Assignment 2

本实验使用的`pytorch`实现

Task 1 RNN简单实现

1.1 实现过程

简单`RNN`由`embedding`/`RNN`/`Linear`三层结构组成：

- 第一层`embedding`：类似词嵌入模型中的`embedding`层，目的是为了将数字映射到高维中，使得信息更加分散，有点像词向量。
- 第二层`RNN`：这里使用两层的`RNN`结构，为了处理序列。
- 第三层线性映射：这是降维操作，将64映射到十进制10位上。

`forward`函数是`task1`重点，需要知道怎么传递输入和生成输出：

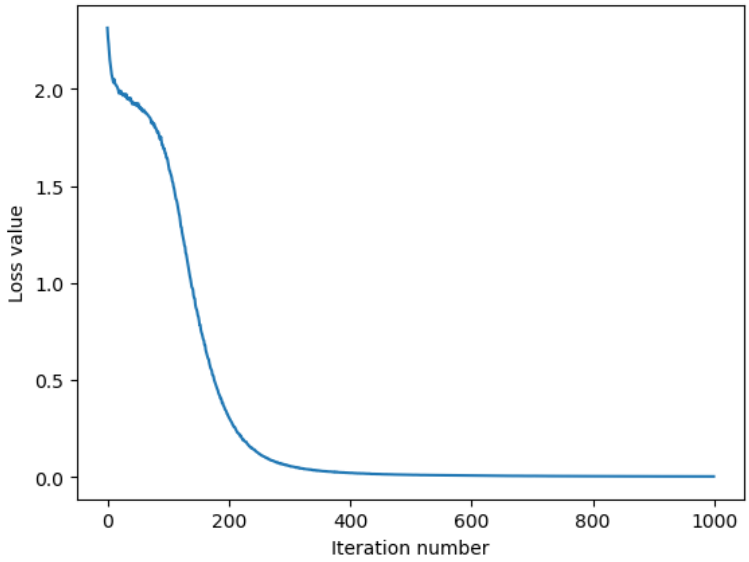
```
def forward(self, num1, num2):
        # num1 (200,11,32)
        # input (11,200,64)
        # output (11,200,64)
        # logits (200,11,10)
    num1 = self.embed_layer(num1)
    num2 = self.embed_layer(num2)
    input = torch.cat((num1, num2), -1).transpose(0,1)
    output, hidden = self.rnn(input)
    logits = self.dense(output)
    logits = logits.transpose(0,1).clone()
    return logits
```

由于`argmax`操作在`evaluate`中，此处只需要输出线性层生成结果。过程中先将两个数字映射到高维，再拼接起来作为`RNN`的输入，最后将`RNN`的输出映射到十进制上。

1.2 Task1 结果

简单加法器测试结果：

- **准确率100%**
- **`loss`趋于0**



上图是记录的`loss`随迭代次数的降低过程，在`>500`后趋于0，收敛稳定。下面是输出的结果，每50输出一次`evaluate`的准确率，在350之后就已经能100%正确了（且是在测试集上）。

```
accuracy is: 0
step 200 : loss 0.3084389865398407
accuracy is: 0.88
step 250 : loss 0.11845769733190536
accuracy is: 0.99
step 300 : loss 0.0564764030277729
accuracy is: 0.995
step 350 : loss 0.030399836599826813
accuracy is: 1
step 400 : loss 0.020080620422959328
accuracy is: 1
step 450 : loss 0.013436054810881615
accuracy is: 1
step 500 : loss 0.010010684840381145
accuracy is: 1
step 550 : loss 0.007755820639431477
accuracy is: 1
step 600 : loss 0.006101652048528194
accuracy is: 1
```

Task 2 RNN优化

由于`task1`中的模型已经可以达到100%的准确率（受GPU限制，最高测试5000位数字），下面改进方法从理论角度入手，对模型进行可能的优化。

优化从**GPU、数字长度、模型选择、初始化、超参**几个方面优化

2.1 GPU性能优化

将`pytorch`在`cpu`上的代码转换为`gpu`上跑，速度快了三四倍左右。

使用了`torch.LongTensor().to(device)`的方法，不知道为什么`torch.tensor()`有问题。

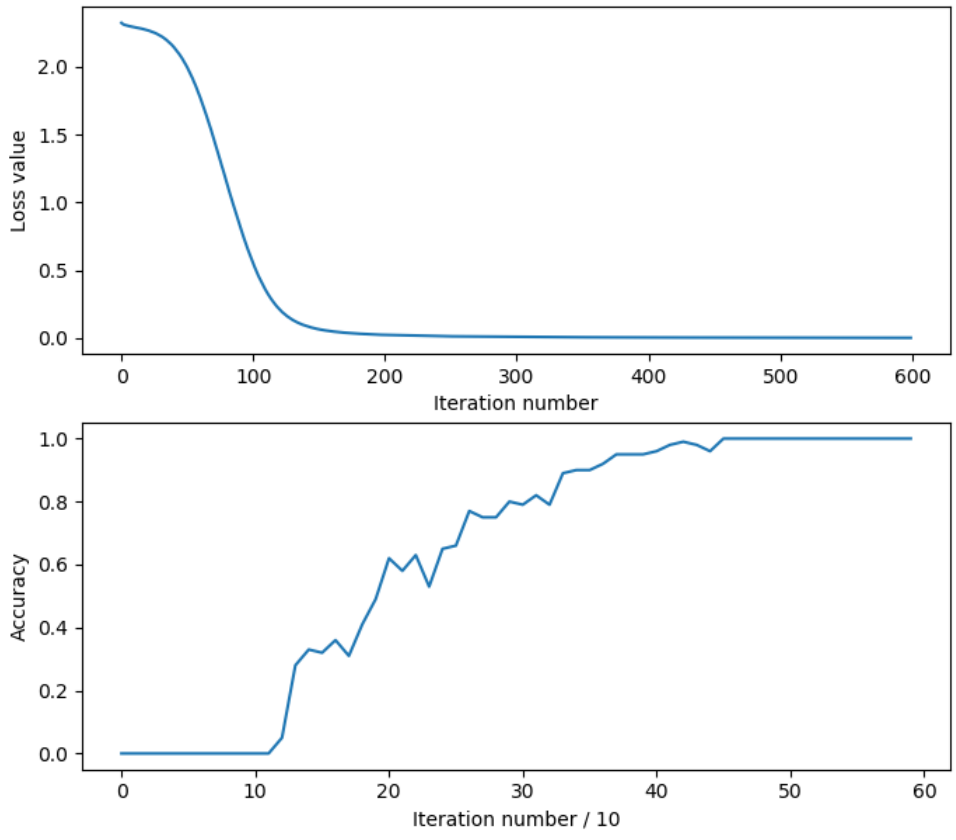
```
cuda0 = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = myPTRNNModel().to(cuda0)
optimizer = torch.optim.Adam(params=model.parameters(), lr=0.001)
train(1000, model, optimizer, cuda0)
loss_draw(model)
```

2.2 数字长度增大

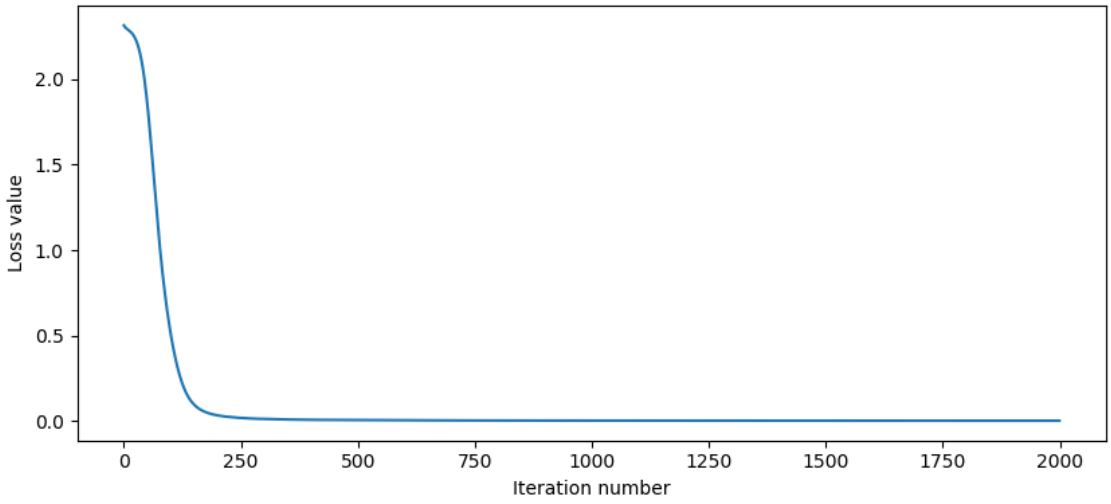
更改了源码中**产生随机整数、增加了整数范围可选参数**：（下面仅为部分改动）

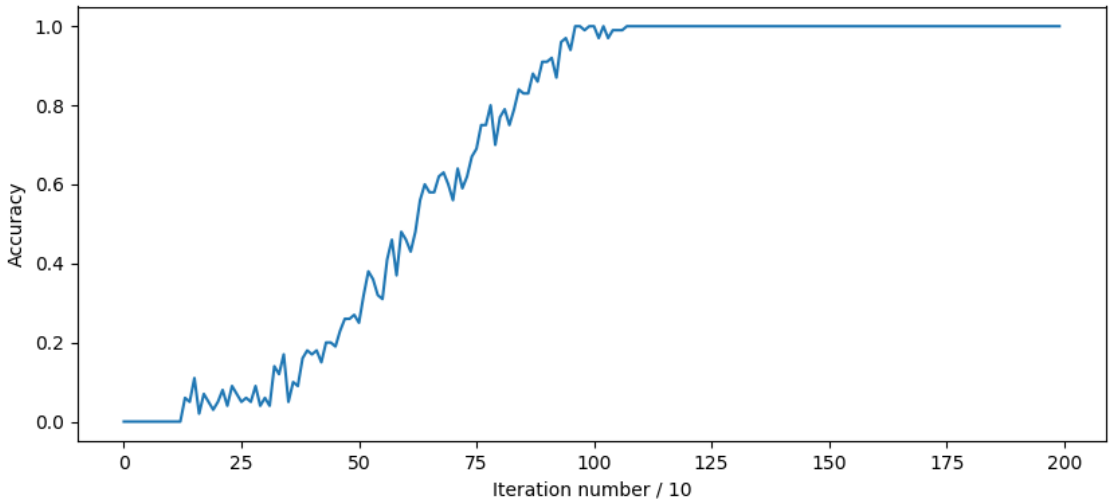
```
# data.py 下面仅为部分改动
def gen_data_batch(batch_size, start, end):
    numbers_1 = [random.randint(start, end) for i in range(batch_size)]
    numbers_2 = [random.randint(start, end) for i in range(batch_size)]
    results = [a+b for a,b in zip(numbers_1,numbers_2)]
    return numbers_1, numbers_2, results
# pt.py 增加可选参数digitlen 下面是部分改动
def train(steps, model, optimizer, device='cpu', digitlen=10):
    loss = 0.0
    accuracy = 0.0
    end_num = 5*10**(digitlen-1)
    for step in range(steps):
        datas = gen_data_batch(batch_size=100, start=0, end=end_num)
        Nums1, Nums2, results = prepare_batch(*datas, maxlen=digitlen+1)
        ......
```

使用`pt_main()`继续测试`rnn`结构，将位数增大到**1000位**，绘制`loss`和`accuracy`的图像，最终准确率达到**100%**。



再增大到**5000位**，绘制图像，依旧可以达到**100%准确率**。



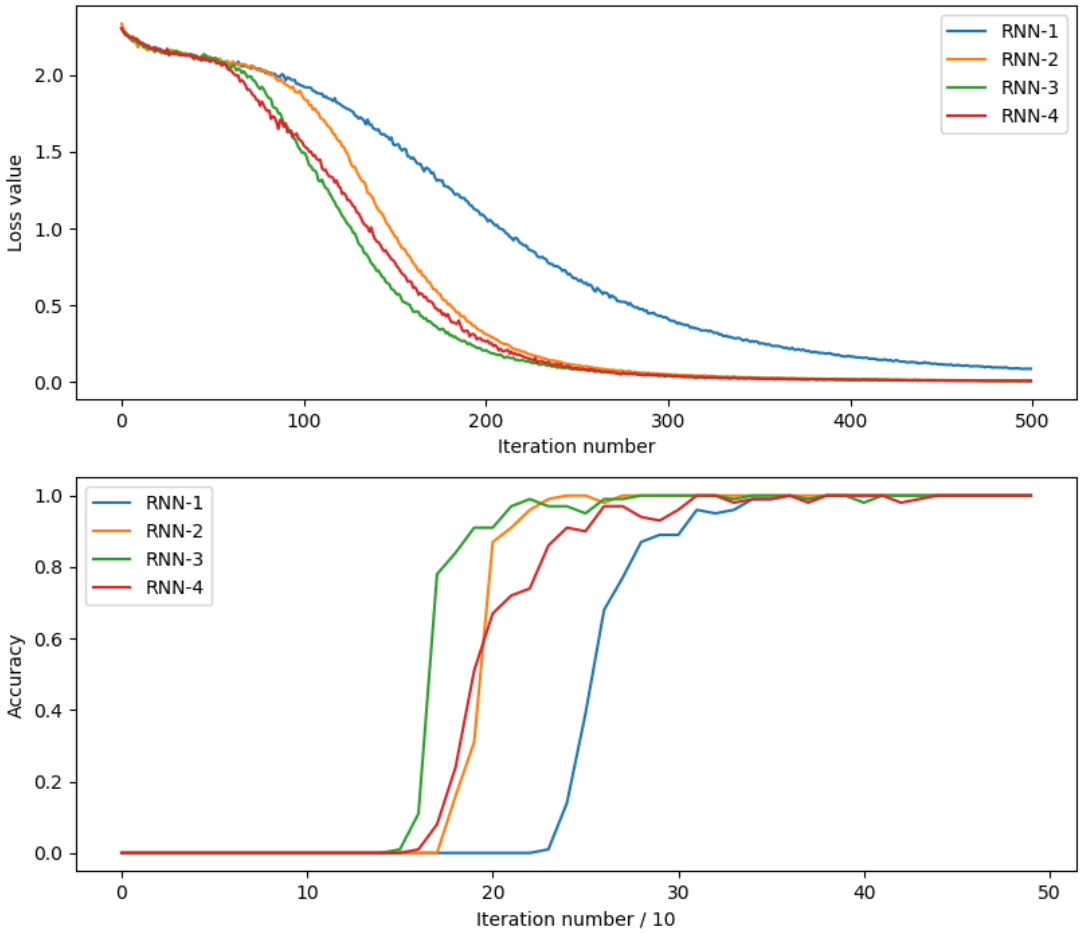


受GPU限制，做到此为止。

2.3 改进RNN模型

2.3.1 RNN多层

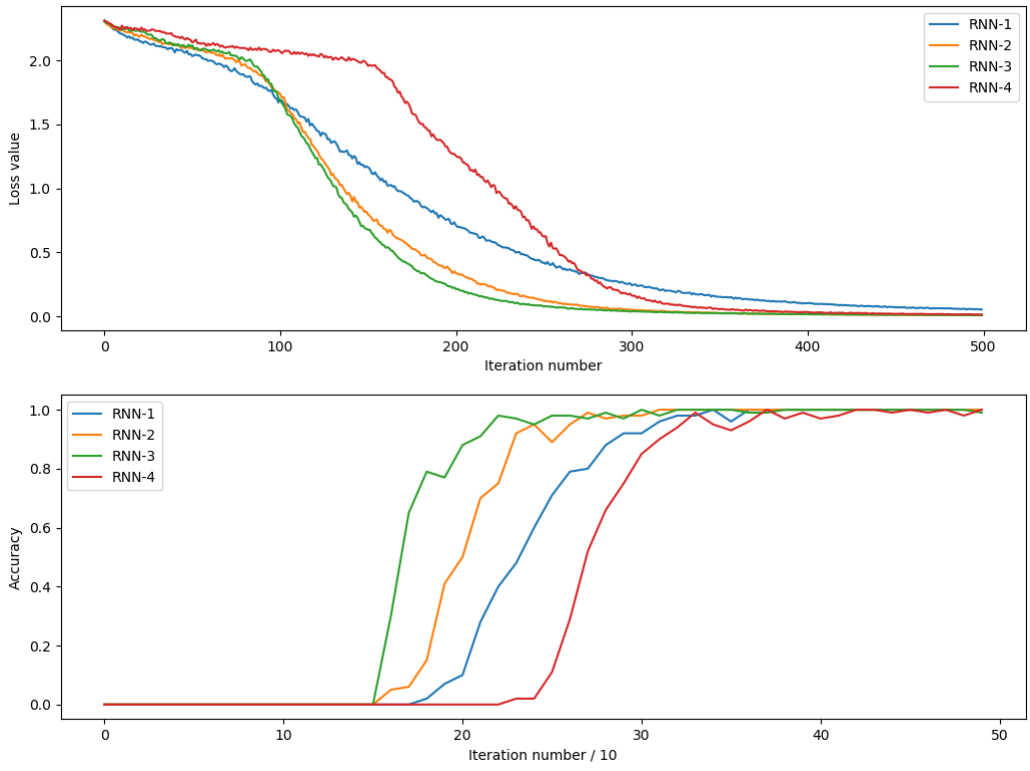
原来的RNN是2层，`self.rnn = nn.RNN(64, 64, layers)`，修改为可变层数，观察1-4的影响，如图，可知：准确率均可到达100%，**RNN-3最佳，更快达到收敛**



2.3.2 更改为GRU/LSTM

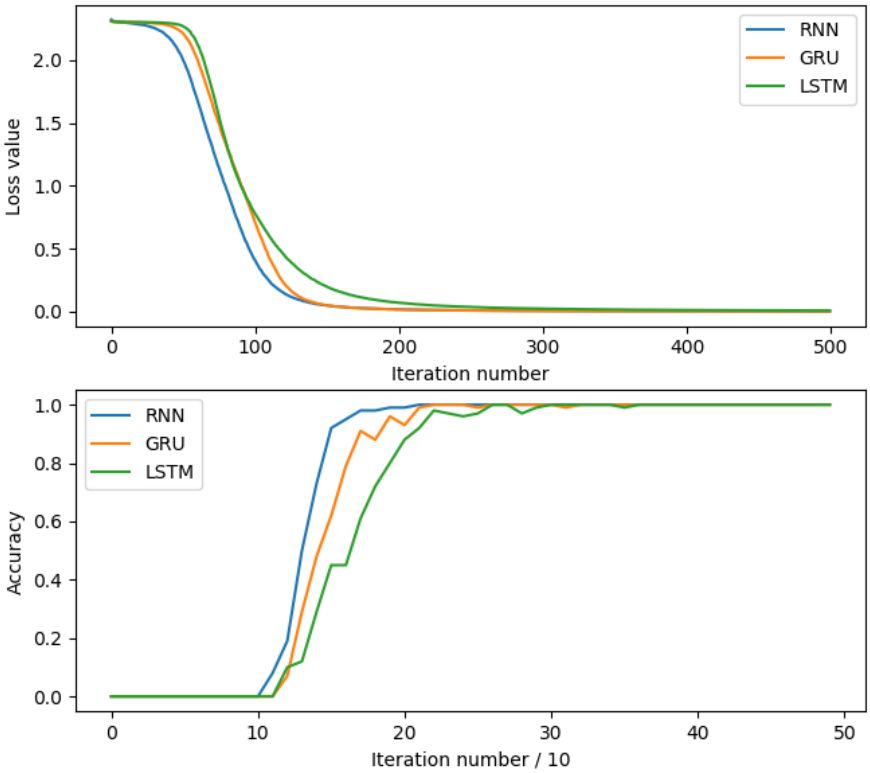
更换`RNN`模型，查看其效果，采用`GRU`来优化`RNN`，同理先判断所需层数，结论相同：

使用3层时效果最佳。下面是GRU层数对模型的影响



接下来在`digitlen=100`进行比较，均在500`steps`内收敛，准确率均为1：

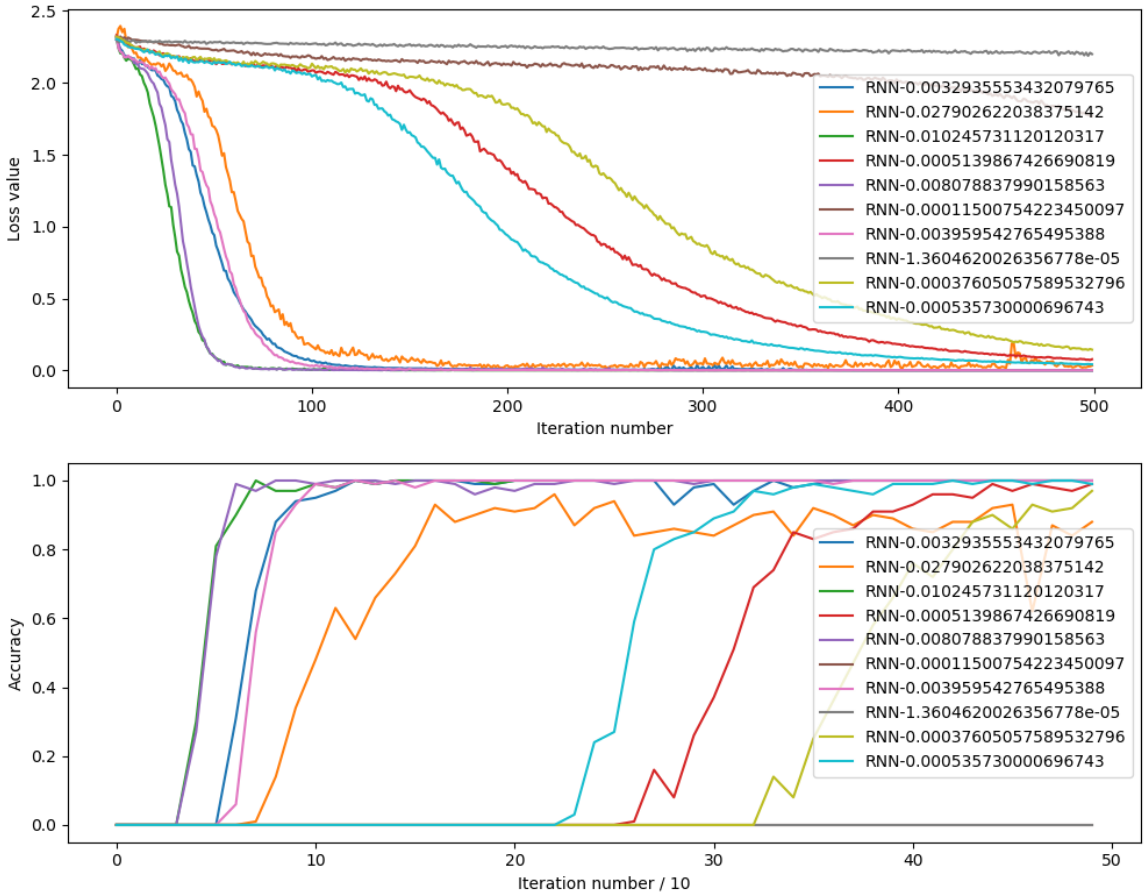
比较发现**还是RNN-3脱颖而出**



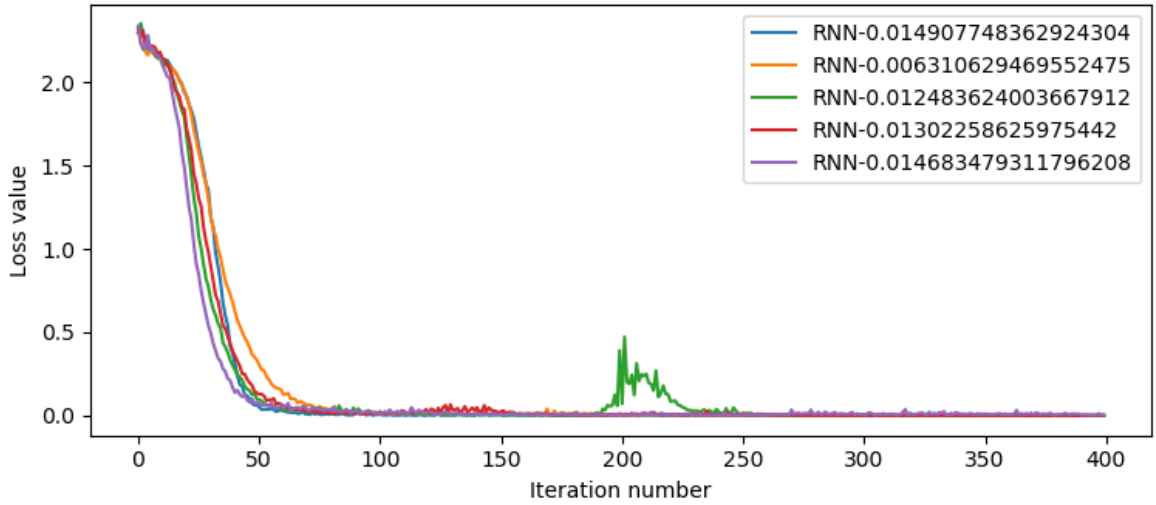
2.3.3 学习率的影响

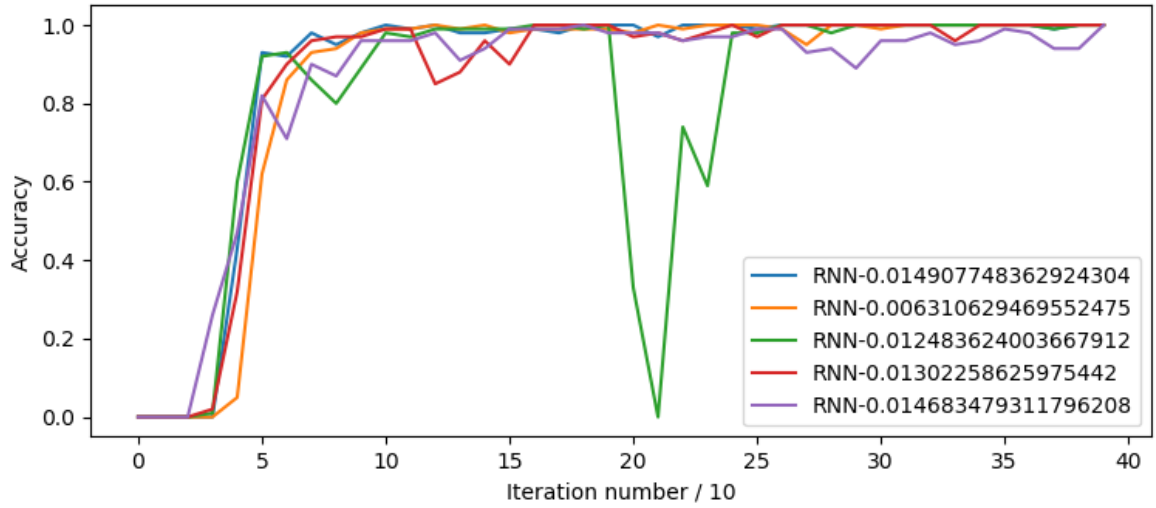
采用RNN-3的模型，在`10^-5~10^-1`之间寻找合适的学习率，如图所示，可知：

- **学习率对该模型的影响很大**
- **最适学习率在-0.01附近**，在100steps前竟然就能收敛
- 学习率过低和过高都有问题



再深挖区间10^-2.2~10^-1.8，效果类似。**所以取lr=0.01即可**

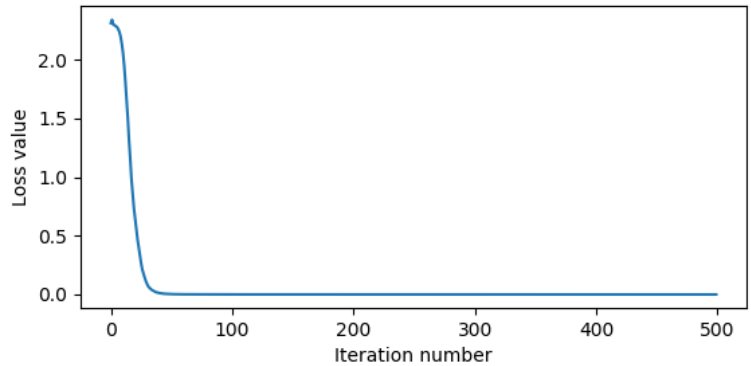


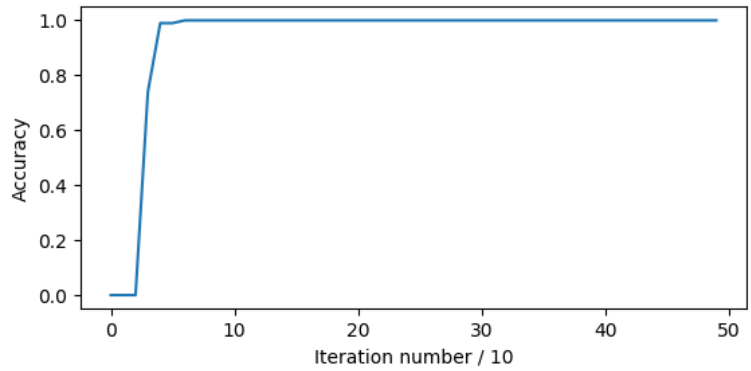


2.4 实验结果

最后选择优化后的模型为 RNN 模型，并且层数为3层，学习率设置为0.01。

能在100steps内快速收敛达到100%准确率，下面测试结果为100位数字情况。





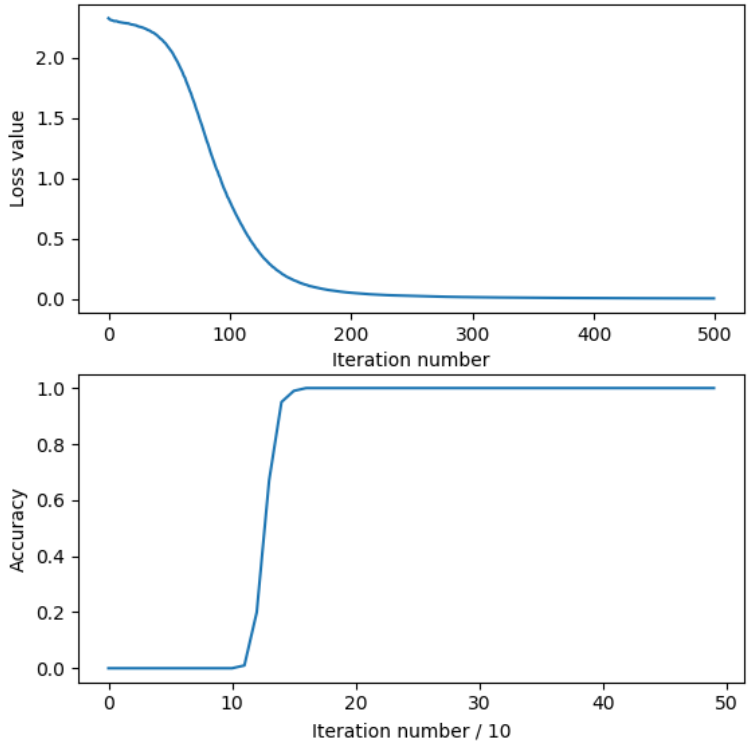
3 总结

优化后的RNN模型如上，能扩展位数至5k，能在100steps内准确率收敛到100%。

运行方法及运行结果：

```
python source.py
```

- `pt_main()`运行结果



- `pt_adv_main()`运行结果