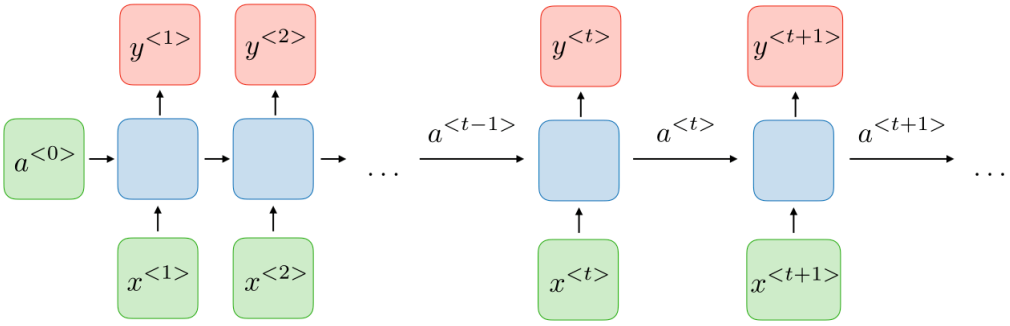
PRML Assignment2

1. Principle

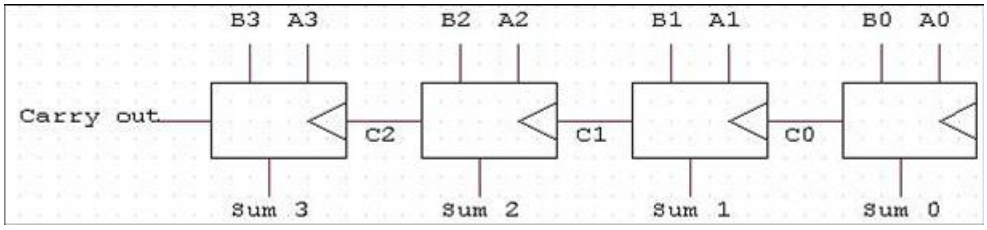
1.1 RNN

Recurrent neural networks, also known as RNNs, are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. They are typically as follows:



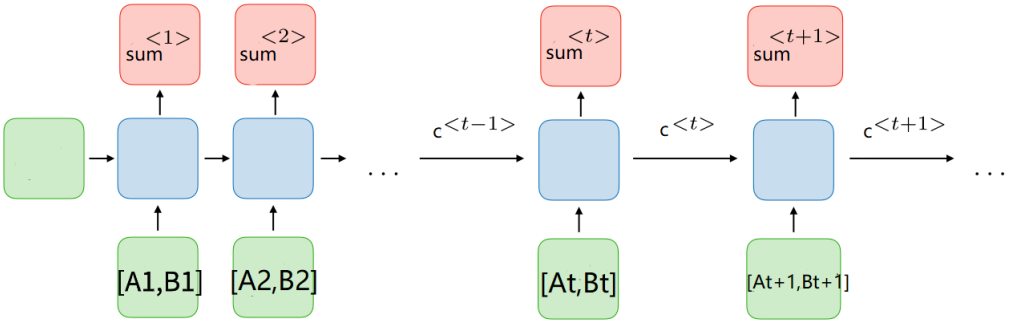
1.2 Binary addition

Binary addition moves from the LSB towards the MSB, with a carry bit passed from the previous addition. It can be concluded that the two **bear similarities**: For binary addition, at each time step, two bits are added and generate a sum and a carry bit, which is passed to the addition at the next time step. At the end, all these sums are combined to get the total sum.



1.3 Connection

Thus, it is suitable to use RNN for binary addition. We want the neural network to move along the binary sequences and remember when it has carried the 1 and when it hasn't, so that it can make the correct prediction.



2. Model

2.1 Basic Model

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

    def forward(self, num 1, num2):
        #logits1,2=(batch_size,seq_len,input_size)
        logits1 = self.embed_layer(num1)
        logits2 = self.embed_layer(num2)
        input_logits = torch.cat([logits1,logits2],dim=2)
        #input_logits=(batch_size,seq_len,2*input_size)
        out,h_n = self.rnn(input_logits)
        logits = self.dense(out)
        return logits
```

2.1.1 Initialization

This model has 4 layers: an embedding layer, two hidden layers and a fully connected layer.

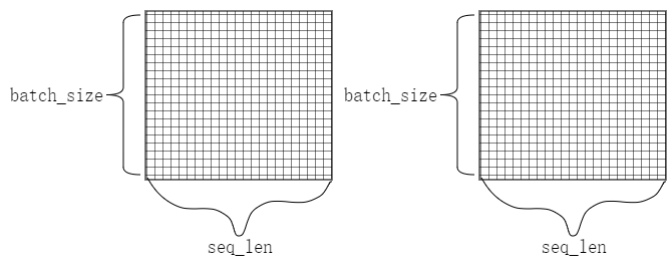
-nn.Embedding(index, dim): index refers to an item in the dictionary. Dim indicates the dimension of the vector each item is mapped to. In this case, the 0-9 digits are mapped to a vector of 32 dimension.

-nn.RNN(input_size, hidden_size, num_layers): input_size indicates the dimension of features, hidden_size shows how many hidden cells lie in the hidden state, num_layers records the number of hidden layers. Batch_first=true means the input for the neural network begins with batch_size. In this case, the input_size=64 because each number is a 32-D vector, 64 features in total. hidden_size=64 because we need the same number of features to describe hidden state. num_layers=2 meas the network has 2 hidden layers.

-nn.Linear(input_size, output_size): this is a fully connected layer which can help get the exact features as the final output, using the features from the previous layer. In this case, the input is 64-D and the output is 10-D. The dimension with the max value indicates which digit it is.

2.1.2 Forward

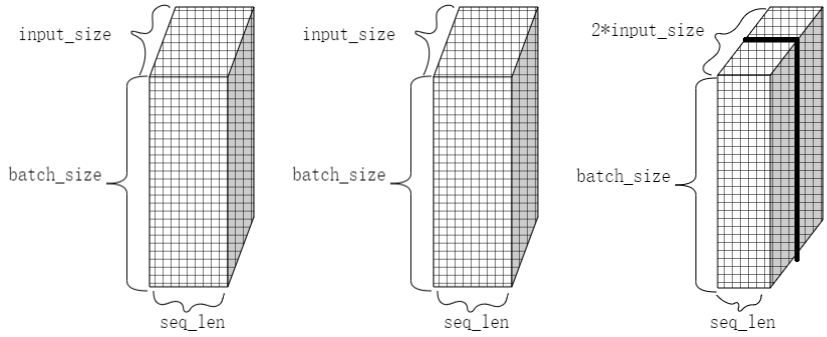
The forward method defines how the layers are organized and how the neural network works. In this case, the inputs num1 and num2 are 2 matrix: Each row refers to a number list from the number's LSB to MSB, there are batch_size rows and the column size is the length of the digits.
num1,2=(batch_size,seq_len)



num1 *num2*

Then, num1 and num2 get their embedding after passing embed_layer. The inputs change from 2D to 3D. **logits1,2=(batch_size,seq_len,input_size=32)**

Use torch.cat to concat the two input in the input_dim dimension.(same as alignment in bit when doing binary addition) Now there is only one input **input_logits=(batch_size,seq_len, 2*input_size=64),** with each floor representing a combination of the matrix expression for two numbers.



logits1 *logits2* *input_logits*

The **input** for rnn layer has two parameters: **input(batch_size,seq_len,input_size)** and **h0(batch_size,num_layers*num_directions,hidden_size)**.

The rnn layer yields two **outputs**: **out(batch_size,seq_len,hidden_size*num_directions)** and **h_n(batch_size,num_layers*num_directions,hidden_size)**

In this case, the input_logits(batch_size,seq_len, 2*input_size=64) is passed throught the rnn layers and h0 is None since there is no initial carry bit.

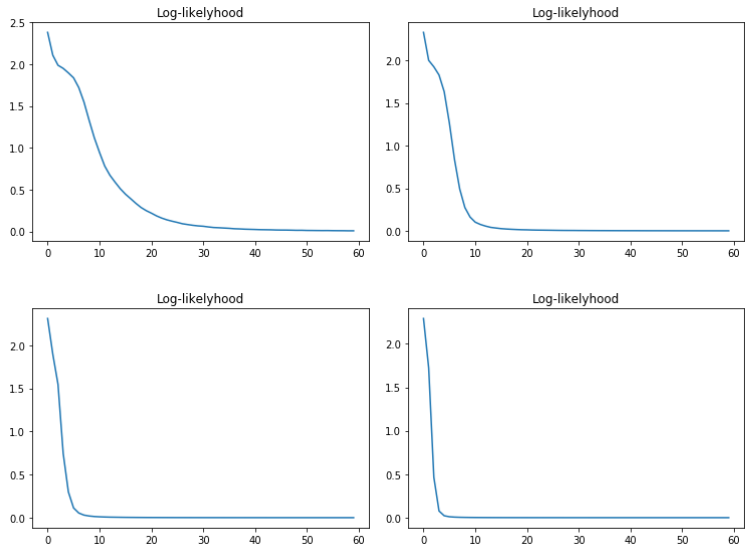
If forward method is defined in the subclass of nn.module, backward method is realized automatically using Autograd.

2.2 Advanced Model

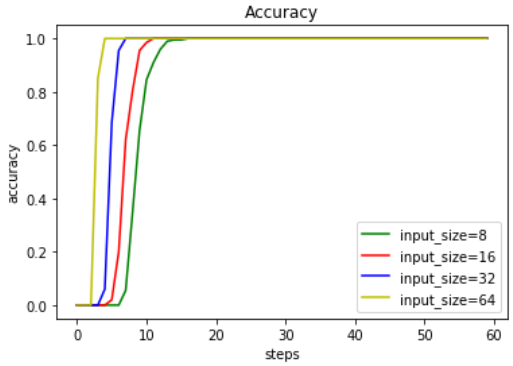
2.2.1 input_size

(seq_len=11,num_layers=2,hidden_size=32)

input_size=8,16,32,64



The accuracy on the test set with the size of 2000 in those 4 cases all reach 100%. As the input_size increases, it takes fewer training steps for loss function to converge and the final loss value gets smaller. However, the training time also gets longer.



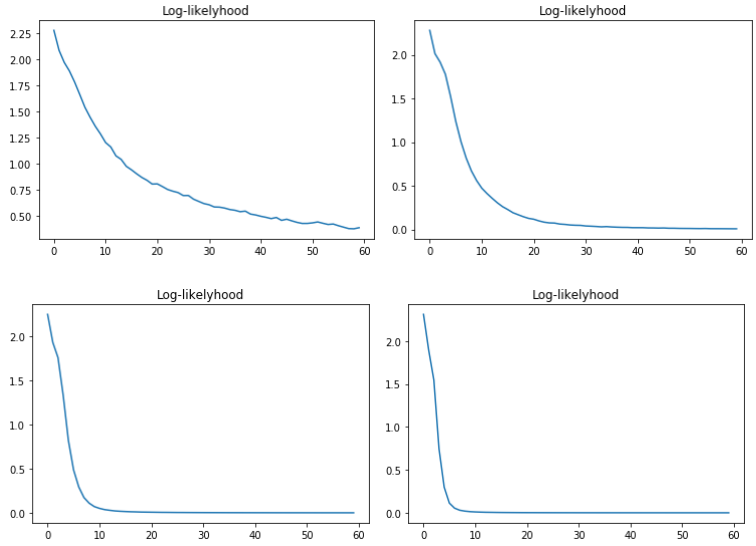
From the chart, we can see that when input_size=64, the accuracy for the model reaches 1 the fastest, a lot better than the others. So, we choose 64 for input_size.

2.2.2 hidden_size

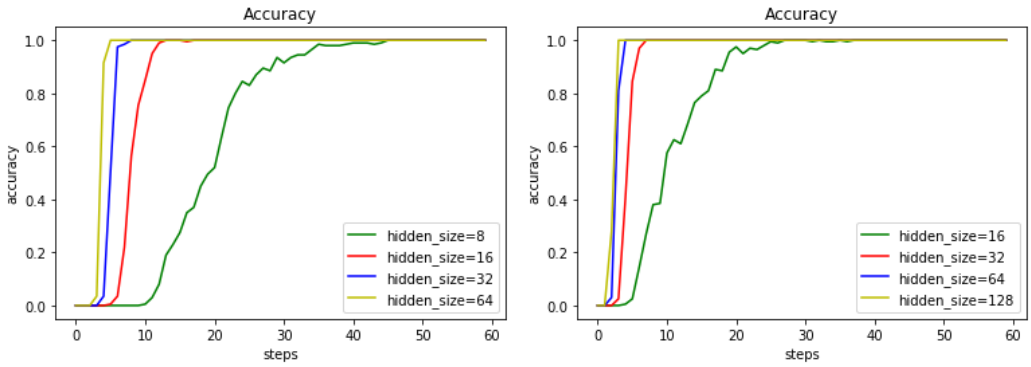
(input_size=64, seq_len=11,num_layers=2)

hidden_size=8,16,32,64

hidden_size	accuracy(test_set=2000)
8	10.6%
16	100%
32	100%
64	100%



Too few hidden layers results in low accuracy. The loss function does not converge. As the hidden_size increases, it takes fewer training steps for loss function to converge and the final loss value gets smaller. However, the training time also gets longer.

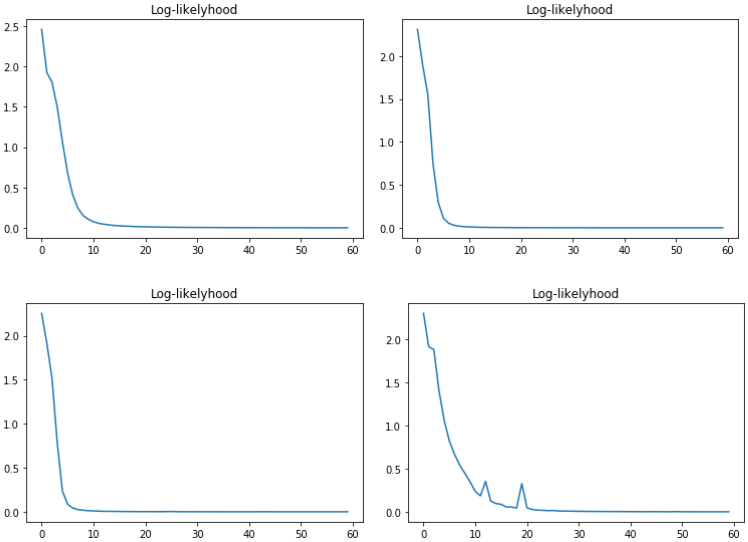


From the chart, the speed for accuracy to reach 1 when hidden_size=64 and 128 are very close. That shows increasing hidden_size from 64 to 128 doesn't help a lot. **According to Sheela and Deepa (2013), one of the major challenges in the design of neural network is the fixation of hidden neurons with minimal error and highest accuracy.** The insufficient hidden neurons will lead to bad stability, that is, the error on the nodes to which their output is connected is high. The excessive hidden neurons will cause over fitting; that is, the neural networks have overestimated the complexity of the target problem. For this case, hidden_size=64 can be a compromised choice. And that also coincide with the length of carry bit(0 or 1, embedding is also 64D.)

2.2.3 num_layers

(input_size=64, seq_len=11,hidden_size=64)

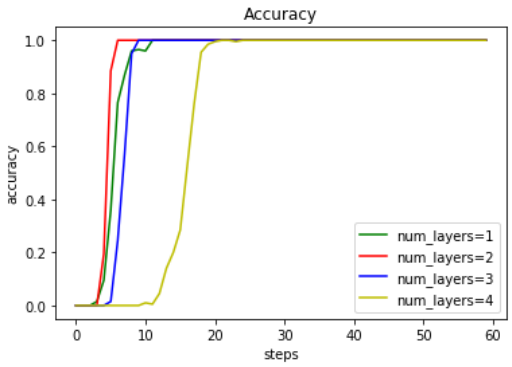
num_layers=1,2,4,8



The accuracy on the test set with the size of 2000 in the first three cases all reach 100%. However, when 8 RNN layers are stacked, the accuracy is only 99.79%, and the loss function is not that smooth. In certain range, as the num_layers increases, it takes fewer training steps for loss function to converge and the final loss value gets smaller. However, the training time also gets longer. **But if there is too many RNN layers, the performance gets worse.** "The deeper, the better" not always holds true. **One one hand, depth allows for longer term dependencies to be learnt well.** However, in this binary adding case, the current state only depends on its last state(carry bit), too many layers may only **results in overfitting**, that is, the neural networks have overestimate the complexity of the target problem. **On the other hand, RNNs are "recurrent".** Because of this prosperity, you don't need to stack them to get powerful functions - you get them over time.

One example is that it's relatively common to have skip-connections between the layers in a deep RNN, so it's able to control itself how much "depth" to use.

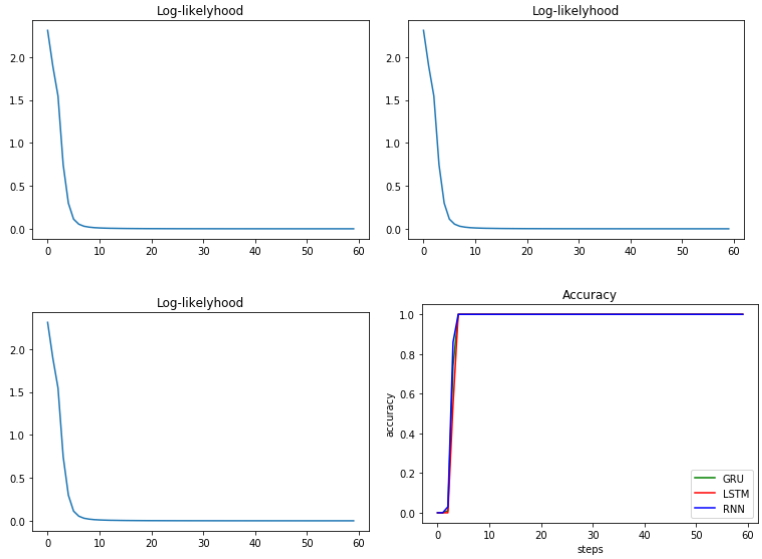
for num_layers=1,2,3,4, the speed for accuracy to reach 1 differs. Judging from the chart, num_layers=2 is the optimum case.



2.2.4 LSTM/GRU

(input_size=64, seq_len=11,hidden_size=64,num_layers=2)

RNN,LSTM,GRU

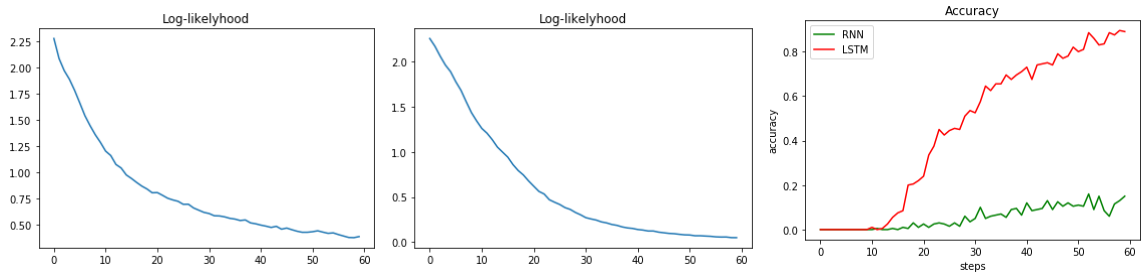


It takes more time to train LSTM/GRU model than RNN model. It is known that LSTM/GRU relieves long distance dependency problem by filtering previous states, however, when it comes to binary addition, only the last state exerts influence on the current state. LSTM/GRU is too complex for this neural network. Actually, there is no obvious difference when looking at the loss curve of the three models. The accuracy is all 100%. Thus, the extra time expense seems unnecessary.

However, **if the hidden_size is very small, LSTM/GRU does increase accuracy** to some extent.

(input_size=32, seq_len=11,hidden_size=8,num_layers=2)

RNN,LSTM



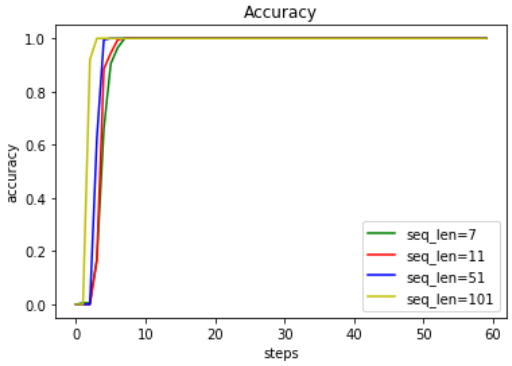
Take LSTM as an example, the loss curve tend to converge , and the accuracy rises from 10.6% to 88.85%. An explanation maybe the insufficient hidden neurons can't exactly preserve the information of the last state. **Using LSTM to select the most important information for the last state and make the best use of these hidden neurons enhance the model. When the hidden neurons are enough, LSTM loses its advantage over it.**

2.2.5 seq_len

(input_size=64,hidden_size=64,num_layers=2)

When seq_len>11, the numbers can't be generated directly , otherwise, it will trigger out-of-bounds problem. Thus, each bit of the number is randomly generated before they are combined together to form a number. Then, add each digit with a carry and get the sum.

seq_len=7,11,51,101



The accuracy for 4 models are all 100%. Adding seq_len can accelerate accuracy to reach 1. That means RNN can be applied to train binary addition for long numbers.

2.2.6 My advanced model

```
class myAdvPTRNNModel(nn.Module):
    def __init__(self,input_size=64,hidden_size=64,num_layers=2,num_classes=10):
        super().__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.num_classes = num_classes
        self.embed_layer = nn.Embedding(self.num_classes, self.input_size)
        self.rnn = nn.RNN(2*self.input_size, self.hidden_size, self.num_layers,
batch_first=True)
        self.dense = nn.Linear(self.hidden_size, self.num_classes)
        self.loss = []
        self.acc = []

    def forward(self, num1, num2):
        h_state = None
        logits1 = self.embed_layer(num1)
        logits2 = self.embed_layer(num2)
        input_logits = torch.cat([logits1,logits2],dim=2)
        out,h_n = self.rnn(input_logits)
        logits = self.dense(out)
        return logits
```

3. Train&Evaluate

3.1 Framework

There is a specific framework for pytorch to train models. First, loss function and optimizer should be selected.

- In this statement, optimizer gets all the parameters (tensors) of the model to be updated. The gradients are "stored" by the tensors themselves (they have a grad and a requires_grad attributes) . Meanwhile ,learning rate is set.

  `optimizer = torch.optim.Adam(params=model1.parameters(), lr=0.001)`

- In this statement, cross entropy loss is applied for the multi-classification problem.

  `losses = nn.CrossEntropyLoss()`

- When `loss.backward()` is called, all it does is compute gradient of the parameters in loss that have `requires_grad = True` and store them in `parameter.grad` attribute for every parameter. Before that, optimizer should be reset because the previous accumulated gradients is still recorded and will exert an effect.

  `optimizer.zero_grad()`

  `loss.backward()`

- In this statement, the optimizer iterates over all parameters (tensors) it is supposed to update and use their internally stored `grad` to update their values.

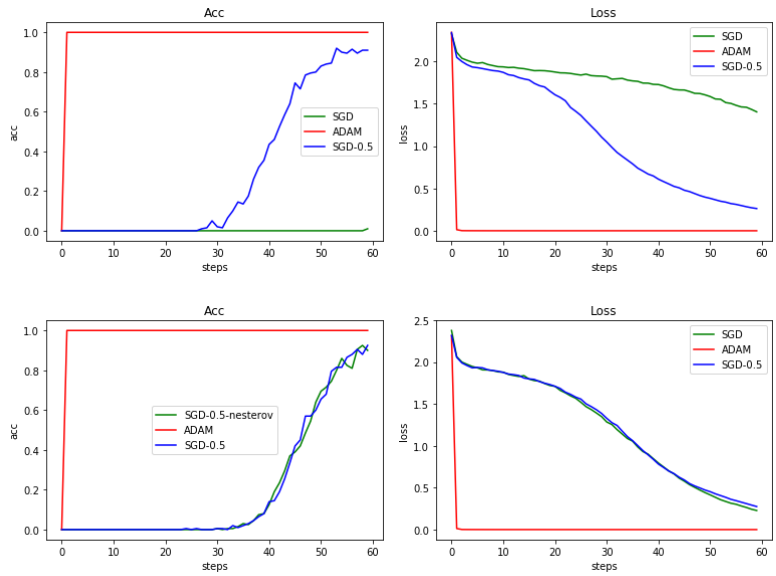
  `optimizer.step()`

3.2 Optimizer

(input_size=64,hidden_size=64,num_layers=2,seq_len=9)

(batch_size=200, epoch=3000)

3.2.1 Algorithm



algorithm	accuracy(test_set=2000)
SGD	0.05%
ADAM	100%
SGD+Momentum=0.5	89.35%
SGD+Momentum=0.5+Nesterov Momentum	90.65%

- SGD

  When training the model, dataset are divided into small batches and gradient update is based on these mini-batches.

- SGD+Momentum=0.5

  SGD doesn't work well because current update direction totally depends on current batch. In SGD, gradient is compared to speed(vector) while learning rate is compared to time. Momentum update means we don't only focus on the speed vector in current position, but also the speed(gradient) of the previous step. The sum of two vectors is the speed vector we actually want.

- SGD+Moment=0.5+Nesterov Momentum

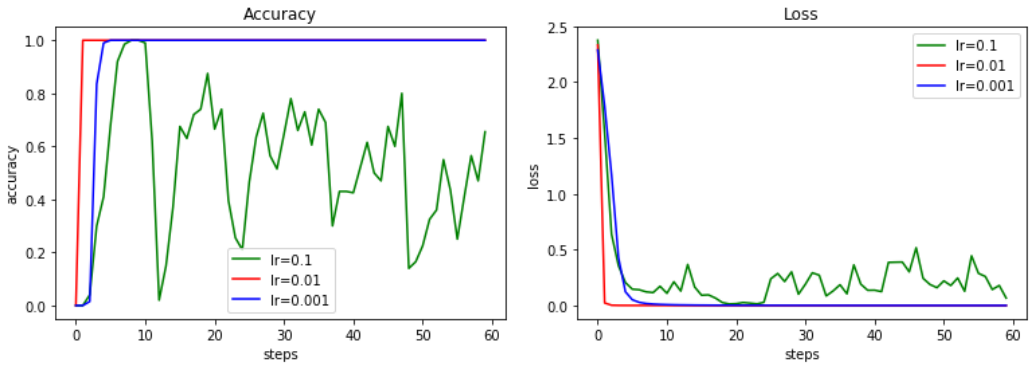
  The difference between Nesterov Momentum and original Momentum is that Nesterov Momentum look ahead gradient step, computing the gradient of Momentum step instead of the speed vector in current position. The former perform better.

- Adam

  This algorithm belongs to **Per-parameter adaptive learning rate methods,** which perform the best among the three due to its flexible learning rate. Our training dataset is sparse because each sample is generated randomly. Thus, the frequency for features differ, (for example, 0 appears most frequently because of padding). It is reasonable to tailor the learning rate to the features, with small learning rate given to frequent features and big learning rate given to infrequent features.

3.2.2 Learning rate

learning rate	accuracy(test_set=2000)
0.1	77.25%
0.01	100%
0.001	100%



Learning rate determines how fast or slow we will move towards the optimal weights. Too small learning rate will slower the loss function to converge, thus taking more epochs to train the network. Too big learning rate will render loss function to vibrate, unable to reach the optimum. In this case, lr=0.1 corresponds to the case where learning rate is too big, the loss function vibrate and the stability for the network is worse. lr=0.01 is an optimum value as loss function converges very quickly.

3.3 train-evaluate

The given `train()` and `evaluate()` generate batches using these statements, they **generate random numbers by range**:

In `train()`:

```
datas = gen_data_batch(batch_size=200, start=0, end=1e10)
Nums1, Nums2, results = prepare_batch(*datas, maxlen=11)
```

In `evaluate()`:

```
datas = gen_data_batch(batch_size=2000, start=0, end=1e10)
Nums1, Nums2, results = prepare_batch(*datas, maxlen=11)
```

`gen_data_batch` call np.random.randint to generate data from range[start,end), but when processing very long numbers, this function can't be called. Thus, we need to generate each bit and combine them to form a number.

In `gen_great_data_batch()`:

```
numbers_1 = [int("".join(np.random.randint(0,10,numdigit).astype('str'))) for
d_i in range(batch_size)]
numbers_2 = [int("".join(np.random.randint(0,10,numdigit).astype('str'))) for
d_i in range(batch_size)]
results = np.add(numbers_1,numbers_2,dtype='int64')
```

The given `my_train()` and `my_evaluate()` generate batches using these statements, they **generate random numbers by digit:**

In `my_train()`:

```
datas = gen_great_data_batch(batch_size, numdigit)
Nums1, Nums2, results = prepare_batch(*datas, maxlen=numdigit+2)
```

In `my_evaluate()`:

```
datas = gen_great_data_batch(batch_size,numdigit)
Nums1, Nums2, results = prepare_batch(*datas, maxlen=numdigit+2)
```

4. Summarization

Steps for this assignment:

1. Get familiar with pytorch.
2. Build a rnn network to realize binary addition.
3. Analyze how each parameter affects the model.
4. Optimize the model based on 3.
5. Test on numbers of different lengths of digits.

5. Demo

`python source.py`

6. Reference

[1] 邱锡鹏《神经网络与深度学习》https://nndl.github.io/

[2] K. Gnana Sheela and S. N. Deepa, Review on Methods to Fix Number of Hidden Neurons in Neural Networks,2013

[3]Alex Graves, Generating Sequences With Recurrent Neural Networks,2014

[4]https://pytorch.org/docs/