PRML Assignment2 报告

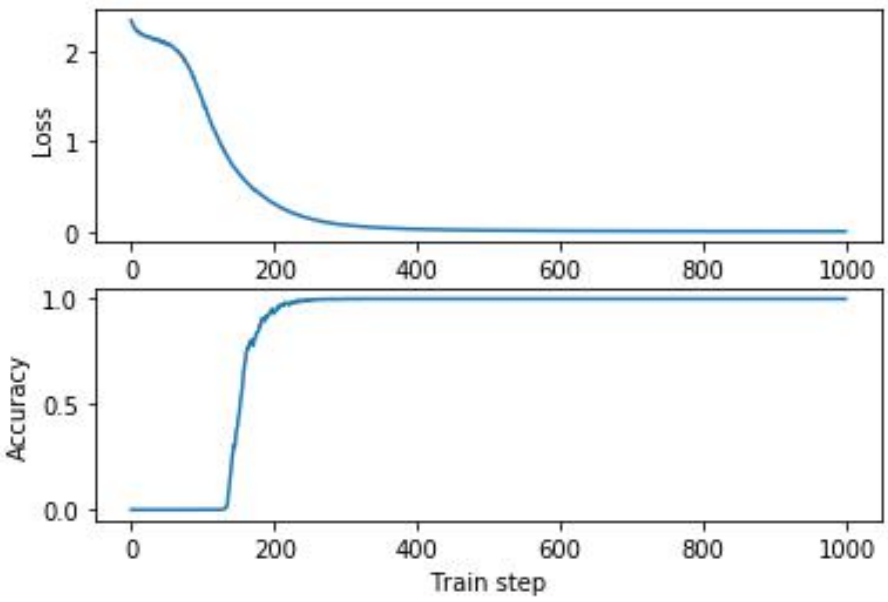
Part 01

第一部分实验要求通过 RNN 算法完成整数求和问题。source.py 调用 pt.py 中 pt_main( )函数实现这一过程。

创建 myPTRNNModel 类 model，里面包含数 number1,number2，通过 embedding 函数将其映射到高维向量中（十进制 10 位×32），将其拼接(torch.cat)的结果 number=[number1;number2]通过 2 层 RNN 结构(self.rnn)，将其结果线性映射到十进制 10 位的向量空间上(self.dense)。将最后的结果作为最大估计进行预测。重复 1000 个 batch，使用交叉熵作为损失函数，使用 torch.optim.Adam 优化。

测试结果如下图：

```
step 0 : loss 2.3032453060150146
step 50 : loss 2.0903074741363525
step 100 : loss 1.7420662641525269
step 150 : loss 0.7945539355278015
step 200 : loss 0.2681187093257904
step 250 : loss 0.08332623541355133
step 300 : loss 0.0363466776907444
step 350 : loss 0.02045026421546936
step 400 : loss 0.013156152330338955
step 450 : loss 0.00974899809807539
step 500 : loss 0.0072812652215361595
step 550 : loss 0.0057766372337937355
step 600 : loss 0.0046830978244543076
step 650 : loss 0.0037573520094156265
step 700 : loss 0.003308959538117051
step 750 : loss 0.002760652685537934 3
step 800 : loss 0.002337432000786066
step 850 : loss 0.0020986744202673435
step 900 : loss 0.0018481172155588865
step 950 : loss 0.0015818466199561954
Accuracy =  1.0
```



总体测试准确率 accuracy=100%，loss 趋近于 0；图像在第 400 个 batch 时接近渐近线。

Part02

0 .根据以给定的 pt_main()程序中代码：

```
model = myPTRNNModel()
optimizer = torch.optim.Adam(params=model.parameters(), lr=0.001)
train(3000, model, optimizer)
```

```
evaluate(model)
```

以及 myPTRNNModel 类的构造，可以得知代码可以对比、修改优化的部分有 cpu->gpu，测试数字的长度，embedding 层向量的维度，RNN 模型的替换，以及学习率 lr 的选取。

通过以下命令：

```
cuda0 = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
torch.LongTensor().to(device)
```

将 pytorch 在 cpu 上的代码放到 gpu 上跑，可增加运行速度。

1 .修改数字长度（pt_adv_main1_add_digital(diglength)）

由于 numpy.random.randint( )函数只能随机取 int32 类型的自然数，而 random 库中自带的 randint( )函数并没有这种限制，所以可以通过修改 data.py 中 gen_data_batch( )函数完成扩大随机数位数的工作。将以下代码：

```
numbers_1 = np.random.randint(start, end, batch_size)
numbers_2 = np.random.randint(start, end, batch_size)
results = numbers_1 + numbers_2
```

替换为下面的代码可以达成目的。

```
numbers_1, numbers_2, results = [], [], []
x, y= 0, 0
for i in range(batch_size):
    x, y = random.randint(start, end), random.randint(start, end)
    numbers_1.append(x)
    numbers_2.append(y)
    results.append(x+y)
```

观察下方生成数据可以发现同一长度的数据，其预测准确度达到 100%所需的 batch 次数波动较小，但总体上与 Part01 中观察的结果一致（step=300 时基本准确度为 100%）

```
The length of digital is : 10
step 0 : loss 2.3566994667053223
step 50 : loss 2.0963032245635986
step 100 : loss 1.6754893064498901
step 150 : loss 0.668658435344696
step 200 : loss 0.20742934942245483
step 250 : loss 0.0740847960114479
step 300 : loss 0.03658248484134674
step 350 : loss 0.021129554137587547
step 400 : loss 0.013902350328862667
step 450 : loss 0.010080700740218163
step 500 : loss 0.007804515305906534
step 550 : loss 0.006071037147194147
When step =  250 Accuracy = 1
```

```
The length of digital is : 10
step 0 : loss 2.318547487258911
step 50 : loss 2.079484701156616
step 100 : loss 1.6717890501022339
step 150 : loss 0.6766113638877869
step 200 : loss 0.20401372015476227
step 250 : loss 0.06723528355360031
step 300 : loss 0.03386310860514641
step 350 : loss 0.01975165866315365
step 400 : loss 0.013100149109959602
step 450 : loss 0.009568310342729092
step 500 : loss 0.007312305737286806
step 550 : loss 0.0058870501816272736
When step =  240 Accuracy = 1
```

```
The length of digital is : 10
step 0 : loss 2.3018453121185303
step 50 : loss 2.06933856010437
step 100 : loss 1.628278136253357
step 150 : loss 0.7264313697814941
step 200 : loss 0.21307605504989624
step 250 : loss 0.0637521743774414
step 300 : loss 0.029709799215197563
step 350 : loss 0.0174189954996109
step 400 : loss 0.01193100493401289
step 450 : loss 0.008648088201880455
step 500 : loss 0.006570360157638788
step 550 : loss 0.005133583210408688
When step =  250 Accuracy = 1
```

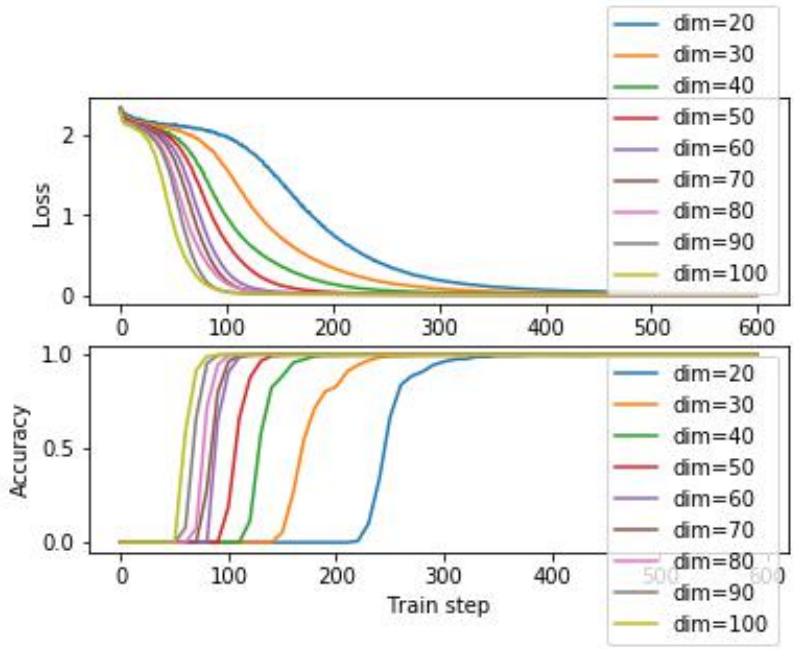
而对于不同长度的数据，其预测准确度达到 100%的次数数据长度关系不大。

```
The length of digital is : 100
step 0 : loss 2.3173506259918213
step 50 : loss 2.054316759109497
step 100 : loss 0.7027155756950378
step 150 : loss 0.10971003025770187
step 200 : loss 0.03183600679039955
step 250 : loss 0.015752211213111877
step 300 : loss 0.009574396535754204
step 350 : loss 0.0066385818645358086
step 400 : loss 0.004928844049572945
step 450 : loss 0.0037849268410354853
When step =  190 Accuracy = 1
```

```
The length of digital is : 150
step 0 : loss 2.322604179382324
step 50 : loss 1.9636037349700928
step 100 : loss 0.6493714451789856
step 150 : loss 0.15321122109889984
step 200 : loss 0.04847566783428192
step 250 : loss 0.02336627058684826
step 300 : loss 0.013847324065864086
step 350 : loss 0.009293586947023869
step 400 : loss 0.006693168077617884
step 450 : loss 0.005102601833641529
When step =  220 Accuracy = 1
```

```
The length of digital is : 200
step 0 : loss 2.317965030670166
step 50 : loss 2.03692626953125
step 100 : loss 0.6254384517669678
step 150 : loss 0.09609684348106384
step 200 : loss 0.03132691979408264
step 250 : loss 0.016237646341323853
step 300 : loss 0.010058354586362839
step 350 : loss 0.006966551300138235
step 400 : loss 0.005122617352753878
step 450 : loss 0.003938736394047737
When step =  190 Accuracy = 1
```

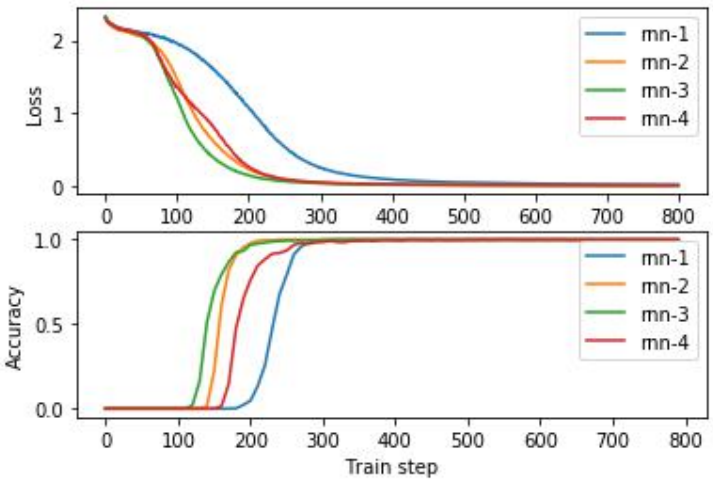
2 .修改数字通过 embedding 层变成的向量的维度 (pt_adv_main2_change_dig())

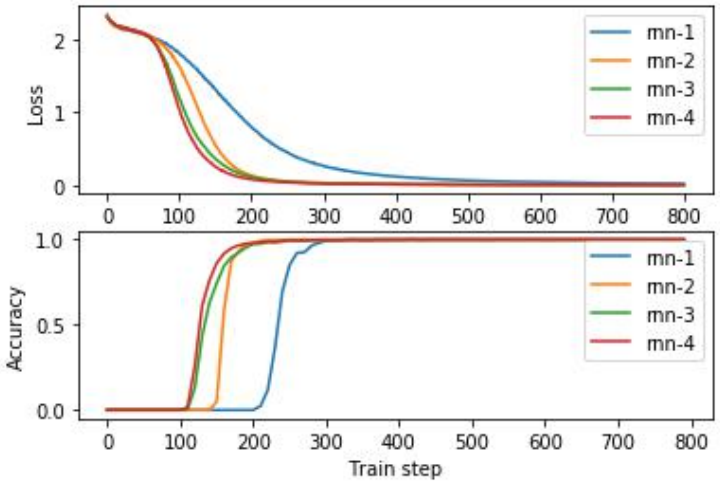


由图示及输出数据可知在 embedding 层数 dim 达到约 70~80 区间范围时，accuracy=1.00 所对应的 step 变化不再明显，层数 dim 在 10~70 范围时随 dim 增加，accuracy=1.00 对应的 step 越小，而当层数 dim 在 70~100 区间中时 accuracy=1.00 对应 step 减小幅度很小。故 dim 可取值在 70~80 之间最优。

3 .修改 RNN 的层数 layers(pt_adv_main3_change_rnn_layer())

RNN-x	1	2	3	4
step	370	260	350	420
RNN-x	1	2	3	4
step	330	250	320	550





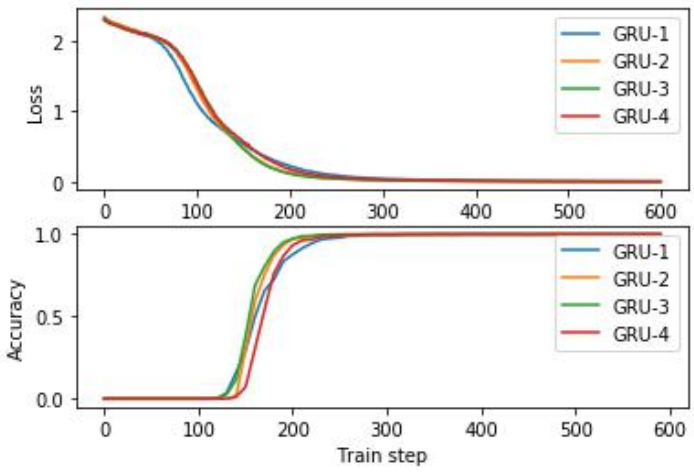
由图示及多组输出数据可知在 RNN 层数 layers=2 时，accuracy=1.00 对应的 step 越小。故 layers=2 最优。

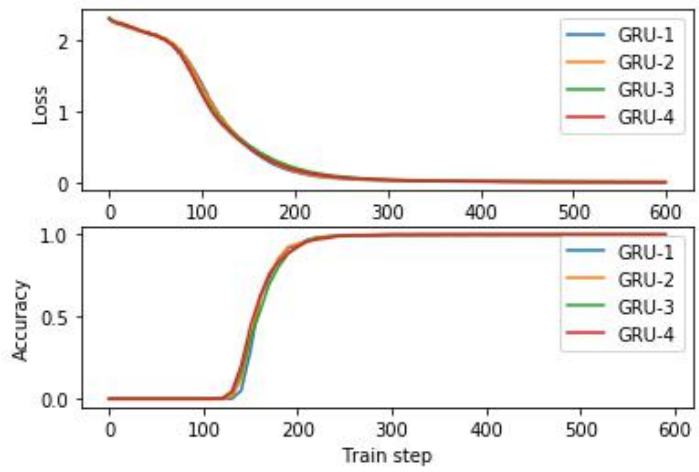
4．更换 RNN 模型为 GRU 或者 LSTM 模型 (pt_adv_main4_change_rnn_to_gru(),pt_adv_main5_change_rnn_to_lstm())

相关测试数据（layers=2，dim=32）如下所示：

GRU：

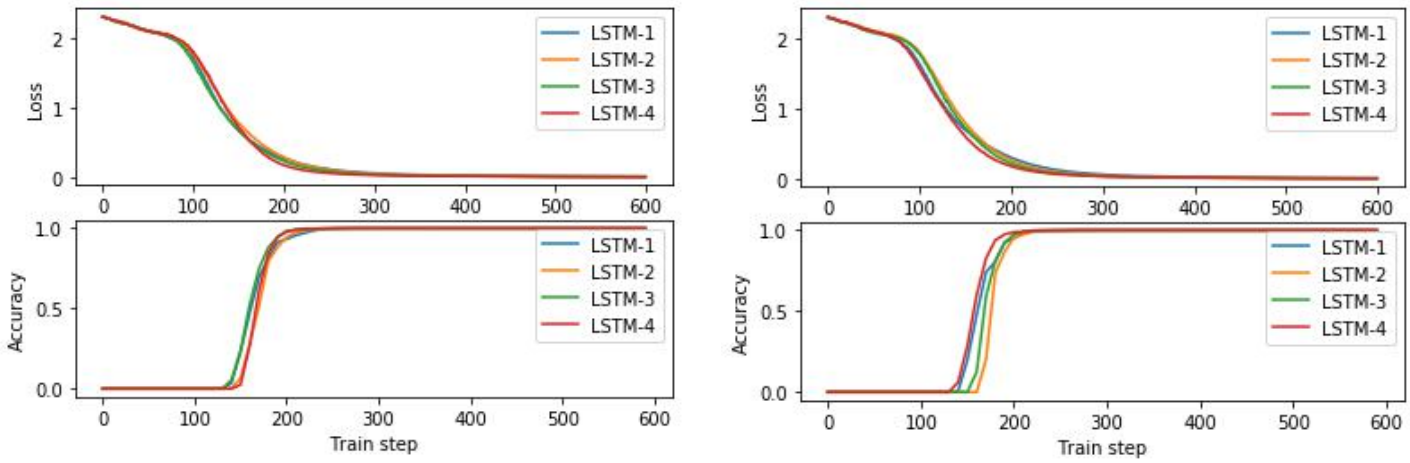
GRU-x	1	2	3	4
step	350	310	260	360
GRU-x	1	2	3	4
step	360	290	310	330





LSTM:

LSTM-x	1	2	3	4
step	300	270	270	260
LSTM-x	1	2	3	4
step	290	260	250	270

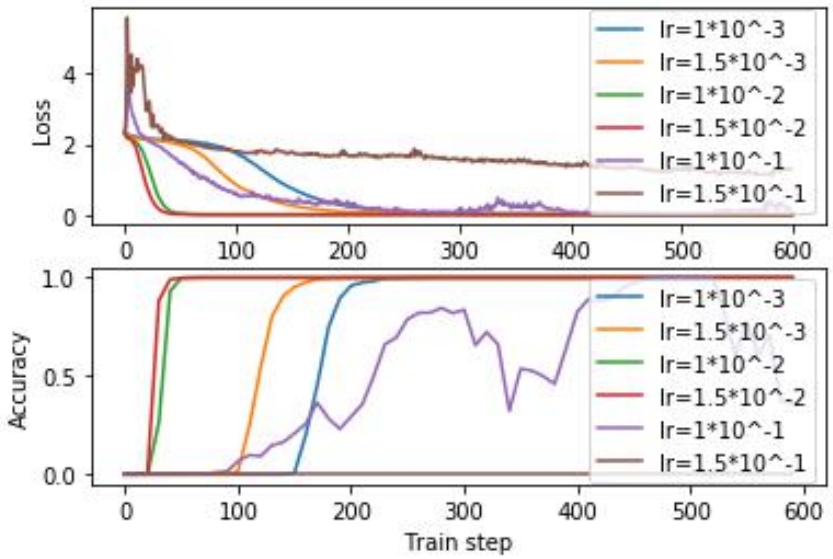


对比 GRU 和 LSTM 层数 layers 数据可知 layers 在 2~3 区间相对更优，对比 RNN 与 GRU、LSTM 数据可知其 accuracy=1.00 对应 batch 次数相差并不明显，但 RNN 相对较优。

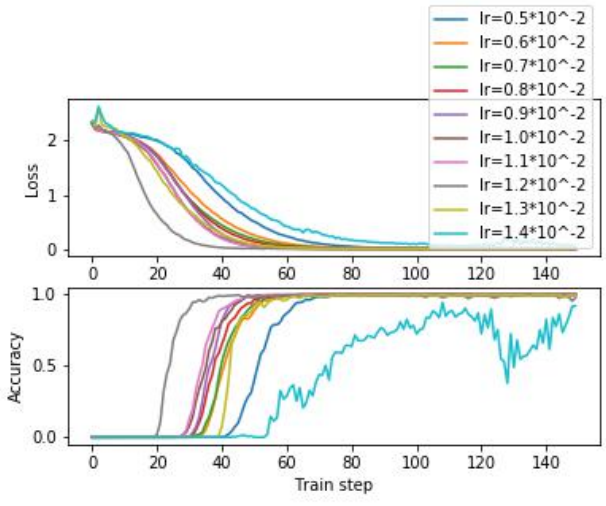
5 .改变 RNN 学习率

(pt_adv_main6_change_lr1(), pt_adv_main7_change_lr2())

学习率对应 RNN（layers=2，dim=32）数据如下所示：



对比可知在 lr 在 10^{-2} 数量级时学习效果最佳，下图为比较 lr 在 10^{-2} 数量级附近学习率，分析可知 lr=1.2*10^{-2} 附近时学习效率最高。



Part03 代码运行命令

```
>>>python3 source.py
```

Part04 参考资料

0 .https://blog.csdn.net/wz2671/article/details/84590843 #RNN 实现二进制加法器案例

1 .https://blog.csdn.net/weixin_42018112/article/details/90084419 #Pytorch nn.Module 模块详解

2 .https://www.cnblogs.com/hancece/p/11177852.html #super().__init__()用法

3 .https://www.cnblogs.com/lindaxin/p/7991436.html #torch.nn.Embedding 解析

4 .https://blog.csdn.net/qq_42079689/article/details/102873766 #PyTorch 的 nn.Linear()详解

5 .https://blog.csdn.net/zhanly19/article/details/96428781 #torch.cat()

6 .https://www.jianshu.com/p/4df3a4262599 #torch.rnn

7 .https://blog.csdn.net/shaopeng568/article/details/95205345 #pytorch to(divice)

8 .https://blog.csdn.net/lkangkang/article/details/89814697 #Pytorch 中 RNN/GRU/LSTM 模型小结