ASSIGNMENT 2

本次作业使用 pytorch 完成

PART I

1. 补全代码

第一部分需要补全代码完成 RNN 网络模型，实现一个 10 位长度数字的加法器，代码如下：

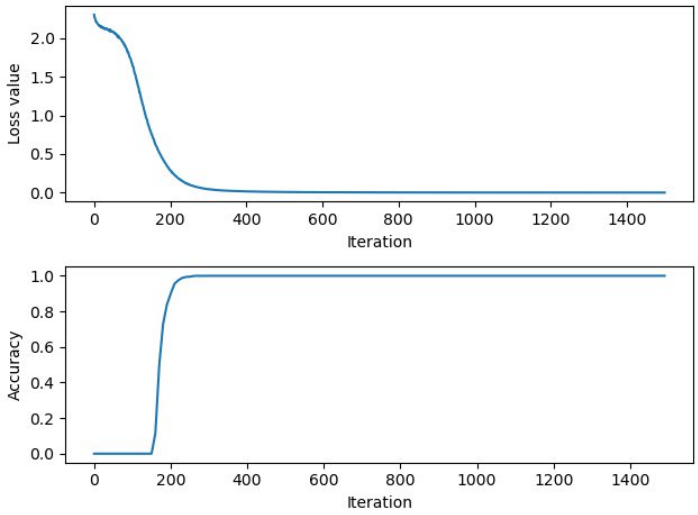
```
class myPTRNNModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.loss = []
        self.accuracy = []
        self.embed_layer = nn.Embedding(10, 32)
        self.rnn = nn.RNN(64, 64, 2)
        self.dense = nn.Linear(64, 10)

    def forward(self, num1, num2):
        '''
        Please finish your code here.
        '''
        x = self.embed_layer(num1)
        y = self.embed_layer(num2)
        sum = torch.cat((x, y), -1).transpose(0, 1)
        sum, h_state = self.rnn(sum)
        logits = self.dense(sum).transpose(0, 1).contiguous()
        return logits
```

原本的代码为网络设计了 4 层结构，第一层 embedding 层将输入的 10 进制数字映射到 32 维，接下来是两层 RNN 层，最后一个线性层再把 RNN 层的输出降维得到网络输出。
我们要做的是填补 forward 函数，依次连接各层即可。此处同时也修改了__init__()函数，添加了两个列表用于记录历史值，方便后面画图。

2. 模型运行结果

输出结果如下：
step 50 : loss 2.0828490257263184 accuracy 0.0
step 100 : loss 1.6677061319351196 accuracy 0.0
step 150 : loss 0.7741047143936157 accuracy 0.0
step 200 : loss 0.28977906703948975 accuracy 0.8995

step 250 : loss 0.10178861021995544 accuracy 0.995
step 300 : loss 0.04467622935771942 accuracy 1.0
step 350 : loss 0.02412053570151329 accuracy 1.0
step 400 : loss 0.01611490733921528 accuracy 1.0
step 450 : loss 0.010942327789962292 accuracy 1.0
step 500 : loss 0.008071268908679485 accuracy 1.0
（此处略微修改了 train 函数的输出模式）
作图示意如下：



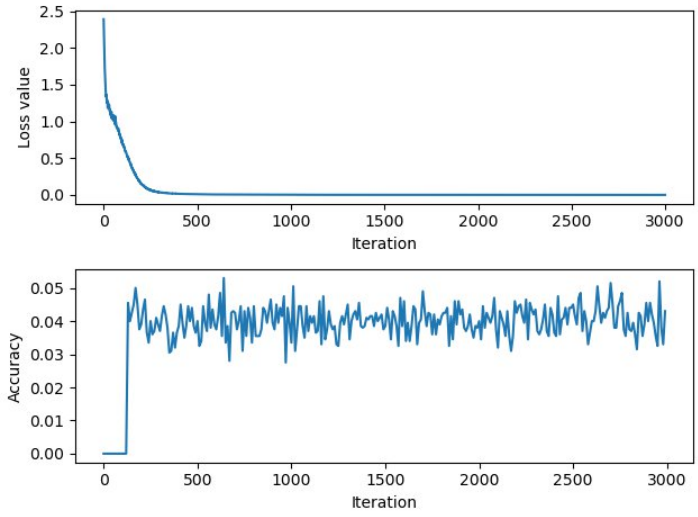
可以看出在 **250** 代到 **300** 代之间 accuracy 收敛到 **1**，大致在 **500** 代之后 loss 收敛到 **0**。

PART II

这一部分需要我们所使用数据的位数长度，评估普通模型的性能，再在普通模型的基础上做改进。

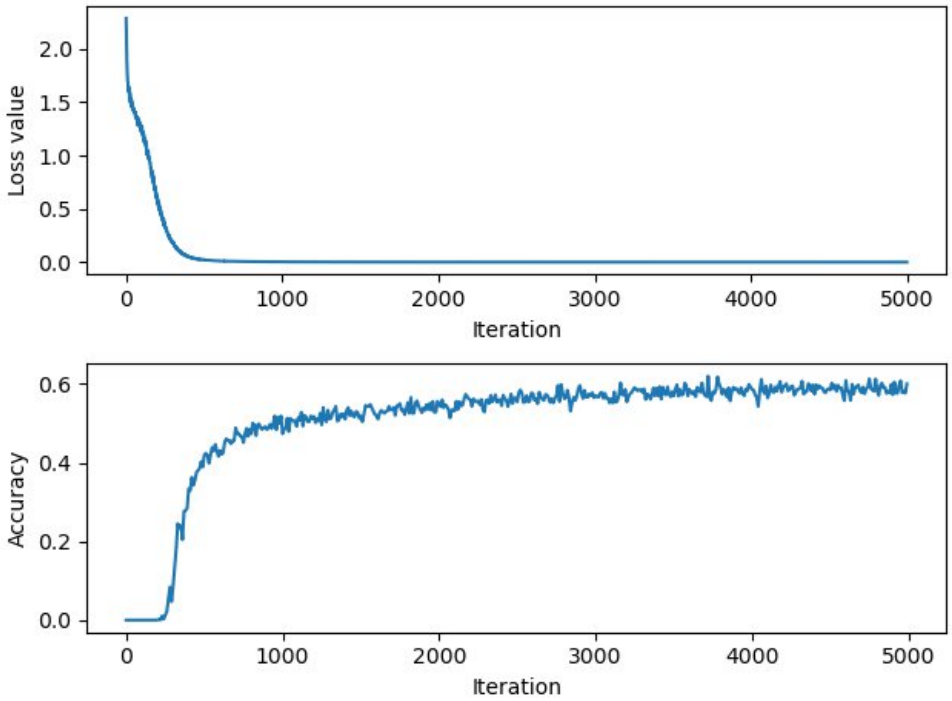
1. 评估普通模型的性能

我们将数据长度分别修改为 **1** 位，**2** 位，**50** 位，**100** 位，**300** 位，观察普通模型的表现。
1 位：



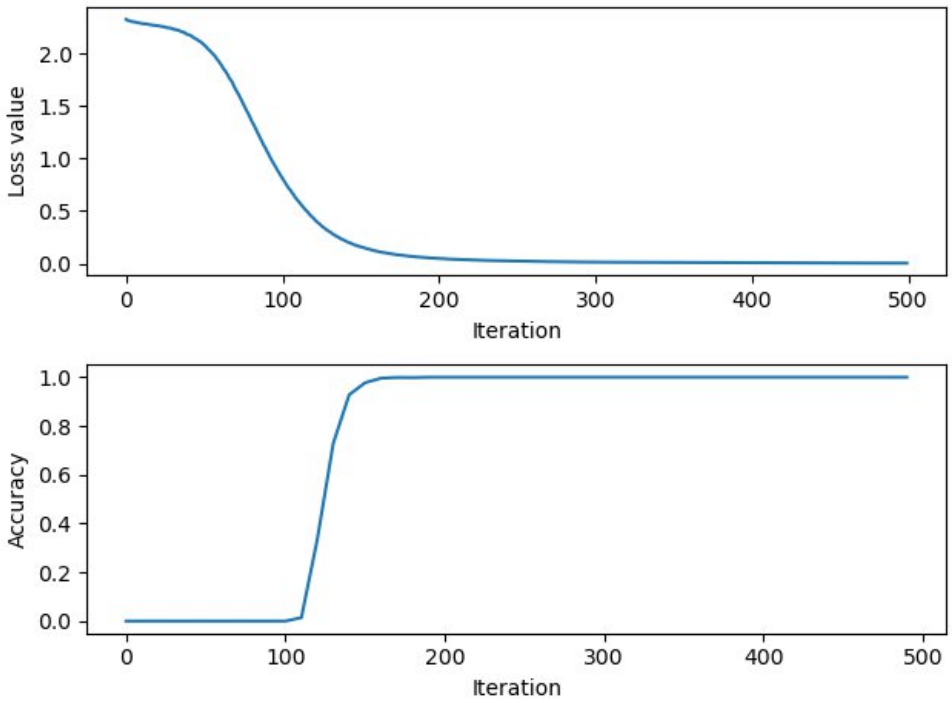
此时可以说模型不具备加法器的能力。考虑到 1 位的加法数据中涉及到进位时高一位上数字都是 0，加法器的进位特征提供的并不够多，模型缺乏能力可以理解。

2 位数据:



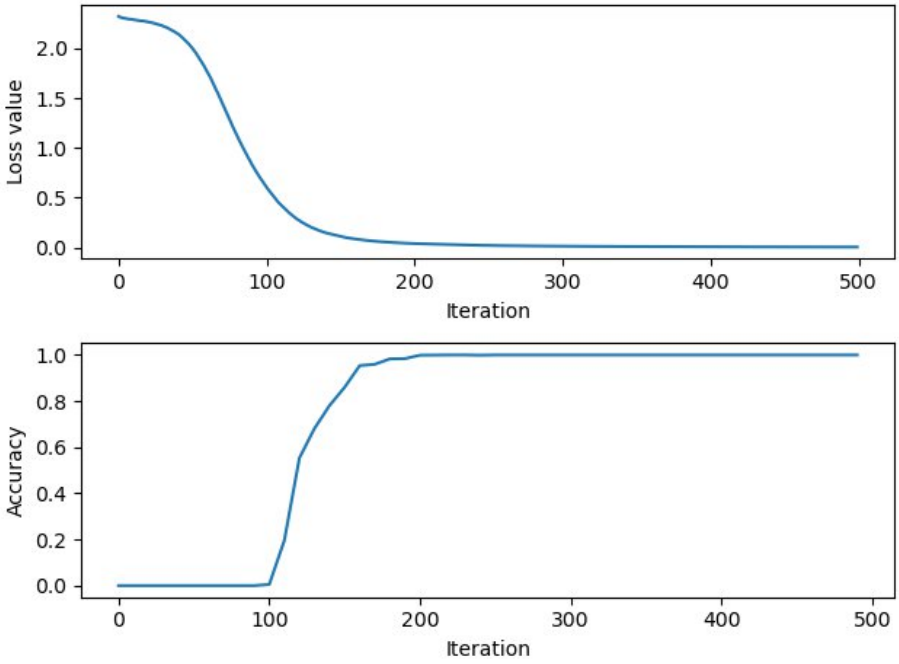
可以看到此时模型具备了部分加法器的能力，准确率最终收敛到 **0.6** 附近。

50 位数据:



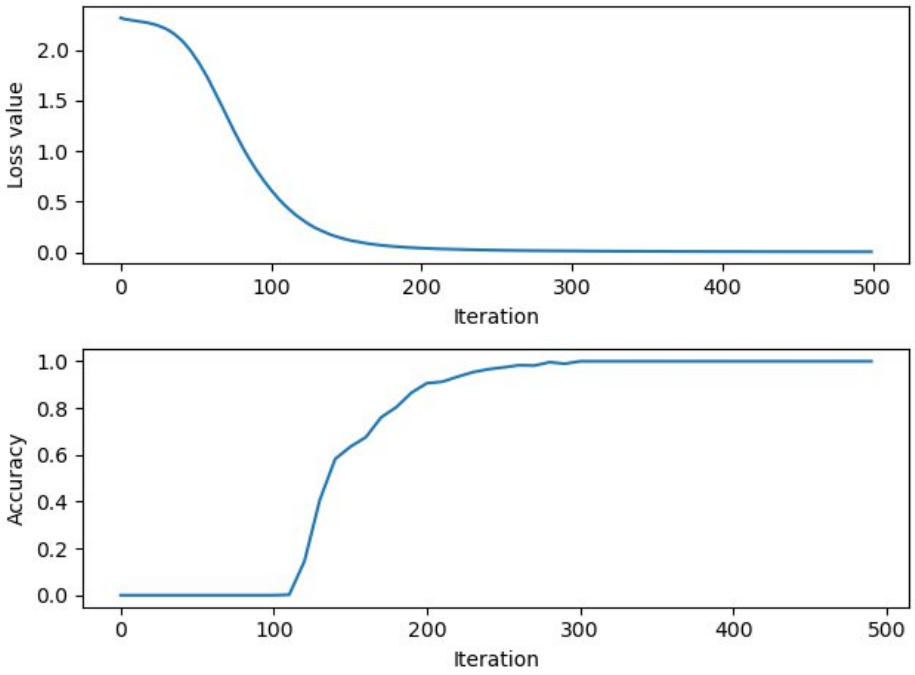
可以看到此时模型的表现要好于数字长度为 **10** 位时，在 **150** 代到 **200** 代之间 accuracy 已经收敛到 **1**，大致在 **350** 代以后 loss 就已经收敛到 **0**

100 位数据



可以看到此时模型的表现略差于 **50** 位，略好于 **10** 位。

300 位数据



表现差于 **100** 位，loss 收敛速度略快于 **10** 位时，accuracy 收敛速度与 **10** 位差不多。

综合以上结果，我们可以发现，普通模型在数据位数**很小**和**很大**时表现**较差**，在**适中**（几十位）时表现**较好**，但是即使在较好情况下也需要**上百代**才能收敛。
接下来的实验我们将选择 **2** 位，**50** 位，**100** 位作为实验数据的长度。

2. 优化：gpu 运行

在上面的实验中，当数据位数上百位时，可以明显感到计算时间变长，为了加速，我们将大部分运算放到 gpu 中进行。
初始化模型处修改代码（以普通模型为例）：

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

evaluate()函数处修改代码：

```
with torch.no_grad():
        logits = model(torch.tensor(Nums1).to(device),
                       torch.tensor(Nums2).to(device))
        logits = logits.cpu()
```

保证不存在 cpu 中数据与 gpu 中数据的直接计算。

train_one_step()函数处修改代码：

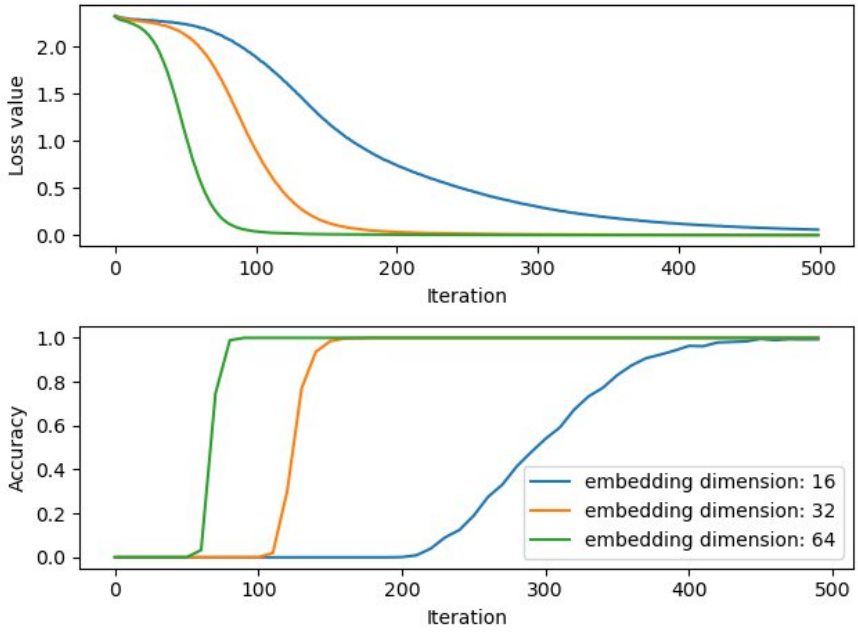
```
logits = model(torch.tensor(x).to(device), torch.tensor(y).to(device))
loss = compute_loss(logits, torch.tensor(label).to(device))
```

3. 优化：收敛迭代次数实验

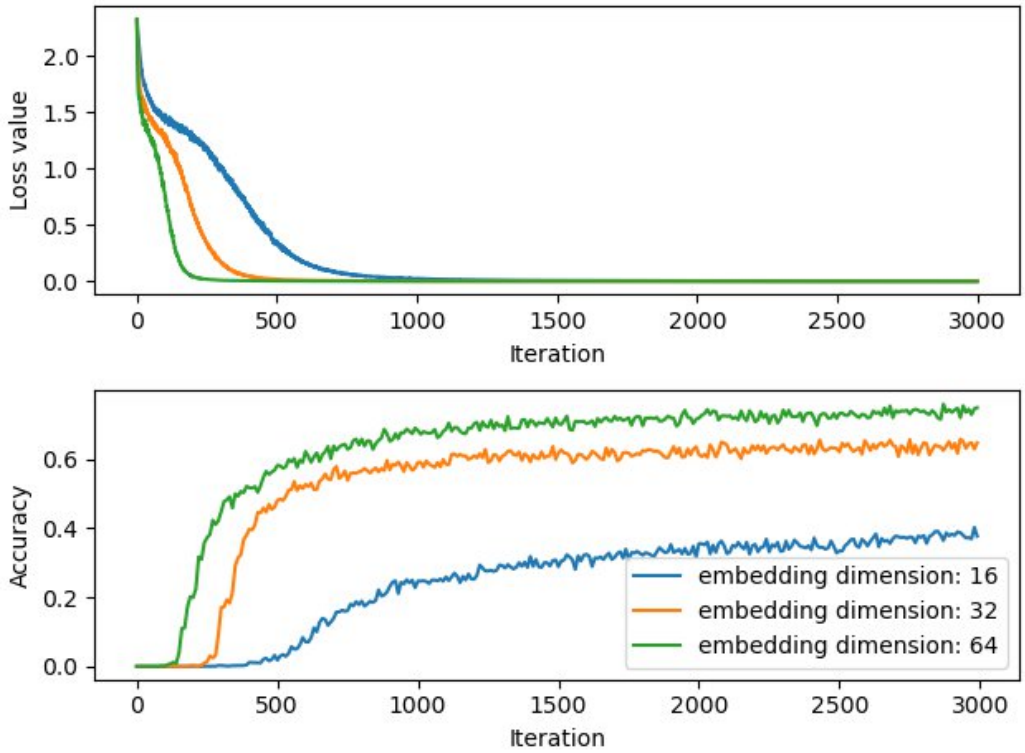
在这部分的实验中，我们保持训练集与测试集的数据位数相同。

3.1 Embedding 层维度比较试验

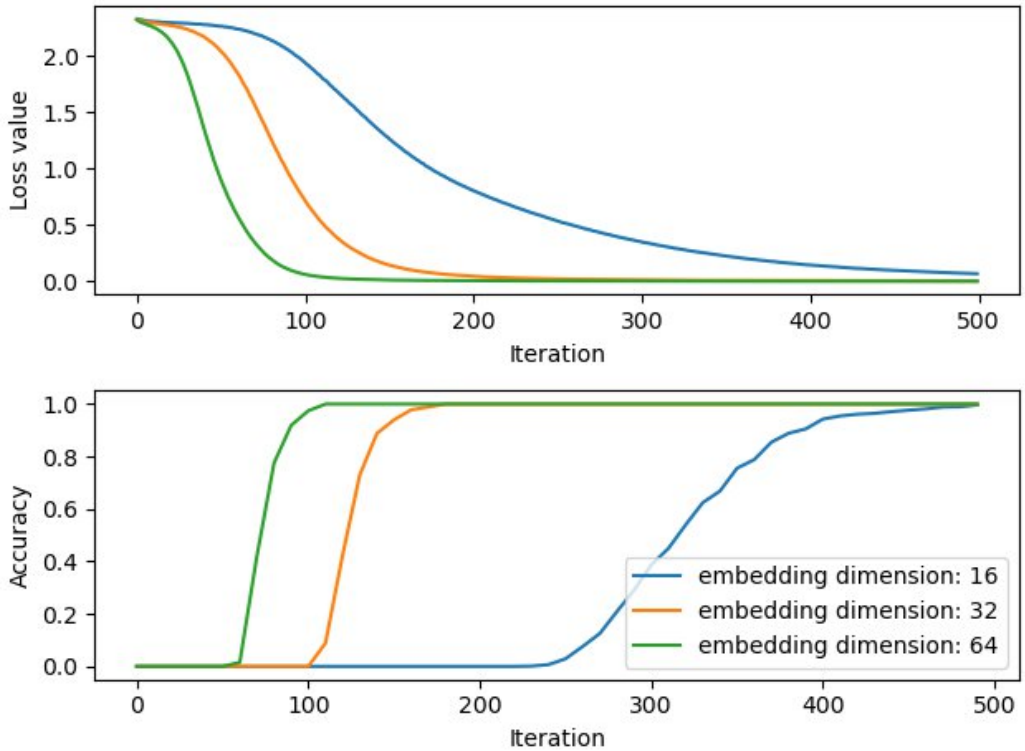
调整 **Embedding 层维度**，试验结果如下：
50 位数据



2 位数据



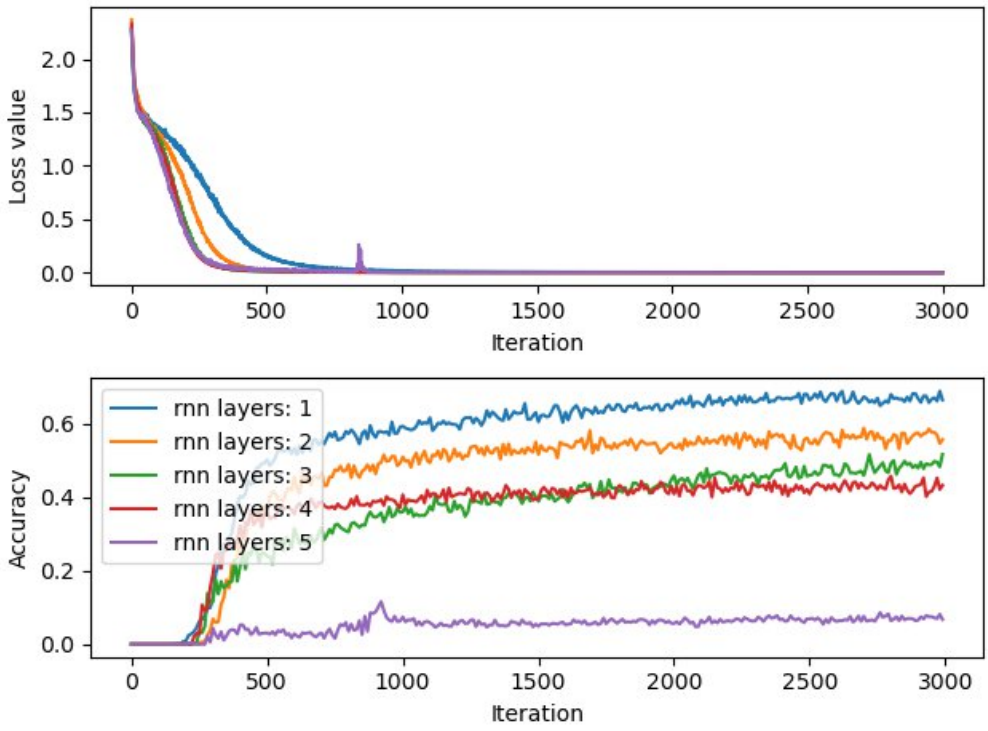
100 位数据



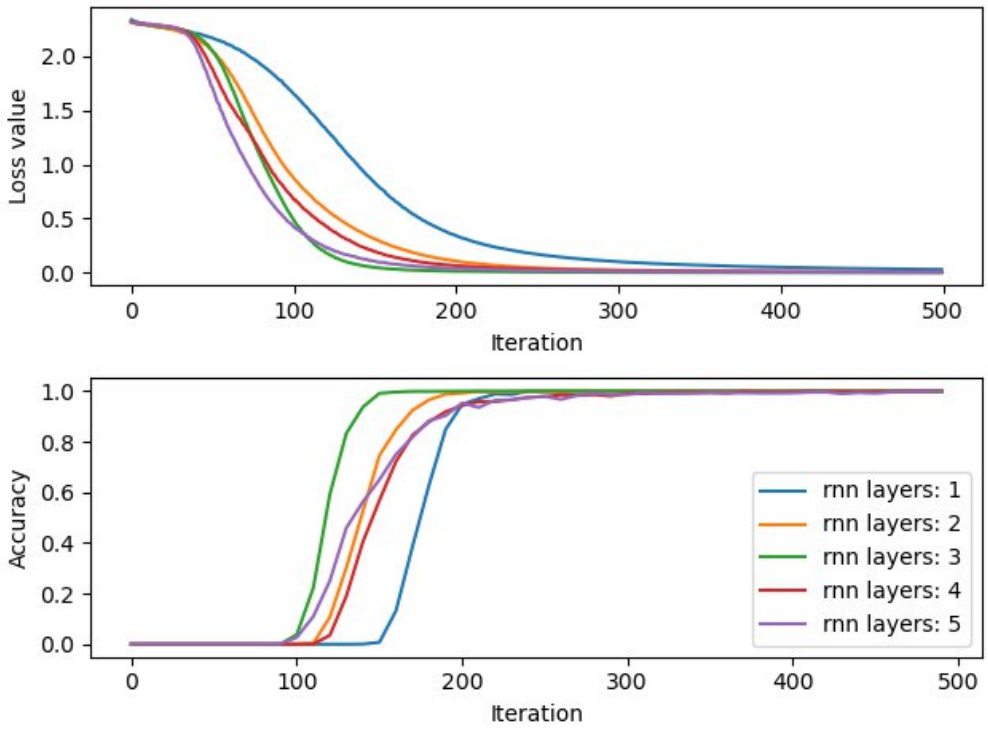
可见将数字映射到更高的维度，使特征更加分散，确实可以增强模型的能力，减小收敛需要的迭代次数。

3.2 RNN 层数

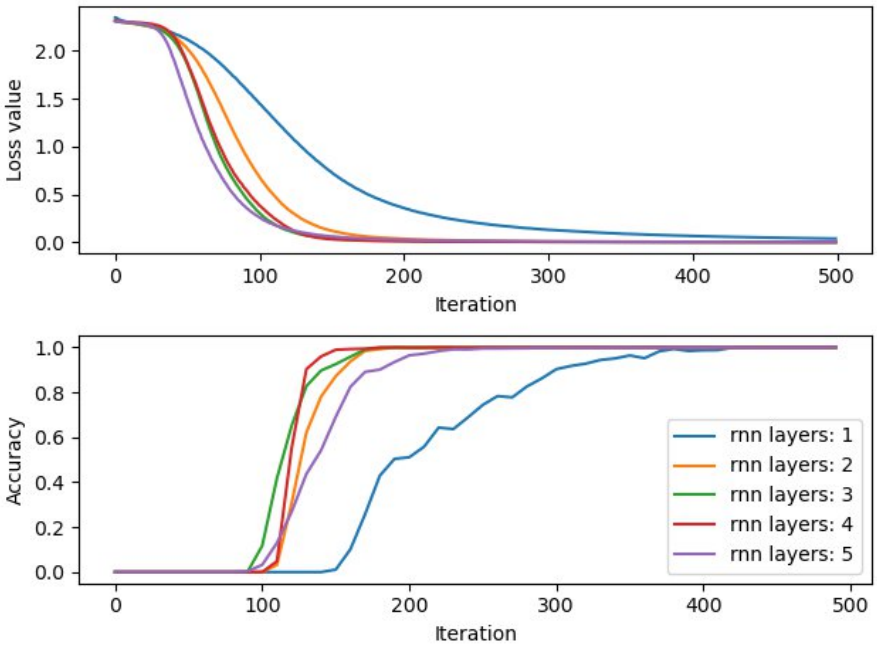
改变 **RNN 层数**，结果如下：
2 位数据：



50 位数据



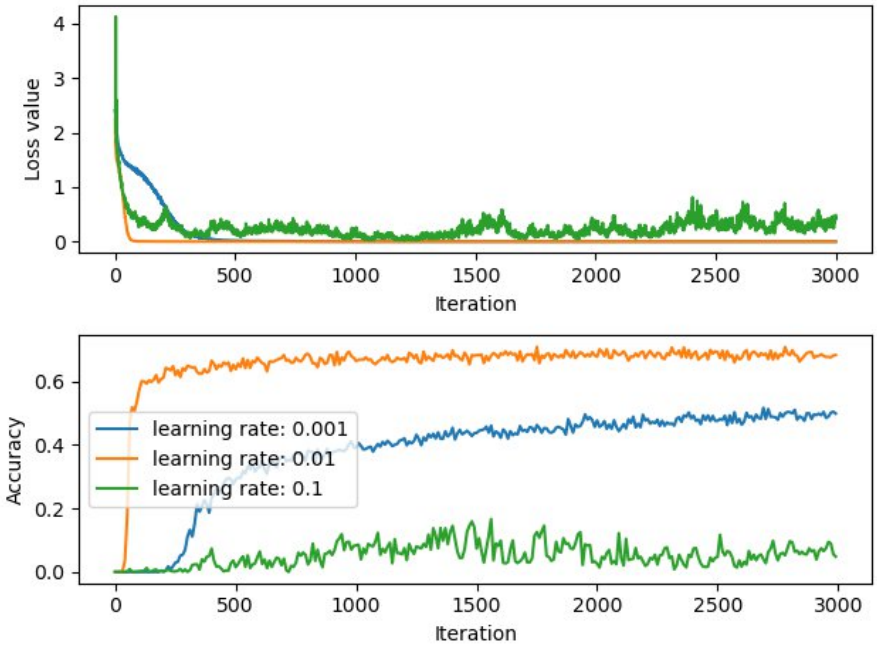
100 位数据



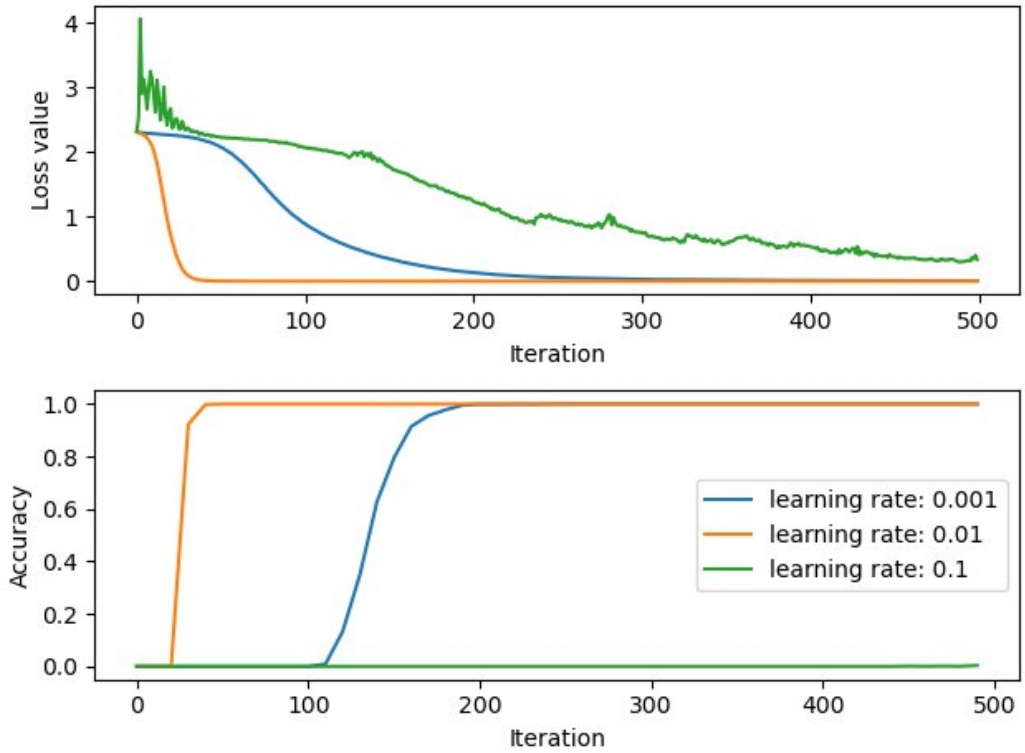
可以发现, 单层 RNN 虽然在数据位数较小时能力最强, 但是大多数情况下收敛速度都最慢。而 5 层 RNN 虽然收敛速率较快，但是在数据位数较小时能力很差。2 到 3 层的 RNN 结构综合而言具有最好的性能。

3.3 学习率

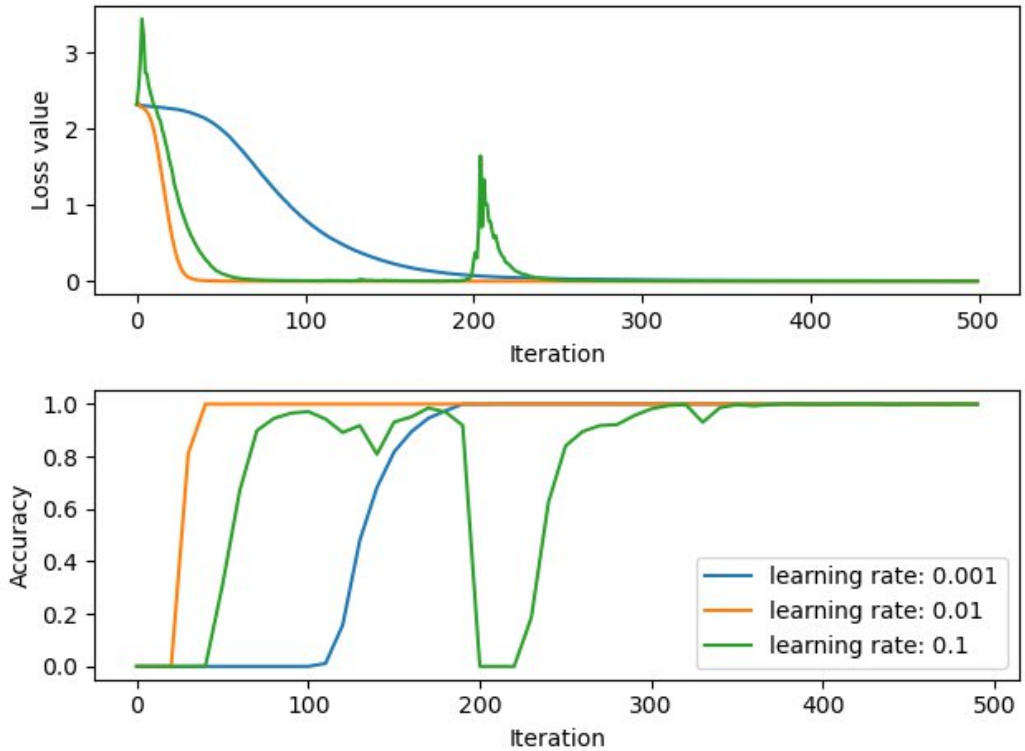
改变**学习率**，结果如下:
2 位数据:



50 位数据:



100 位数据:

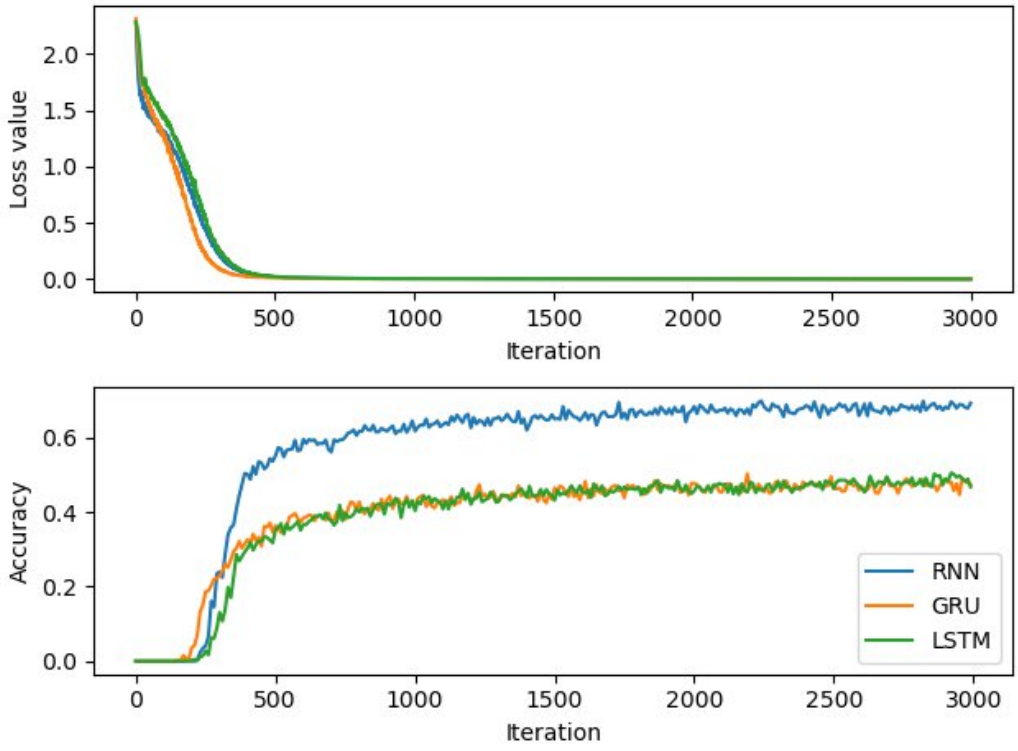


三种学习率中，0.01 毫无疑问具有最好的性能，也即学习率不能过大也不能过小，过大会导致训练过程不稳定，模型的能力会出现较大波动，过小则会导致收敛速度偏慢。

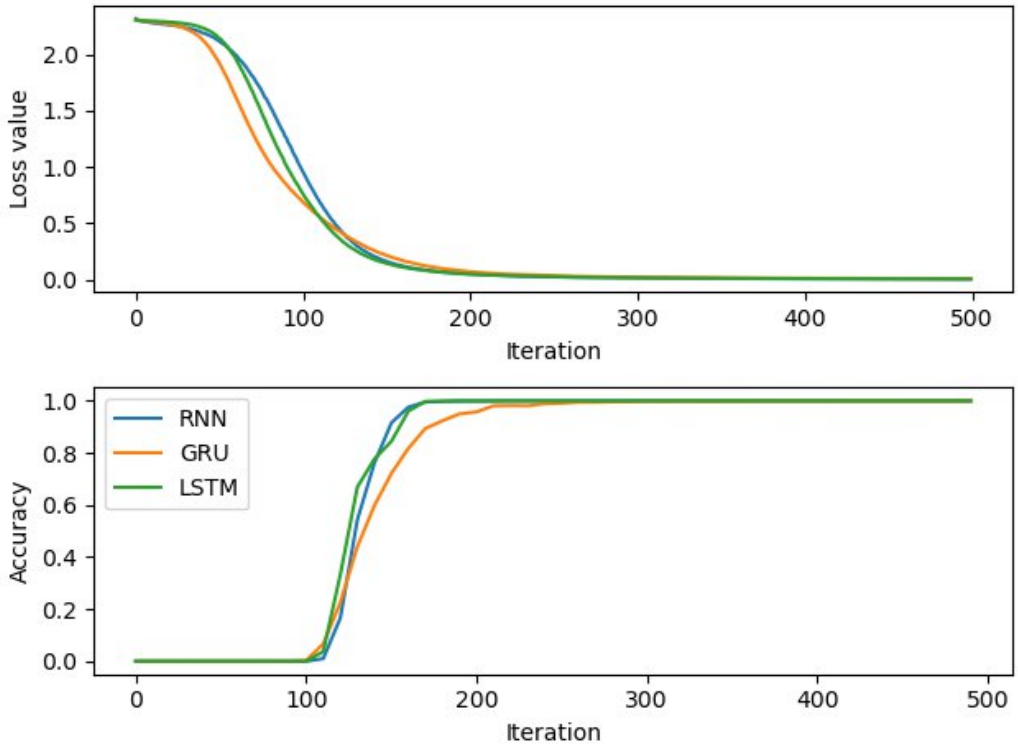
3.4 网络类型

分别使用普通 **RNN**，**GRU** 和 **LSTM**，结果如下：

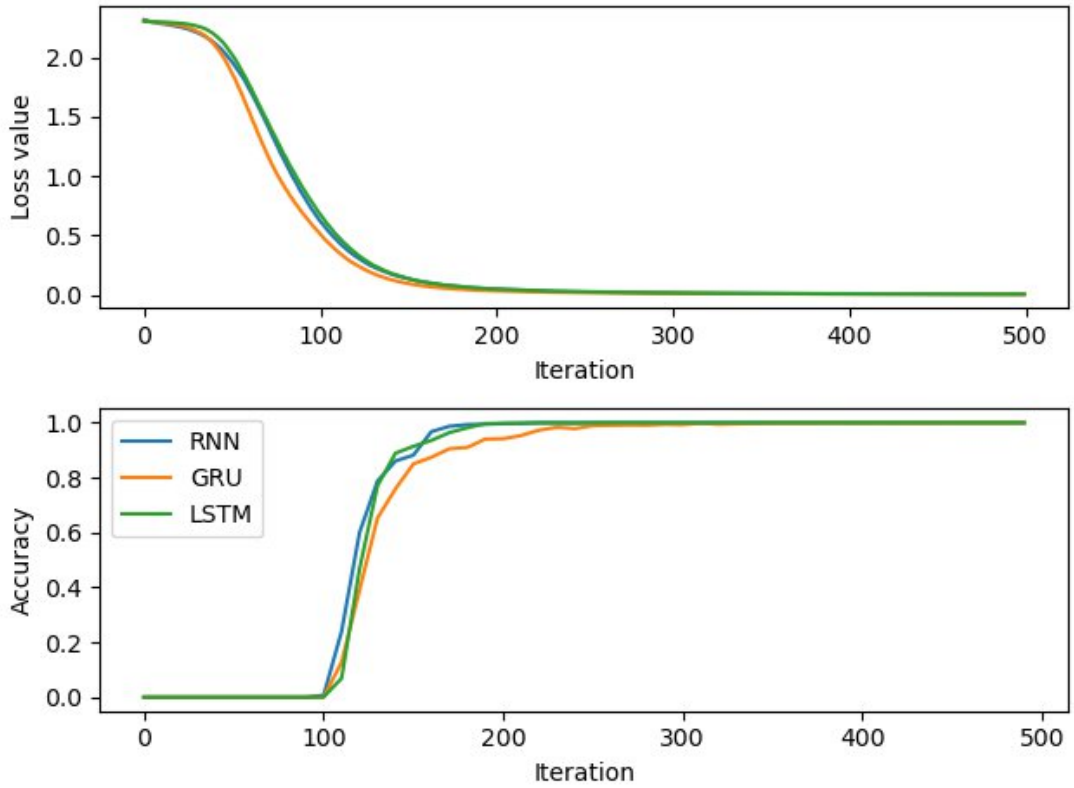
2 位数据:



50 位数据:



100 位数据



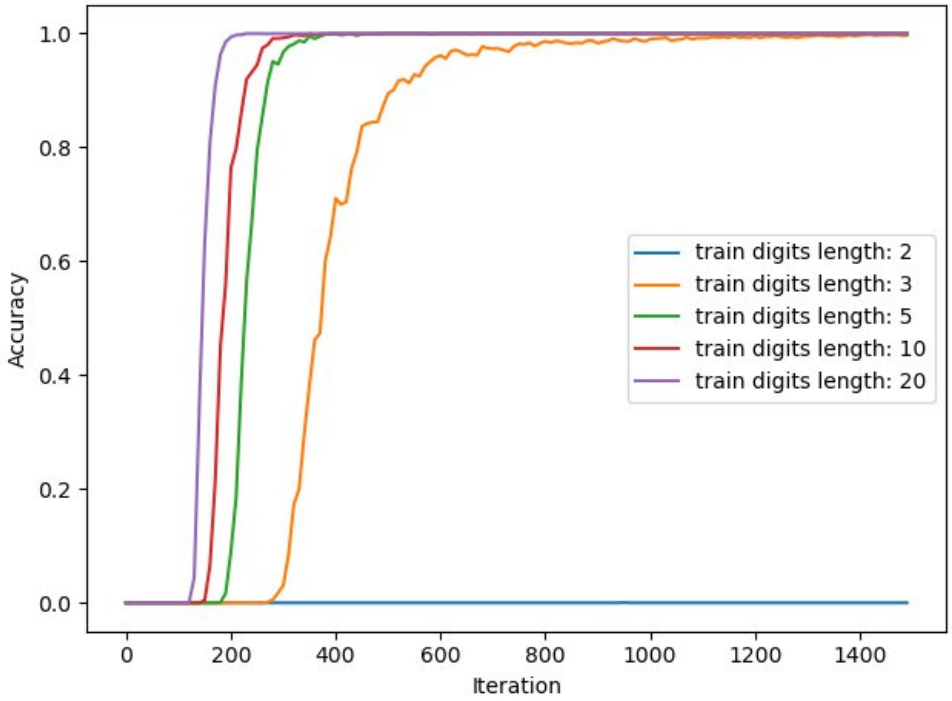
可以看出多数情况下，对于加法器这个问题，三种 RNN 结构的能力是差不多的，但是在数据较短的情况下，普通 RNN 网络反而具有更好的性能。推测这是由于对于加法器而言，每一位的计算理论上仅仅与上一位相关，依赖关系仅仅存在于短期，不需要构建长程依赖也能解决问题。所以 LSTM 网络与 GRU 网络并没有展现出优越性。

4 优化：泛化能力实验

对于具有加法能力的加法器而言，应当能够计算任意位数的加法。保持训练集与测试集位数相同的做法无疑应当是一种多余的限制。这一部分我们将探索训练集与测试集位数不同时模型的能力。

4.1 能力测验

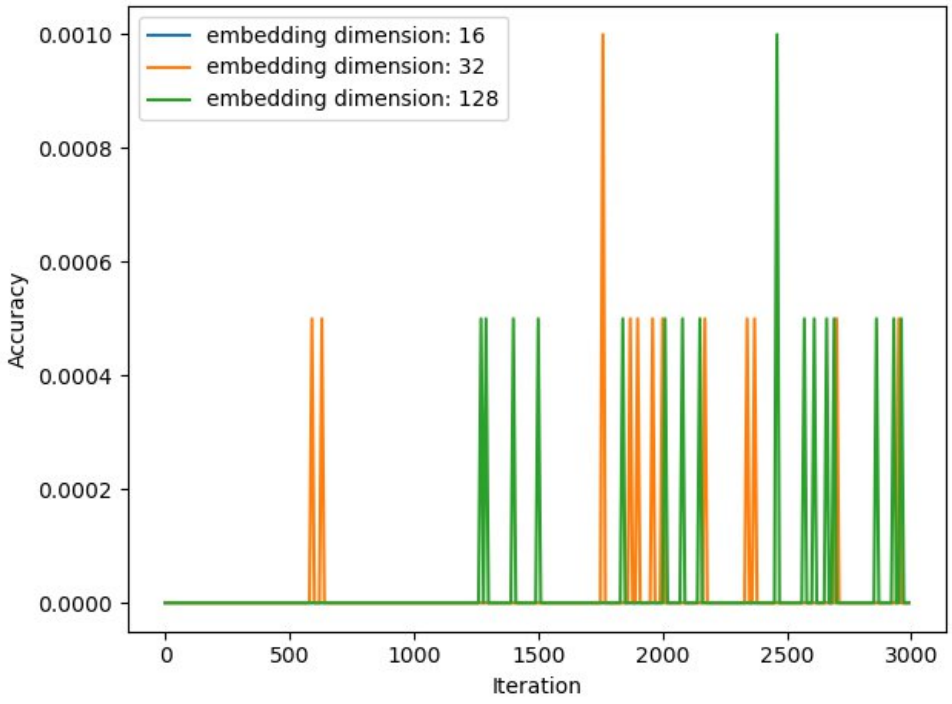
固定测试集位数为 50，改变训练集位数，以未优化的普通模型为基准模型，结果如下：



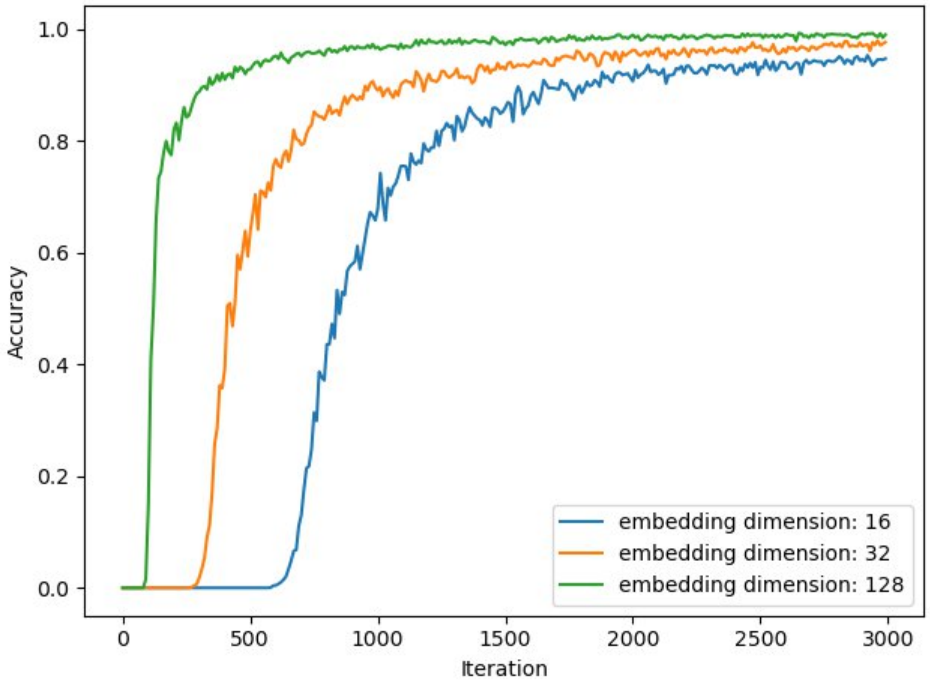
可见当训练集位数为 **2** 时，模型完全没有加法器能力，而当训练集位数为 **3** 时，模型具备加法器能力，但是收敛速度较慢。
接下来我们就用 **2** 和 **3** 作为训练集数据位数开展实验。

4.2 Embedding 层维度

2 位数据:



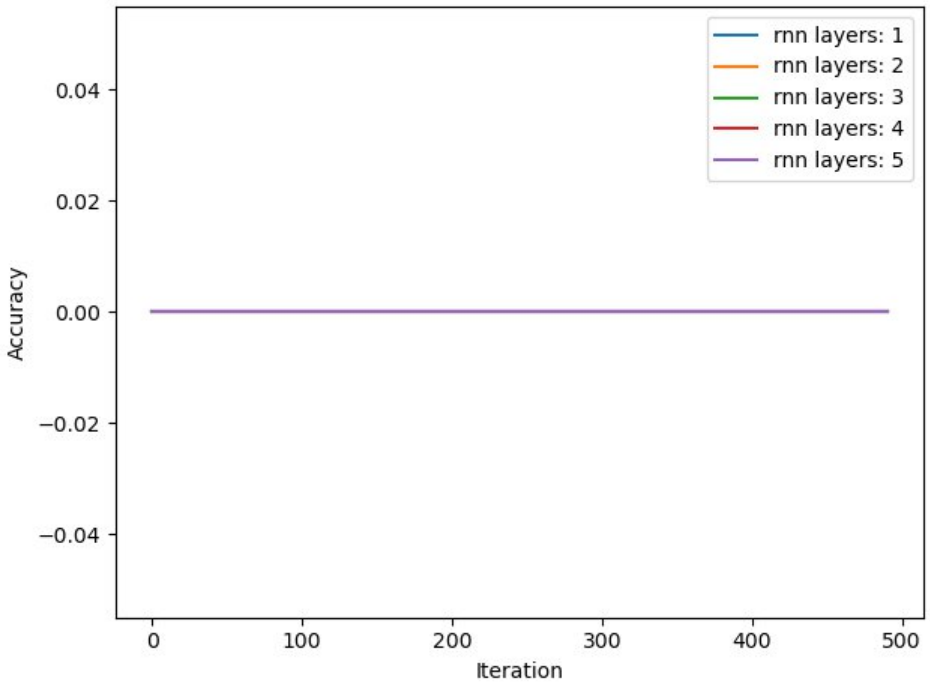
3 位数据:



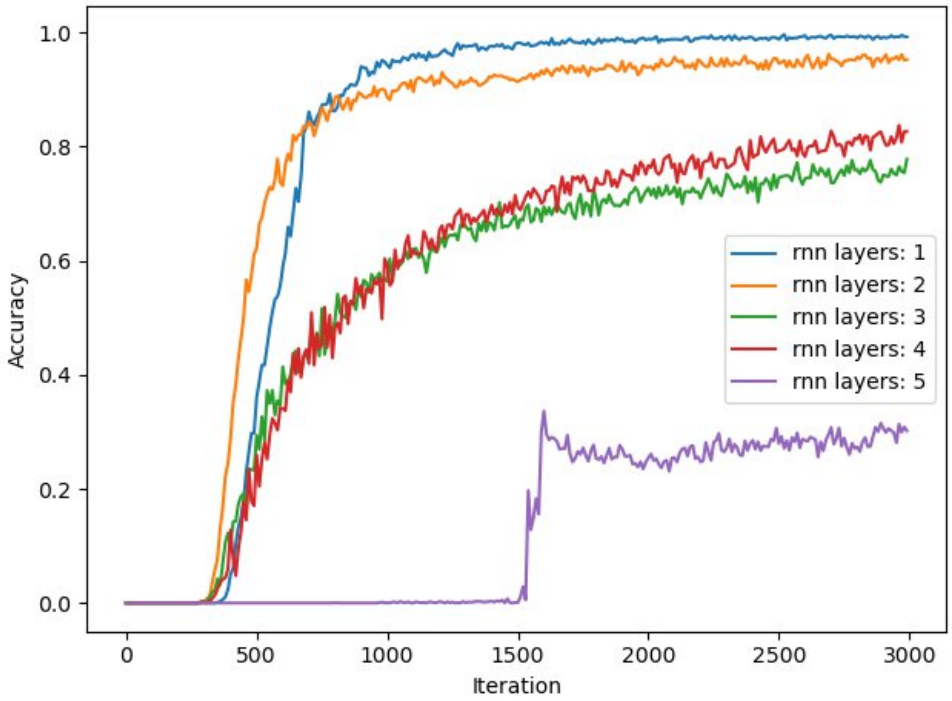
增大或者减小 Embedding 层维度，模型使用 2 位训练集始终无法具备加法能力，但是当使用 3 位训练集时，较高的维度具有更快的收敛速度。说明更高的维度有助于提高模型的泛化能力，但是不能带来根本改变。

4.3 RNN 层数

2 位数据



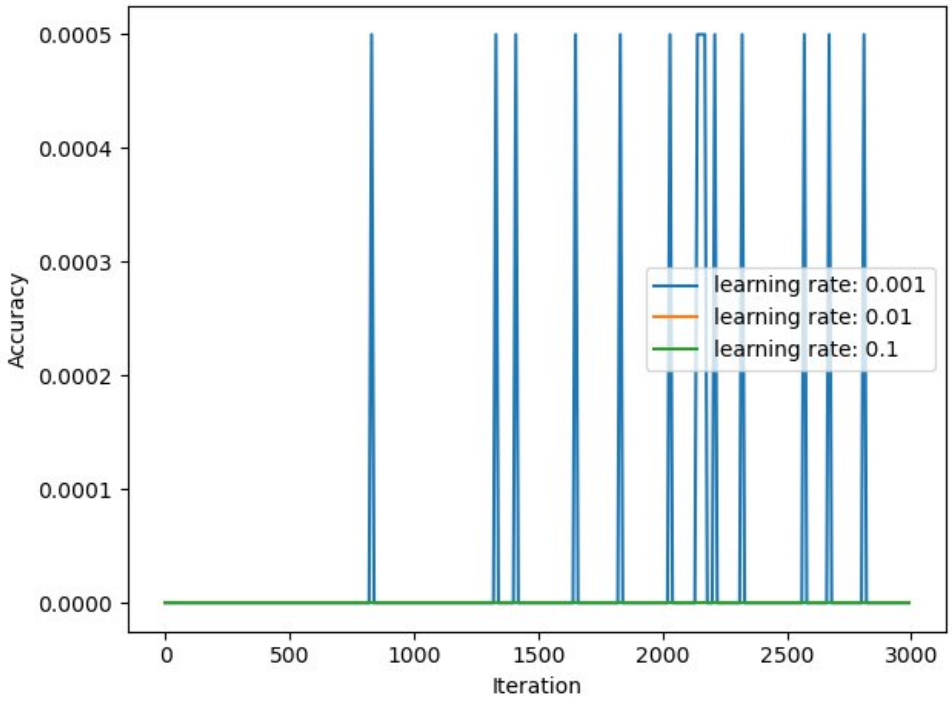
3 位数据



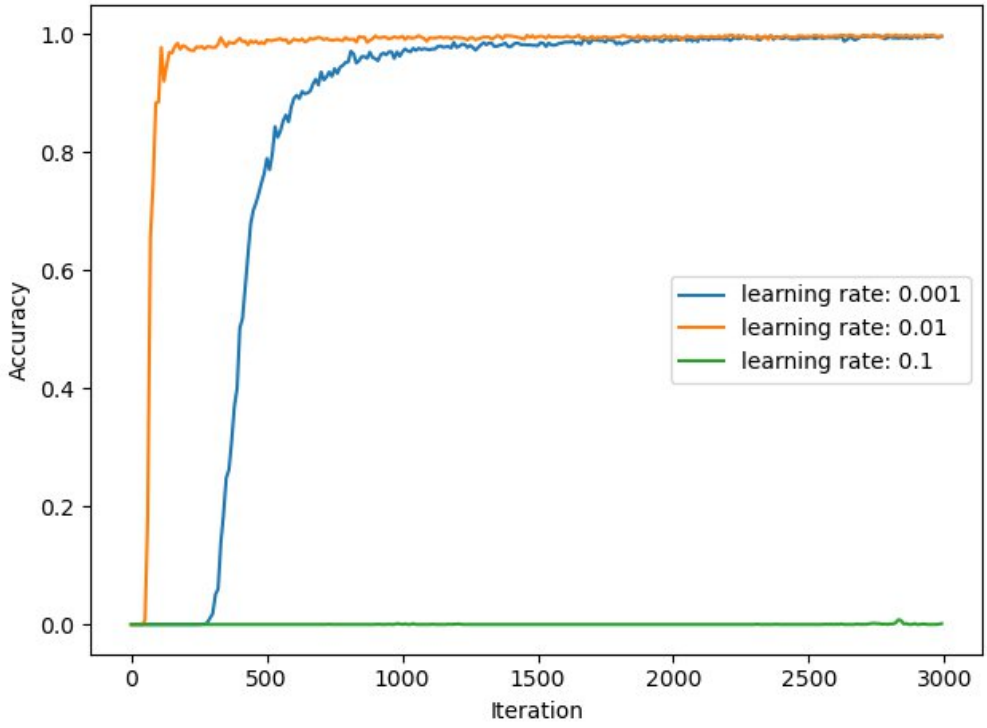
可见 1 或 2 层 RNN 具有更好的泛化能力，但依旧无法使模型在训练集只有 2 位时具备加法器能力。

4.4 学习率

2 位数据



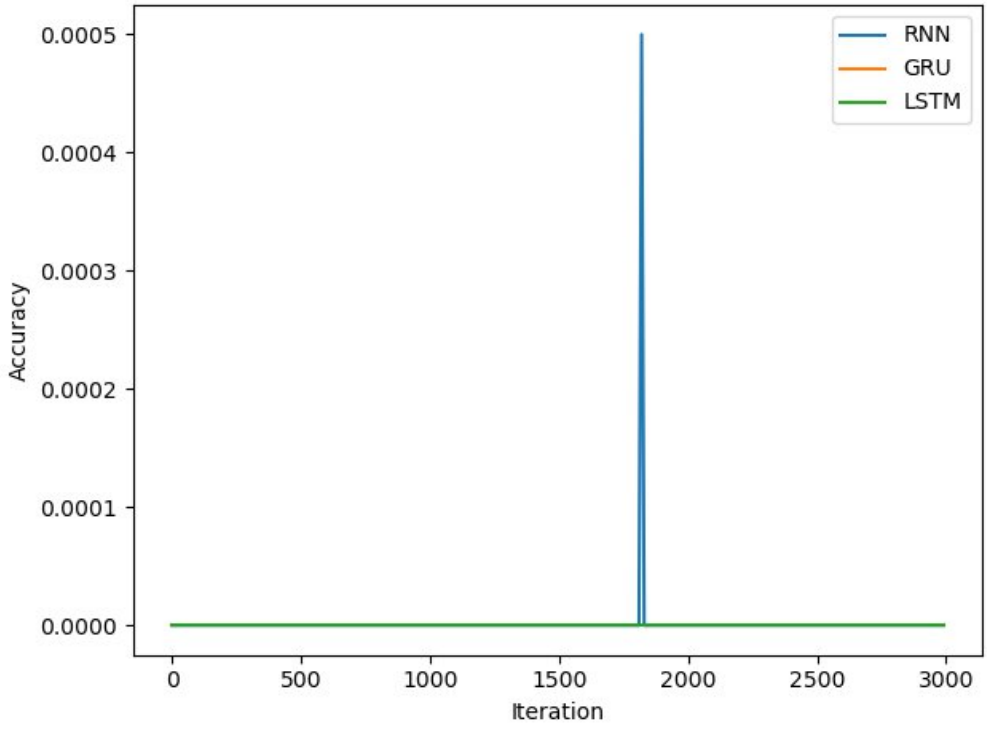
3 位数据



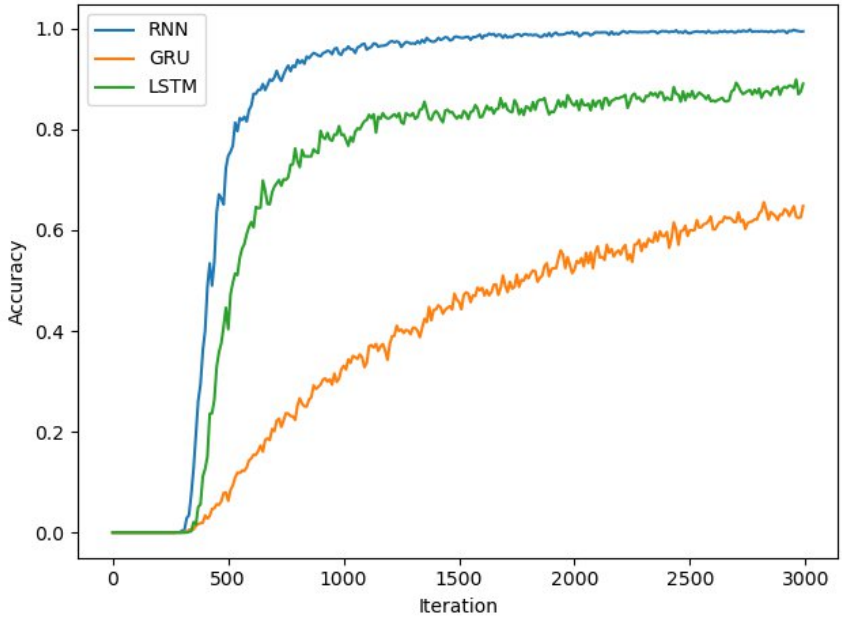
学习率 0.01 的模型具有更好的泛化能力，但是依旧对 2 位训练集无能为力。

4.5 RNN 类型

2 位数据



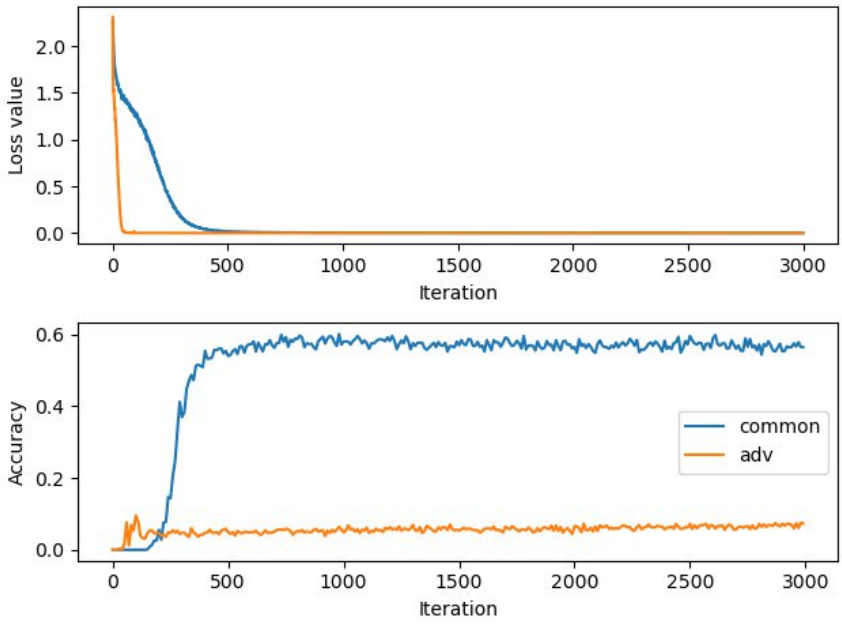
3 位数据



可见普通 RNN 网络具有更好的泛化能力。

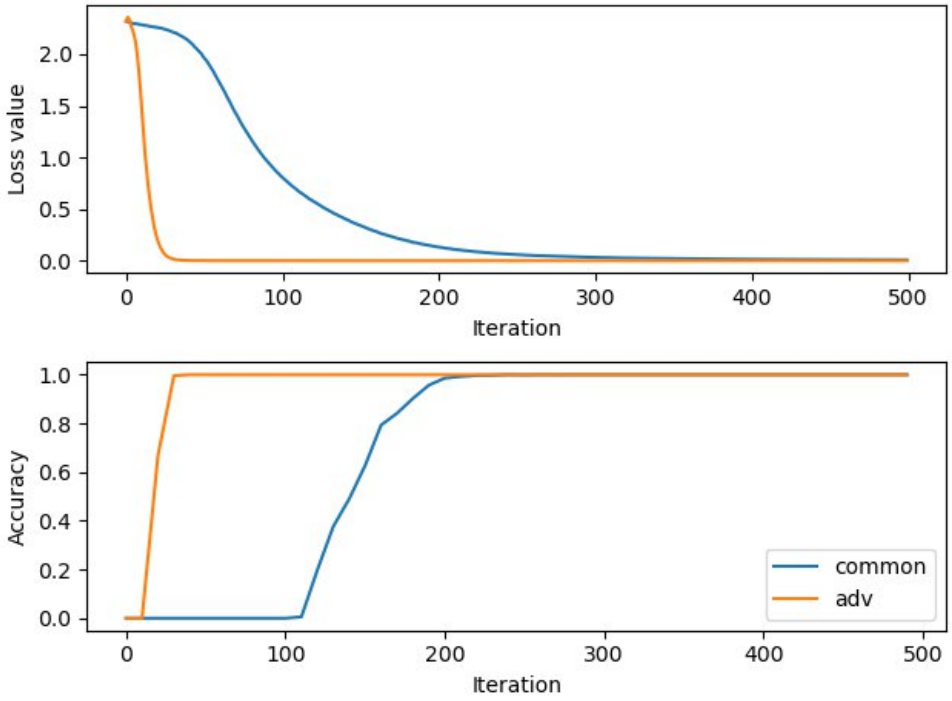
5 总结

综合以上实验，我们可以给出优化后的网络模型:
Embedding 层维度：64
RNN 类型：普通 RNN 网络
RNN 层数：2
学习率：0.01
分别以 2，50，100 位数据进行试验，结果如下:
2 位数据

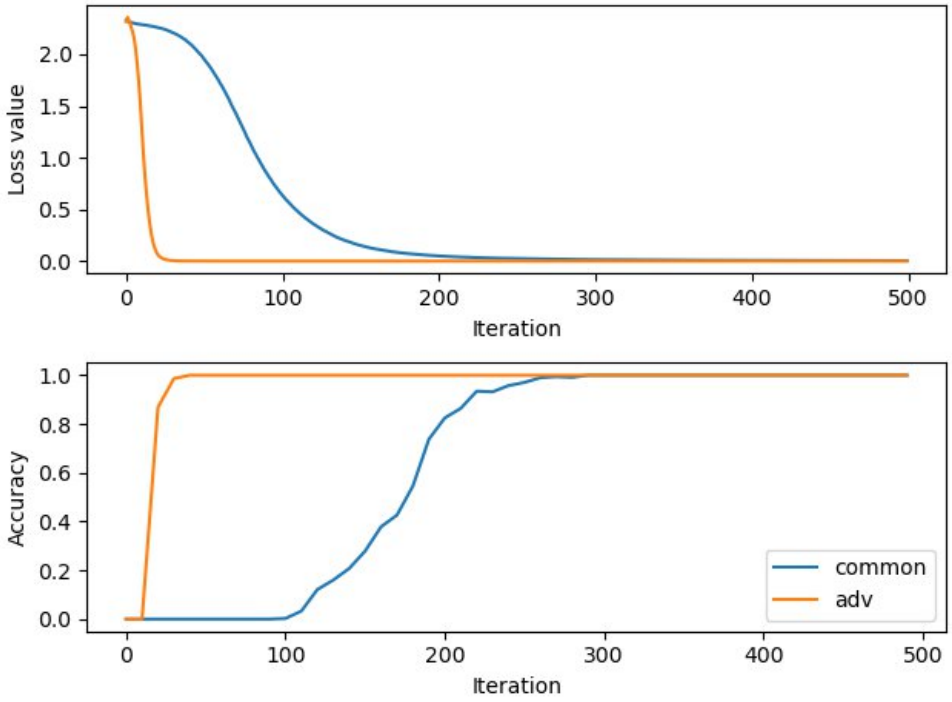


发现优化模型的准确率并不理想，思考之后我认为是多个优化导致 loss 收敛速度过快，使准确率没有足够的时间达到一个较高水平。这就涉及到一个取舍问题，需要对迭代速率进行一些妥协，放弃一些优化，才能在这种情况获得可以接受的准确率。

50 位数据



100 位数据



可见优化模型在多束情况下能力有了较大增强。

运行指令 python source.py