PRML Assignment-2 Report

> This document contains the report of assignment-2 in PRML course.
>
> In this assignment, you are required to design a RNN-based neural network to simulate an adder.

Table of Contents

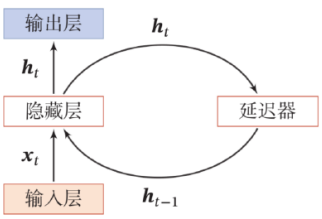
PRML Assignment-2 Report

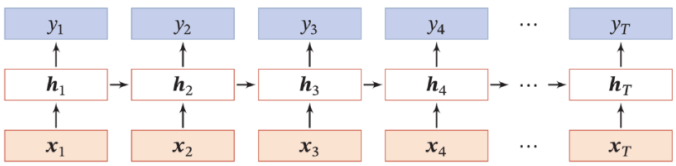
- Table of Contents
- Preparation
  - 1. RNN原理
  - 2. LSTM模型
  - 3. GRU模型
- Part I. Construct Model With PyTorch
  - 1. 安装PyTorch
  - 2. RNN加法器模型
  - 3. 实验观察
- Part II. Improve Model
  - 1. 不同数据长度
  - 2. 不同学习率
  - 3. 不同隐藏模型
  - 4. 不同网络层数
  - 5. 最优组合模型

Preparation

1. RNN原理

RNN（Recurrent Neural Networks)是能够处理序列输入任务的模型，在模型的训练过程中，相比其他神经网络，RNN引入了隐藏状态，也就是所谓的记忆能力：通过使用带自反馈的神经元，能够处理任意长度的时序数据。





左边为RNN的原理图，右边为RNN按照时间展开的原理图。用公式描述为：

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t)$$

对于一个简单的循环网络(Simple Recurrent Network)来说，其状态更新公式为：

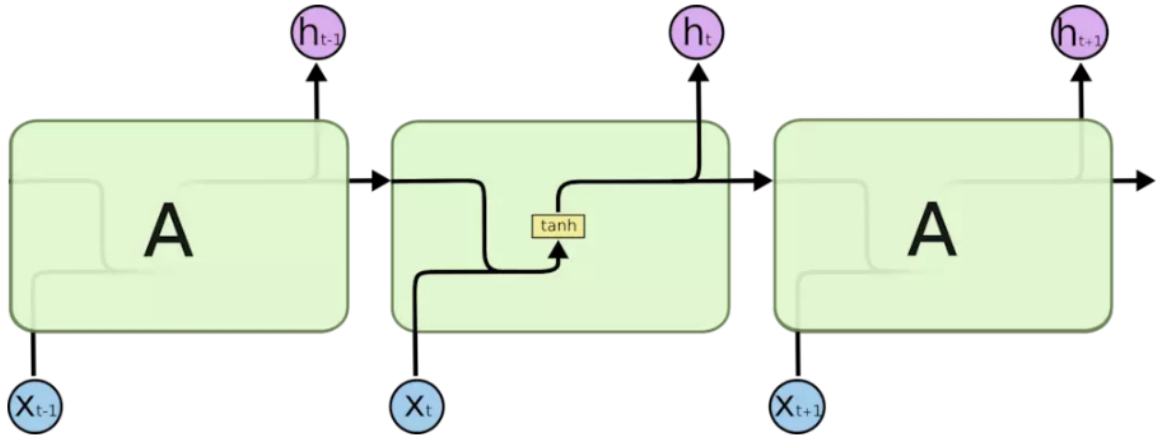
$$\mathbf{h}_t = f(\mathbf{U}_h \mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t + \mathbf{b}_n)$$

其中，$\mathbf{x}_t$为 t 时刻下输入， $\mathbf{h}_t$为 t 时刻下隐状态 , f为非线性激活函数(tanh等)。预测公式为：

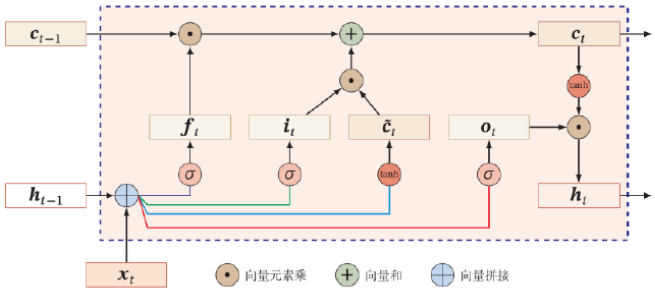
$$\mathbf{y}_t = softmax(\mathbf{W}_y[\mathbf{h}_t,\ 1]^T)$$

2. LSTM模型

LSTM是一种RNN的特殊类型，可以学习长期依赖信息。在标准的RNN中，重复的模块只有一个非常简单的结构，如tanh层：



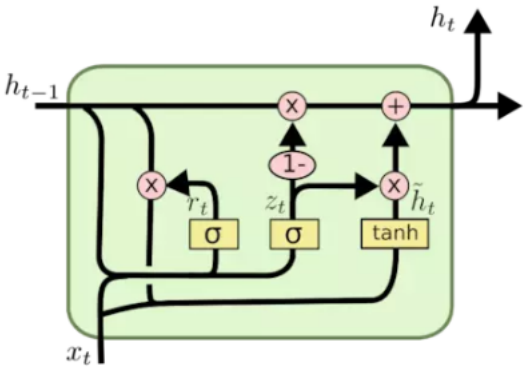
而对于LSTM来说，重复的模块具有一个不同的结构。不同于单一的神经网络层，整体上除了h在随时间流动，细胞状态c也在随时间流动，细胞状态c就代表这长期记忆。以下为LSTM示意图：



$$\mathbf{i}_t = \sigma(W_i\mathbf{x}_t + U_i\mathbf{h}_{t-1} + \mathbf{b}_i),$$
$$\mathbf{f}_t = \sigma(W_f\mathbf{x}_t + U_f\mathbf{h}_{t-1} + \mathbf{b}_f),$$
$$\mathbf{o}_t = \sigma(W_o\mathbf{x}_t + U_o\mathbf{h}_{t-1} + \mathbf{b}_o),$$
$$\tilde{\mathbf{c}}_t = \tanh(W_c\mathbf{x}_t + U_c\mathbf{h}_{t-1} + \mathbf{b}_c)$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t,$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t),$$

3. GRU模型

Gated Recurrent Unit (GRU)是LSTM的一个变体，它将忘记门和输入门合成了一个单一的更新门。同样还混合了细胞状态和隐藏状态，和其他一些改动。



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Part I. Construct Model With PyTorch

> In this part, you are required to learn how to write your source code based on Tensorflow 2.0 (TF) or PyTorch (PT). Source code based on TF or PT is available in directory `example` . You are required to clone one of them to your own directory and finish the rest of codes. (If you find any mistakes from the example source code, please contact TA @xuyige.)

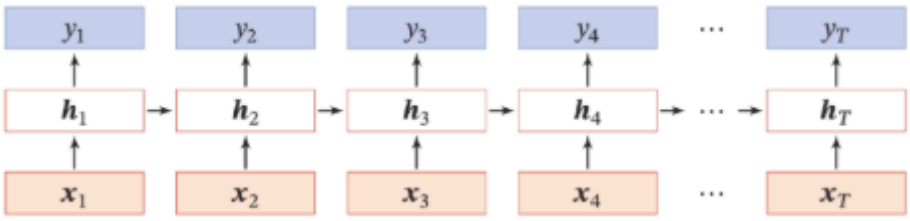
1. 安装PyTorch

首先查询机器的Cuda版本（10.0），然后使用命令行完成安装即可。

```
$  conda install pytorch torchvision cudatoolkit=10.0 -c pytorch
```

2. RNN加法器模型

利用RNN构建一个加法器模型，将两个数A和B通过训练进行相加并得到结果。对比二进制加法器可以发现，RNN在处理时是有相似之处的：



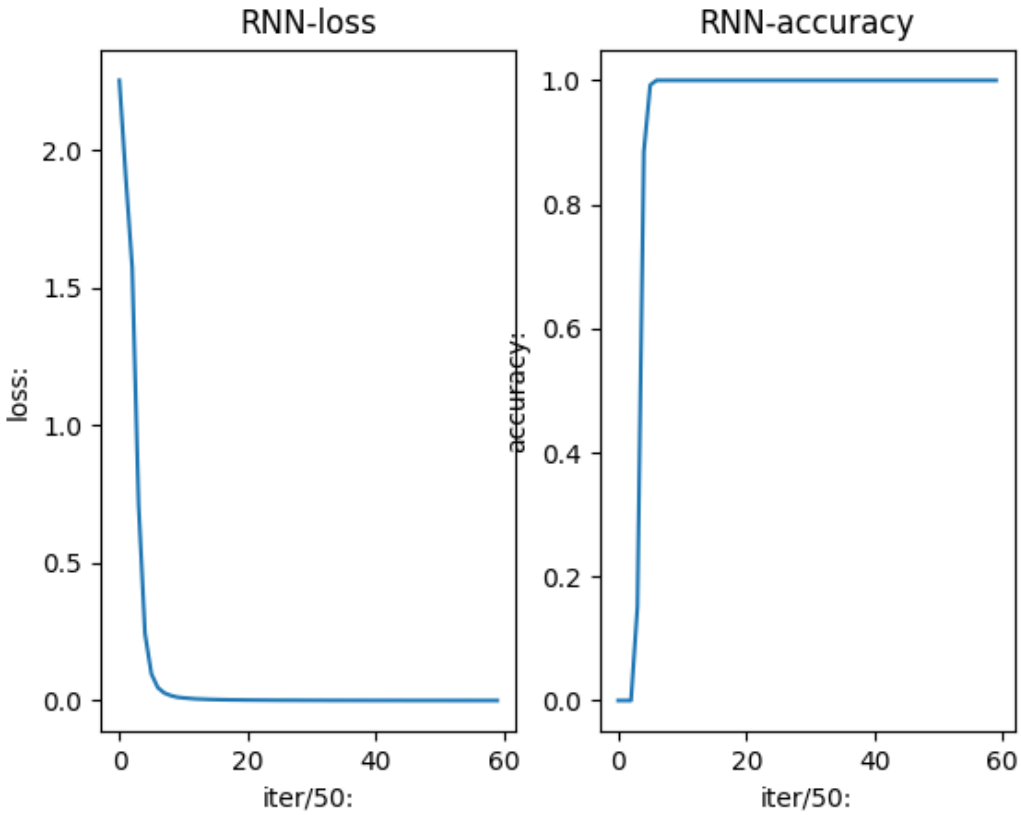
查看代码`myPTRNNModel`部分，可以发现，初始化的时候定义了输入层的embed_layer，将10个数字embedding成32维，并且定义了一个输入和输出层维度都是64维的两层RNN网络。全连接输出层定义了一个输入64维映射到10维的线性全连接层（$y = Ax + b$）。

```
    def __init__(self):
        super().__init__()
        # nn.Embedding(num_embeddings, embedding_dim, padding_idx=None,
max_norm=None, norm_type=2, scale_grad_by_freq=False, sparse=False)
        self.embed_layer = nn.Embedding(10, 32)
        # nn.RNN(input_size, hidden_size, num_layers=1, nonlinearity=tanh,
bias=True, batch_first=False, dropout=0, bidirectional=False)
        self.rnn = nn.RNN(64, 64, 2, batch_first=True)
        # nn.Linear（in_features, out_features, bias=True ）
        self.dense = nn.Linear(64, 10)
```

RNN各层已经定义好了，我们只需要完成forward函数。对于forward过程来说，通过查资料可以知道，简单的RNN网络就是将输入进行embedding，然后将embedding的结果进行连接，然后扔进RNN，然后将RNN层的结果扔进全连接层，得到的结果就是最终的结果。

3. 实验观察

为了更好地观察实验结果，将loss和accuracy在训练过程中的变化分别用展示出来：



可以看到，对于这个问题，RNN模型的效果是非常好的，甚至远不用预设的3000个steps（从输出来看大概600个steps）就已经达到了精度为1的结果。分析认为，加法器模型对于RNN来说，每一位的输入仅仅位当前位的两个输入和上一个位的进位，得到了当前位的隐藏状态，当前隐藏状态的个位就是该层输出，因此比较简单（即依赖距离太短）。

> 注：这里为了画图方便，修改了部分函数的返回值，并且在每50个steps时，记录训练的loss和accuracy。

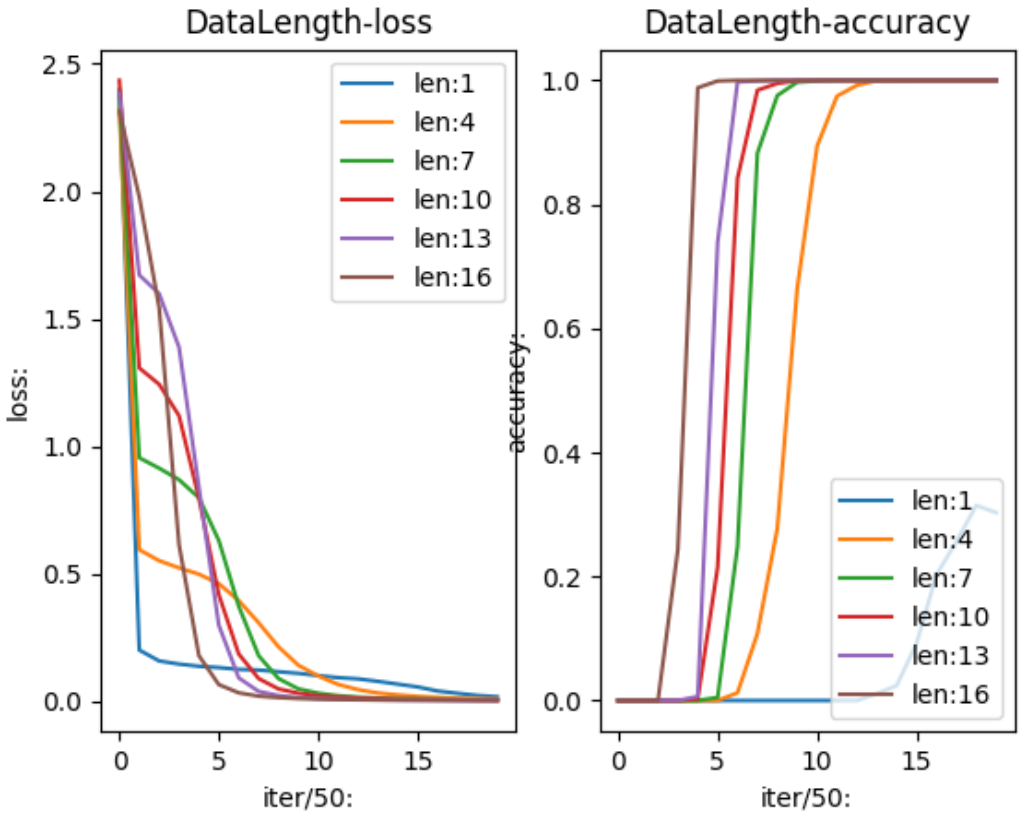
Part II. Improve Model

> In this part, you are required to modify the length of digits. You need to analyze the performance of your model and then improve your model.

考虑到RNN对这个问题的效果比较好，甚至在几百个steps范围内就能达到1的准确率，为了之后的训练比较方便，以下的实验将steps的默认值从3000改为1000。

1. 不同数据长度

考虑到python的`int`最大大概在`10^19`左右，为了考察不同长度的输入数据对模型的影响，我写了一个`compare_length()`函数，比较了当数据长度分别为`1,4,7,13,16`时，原来的RNN模型的表现(RNN, lr=0.001, steps=1000)如下图所示。

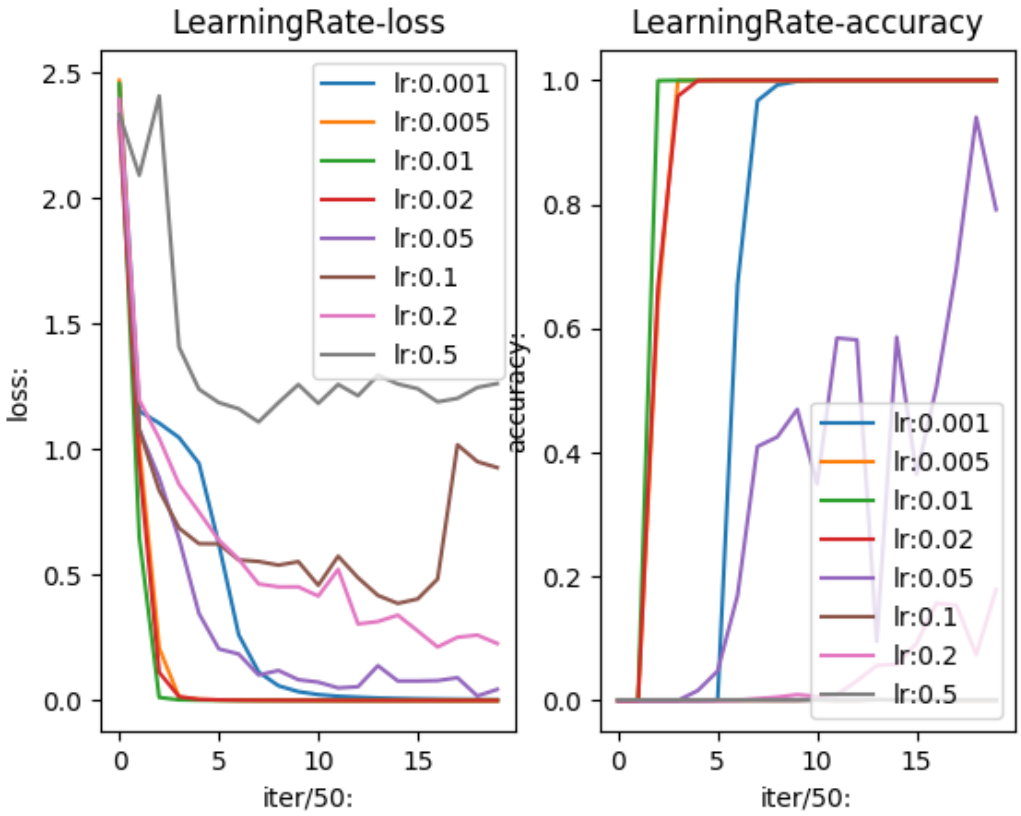


从图中可以看到，其他情况不变的情况下，随着数据长度的不断增加，训练的准确率收敛到1所需的steps不断减小，而数据长度为1时，训练的准确度竟然无法在1000steps以内到达1，这可能是因为数据长度只有一位时，进位信息反而对当前位的学习造成了干扰。

2. 不同学习率

因为这个模型对该问题的高准确率预测，我能想到的improve第一个就是观察学习率的提升对于模型训练过程和结果的影响：

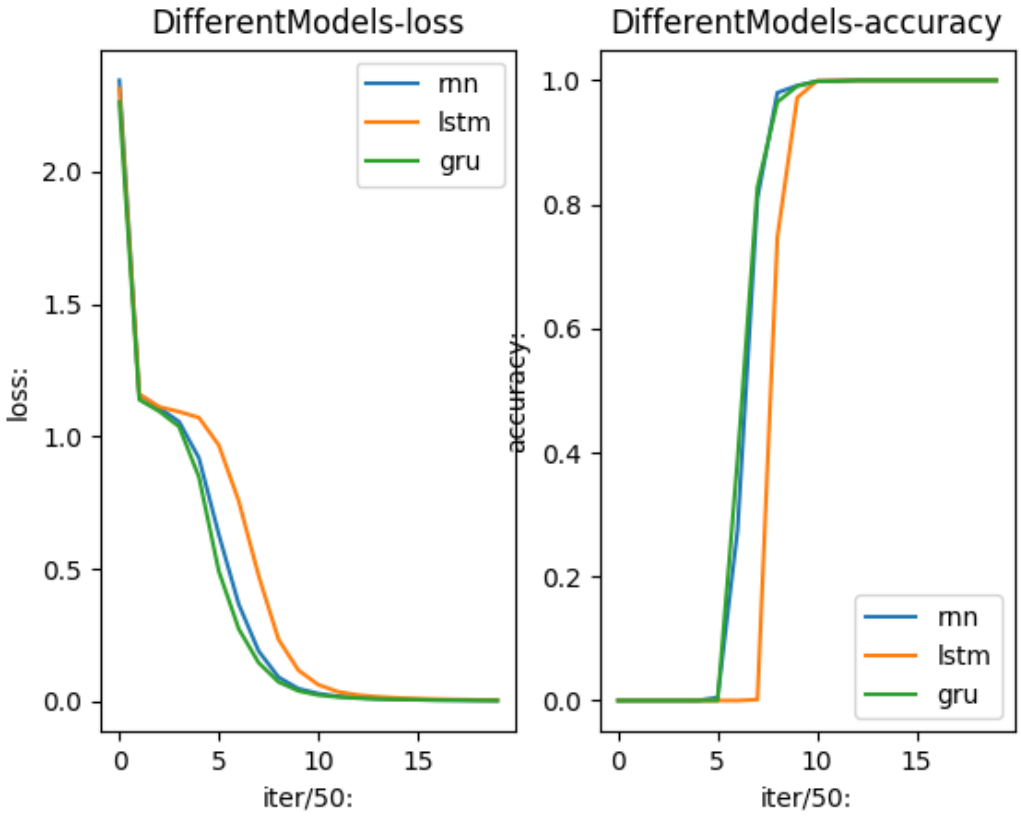
我写了`compare_learning_rate()`函数对不同学习率的RNN模型的表现进行测试(RNN, steps=1000, date length=9)，比较了learning_rate分别为`[0.001, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5]`时模型的表现，得到的结果如下：



从图中可以看到，随着学习率的提高，准确度在lr=0.01之前能够减小达到1所需的steps，而从lr=0.01开始之后，明显到达accuracy=1所需的steps显著增加，当lr取0.05以上时，accuracy和loss的波动大幅增加，并且最终的准确率甚至无法在1000steps内到达1，这可能是因为出现了过拟合的情况。因此，对于当前模型来说，lr设置成0.01是其他参数不变的情况下最优的结果。

3. 不同隐藏模型

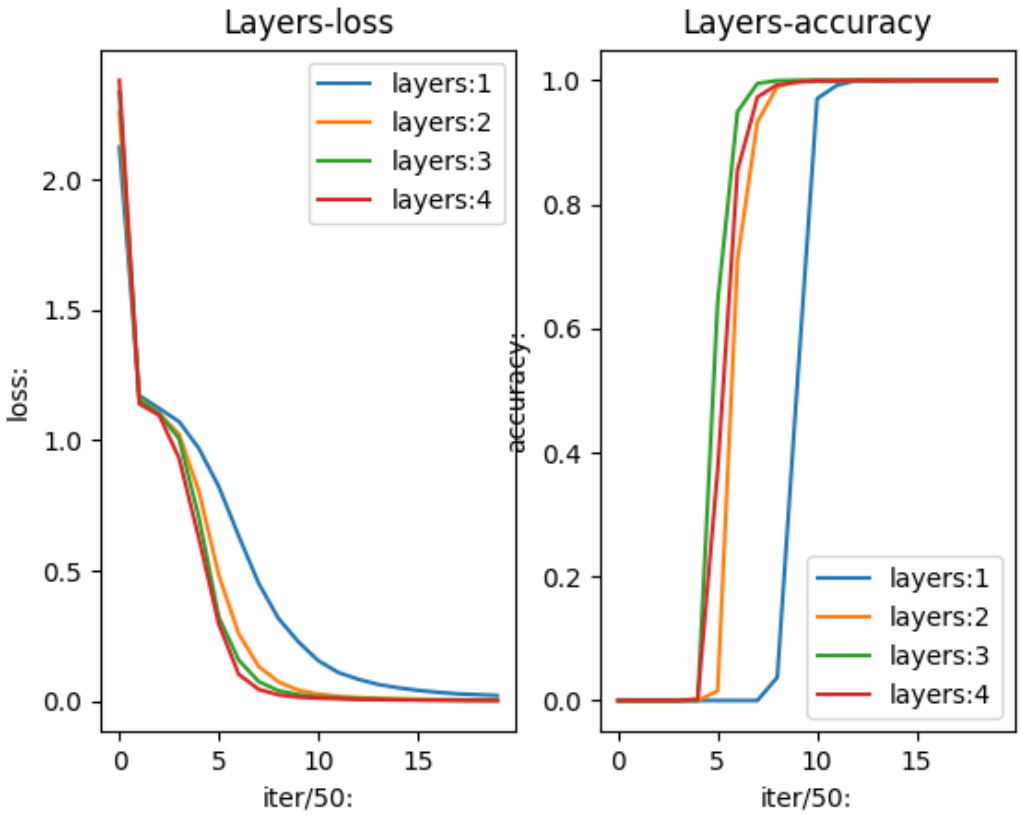
查阅资料得知，在选择不同的模型作为隐藏层的网络结构时，常见的有三种选择，RNN, LSTM, GRU。相关的介绍见前文的预备知识。因此，在`myAdvPTRNNModel()`里我定义了根据输入决定选用不同的隐藏层模型，并且在同样的参数情况下测试了模型的表现(lr=0.001, steps=1000, datalen=9)，结果如下：



从上图可以看出，三种模型在当前情况下都具有差不多的快速收敛能力，也最终都能达到相同的loss和accuracy。可能是参数设置(lr和data length）的问题，并未使得三种模型表现出明显的区分度。但总的来说，还是能得到gru>=rnn>lstm的结论。

4. 不同网络层数

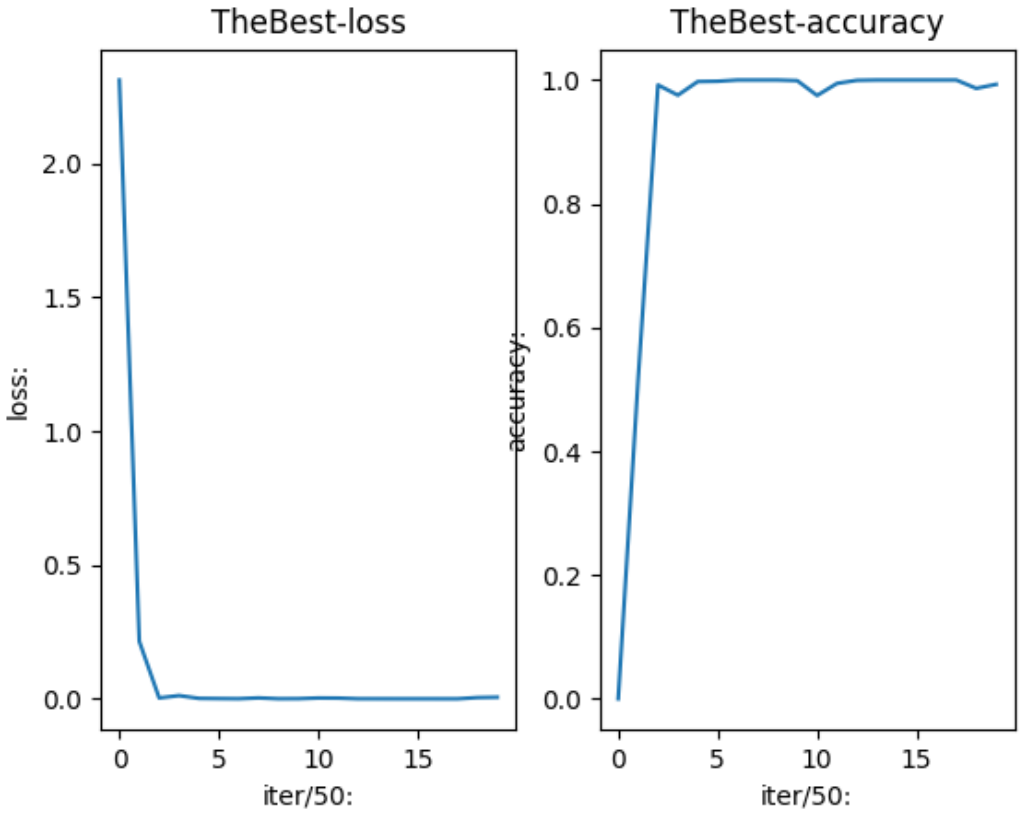
为了探索更多的可能性，我比较了在其他条件相同下，不同网络层数对结果的影响。在(lr=0.001，steps=1000, data length=9)的情况下，测试了layers分别为1, 2, 3, 4的结果，如下：



从结果来看，对于当前参数设置，2/3/4层网络层的结构显然对于加快学习速度有所帮助。多层网络结构有利于提升该模型。

5. 最优组合模型

经过前面的探索，我"粗暴"地将控制变量法得到的最优的参数放在一起，构建出一个(lr=0.01，steps=1000，model='gru'，data_length='16'，layers=3)的网络模型，最终得到的训练结果如下：



从结果来看，模型的收敛速度比之前任何单项参数的改变都要更快，但是显然存在过拟合导致的准确率抖动的问题。此外，模型的最终准确率仍然能够达到1（在1附近极小幅度波动），**这说明各种局部最优的参数组合在一起得到的结果并不一定是全局最优的。为了得到全局最优的模型，应该不断尝试组合不同局部最优参数，进行模型结果分析。此处受限于时间关系，不做进一步深入讨论。**