CS324 Assignment2 Report

SID：12011725

Name：彭英智

Part I: PyTorch MLP (30 points)

1. Method

1.1 Numpy MLP

In this assignment, I changed part of my preious numpy MLP code. The MLP model part remains the same, but the train() method is changed its training strategy to match my pytorch code. In assignment 1, the model testing is in training turns but now it moves after training every $eval_freq$ step of $epoch$.

1.2 Pytorch MLP

Aligning with the previous MLP work using Numpy, the new MLP using Pytorch has the same hyper-parameters, model parameter initialization, model architecture, traing and test dataset, and training strategy.

2. Preparation

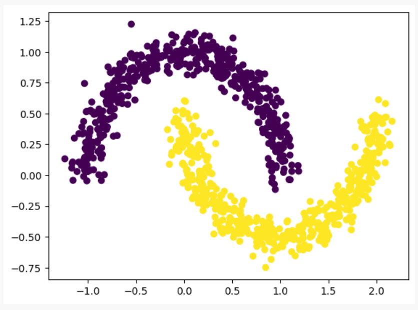
2.1 Dataset

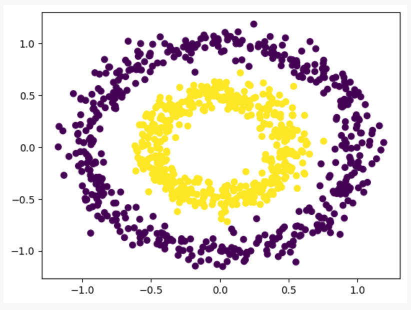
2.1.1 sklearn dataset

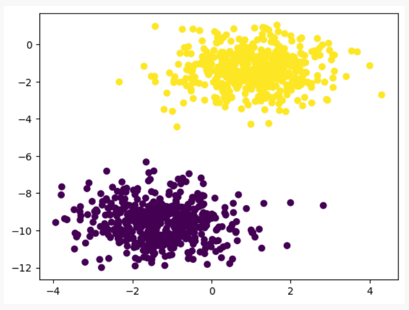
I use three particular dataset in $sklearn.datasets:make\_moons$, $make\_circlyes$, and $make\_blobs$, the dataset initialization parameter is shown below.

```
x, y = datasets.make_moons(n_samples=dataset_size, noise=0.08, shuffle=True,
random_state=1)
x, y = datasets.make_circles(n_samples=dataset_size, noise=0.08, factor=0.5,
shuffle=True, random_state=1)
x, y = datasets.make_blobs(n_samples=1000, n_features=2, centers=2, random_state=2)
```

All of the initialization of dataset have $random\_state$ to get exact same trainig and testing data. And the dataset virsualization is shown below.







To transfer sklearn to pytorch dataloader, I use $sklearn.model\_selection$ and $TensorDataset$ to correctly splict and load the sklearn dataset using pytorch.

2.1.2 CIFAR10 dataset

```
transform_train = transforms.Compose([transforms.ToTensor(),
     transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))])

transform_test = transforms.Compose([transforms.ToTensor(),
     transforms.Normalize((0.4941, 0.4850, 0.4502), (0.2467, 0.2429, 0.2616))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform_train)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
transform=transform_test)

# Batch-Size is 128
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE,
shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(testset, batch_size=BATCH_SIZE, shuffle=True,
num_workers=2)
```

2.2 Pytorch MLP Model Design

Initialization & Architecture

```
self.input_layers = nn.Sequential()
        self.hidden_layers = nn.Sequential()
        self.output_layers = nn.Sequential()
        for index in range(len(n_hidden)):
            if index == 0:
                self.input_layers.add_module("input_layer",
                                             nn.Linear(in_features=n_inputs,
out_features=int(n_hidden[index]),
                                                       bias=True))
                self.input_layers.add_module("input_relu",
                                             nn.ReLU())
            if index == len(n_hidden) - 1:
                self.output_layers.add_module("output_layer",

 nn.Linear(in_features=int(n_hidden[index]),out_features=n_classes,
                                                        bias=True))
            else:
                self.hidden_layers.add_module("hidden_layer" + str(index),

 nn.Linear(in_features=int(n_hidden[index]),
                                                        out_features=int(n_hidden[index
+ 1]), bias=True))
                self.hidden_layers.add_module("hidden_relu" + str(index),
                                              nn.ReLU())
```

Model Parameter Initialization

```
for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, mean=0, std=0.001)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')

            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
```

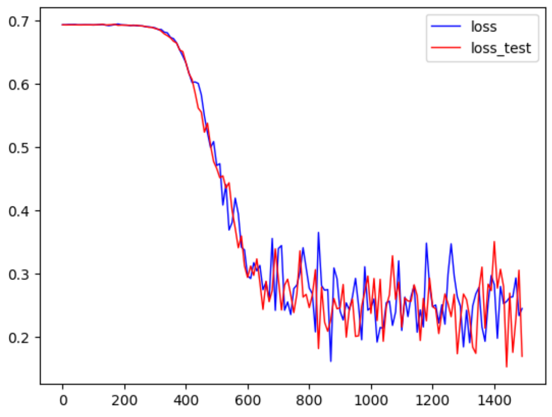
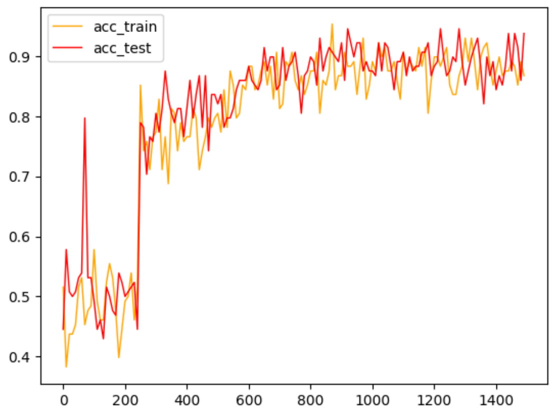
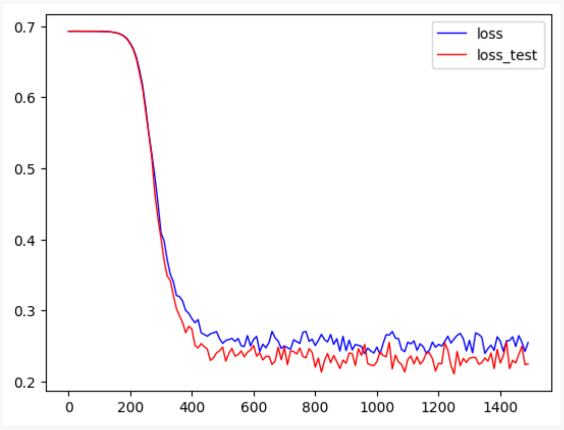
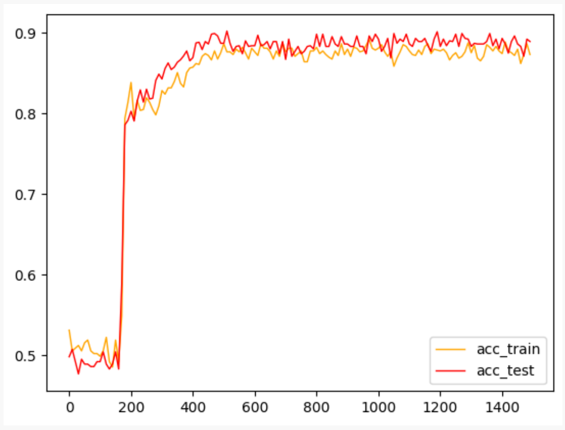
3. Training & Evaluation & Analysis

3.1 sklearn dataset (task 1 & 2)

The following below shows the training and testing result of MLP in Pytorch and Numpy.

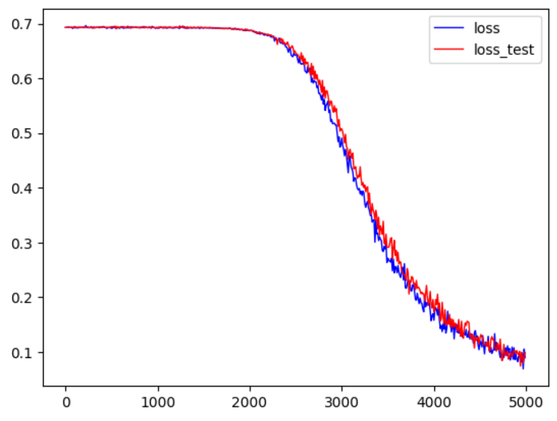
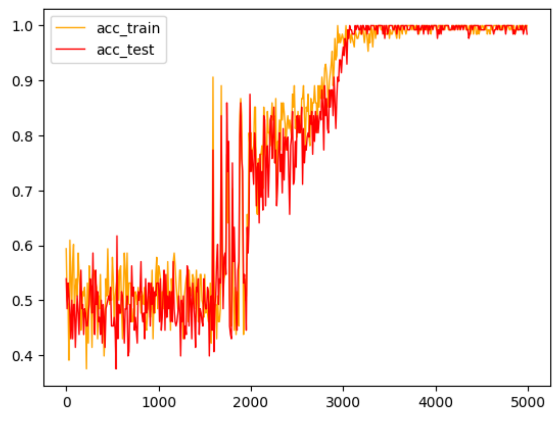
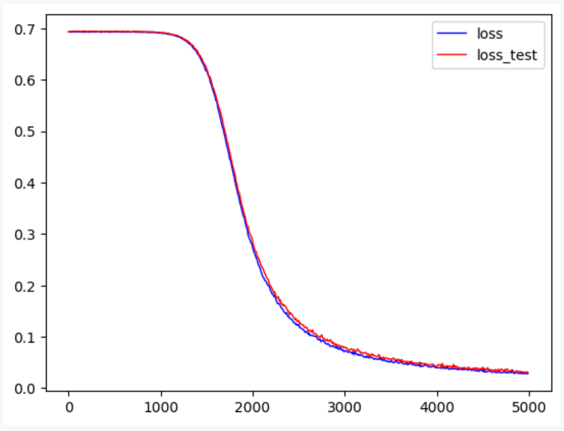
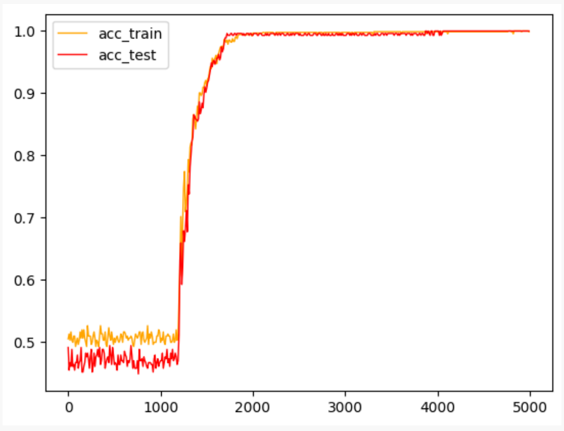
Make_Moon Dataset

```
N_INPUTS = 2
N_CLASSES = 2
HIDDEN_LAYER = '20'

LEARNING_RATE = 0.01
EPOCH = 1500
EVAL_FREQ = 10

dataset_size = 1000
TEST_SIZE = 0.2
BATCH_SIZE = 128
OPTIMIZER = 'SGD'
```

	Train & Test Loss	Train & Test Acc
Numpy		
Pytorch		

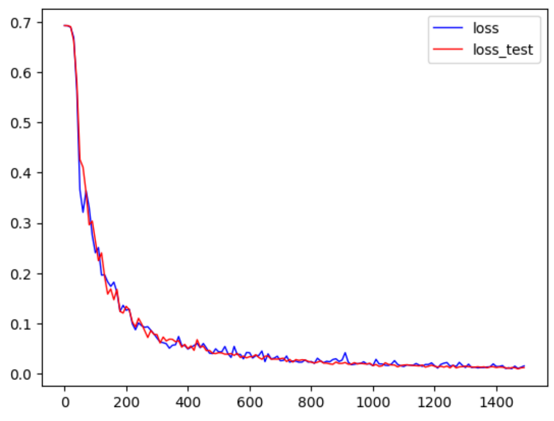
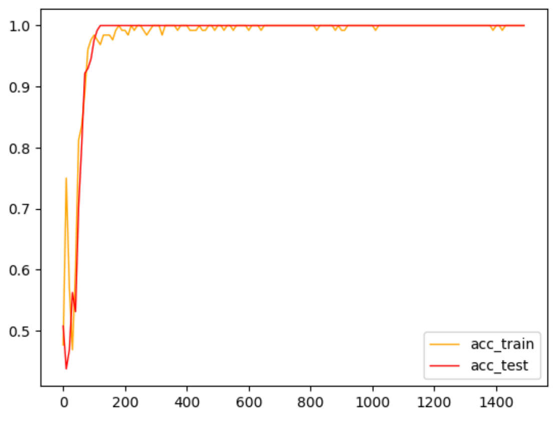
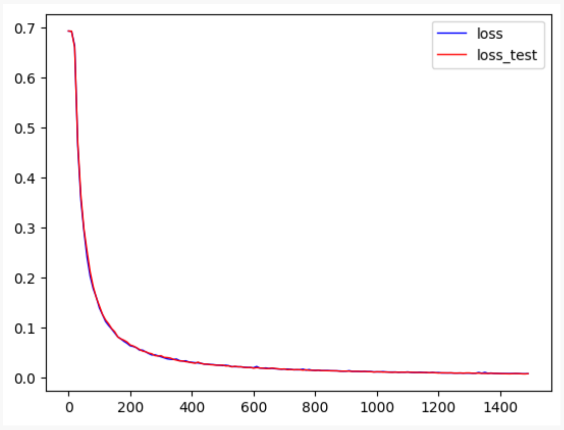
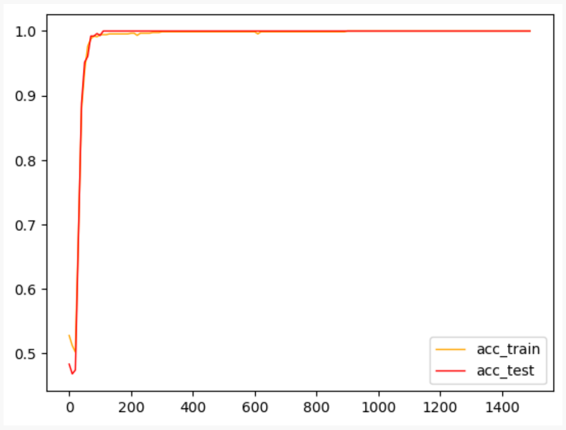
Make_Circle Dataset

```
N_INPUTS = 2
N_CLASSES = 2
HIDDEN_LAYER = '20'

LEARNING_RATE = 0.01
EPOCH = 5000
EVAL_FREQ = 10

dataset_size = 1000
TEST_SIZE = 0.2
BATCH_SIZE = 128
OPTIMIZER = 'SGD'
```

	Train & Test Loss	Train & Test Acc
Numpy		
Pytorch		

Make_Blob Dataset

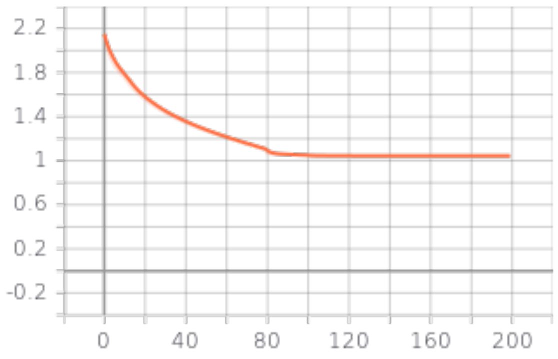
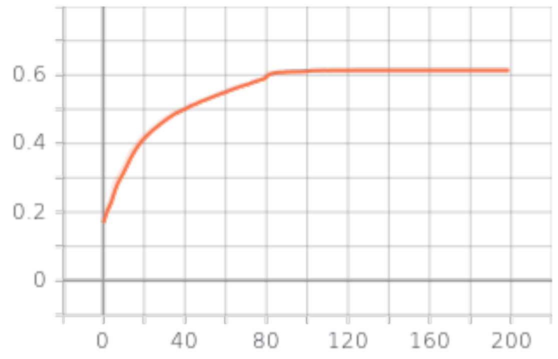
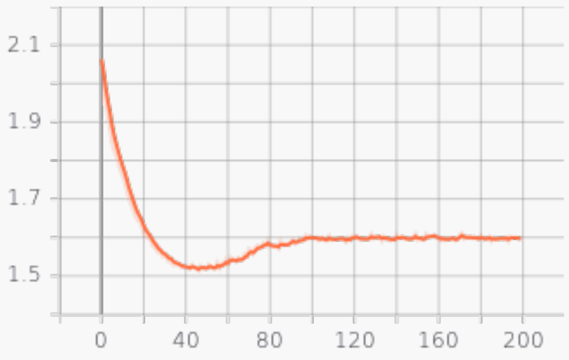
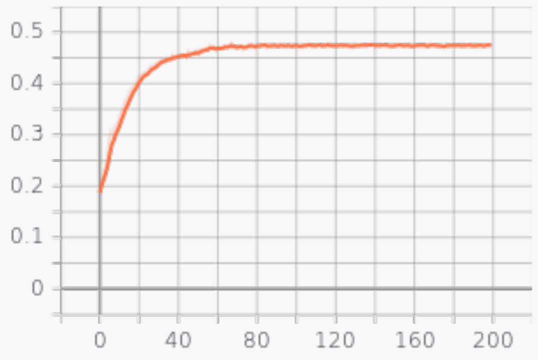
```
N_INPUTS = 2
N_CLASSES = 2
HIDDEN_LAYER = '20'

LEARNING_RATE = 0.01
EPOCH = 1500
EVAL_FREQ = 10

dataset_size = 1000
TEST_SIZE = 0.2
BATCH_SIZE = 128
OPTIMIZER = 'SGD'
```

	Train & Test Loss	Train & Test Acc
Numpy		
Pytorch		

3.2 CIFAR10 dataset (task 3)

Hyper-Parameter

```
N_INPUTS = 32 * 32 * 3
N_CLASSES = 10
HIDDEN_LAYER = '64, 64, 64'
LEARNING_RATE = 0.01
EPOCH = 1500
EVAL_FREQ = 10
dataset_size = 1000
TEST_SIZE = 0.2
BATCH_SIZE = 128
```

	Loss	Acc
Train		
Test		

4. Analysis & Difficulties

4.1 sklearn dataset (TASK 1 & 2)

In three dataset, with the same training and evaluation strategy, the training and testing curves of Numpy MLP and Pytorch MLP converge in the same epoch range, which certifies that the performance and architecture of two MLP is relatively the same as the architecture and hyper-para is designed to be same at beginning.

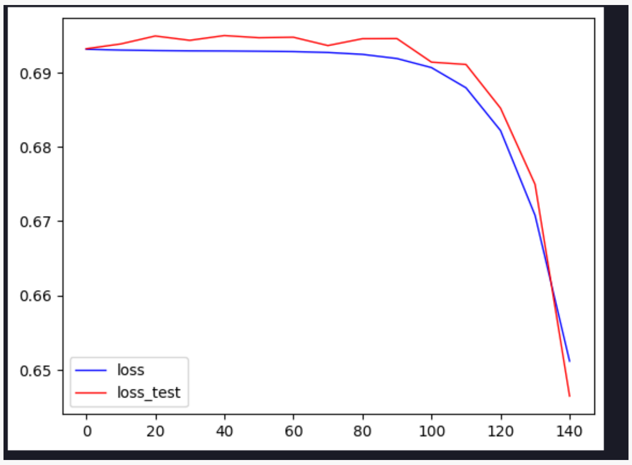
However, the curves of Numpy MLP is more jittery than the curves of Pytorch MLP. There are two possible reasons for this phenomenon.

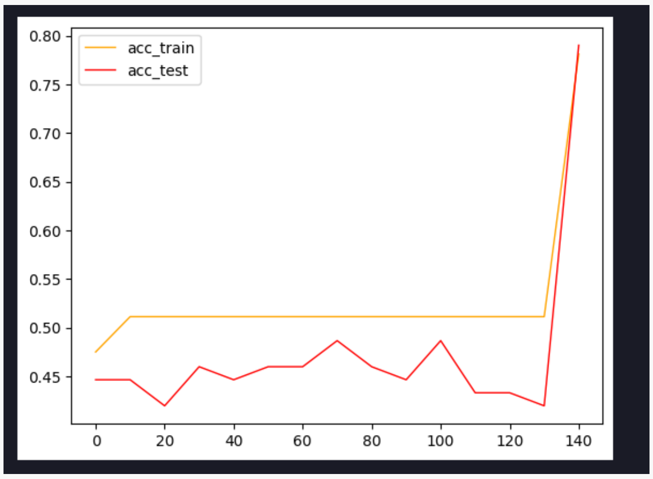
The first reason is that the dataloader of these two model is little bit different. The Numpy MLP does not use pytorch dataloader, in each epoch, I just randomly select $batch\_size$ number of data to train the model which may cause duplicated data usage in training. And Pytorch MLP uses pytorch dataloader which can avoid choosing duplicated data in dataset.

The second reason is that pytorch may have some methods to optimize the training performance, so the curves of Pytorch MLP seems more perfect that Numpy MLP. For instance, the SGD optimizer fuses **momentum gradient**, **weight decay** and **NAG** to improve the performance of training.

Choise of Batch_Size

It is worth noting that with the same hyper-para, the pytorch MLP trains much slower than Numpy MLP, about 10 to 50 times time cost in the same device. I tried to search the reason, and finialy find that the $batch\_size$ has a huge impact on the training time of pytorch MLP. In my experiment, the $batch\_size$ is 128 and I need 1500 $epochs$ to train my model in Numpy and Pytorch. However, when I reduce the $batch\_size$ to 32, the training time per $epoch$ improved so much and the model only need 150 $epcoch$ to be trained perfectly!





4.1 CIFAR10 dataset (TASK 3)

I tried my best to improve the performance on this dataset, but the result is still not good.

I use $torch.optim.lr\_scheduler.MultiStepLR$ to optimize the $learning\_rate$ of $optimizer$, and I use more $hidden\_layers$ and increase the training $epoch$ to help the MLP to learn more features.

```
N_INPUTS = 32 * 32 * 3
N_CLASSES = 10
HIDDEN_LAYER = '64, 64, 64'
LEARNING_RATE = 0.01
EPOCH = 1500
EVAL_FREQ = 10
dataset_size = 1000
TEST_SIZE = 0.2
BATCH_SIZE = 128
```

The reason why MLP can't fit for CIFAR10 dataset might be that the dimension of a image in this dataset is $32 * 32 * 3$ and it has many area features but MLP can't well extract this kindof features (unlike CNN). And the CIFAR10 contains 10 classes for MLP to classify, which also decrease the performance of MLP.

Part II: PyTorch CNN (30 points)

1. Method & Formulation

The overall architecture of the network is implemented according to slides.

$Input: (N, C_{in}, H_{in}, W_{in})$

$Output: (N, C_{out}, H_{out}, W_{out})$

$H_{out} = \lfloor \frac{H_{in}+2\times padding[0]-kernel[0]}{stride[0]} + 1 \rfloor$

$W_{out} = \lfloor \frac{H_{in}+2\times padding[1]-kernel[1]}{stride[1]} + 1 \rfloor$

According to the model structure, each layer’s parameters are computed as follows:

0. $Input: (Batch\_size, 3, 32, 32)$

1. ***conv layer***

   $Input:(k=3\times3,s=1,p=1,in=64,out=64)$

   $H_{out}=\lfloor\frac{32+2\times1-3}{2}+1\rfloor=16$

   $W_{out}=\lfloor\frac{32+2\times1-3}{2}+1\rfloor=16$

   $Output:(Batch\_Size,64,16,16)$

2. ***maxpool layer***

   $Input:(k=3\times3,s=1,p=1,in=3,out=64)$

   $H_{out}=\lfloor\frac{32+2\times1-3}{1}+1\rfloor=32$

   $W_{out}=\lfloor\frac{32+2\times1-3}{1}+1\rfloor=32$

   $Output:(Batch\_Size,64,32,32)$

3. ***conv layer***

   $Input:(k=3\times3,s=1,p=1,in=64,out=128)$

   $H_{out}=\lfloor\frac{16+2\times1-3}{1}+1\rfloor=16$

   $W_{out}=\lfloor\frac{16+2\times1-3}{1}+1\rfloor=16$

   $Output:(Batch\_Size,128,16,16)$

4. ***maxpool layer***

   $Input:(k=3\times3,s=2,p=1,in=128,out=128)$

   $H_{out}=\lfloor\frac{16+2\times1-3}{2}+1\rfloor=8$

   $W_{out}=\lfloor\frac{16+2\times1-3}{2}+1\rfloor=8$

   $Output:(Batch\_Size,128,8,8)$

5. ***conv layer***

   $Input:(k=3\times3,s=1,p=1,in=128,out=256)$

   $H_{out}=\lfloor\frac{8+2\times1-3}{1}+1\rfloor=8$

   $W_{out}=\lfloor\frac{8+2\times1-3}{1}+1\rfloor=8$

   $Output:(Batch\_Size,256,8,8)$

6. ***conv layer***

   $Input:(k=3\times3,s=1,p=1,in=256,out=256)$

   $H_{out}=\lfloor\frac{8+2\times1-3}{1}+1\rfloor=8$

   $W_{out}=\lfloor\frac{8+2\times1-3}{1}+1\rfloor=8$

   $Output:(Batch\_Size,256,8,8)$

7. ***maxpool layer***

   $Input:(k=3\times3,s=2,p=1,in=256,out=256)$

   $H_{out}=\lfloor\frac{8+2\times1-3}{2}+1\rfloor=4$

   $W_{out}=\lfloor\frac{8+2\times1-3}{2}+1\rfloor=4$

   $Output:(Batch\_Size,256,4,4)$

8. ***conv layer***

   $Input : (k = 3 \times 3, s = 1, p = 1, in = 256, out = 512)$

   $H_{out} = \lfloor \frac{4+2\times1-3}{1} + 1 \rfloor = 4$

   $W_{out} = \lfloor \frac{4+2\times1-3}{1} + 1 \rfloor = 4$

   $Output : (Batch\_Size, 512, 4, 4)$

9. ***conv layer***

   $Input : (k = 3 \times 3, s = 1, p = 1, in = 256, out = 256)$

   $H_{out} = \lfloor \frac{4+2\times1-3}{1} + 1 \rfloor = 4$

   $W_{out} = \lfloor \frac{4+2\times1-3}{1} + 1 \rfloor = 4$

   $Output : (Batch\_Size, 512, 4, 4)$

10. ***maxpool layer***

    $Input : (k = 3 \times 3, s = 2, p = 1, in = 512, out = 512)$

    $H_{out} = \lfloor \frac{4+2\times1-3}{2} + 1 \rfloor = 2$

    $W_{out} = \lfloor \frac{4+2\times1-3}{2} + 1 \rfloor = 2$

    $Output : (Batch\_Size, 512, 2, 2)$

11. ***conv layer***

    $Input : (k = 3 \times 3, s = 1, p = 1, in = 512, out = 512)$

    $H_{out} = \lfloor \frac{2+2\times1-3}{1} + 1 \rfloor = 2$

    $W_{out} = \lfloor \frac{2+2\times1-3}{1} + 1 \rfloor = 2$

    $Output : (Batch\_Size, 512, 2, 2)$

12. ***conv layer***

    $Input : (k = 3 \times 3, s = 1, p = 1, in = 512, out = 512)$

    $H_{out} = \lfloor \frac{2+2\times1-3}{1} + 1 \rfloor = 2$

    $W_{out} = \lfloor \frac{2+2\times1-3}{1} + 1 \rfloor = 2$

    $Output : (Batch\_Size, 512, 2, 2)$

13. ***maxpool layer***

    $Input : (k = 3 \times 3, s = 2, p = 1, in = 512, out = 512)$

    $H_{out} = \lfloor \frac{2+2\times1-3}{2} + 1 \rfloor = 1$

    $W_{out} = \lfloor \frac{2+2\times1-3}{2} + 1 \rfloor = 1$

    $Output : (Batch\_Size, 512, 1, 1)$

14. ***linear layer*** $(512, 10)$

    $Output : (Batch\_Size, 10)$

2. Preparation

2.1 Dataset

Use CIFAR10 dataset to train and test the model.

```
transform_train = transforms.Compose([transforms.ToTensor(),
     transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2470, 0.2435, 0.2616))])

transform_test = transforms.Compose([transforms.ToTensor(),
     transforms.Normalize((0.4941, 0.4850, 0.4502), (0.2467, 0.2429, 0.2616))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
transform=transform_train)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
transform=transform_test)

# Batch-Size is 128
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE,
shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(testset, batch_size=BATCH_SIZE, shuffle=True,
num_workers=2)
```

2.2 Optimizer

Use three optimizer to train the model.

```
if optimizer == 'SGD':
      optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
elif optimizer == 'ADAM':
      optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
elif optimizer == 'RMSprop':
      optimizer = torch.optim.RMSprop(model.parameters(), lr=learning_rate)
```

2.3 Visualization Tools

Use $torch.utils.tensorboard$ to visualize the **real-time** train and test result.

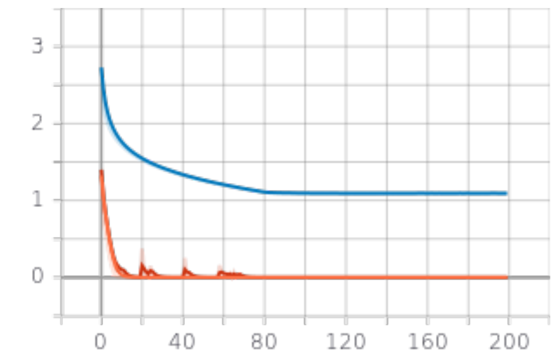
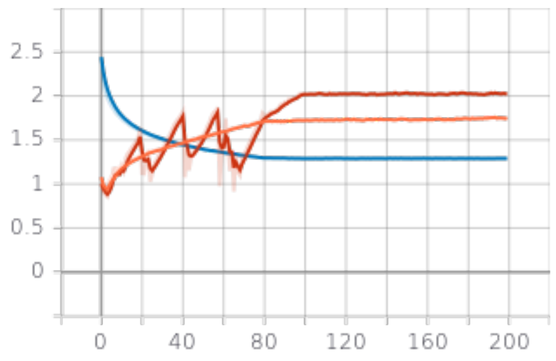
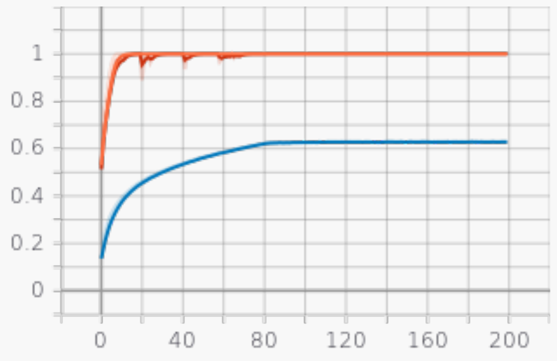
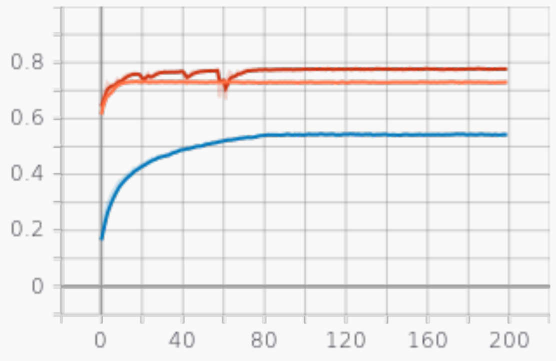
2.5 Model Parameter Initialization

```
for m in self.modules():
  if isinstance(m, nn.Linear):
    nn.init.xavier_normal_(m.weight)
    nn.init.constant_(m.bias, 0)
  elif isinstance(m, nn.Conv2d):
    nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
  elif isinstance(m, nn.BatchNorm2d):
    nn.init.constant_(m.weight, 1)
    nn.init.constant_(m.bias, 0)
```

3. Training & Evaluation

3.1 Optimizer Comparation

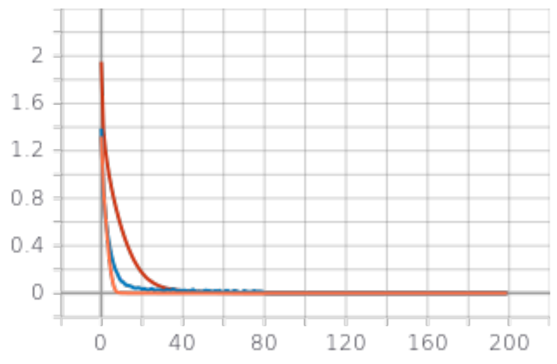
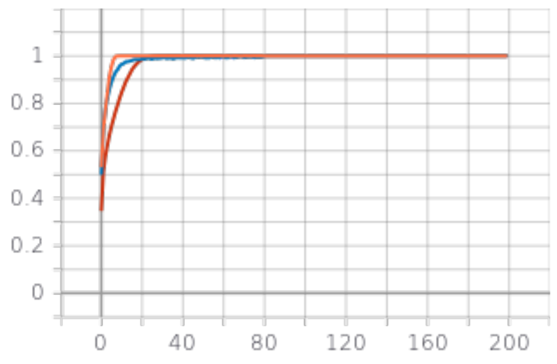
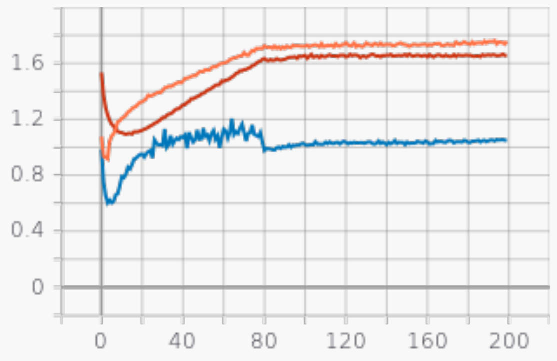
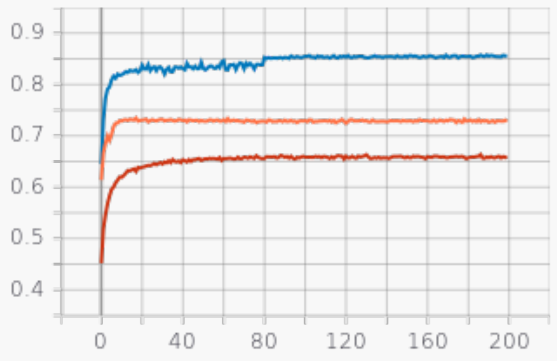
```
BATCH_SIZE = 512
MAX_EPOCHS = 200
EVAL_FREQ = 500
```

	Loss	Acc
Train		
Test		

✓ ○ ADAM/lr=0.0001
✓ ○ SGD/lr=0.0001
✓ ○ RMSprop/lr=0.0001

3.2 Learning Rate Comparation

```
BATCH_SIZE = 512
MAX_EPOCHS = 200
EVAL_FREQ = 500
```

	Loss	Acc
Train		
Test		

✓ ○ lr=0.0001

✓ ○ lr=0.001

✓ ○ lr=1e−05

4. Analysis & Difficulties

4.1 Optimizers

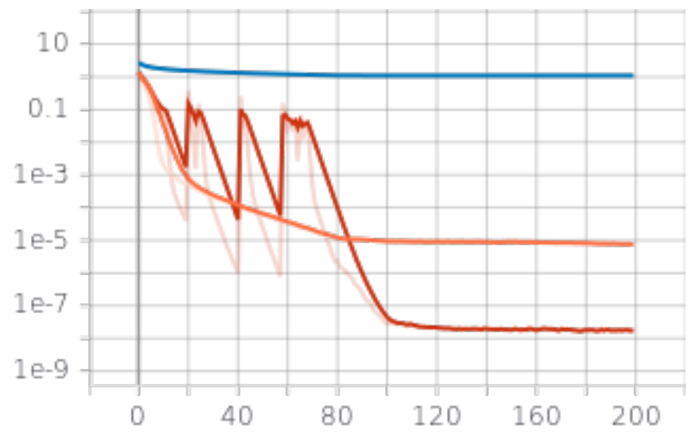
Above table shows curves in training and test of three different optimizer using the same training strategy and hyper-parameters.

It is shown that SGD has the worst performance, while $ADAM$ and $RMSprop$ have almost the same performance in training, and $RMSprop$ has a higher performance than $ADAM$ in **test accuracy**.

However, since the $ADAM$ conbines $RMSprop$ and **momentum gradient** together to improve the gradient decent ability, it should have higher performance than $RMSprop$. And I haven't find the reason yet.

But if we focus on the **training acc curves** of $ADAM$ and $RMSprop$, $RMSprop$ is more unstable than $ADAM$, which may because based on $RMSprop$, $ADAM$ has **momentum gradient** to make the process of gradiant decnet more smoothly.

And the picture below shows the **training loss cures in toggle of y axis**



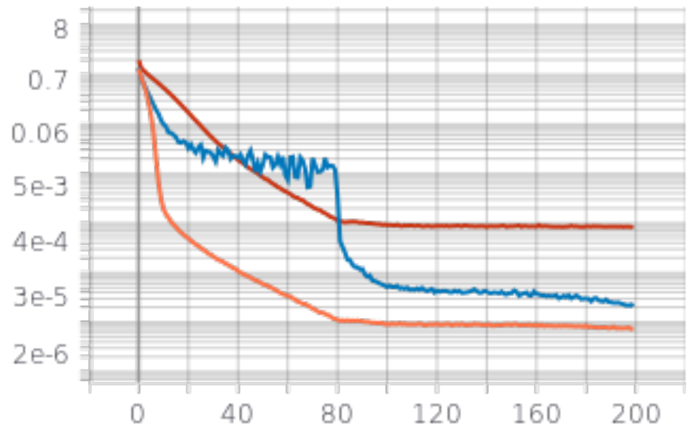
The rad curves is $RMSprop$, the orange one is $ADAM$, which can also show that the **momentum gradient** of $ADAM$ helps the model training in gradient decent.

4.2 Learning Rate

The curves of different $learn\_rate$ shows that $learn\_rate$ at 0.001 has the best performance, and if we increase or decrease the $learn\_rate$ too much (10 times in my experiment), the performance is much worse.

So it is very important to choose a proper $learn\_rate$.

And the picture below shows the **training loss cures in toggle of y axis**



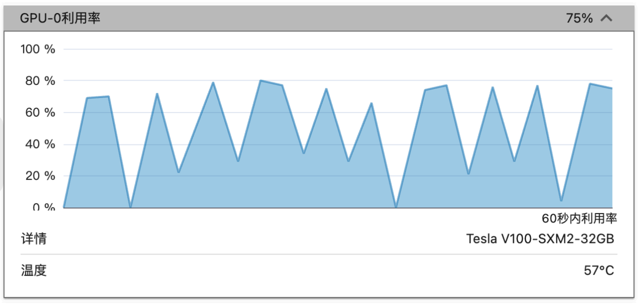
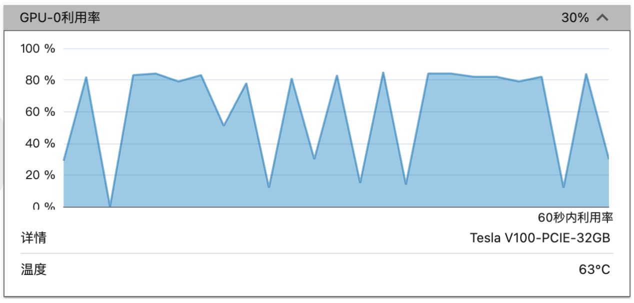
The loss of blue curves (lr=0.01) has a rapid decrease after 80 epoch, that is beacause I use $lr\_scheduler.MultiStepLR$ to change the $learning\_rate$ in 80 epoch and 120 epoch. I also apply the $schedule$ in another two model, but the curevs shows that it is not working well. Also, the decrease of learning rate at 120 epoch can't improve the loss decrease.

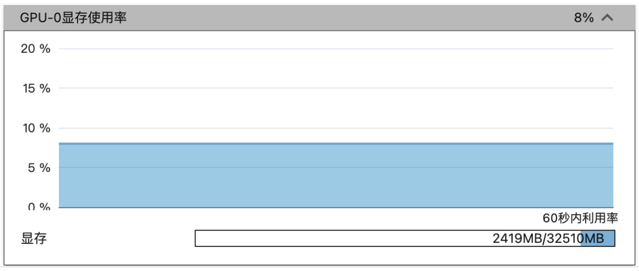
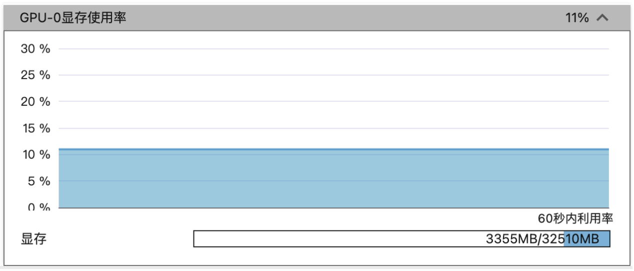
It shows that a good training strategy may be that have a relatively higher learning rate in early training epoch, and decrease the learning rate to better reach the **global** minimum of loss.

4.3 Choice of BatchSize

Extending to the analysis of $Batch\_Size$ in PART 1. This turn I made a further research on $Batch\_Size$

The trainning device is **8 vCPU + 64 GiB + 1 x Tesla V100-PCIE-32GB**

Batch_Size = 256	Batch_Size = 512
 GPU-0利用率 75% 详情 Tesla V100-SXM2-32GB 温度 57°C	 GPU-0利用率 30% 详情 Tesla V100-PCIE-32GB 温度 63°C

Batch_Size = 256	Batch_Size = 512
 GPU-0显存使用率 8% 显存 2419MB/32510MB	 GPU-0显存使用率 11% 显存 3355MB/32510MB

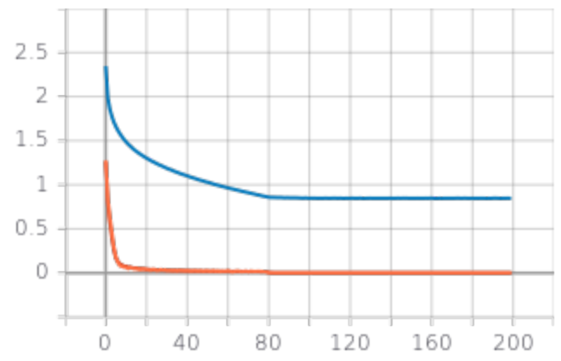
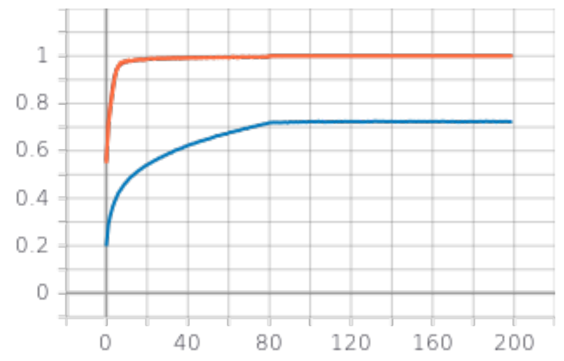
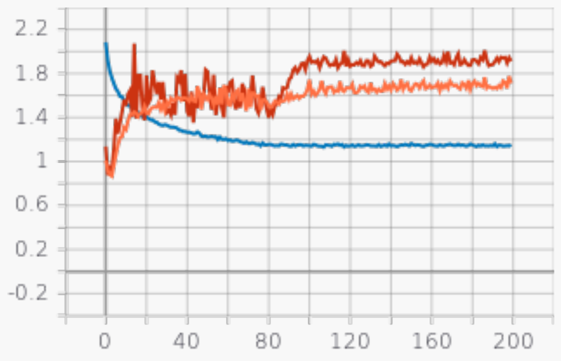
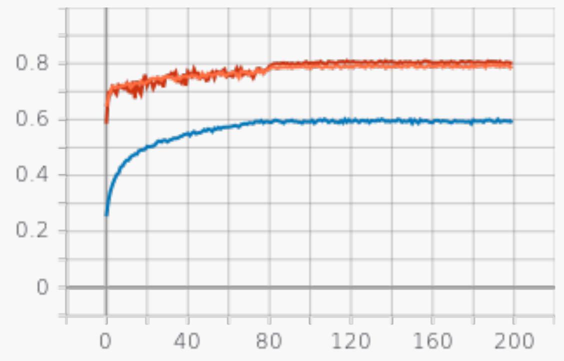
Batch_Size = 256	Batch_Size = 512

`96%|██████████| 192/200 [23:31<00:58,  7.26s/it]`
`epoch is 192, test_loss is 1.673699, test_acc is 0.790430`
`96%|██████████| 193/200 [23:38<00:51,  7.29s/it]`
`epoch is 193, test_loss is 1.663173, test_acc is 0.794141`
	`94%|██████████| 189/200 [18:50<01:02,  5.67s/it]`
	`epoch is 189, test_loss is 1.757815, test_acc is 0.728504`
	`95%|██████████| 190/200 [18:55<00:56,  5.68s/it]`
	`epoch is 190, test_loss is 1.744961, test_acc is 0.729751`

It shows that when batch_size decrease the training time for one epoch is much lower and the GPU memory usage is lower, but GPU usage remains the same.

And the GPU memory usage is in linear ralationship but trainning time is not linear.

And below is the **batch_size = 256** model training and evaluation of **three optimizers**.

	Loss	**Train**
Train		
Test		

Comparing with the Batch_Size=512, the Test Loss and Acc Curves are more unstable. Which may because the batch is small so the direction of fradient decent is not as stable as Batch_Size=512.

Part III: PyTorch RNN (40 points)

1. Method

Briefly speaking, a RNN cell is just adding a hidden weight transfer between RNN cell from normal MLP cell.

The formulation of RNN is shown below.

$$h^{(t)} = tanh(W_{hx}x^{(t)} + W_{hx}x^{(t-1)} + b_h) \tag{1}$$

$$o^{(t)} = (W_{ph}h^{(t)} + b_o) \tag{2}$$

$$\tilde{y}^{(t)} = softmax(o^{(t)}) \tag{3}$$

$$Loss = -\sum_{k=1}^{K} y_k log(\tilde{y}_k^{(T)}) \tag{4}$$

Where $h^{(t)}$ is hidden_layer of time t, o^t is the out put of time t, $\tilde{y}^{(t)}$ is the classify property, $Loss$ is calculated using cross entropy.

2. Preparation

2.1 Dataset

Using PalindromeDataset.

2.2 Model Design

According to the slide and formulation, the model code design is shown blow.

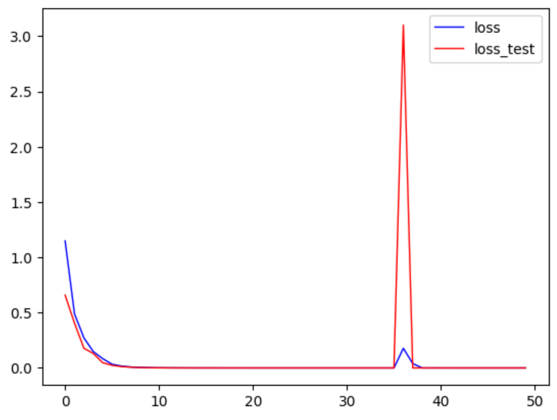
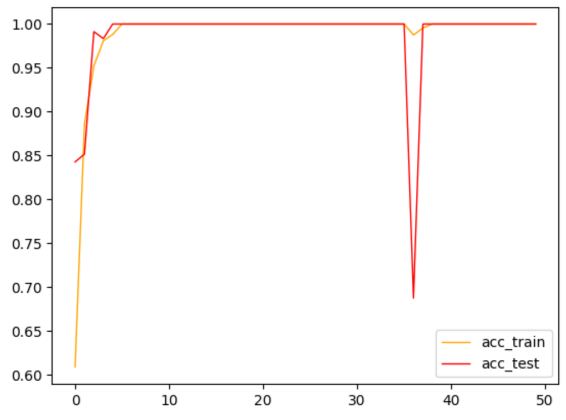
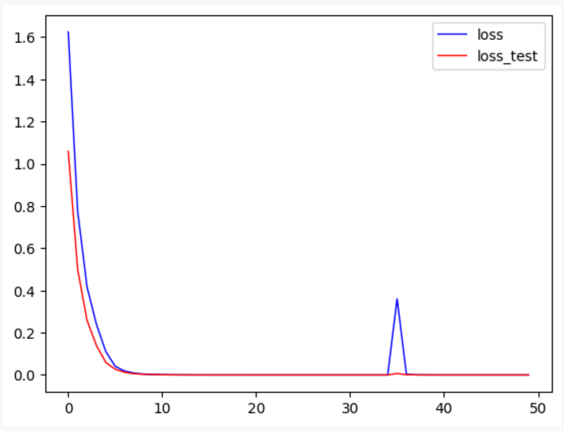
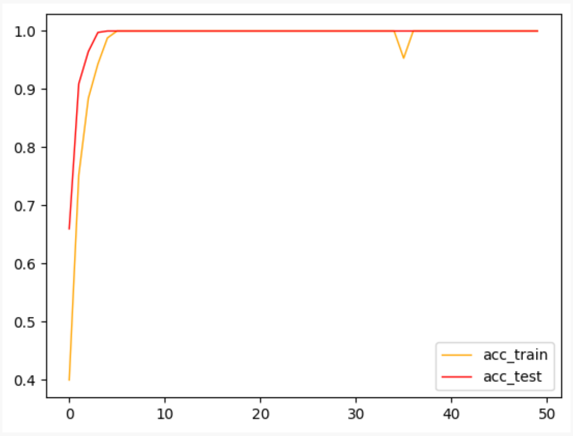
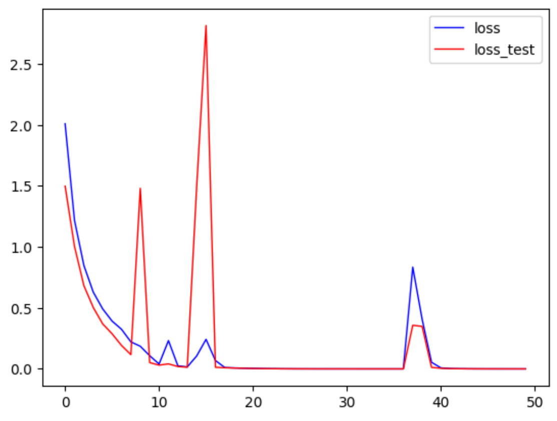
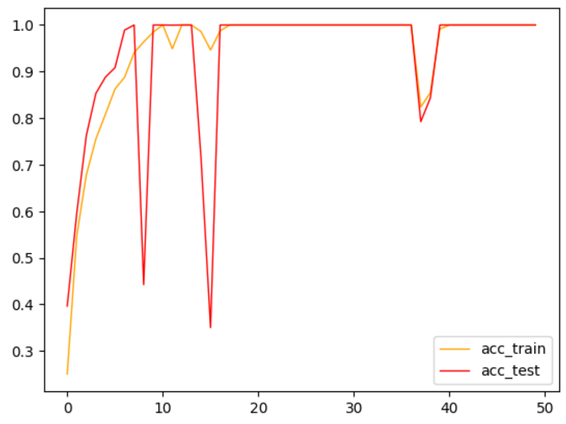
```python
# initialization
def __init__(self, seq_length, input_dim, hidden_dim, output_dim, batch_size):
        super(VanillaRNN, self).__init__()
        # Initialization here ...
        self.seq_length = seq_length
        self.batch_size = batch_size
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.w_x2h = nn.Linear(in_features=input_dim, out_features=hidden_dim,
bias=True)
        self.w_h2h = nn.Linear(in_features=hidden_dim, out_features=hidden_dim,
bias=False)
        self.w_h2o = nn.Linear(in_features=hidden_dim, out_features=output_dim,
bias=True)
        self.tanh = nn.Tanh()
```

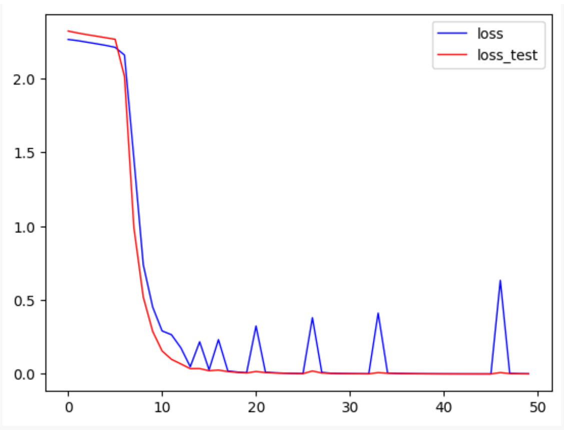
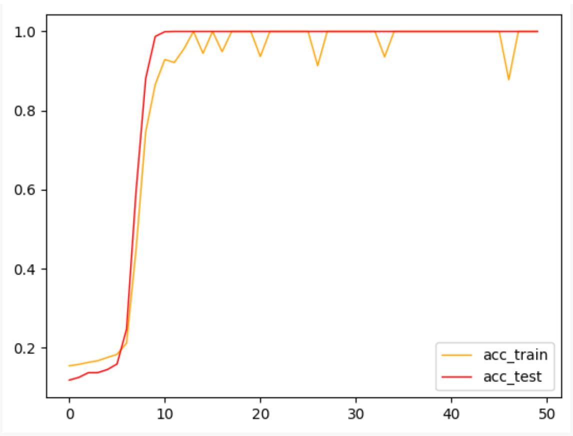
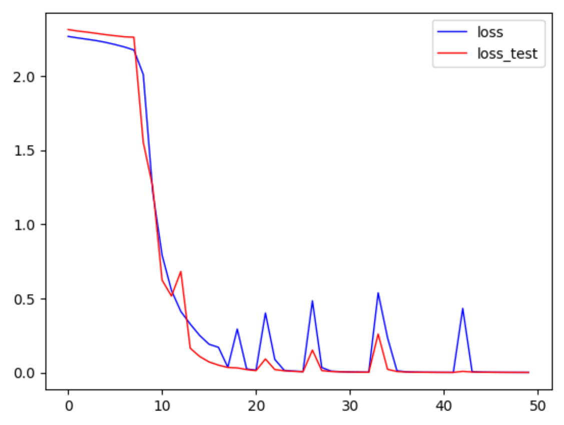
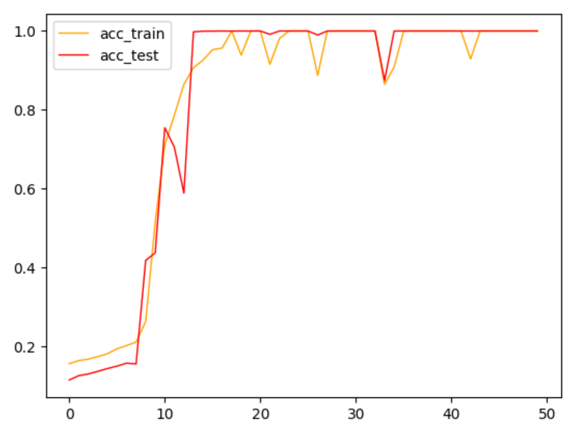
```python
# forward
def forward(self, inputs, state=None):
        # Implementation here ...
        inputs = torch.t(inputs)
        if state is None:
            state = torch.zeros([self.batch_size,
self.hidden_dim],device=inputs.device)
        else:
            state = state
        outputs = []
        for x in inputs:
            x = torch.unsqueeze(x, dim=1)
            state = self.tanh(self.w_x2h(x) + self.w_h2h(state))
        out = self.w_h2o(state)
        return out
```

The dimention of input data is $(Batch\_Size, set\_length)$, and we need to add a dimention to match the RNN input dimension $(Batch\_Size, set\_length, 1)$, output dimention of output data is $(Batch\_Size, 1)$

3. Training & Evaluation

```
input_length = 10
input_dim = 1
num_classes = 10
num_hidden = 128
batch_size = 128
learning_rate = 0.001
train_steps = 100
max_norm = 10.0
max_epochs = 50
eval_freq=1
```

	Loss	Acc
Length = 3		
Length = 5		
Length = 7		

Length = 10		
Length = 15		

4. Analysis & Difficulties

The table above shows that when the length=5, RNN has the best performance, while the length increases or decreases, the performance and the training curves get worse. This phenomena matches the character of RNN that RNNs have a limited memory. Too short length might cause RNN can't fully learning the sequence, and too long length may cause RNN lose the memory of previous features.