 训练开始时间 + 精度 + 轮数”。这样，在按文件名排序后，可以清楚的看到模型的精度变化过程及最优精度。

2.5 训练加速

训练过程中需要使用用字典格式存的 Word2Vec，但直接从原始数据中生成这个字典需要花费相当长的时间。因此，我使用 pickle 库将生成好的字典以二进制形式存到文件中，程序读取这个二进制文件仅需要很短的时间。这个模型很小，因此 GPU 的利用率抖动十分明显。为了提高 GPU 的利用率，我采用了多进程生成训练数据 + 同时训练多个模型的方法。

3 对比实验

3.1 Dropout Rate

此次任务的特点之一是极易过拟合，因此我选择首先对各种模型寻找尽量优的 dropout rate。基于 RNN 的模型的 dropout 在 RNN 中执行，其余模型的 dropout 体现在之前的源代码中。CNN 的 channel 数为 128，RNN 中间结果长度 128，MLP 隐层长度 512，测试结果如表 1所示 (注：此表中的准确率并不是模型的最终准确率，最终准确率见 “实验结果 “一节)。

模型		0	10%	30%	50%	70%	90%
CNN	SimpleCNN	0.5916	0.5987	0.5969	0.5925	**0.6014**	0.5934
	DeepCNN	0.4143	0.4681	0.4645	**0.4861**	0.4533	0.4610
RNN	BasicRNN	0.5251	0.5117	0.5162	0.5103	**0.5359**	0.5301
	MPRNN	0.5628	**0.5660**	0.5552	0.5444	0.5561	0.5435
MLP	FC	0.5337	0.5162	0.5130	0.5242	**0.5498**	0.5485
	BasicMLP	0.5507	0.5530	**0.5570**	0.5507	0.5566	0.5462
	MPMLP	**0.5153**	0.4430	0.3748	0.2751	0.1548	0.2455

表 1: 模型准确率与 dropout rate 的关系，其中 MPMLP 中仅使用 max-pooling, 未使用 min-pooling

通过表 1可以看出，不同的模型的最优 dropout rate 不尽相同，但 dropout rate 带来的准确度的影响小于模型本身的差异。

在上述模型中，SimpleCNN 的表现最好，因此之后的实验以该模型为主。

3.2 文章长度

新闻的大结构为 “标题-总-分-总”，因而新闻的前面一小部分即可决定新闻的总体情感。例如，尽管下面一则新闻 [2] 很长，但是第一段便已经交代清楚整个事件。因此，仅截取新闻最前面的一部分内容也可以达到较好的效果。

客机起飞半小时后机舱冒烟返航

昨天上午 8 点 25 分左右，由首都国际机场起飞半小时的一架美航 AA186 次航班机舱冒烟，不得不折返降落。事后，返航乘客被安排到附近酒店。下午 5 点多，乘客接到航空公司答复，改乘今天上午 10 点多的航班再次起飞。“希望这次能顺利到达。”乘客表示。

机舱冒烟客机返航

“飞机飞着飞着就冒烟了，北京我又回来了!”昨天上午，一位网友在微博上透露，自己乘坐美国航空公司的 AA186 次航班由北京去芝加哥，但飞机起飞后客舱内突然冒烟，最终飞机不得不返航。“看来这个元旦我必须在北京度过了”，“真是悲剧，飞机都起飞了，然后紧急返航，此刻我又在 T3 了”，另一位网友“全熊小能猫”也用微博直播着这次遭遇。当记者联系上她时，她已被航空公司安排到附近酒店。“我姓黄，是在美国上学的留学生，这次去芝加哥是中途换乘。”黄女士告诉记者，事发时大约上午 8 点 25 分，飞机已经起飞半小时，“突然机舱内飘来一股浓浓的异味儿，有点像烟味儿。紧接着广播中机长的声音证实了这一点，说是飞机临时出现机械故障，导致机舱冒烟，需要返航降落”。记者查询发现，美航 AA186 次航班由首都机场正点起飞时间为 7 点 55 分。

留学生临时当翻译

昨天上午 8 点 55 分，AA186 次航班在起飞一小时后又重新降落在首都国际机场，整个过程无人受伤。黄女士称，飞机落地之后，机舱门迟迟未打开，部分乘客有些着急，不断向身边的乘务员询问为什么。“但那些空姐不懂中文，所以机舱内气氛一度有些混乱。于是我们这些懂英文的乘客就帮忙翻译了一下，这才让机舱内的紧张气氛平息了下来”。黄女士称，她帮忙翻译的大概意思是说，“飞机已经安全落地，请乘客耐心等待，地面空勤人员正在将舷梯接过来，一会儿就可以下飞机了”。微博上，黄女士也将这次特殊的“翻译”经历晒了出来：机上空姐不会讲中文，还找我当翻译，体验了一次当小喇叭广播手的感觉!

乘客今天上午再起程

事后，航空公司将 AA186 全部乘客安排到酒店。“这架飞机是波音 777，而且基本满员，乘客估计得有近 300 人了。外籍乘客去了希尔顿酒店，国内乘客去了丽都酒店。”黄女士透露，在酒店呆了很长时间，航空公司却一直无人出面解释和答复下一步如何，而赔偿问题更是无从谈起。

昨天下午，记者联系到丽都酒店负责协调返航乘客的负责人，他称自己仅是受美航公司委托负责协调乘客的，“飞机何时才能再次起飞，我并不知晓”。首都机场负责乘客改签工作的美航工作人员透露，目前尚未接到上级有关这批乘客改签起飞的具体通知。至于飞机返航的具体原因和赔偿问题等，让记者采访美航北京办事处的媒体传媒部。但昨天下午，记者连续拨打该办事处电话十几次均无人接听。首都机场美航公司工作人员也拒绝为记者转达采访意图。

昨天下午 5 点半，黄女士致电记者，称航空公司刚通知她，他们将乘坐今天上午 10 点多的航班再次起飞。“一大早就得去机场等候，由工作人员带领去取行李。希望这次能顺利到达。”黄女士提前给自己送上了一份新年祝福。

晨报 96101 热线新闻

记者岳亦雷

线索：祝先生

新闻链接

该次航班曾有冒烟返航记录

2010 年 12 月 16 日上午 10 时许，美国航空公司由北京飞往芝加哥的 AA186 航班，起飞 1 小时后客舱突然冒烟，机长决定返航。首都机场解释称，返航是因空调系统故障，机场运行未受影响。美国航空公司北京代表处表示，故障未造成人员伤亡。

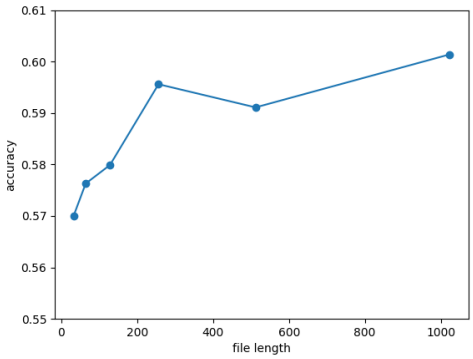

图 9: SimpleCNN 的准确率与文章长度之间的关系

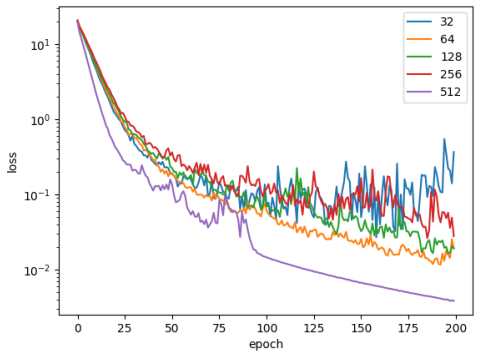

图 10: SimpleCNN 的训练 loss 与文章长度之间的关系

然而，总的来说，文章截取长度越长，训练效果越好。我认为这和训练集太小有关。截取文章过短，则模型训练接收到的语料库很小，因而无法充分训练。loss 的下降速度与文章截取长度的关系不大。

3.3 Optimizer 的选择

常见的 optimizer 有 Adam 和 SGD 两种。在 pytorch 中允许 Adam 不提供任何参数，但是 SGD 必须提供动量参数，且动量参数设置不佳会导致模型不收敛。如在我的 SimpleCNN 中，若将 SGD 设为 0.9 模型便无法收敛，设为 0.99 则可以正常运行。然而，如图 11所示，即便设置了合适的动量参数，SGD 的收敛速度仍慢于 CNN。因此，在一般情况下，optimizer 应选用 Adam。

3.4 随机种子的影响

我使用不同的随机种子，运行了 10 遍同一个 SimpleCNN，结果如图 12所示，这 10 组的标准差为 0.2%，极差为 0.72%，小于模型之间的差距，因此炼丹的时候还是要认真设计模型，调整参数，而不能寄希望于“玄学”。

3.5 CNN

SimpleCNN vs DeepCNN 从表 1中可以看出，SimpleCNN 的效果明显好于 DeepCNN，这主要是因为 DeepCNN 模型表达能力过强，进而发生了严重的过拟合问题。一个已经使用了预训练的 word2vec 的情感分类模型，本身需要拟合的函数不是特别复杂，因而层数较低的 CNN 就可以胜任。

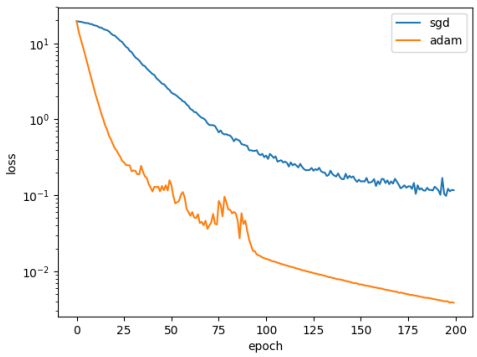

图 11: SimpleCNN 使用不同 optimizer 时的 loss 变化情况

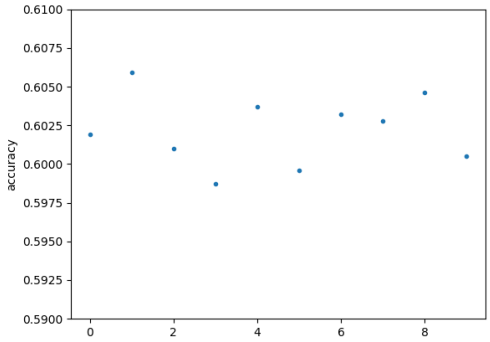

图 12: 用相同参数，不同随机种子训练 SimpleCNN

channel 的选择 图 13中展示了 SimpleCNN 的准确率与 channel 个数之间的关系。channel 数较小时，模型的表达能力较弱，如在 channal 数为 32 时，训练集上最高仅能达到 2296/2304 的准确率，因而在测试集上的表现也就不是特别理想。而在 channel 数太大时，模型表达能力过强，因而更易发生过拟合的问题。

dropout 层的位置 对于 SimpleCNN，dropout 层有以下三种添加方式。这三种添加方式的准确度如图 14所示。从图中可以看出，在 dropout rate 不太大时，三种添加方法的效果基本相同。但是如果选择在输入模型之前就进行 dropout，则 dropout rate 不能设得太大。设得太大会导致训练无法收敛。从直观上讲，面对一篇生词率很高的文章，即便有很强的理解能力，也无法理解文章所讲的内容，因此模型也就无法在 dropout rate 很大时收敛。

```
def forward(self, x, _):
    x = x.unsqueeze(1)
    # pos 1
    x = [conv(x) for conv in self.conv]
    x = [F.relu(i.squeeze(3)) for i in x]
    # pos 2
    x = [F.max_pool1d(i, i.size(2)).squeeze(2) for i in x]
    x = torch.cat(x, 1)
    # pos 3
    x = self.fc(x)
    return x
```

Activation function 的选择 深度学习中常用的 activation function 有 Sigmoid，RELU，TANH，PRELU 等，使用它们进行训练的收敛速度、准确率均有差异。

3.6 RNN

BasicRNN vs MPRNN 从表 1中可知，MPRNN 的效果明显好于 BasicMLP。这是因为 MPRNN 使用了全局每个时间的输出，因而对中间内容有更全的保留。但是，MPRNN 较 SimpleCNN 的准确率

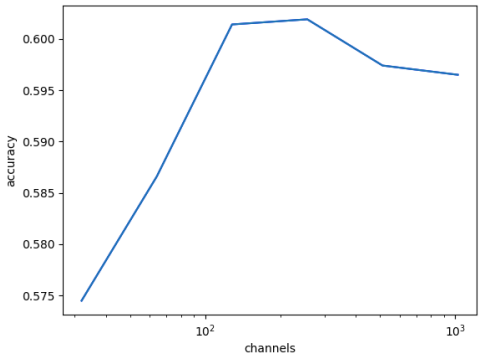

图 13: SimpleCNN 的准确率与 channel 数之间的关系。取 dropout rate=0.7

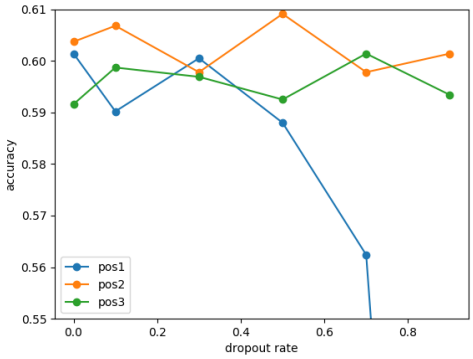

图 14: SimpleCNN 的准确率与 dropout 层位置的关系。

仍有一定距离，这也与过拟合有关。

LSTM vs GRU GRU 在基本保留了 LSTM 的功能的同时，大大简化了网络结构。从图 15中可以看出，二者的准确率基本是一样的。由于按照我的实现，程序瓶颈并不在 GPU 的运算上，故没有看到这二者在运行时间上的差异。

然而，如图 16所示，相较于 GRU，LSTM 的收敛速度快，达到相同准确率基本只需要 GRU 的一半 epoch 数。

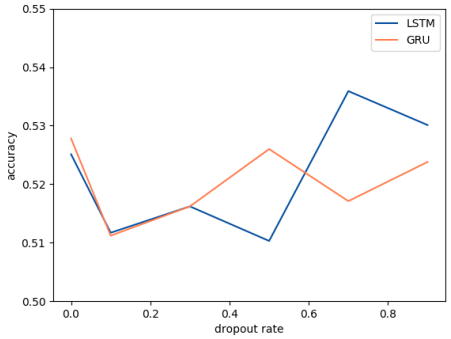

图 15: BasicRNN 中 LSTM 与 GRU 效果对比，取 128 个 channel

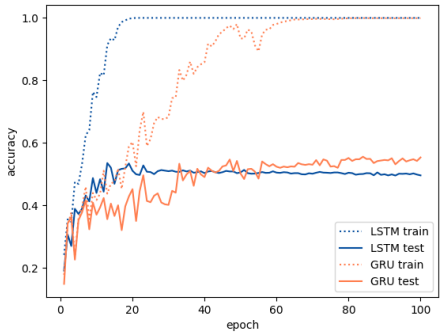

图 16: BasicRNN 中 LSTM 与 GRU 训练过程对比，取 128 个 channel，dropout rate70%

中间层维数的选择 如表 2所示，在我进行测试的范围内，中间层维数越高，模型准确率越好。但是受计算资源的限制，我无法继续增大维数进行测试。

3.7 MLP

FC vs MLP 从理论上讲，FC 的模型中没有非线性层，因而可能无法拟合某些函数。但是，我的 FC 模型与带非线性层的 BasicMLP 模型的表现基本相似。在表 1中，二者 dropout 层的位置不同，直接

维数	32	64	128	256	512
准确率	0.5350	00.5575	0.5476	0.5628	0.5660

表 2: MPRNN 准确率与维数之间的关系

对比不严谨。若将 BasicMLP 的 dropout 层也放在模型输入的位置（如下面的代码所示），最优精度则较为接近。结果如表 3所示。

```
def forward(self, x, _):
    x = x.view(x.shape[0], -1)
    x = self.dropout(x)
    x = self.fc1(x)
    x = self.sigmoid(x)
    x = self.fc2(x)
    return x
```

模型	0	10%	30%	50%	70%	90%
FC	0.5337	0.5162	0.5130	0.5242	**0.5498**	0.5485
BasicMLP	0.5453	**0.5525**	0.5458	0.5516	0.5521	0.5498

表 3: FC、MLP 的准确率与 dropout rate 之间的关系

MPMLP MPMLP 直接对输入数据做 min/max pooling, 这种做法会导致包括上下文信息在内的大量信息的丢失。对其他模型，我至多进行了 200 轮的训练，但对这个模型进行 1000 轮训练，loss 也仅为其他模型训练十多轮的效果。如图 17所示，虽然在训练后期，训练集的准确率仍在稳步上升，loss 仍在稳步下降，但测试集准确率在下降，因此判断继续训练所带来的提升是建立在过拟合的基础之上的。

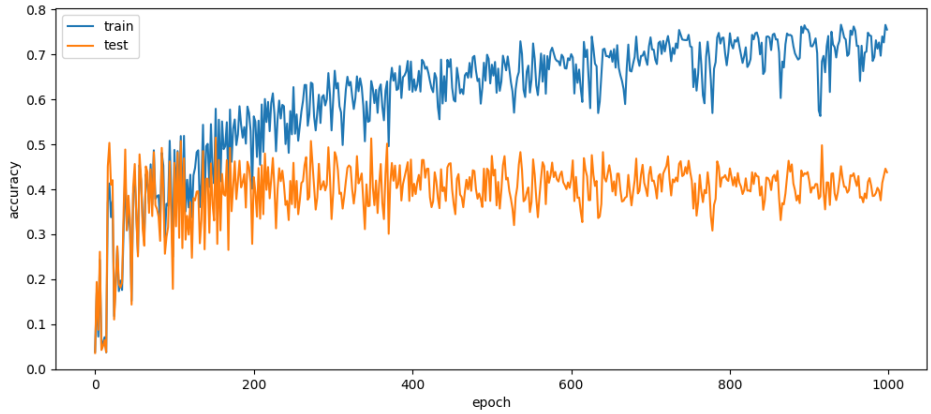

图 17: MPMLP 训练过程中的准确率变化情况

4 一些说明

严谨的讲，应将数据划分成训练集，验证集，测试集三个集合，用训练集进行训练，挑出在验证集上表现最好的模型，再在测试集上进行测试。但是，此次作业标签分布极不均衡，因此想要反应标签的正确分布，使验证集有一定的代表性，验证集中必须要有大量的数据。而这次作业所给的用以划分训练集和验证集的文章仅有 2302 篇，本身就是严重不足的，再从中划分出大量数据作为验证集的代价实在过大。因此，我此次实验略有不严谨的将助教所给的训练集作为我的训练集，测试集作为我的验证集。如果另有数据，我也想知道自己的模型在真正的测试集上的表现。

5 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。**

   在实验过程中，我尝试多次对模型进行超长轮数的训练，但发现由于过拟合问题的存在，模型的最优准确率几乎全部出现在 100 个 epoch 以内，因此我将训练轮数大致固定在 100-200 个 epoch 级别由于不同模型、不同参数的收敛速度存在区别，可能有部分模型的最优点出现在 200 轮之后，因此我在训练过程中记录了每一次刷新准确率时的 epoch 数，并借此判断模型是否有达到更优准确率的潜力。但是，实践过程中，我并没有发现太多有继续提升潜力的模型。

   固定迭代次数实现较为简单，且若最后检查一下是否有继续训练的必要，基本不会导致丢失最优解。然而，固定迭代次数会导致模型进入过拟合还继续训练，从而导致计算资源的浪费（~~但可以选择直接掐掉~~）。倘若选择在验证集准确率长期不提升后终止程序，则可节约计算资源，但是这个阈值的设置特别重要，要防止在模型进入”低谷“和”瓶颈期“的时候终止。这次作业的模型很小，多训练若干轮也无妨，因此我选择了固定迭代次数。

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？（现有的初始化方法为零均值初始化，高斯分布初始化，正交初始化等）**

   在执行构造函数的时候，pytorch 会自动做初始化，因此不需要额外写初始化代码。翻阅 pytorch 源码，可以发现大多数 layer 都使用的零均值初始化。在 pytorch 中设置 layer 的初始化方法较为复杂，故我没有对不同的初始化方法做进一步测试。在 CNN 中，绝不可以把所有权重初始化为 0，因为若所有 channel 的初值相同，则反向传播的过程中，他们的变化都一样，因而失去了设置多个 channel。对于层数较深的神经网络，一般采用高斯分布初始化 +relu 激活函数。

```
class Linear(Module):
    def reset_parameters(self):
        init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in)
            init.uniform_(self.bias, -bound, bound)

class RNNBase(Module):
    def reset_parameters(self):
        stdv = 1.0 / math.sqrt(self.hidden_size)
        for weight in self.parameters():
```

```
            init.uniform_(weight, -stdv, stdv)

class _ConvNd(Module):
    def reset_parameters(self):
        init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in)
            init.uniform_(self.bias, -bound, bound)
```

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。**

   扩充训练集，提前停止，dropout，正则化

4. **试分析 CNN，RNN，全连接神经网络（MLP）三者的优缺点**

   如表 4所示。

   CNN 是我所有模型中准确率最高的，在其他方面也没有明显的问题。

   RNN 对算力要求较高，因为想要”记住“文章之前出现过的内容，中间层必须足够大。我实现的 MPRNN 使用了中间过程中的输出，在一定程度上减小了对记忆深度的要求，因而较 BasicRNN 有较好的效果。

   全连接神经网络参数数量大，但因为深度小，参数复用率低，运行速度和收敛速度都是三类模型中最为出色的。但是，全连接的实现方式，使得它过拟合问题最为严重，且它是三种模型中唯一不支持变长的。MPMLP 是加了池化层的 MLP，它参数数量极小，但由于 pooling 丢失了太多信息，无法达到很高的精度。

模型	运行速度	模型大小	收敛速度	准确率
CNN	中	中	中	高
RNN	慢	中	中	中
MLP	快	大	快	中
MPMLP	快	小	慢	低

表 4: 模型性能比较

6 实验结果

按作业说明文档中的公式进行计算，F-score 取 macro，且若 TP=0，将此类的 F-score 视为 0. 在训练过程中，我仅考虑出现次数最多的标签，而忽略标签的具体分布，因此结果的相关系数可能偏低。

模型		代码中名称	准确率	F-score	相关系数
CNN	SimpleCNN	CNN	0.6176	0.3262	0.6490
	DeepCNN	DeepCNN	0.4942	0.2765	0.4650
RNN	BasicRNN	RNN	0.5364	0.2892	0.5528
	MPRNN	MPRNN	0.5736	0.3522	0.6001
MLP	FC	FC	0.5579	0.2796	0.4519
	BasicMLP	MLP	0.5651	0.2796	0.5997
	MPMLP	MMPMLP	0.5220	0.3182	0.5013

7 实验总结

我在此次作业中进行了较为细致的调参，感受到了”炼丹“的玄学性。但在玄学之上，我发现在模型能够收敛的前提下，调参带来的收益是远远小于设计一个好的模型的。

另外，我也体验到计算资源所带来的优越性，因为能使用 GPU，我可以比周围同学做更多的实验、训练更大的模型，在一定程度上提高了我最终提交的模型的准确度。

为了能拥有尽量多的训练数据，我这次将助教给的整个测试集视为了验证集。经过和同学的讨论，我意识到了这样做的错误的严重性，以后做类似任务的时候，我会改正。

参考文献

[1] Yoon Kim, ”Convolutional Neural Networks for Sentence Classification”, *arXiv:1408.5882* , Aug. 2014

[2] 客机起飞半小时后机舱冒烟返航 `http://news.cntv.cn/20120103/100422.shtml`