# PRML Assignment 2

May 6, 2020

## Introduction

In this assignment, we are going to simulate an adder(a model with arithmetic add function) using RNN-based neural network. And we will reorganize the train data set and modify the architecture of the model to see their performance on this task. Further more, we will do some experiment using different hyper parameters and try to find the parameters with best performance for each model.

## Dataset

The code for data generation has been provided in advance, in the *handout/data.py*. We generate the data in the following way: We specify the upper and lower bounds of the data, then we randomly generate two sample, calculate their sum, and padding them to a preset length(padding means add additional zeros in front of the number). we need to make sure that their sum's length won't exceed the preset length. In order to do that, we can simply make the preset length larger than the upper bound. After generate two number and their sum, we transfer these three number into digit list and reverse the list, make the lower digit in the front, we do this because considering the architecture of RNN, it only receive the information from the previous unit, and in the add process, the higher digit receive the digital carry from the lower digit, so we must reverse the digits, thus make the lower digits enter the RNN earlier than the higher digits.

## Models

### Basic RNN

RNN is a kind of neural network that can accept sequence data input, and it will transfer some information of previous input to later input. Its state is called hidden state, and it's calculate in the following way:

$$h_t = tanh(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh}) \quad (1)$$

in this equation, $x_t$ stands for the input of step $t$, and $h_{t-1}$ stands for the hidden state of last unit, and $W$ and $b$ are the model parameters. The $tanh$ can be replaced by other nonlinear functions such as $ReLU$. Also we can implement multi-layers RNNs (also called stacked RNN, with the second layer RNN taking in outputs of the first RNN as input.)

### LSTM

LSTM, the full name is Long Short-Term Memory neural network, is a kind of improved RNN, its architecture is shown in Figure1:

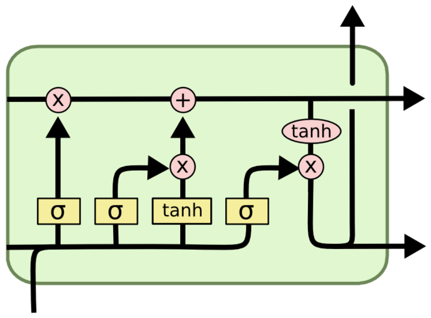


Figure 1: LSTM Architecture

it add the cell state, an input gate, an output gate and a forget gate based on basic RNN, it allows LSTM to deal the longer memory length and alleviate the gradient vanishment problem. Also, LSTM can be used as a complex nonlinear unit to construct larger deep neural networks.

## GRU

GRU (Gate Recurrent Unit) is also a improved RNN-based neural network. Its architecture is shown in figure2:

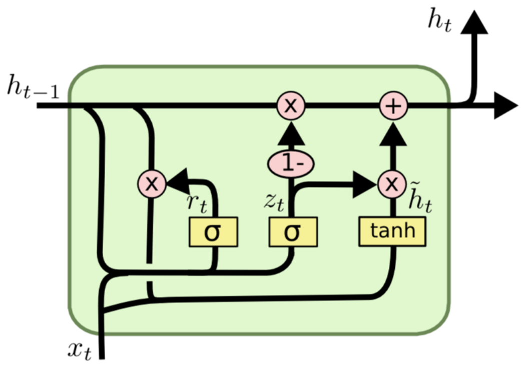


Figure 2: GRU Architecture

its motivation is similar to LSTM, they are both want to receive the information of more previous unit(rather than only last one), so they add the gate with probability that will mix some information of previous hidden state to the current hidden state. LSTM add the cell state to do the procedure, while the GRU simplify the architecture, directly using the hidden state to calculate. According to the paper, Their performance are similar, but the GRU is less computationally intensive, so it's a good alternatives to LSTM.

## Model Implement

We use pytorch in this assignment, it is an open source machine learning library, and we can build the neural networks easily based on it. In our code, our model class implement the *nn.module* class, and we rewrite the *__init()__* and *forward()* function, then we can simply use *model(input)* do run the model. Pytorch would automatically propagate the input forward, calculate the results, and retain some gradient-related information where needed. Furthermore, pytorch package some common models, so users can directly call these models. For example, if you want to use LSTM, you can just call *LSTM == nn.LSTM(input_size, hidden_size, num_layers)*, and it can be used as LSTM neural network. A small tips here is in this assignment, the data are generated in shape of $(batch, seq\_len, dim)$, but the RNN in pytorch take the input standard as $(seq\_len, batch, dim)$, since these kind of input are faster to calculate in memory. We can add $batch\_first = True$ setting to make it work on our data.
Then how can we use RNN to simulate an adder? The answer is simple, we map the digit to higher dimension, and then concatenate two input into one vector, put it into the model, and map the output two 10 dimension again. Using the softmax function to determine the digit. The procedure above can be described as follow:

**Algorithm 1** Train a RNN to be an adder

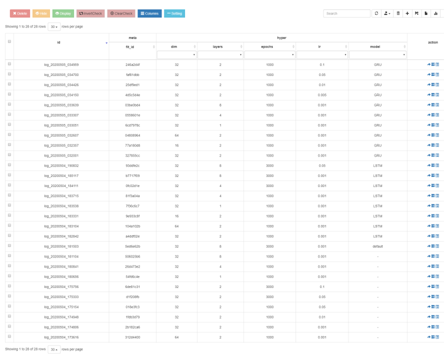
```
Result: A well-trained model
step = 0
while $step \leq EPOCHS$ do
    x, y, sum=GenerateData()
    x = embedding(x)
    y = embedding(y)
    input = concatenate(x,y)
    output = RNN(input)
    output = Linear(output)
    prediction = Softmax(output)
    loss = ComputeLoss(sum, prediction)
    loss.backward()
    optimizer.step()
    step += 1
end
```

In this way, we can train our RNN to learn the digital carry rules, thus play the function of an adder. You can use following command to run the code(in default setting):

```
python source.py
```

## Experiments

In this assignment, we use **fitlog** to record the experiment results. The results tableau is shown in figure 3:



Showing 1 to 28 of 28 rows 30 rows per page

| id | meta | hyper | | | | | action |
|---|---|---|---|---|---|---|---|
| | fit_id | dim | layers | epochs | lr | model | |
| log_20200505_034959 | 246a2d4f | 32 | 2 | 1000 | 0.1 | GRU | |
| log_20200505_034700 | faf81dbb | 32 | 2 | 1000 | 0.05 | GRU | |
| log_20200505_034426 | 25df9ed1 | 32 | 2 | 1000 | 0.01 | GRU | |
| log_20200505_034150 | 4d5c5d4e | 32 | 2 | 1000 | 0.005 | GRU | |
| log_20200505_033639 | 03be0bd4 | 32 | 8 | 1000 | 0.001 | GRU | |
| log_20200505_033307 | 0558601e | 32 | 4 | 1000 | 0.001 | GRU | |
| log_20200505_033051 | 6cd7978c | 32 | 1 | 1000 | 0.001 | GRU | |
| log_20200505_032607 | 04808964 | 64 | 2 | 1000 | 0.001 | GRU | |
| log_20200505_032357 | 77a180d8 | 16 | 2 | 1000 | 0.001 | GRU | |
| log_20200505_032001 | 327855cc | 32 | 2 | 1000 | 0.001 | GRU | |
| log_20200504_190832 | 93ddfe2c | 32 | 8 | 3000 | 0.05 | LSTM | |
| log_20200504_185117 | b7717f09 | 32 | 8 | 3000 | 0.001 | LSTM | |
| log_20200504_184111 | 0fc02d1e | 32 | 4 | 3000 | 0.001 | LSTM | |
| log_20200504_183715 | 81f3a04a | 32 | 4 | 1000 | 0.001 | LSTM | |
| log_20200504_183538 | 7f36c6c7 | 32 | 1 | 1000 | 0.001 | LSTM | |
| log_20200504_183331 | 9e933c8f | 16 | 2 | 1000 | 0.001 | LSTM | |
| log_20200504_183104 | 104a102b | 64 | 2 | 1000 | 0.001 | LSTM | |
| log_20200504_182842 | a4ddf02e | 32 | 2 | 1000 | 0.001 | LSTM | |
| log_20200504_181503 | 5ed6e62b | 32 | 8 | 3000 | 0.001 | default | |
| log_20200504_181104 | 506025b6 | 32 | 8 | 1000 | 0.001 | - | |
| log_20200504_180841 | 26d473e2 | 32 | 4 | 1000 | 0.001 | - | |
| log_20200504_180656 | 54fd6c4e | 32 | 1 | 1000 | 0.001 | - | |
| log_20200504_175756 | 6de81c31 | 32 | 2 | 3000 | 0.1 | - | |
| log_20200504_175333 | d1f208fb | 32 | 2 | 3000 | 0.05 | - | |
| log_20200504_175154 | 018e3fc3 | 32 | 2 | 1000 | 0.05 | - | |
| log_20200504_174948 | 1fdb3d79 | 32 | 2 | 1000 | 0.01 | - | |
| log_20200504_174806 | 2b182ca6 | 32 | 2 | 1000 | 0.001 | - | |
| log_20200504_173616 | 312d4400 | 64 | 2 | 1000 | 0.001 | - | |

Showing 1 to 28 of 28 rows 30 rows per page

Figure 3: Fitlog Results Tableau

## RNN Results

The basic RNN setting is:

$$\begin{aligned} &hidden\_dim = 32 \\ &layers = 2 \\ &epochs = 1000 \\ &learning\_rate = 0.001 \end{aligned}$$

And the result is shown in Figure 4:

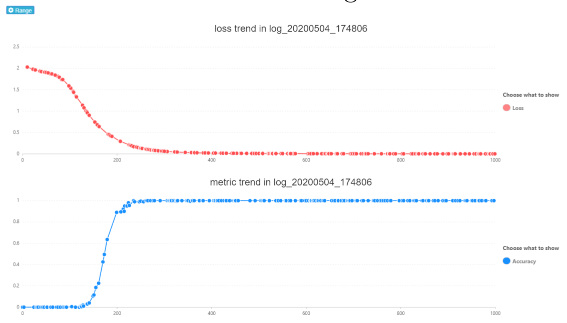


Figure 4: RNN Result

we can see that the loss came to convergence to less than 0.005 in less than 300 steps, and the accaurcy directly increase to 1. The loss decrease sharply during 100-200 epochs, Therefore, the accuracy increased sharply too. And after 300 epochs, the loss changes slightly, which would not affect the performance of accuracy(since it's prediction is based on softmax, so it won't change under small changes), so we can modify the range of epochs, get the more beautiful training curve shown in Figure 5:

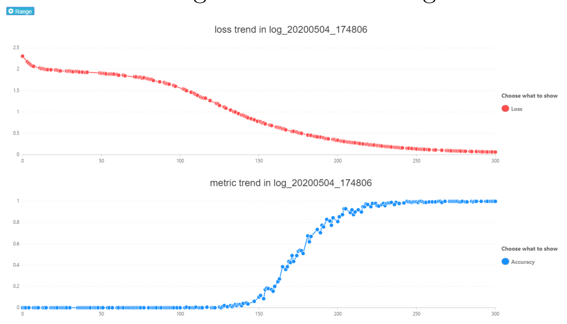


Figure 5: RNN Result (Epoch Range 0-300)

## LSTM Results

The basic LSTM setting is same to RNN's:

$$\begin{aligned} &hidden\_dim = 32 \\ &layers = 2 \\ &epochs = 1000 \\ &learning\_rate = 0.001 \end{aligned}$$

and it's result is shown in Figure 6:

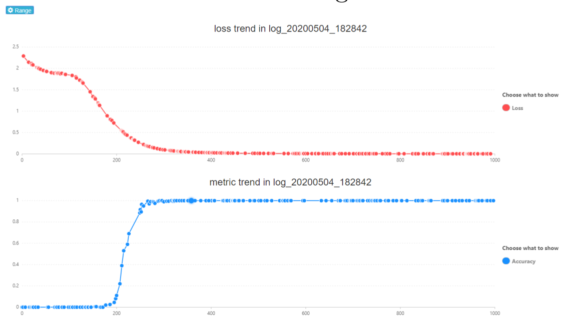


Figure 6: LSTM Result

We can find that its loss came to converge in about 400 epochs. And its accuracy increase to 1 in about 200 epochs, that means LSTM aslo works in this assignment. If we want to comparing it with RNN, we can also modify the plot range, shown in Figure 7:

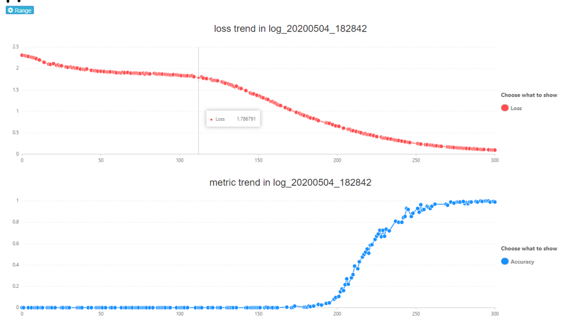


Figure 7: LSTM Result (Epoch Range 0-300)

Comparing the figures of RNN and LSTM, we can find that the loss of LSTM decreases slower than RNN. We think this is because the architecture of LSTM is more complex, it contains more parameters, so it's harder to train under the same learning rate (since the gradient is vanishing during the backward procedure, so the change of loss will be smaller under more complex architecture).

## GRU Results

The basic GRU setting is same to previous two models, similar to LSTM, it can work on this task. and its results are shown in Figure 8 and the result of epoch 0-300 is shown in Figure 9:

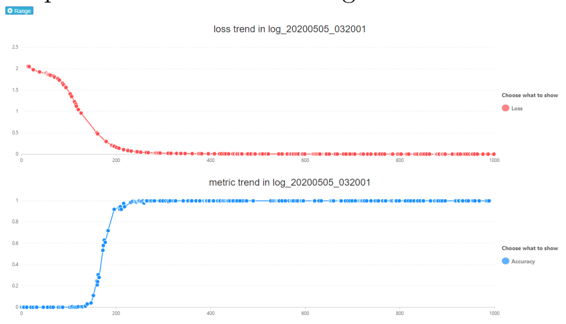


Figure 8: GRU Result

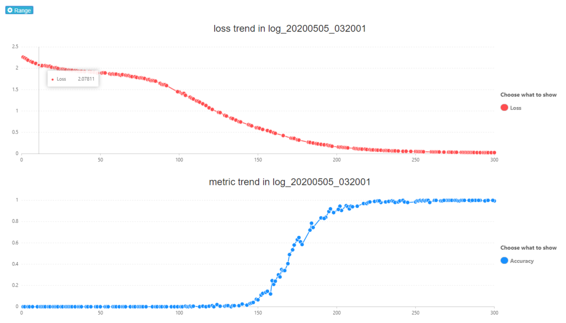


Figure 9: GRU Result (Epoch range 0-300)

We can find that the train curve of GRU is better than LSTM (train faster, loss converge faster, accuracy increase faster, since it has less parameters). Its performance is close to RNN, but in this train procedure, its loss and metric are not so smooth as other models. Sometimes the loss and metric may perform worse than previous epoch, that's may because the previously state was in the local optimal solution instead of the global optimal solution, and this situation might not happen in another training under same settings.

## Reorganize Data

### Modify Digit Length

In the original setting, the length of digits is 11, So we modify the data generation code, make it can generate digits of any length, and we find that model can learn the adder function for large number two, the train time in longer, but the loss and accuracy change trend are similar to small length. The results of longer length are are shown in Figure 10 and 11(the experiment setting is basic GRU setting):

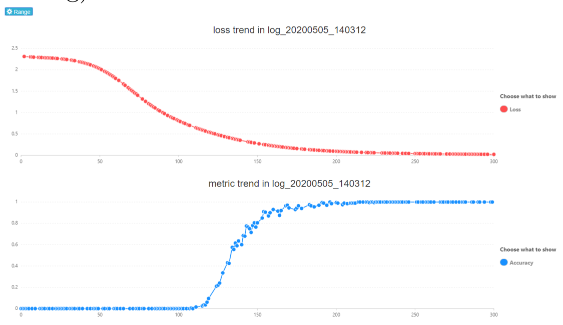


Figure 10: Digit Length 100

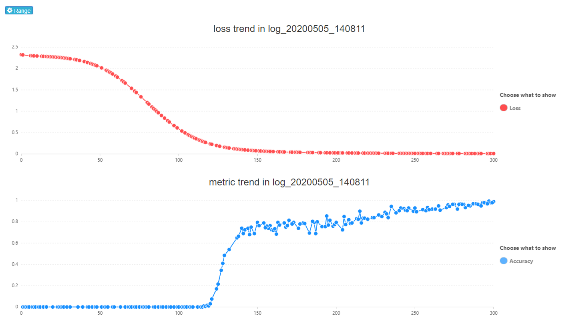


Figure 11: Digit Length 1000

Comparing to the previous result, we can find that the curve are similar, with slight difference. That's may because this task is simple, so there's no long dependency to previous unit.

### Modify The Distribution of Data

Although the model perform perfectly on the previous task, we can not conclude that the model have learned how to add. Since the train data and the test data are under the same distribution(not means they are in the same range, but their digital length are same). For example, in the original setting, $Train\_data \in (0, 555555555)$, $Test\_data \in (555555555, 999999999)$, they are not in same range, but both of their digital length are 9, so the model will learn how to carry on each digit. But if you modify the train data to smaller length, the performance will become worse. The performance is shown in Figure 12:

| Data Distribution | Accuracy on Testset |
|---|---|
| Train data : $(0,10^{10})$<br>Test data : $(0,\ 10^{10})$ | 1 |
| Train data : $(0,10^{2})$<br>Test data : $(0,\ 10^{10})$ | 0.7635 |
| Train data : $(0,10^{5})$<br>Test data : $(0,\ 10^{100})$ | 1 |

Figure 12: Results on Different Data Distributions

We find that when the train data is very small (less than 100), the training will not be sufficient (in 1000 epochs). We find that the wrong prediction examples are often wrong on the third digit, which means the model didn't learn how to carry digit on the third digit (which could also been explained as when it comes to higher digit, the model will choose to output 0 or 1 to get the loss decrease, rather than learning the rules of digital carry). **But when the train set's digital length is larger than 3, the model will perform perfectly no matter how long the test set digital length is**.

## Modify Hyper Parameters

### Hidden State Dimension

In this assignment, we map ten-dimensional digit to higher dimension vectors, as it's easy to understand, the larger the dimension is, the more information it can contains. As we modify the hidden state dimension, the results are shown in Figure 13,14,15.

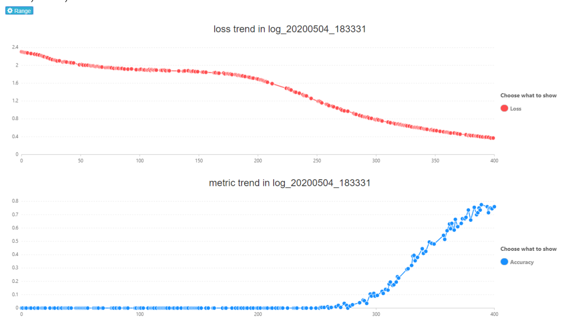


Figure 13: Hidden State Dimensions = 16

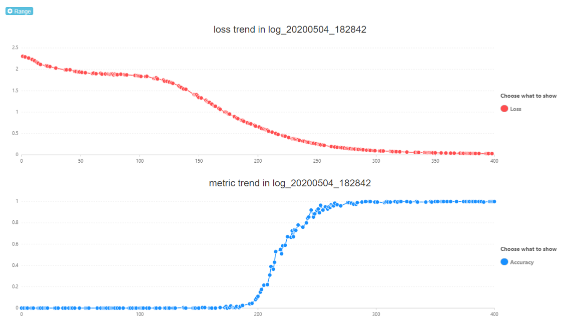


Figure 14: Hidden State Dimensions = 32

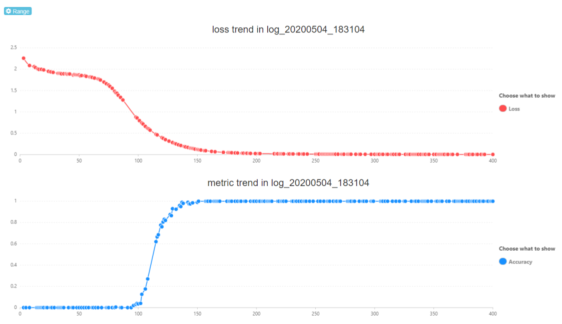


Figure 15: Hidden State Dimensions = 64

We can find that with higher dimension, the faster loss come to converge, but the training time for each epoch is increasing too, so it's you win some, you lose some.

### RNN Layers

Why we build a multi-layers RNN? The answer is to make the model deeper. With the additional hidden layer, it can understand the combination of the learned representation from the previous layer and create a new representation at a high level of abstraction. **But in this task, there's no need to stack multiple layers, since the task is too simple to be handled by few layers.** The results are shown in the Figure 16:

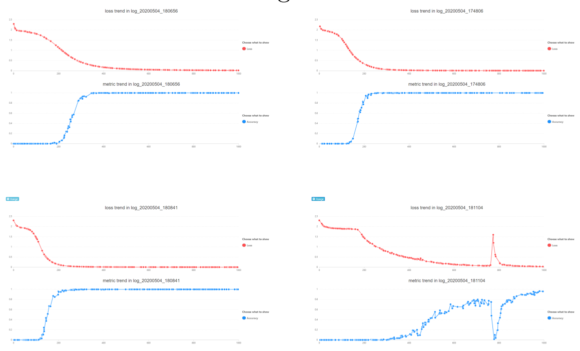


Figure 16: Layers = 1,2,4,8 (from top left to bottom right)

As we can see, when the $layers = 1, 2, 4$, they all can handle the task, and the model with deeper layers improve accuracy faster (since it has stronger representation ability), but when $layers = 8$, the model is too deep for this task. The model might not learn the correct optimize direction, so it fluctuate violently, deviating from convergence, the stronger data and longer epochs are needed in this situation, which is not cost-effective in this task.

### Learning Rate

Proper learning rate may help model's loss come to convergence fast, while the improper learning rate may cause the train very slow (when the rate is too small) or cause the model's loss could not decrease (when the rate is large, gradient change make it farther from optimal). So we do some experiments to see its impact, shown in Figure 17:

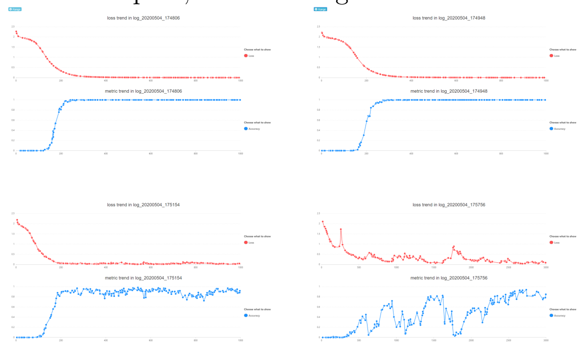


Figure 17: Learning Rate = 0.001,0.01,0.05,0.1 (from top left to bottom right)

As we can see, the learning rate about $[10^{-3}, 10^{-2}]$ is suitable for this model, the convergence is very fast. but when the learning rate is larger than 0.1, it will never converge and the training is fail.

## Conclusion: How to Improve Model

**Choose the suitable model architecture.** RNN is used in this task, so we can consider whether to use basic RNN, LSTM or GRU, or other variant RNN models. When considering a problem that the input is a sequence, there are some other models that can also be considered, such as the transformer and BERT which are based on the attention mechanism.

**Adjust the hyper parameters.** First, we must consider the hyper parameters of the model architecture, such as data dimension, layers, etc., which will affect the expression ability of the model, while too many parameters may also cause difficulties in training. After that, the main issue to be considered are the hyper parameters during the training process. The epoch need not be too much, and can also be set to automatically end the training after the loss converges. The learning rate should be appropriate, and some learning rate decay policy can be adopted to make the learning rate decay when the epoch increase to ensure stable loss. For example, using exponential decay will make the learning rate decay in a exponential curve.

**Use GPU for training.** GPU has stronger computing power and memory than CPU, and Nvidia added a tensor operation unit to the 20 series graphics cards and professional TPU, which can make the tensor operation in the machine learning model much faster, so it can speed up our training.

## References

[1] https://pytorch.org/

[2] https://fitlog.readthedocs.io/zh/latest/index.html