CS324 Assignment 3 Report

SID：12011725

Name：彭英智

Part I: PyTorch LSTM (40 points)

1. Method & Preparation

1.1 LSTM Model

The overall architecture of the network is implemented according to slides. The formulation of LSTM is shown below.

$$
\begin{aligned}
h^{(t)} &= \tanh\left(W_{hx}x^{(t)} + W_{hx}x^{(t-1)} + b_h\right) \\
o^{(t)} &= \left(W_{ph}h^{(t)} + b_o\right) \\
\tilde{y}^{(t)} &= \operatorname{softmax}\left(o^{(t)}\right) \\
\text{Loss } &= -\sum_{k=1}^{K} y_k \log\left(\tilde{y}_k^{(T)}\right) \\
g^{(t)} &= \tanh\left(W_{gx}x^{(t)} + W_{gh}h^{(t-1)} + b_g\right) \\
i^{(t)} &= \sigma\left(W_{ix}x^{(t)} + W_{ih}h^{(t-1)} + b_i\right) \\
f^{(t)} &= \sigma\left(W_{fx}x^{(t)} + W_{fh}h^{(t-1)} + b_f\right) \\
o^{(t)} &= \sigma\left(W_{ox}x^{(t)} + W_{oh}h^{(t-1)} + b_o\right) \\
c^{(t)} &= g^{(t)} \odot i^{(t)} + c^{(t-1)} \odot f^{(t)} \\
h^{(t)} &= \tanh\left(c^{(t)}\right) \odot o^{(t)} \\
p^{(t)} &= \left(W_{ph}h^{(t)} + b_p\right) \\
\tilde{y}^{(t)} &= \operatorname{softmax}\left(p^{(t)}\right),
\end{aligned}
\tag{1}
$$

```
class LSTM(nn.Module):
    def __init__(self, seq_length, input_dim, hidden_dim, output_dim, batch_size):
        super(LSTM, self).__init__()
        # Initialization here ...
        self.batch_size = batch_size
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.hidden_dim = hidden_dim
        self.layer_num = seq_length
```

```
        self.Wgx = nn.Linear(self.input_dim, self.hidden_dim, bias=True)
        self.Wgh = nn.Linear(self.hidden_dim, self.hidden_dim, bias=False)
        self.Wix = nn.Linear(self.input_dim, self.hidden_dim, bias=True)
        self.Wih = nn.Linear(self.hidden_dim, self.hidden_dim, bias=False)
        self.Wfx = nn.Linear(self.input_dim, self.hidden_dim, bias=True)
        self.Wfh = nn.Linear(self.hidden_dim, self.hidden_dim, bias=False)
        self.Wox = nn.Linear(self.input_dim, self.hidden_dim, bias=True)
        self.Woh = nn.Linear(self.hidden_dim, self.hidden_dim, bias=False)
        self.Wp = nn.Linear(self.hidden_dim, self.output_dim, bias=True)

    def forward(self, inputs):
        # Implementation here ...
        ht = torch.zeros([self.batch_size, self.hidden_dim], device= 'cuda:0')
        ct = torch.zeros([self.batch_size, self.hidden_dim], device= 'cuda:0')
        inputs = torch.t(inputs)
        for x in inputs:
            x = torch.unsqueeze(x, dim=1)
            gt = torch.tanh(self.Wgx(x) + self.Wgh(ht))
            it = torch.sigmoid(self.Wix(x) + self.Wih(ht))
            ft = torch.sigmoid(self.Wfx(x) + self.Wfh(ht))
            ot = torch.sigmoid(self.Wox(x) + self.Woh(ht))
            ct = gt * it + ct * ft
            ht = torch.tanh(ct) * ot
        out = self.Wp(ht)
        return out
```

1.2 RNN Model

I use the RNN model implemented in Assignment 2. The formulation of RNN is shown below.

$$\begin{aligned} h^{(t)} &= \tanh\left(W_{hx}x^{(t)} + W_{hx}x^{(t-1)} + b_h\right) \\ o^{(t)} &= \left(W_{ph}h^{(t)} + b_o\right) \\ \tilde{y}^{(t)} &= \operatorname{softmax}\left(o^{(t)}\right) \\ \text{Loss} &= -\sum_{k=1}^{K} y_k \log\left(\tilde{y}_k^{(T)}\right) \end{aligned} \tag{2}$$

1.3 Dataset

Using PalindromeDataset.

1.4 Hyper-parameter & Environment

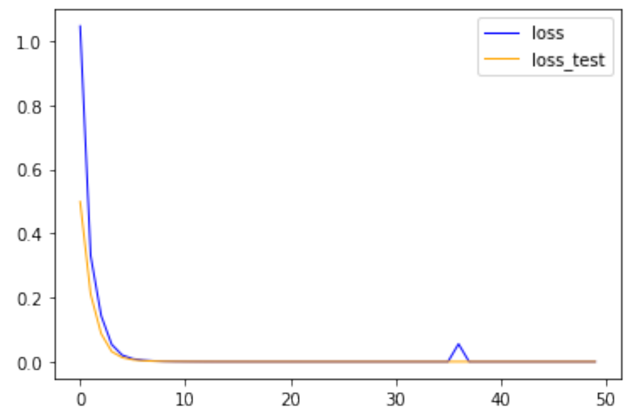
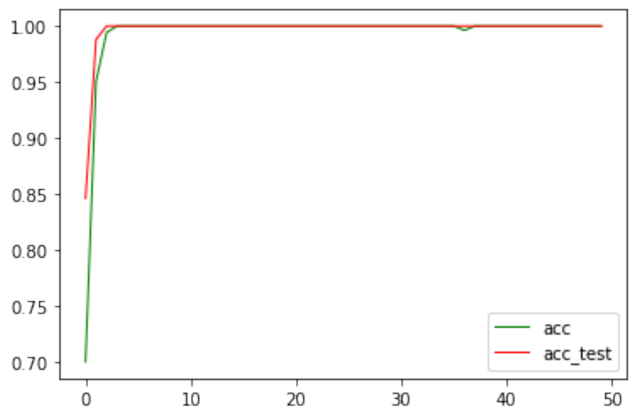
```
# hyper-paremeter of LSTM and RNN
input_length = 3,5,7,10,15
input_dim = 1
num_classes = 10
num_hidden = 128
batch_size = 128
learning_rate = 0.001
train_steps = 100
eval_freq = 1
max_epochs = 50
max_norm = 10.0
seed = 0
```

```
# Training & Testing Environment
CPU AMD EPYC 7451
GPU RTX3090 24G

conda 23.5.0
python 3.9.7
torch 2.0.1
CUDA: 11.6
```

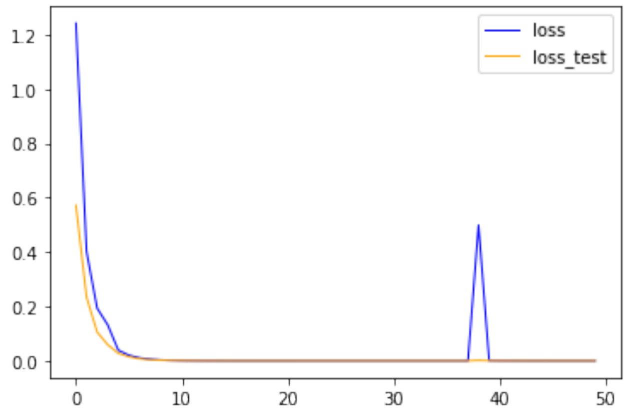
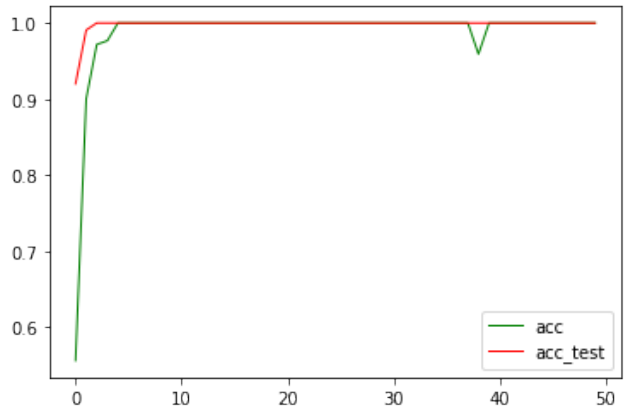
2. Experimental Results

2.1 LSTM in different length T

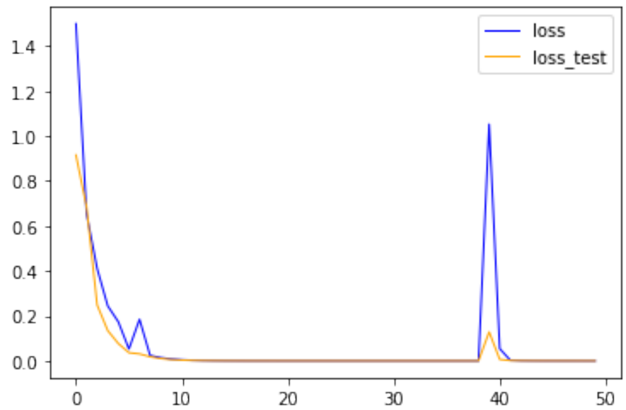
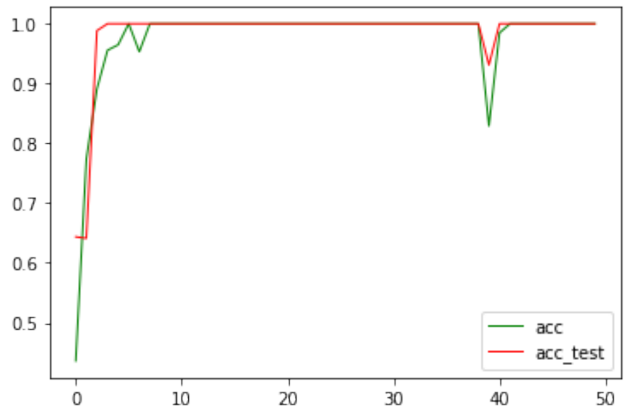
T = 3

loss	acc
	

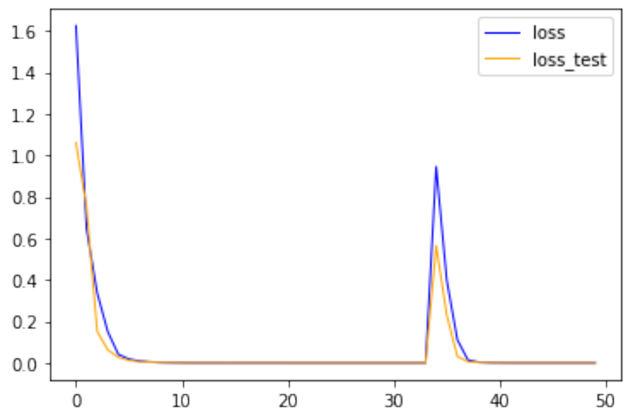
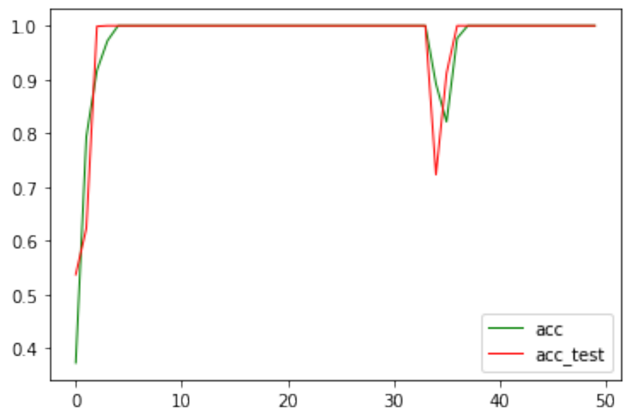
T = 5

loss	acc
	

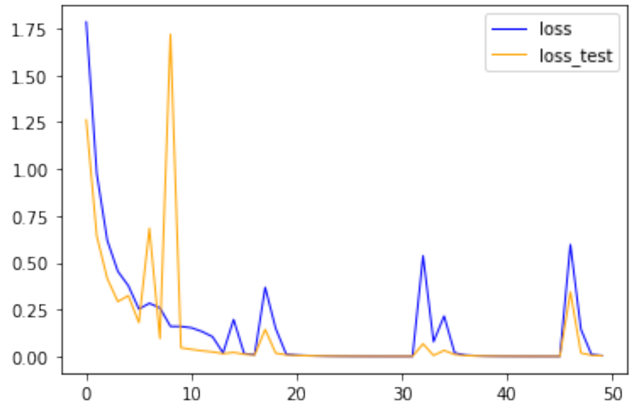
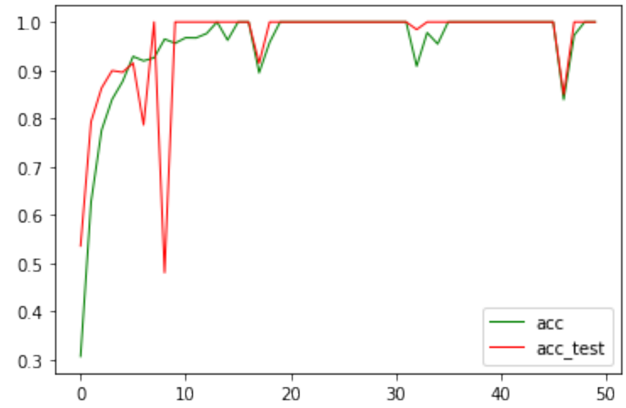
T = 7

loss	acc
	

T = 10

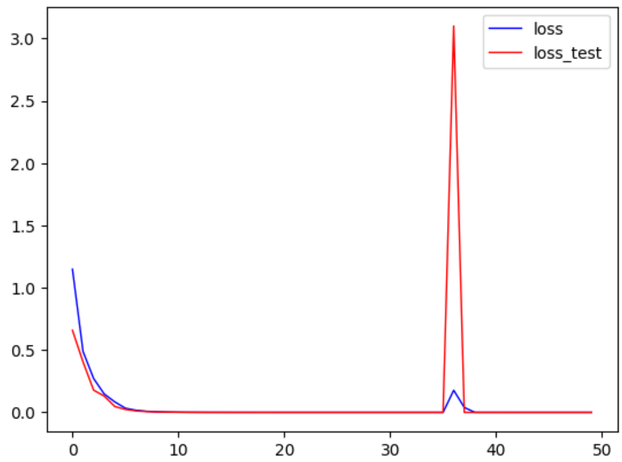
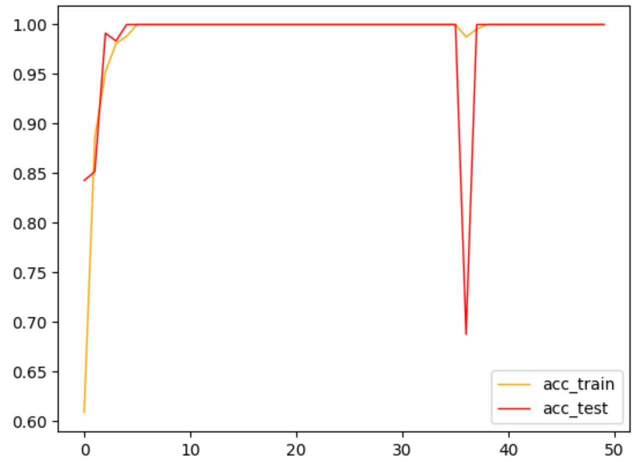
loss	acc
	

T = 15

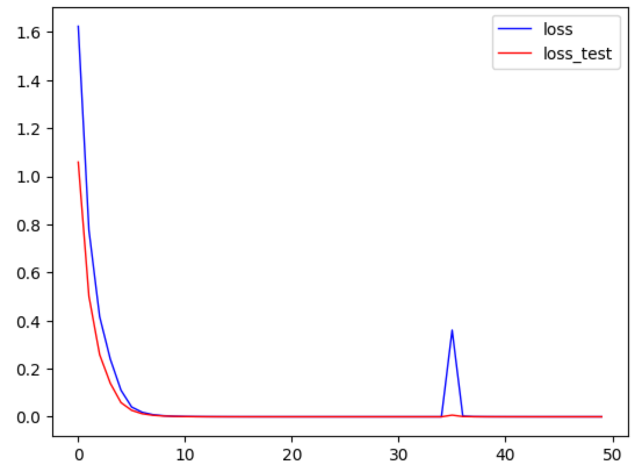
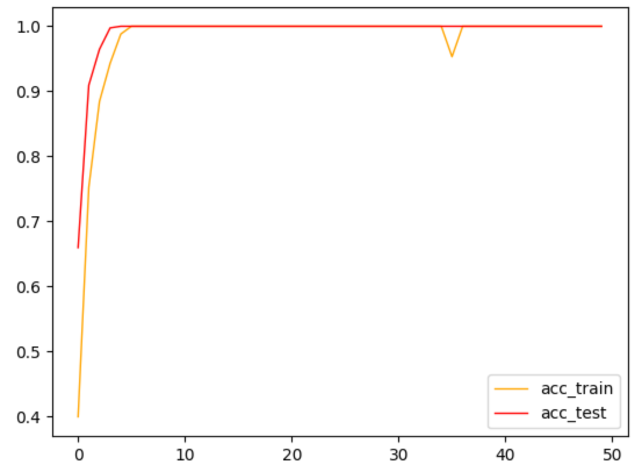
<table>
<tr><th>loss</th><th>acc</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

2.2 RNN in different length T

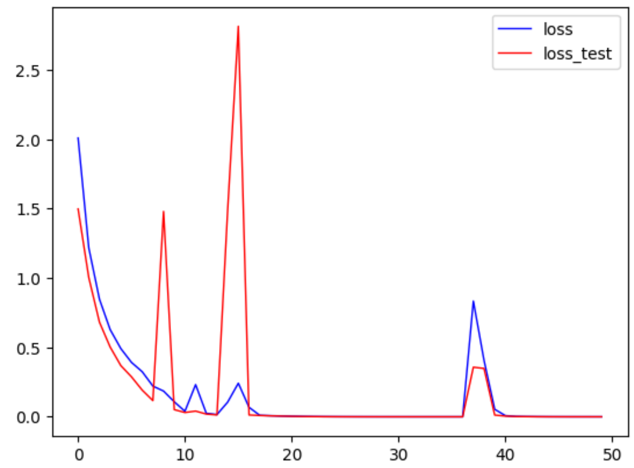
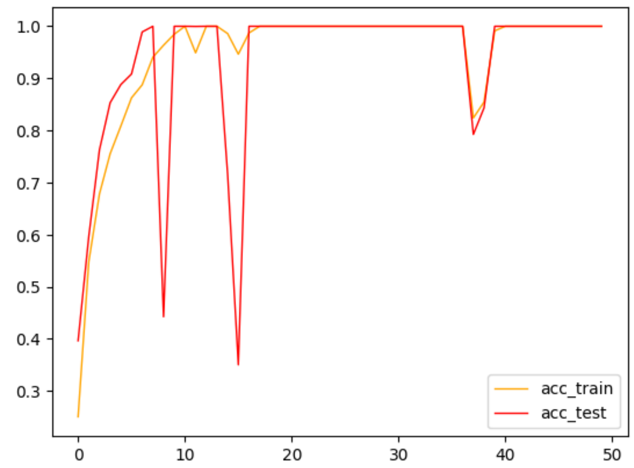
T = 3

<table>
<tr><th>loss</th><th>acc</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

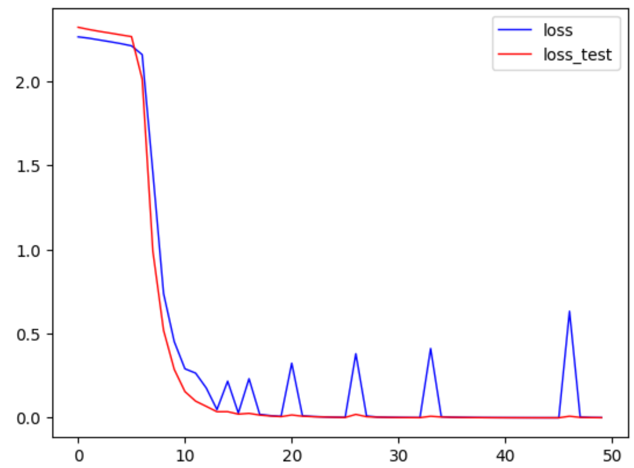
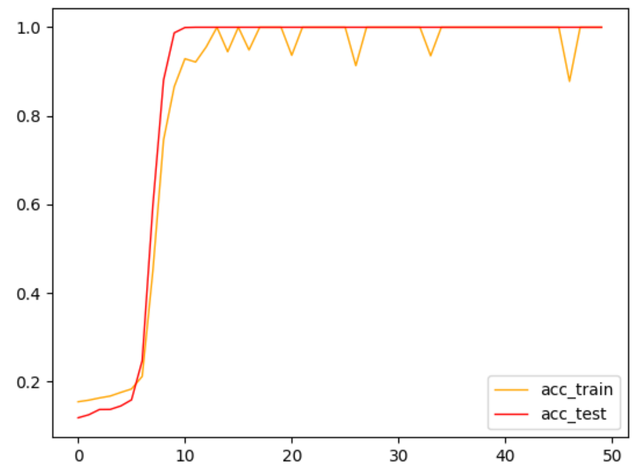
T = 5

loss	acc
	

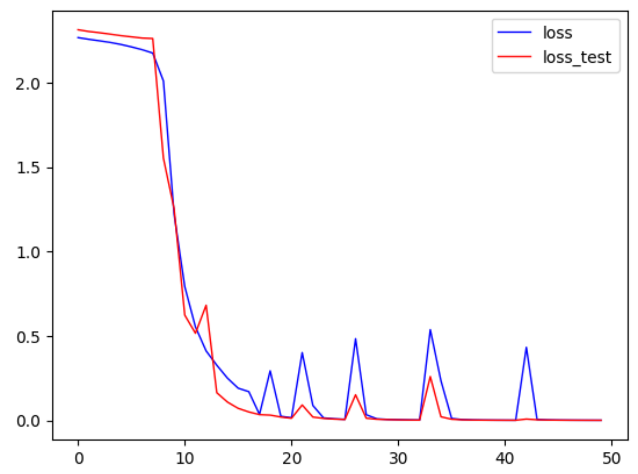
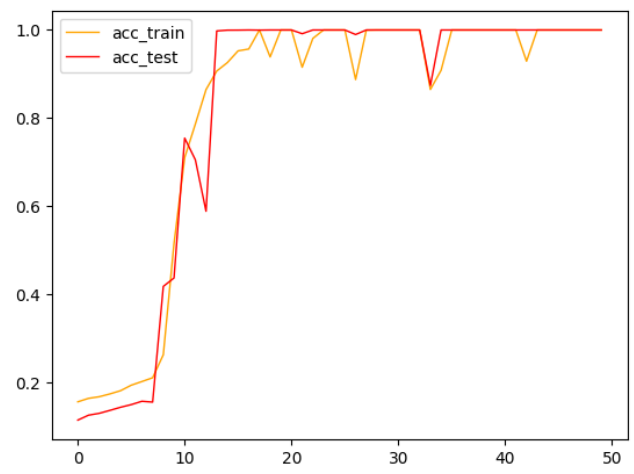
T = 7

loss	acc
	

T = 10

loss	acc
	

T = 15

<table>
<tr><th>loss</th><th>acc</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

3. Analysis

- Compare with the performance in different T of LSTM, it is found that when the length=5, LSTM has the best performance, while the length increases or decreases, the performance and the training curves get worse.
- Compare the LSTM with RNN, it is found that at the same length T, LSTM has more smooth training and testing loss and accuracy curves. Since RNN only has 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. However, LSTM has long short memory, so it won't lose the long memory.

Part II: Generative Adversarial Networks (60 points)

1. Method & Preparation

1.1 GAN model

The overall architecture of the network is implemented according to slides. The implement code of GAN is shown below.

```
class Generator(nn.Module):
    def __init__(self, latent_dim):
        super(Generator, self).__init__()

        self.fc1 = nn.Linear(latent_dim, 128)
        self.fc2 = nn.Linear(128, 256)
        self.fc3 = nn.Linear(256, 512)
        self.fc4 = nn.Linear(512, 1024)
        self.fc5 = nn.Linear(1024, 784)

        self.BN1 = nn.BatchNorm1d(256)
        self.BN2 = nn.BatchNorm1d(512)
        self.BN3 = nn.BatchNorm1d(1024)
```

```
        self.leaky_relu1 = nn.LeakyReLU(0.2)
        self.leaky_relu2 = nn.LeakyReLU(0.2)
        self.leaky_relu3 = nn.LeakyReLU(0.2)
        self.leaky_relu4 = nn.LeakyReLU(0.2)


        # Construct generator. You should experiment with your model,
        # but the following is a good start:
        #   Linear args.latent_dim -> 128
        #   LeakyReLU(0.2)

        #   Linear 128 -> 256
        #   Bnorm
        #   LeakyReLU(0.2)

        #   Linear 256 -> 512
        #   Bnorm
        #   LeakyReLU(0.2)

        #   Linear 512 -> 1024
        #   Bnorm
        #   LeakyReLU(0.2)

        #   Linear 1024 -> 784
        #   Output non-linearity

    def forward(self, z):
        # Generate images from z
        z = self.leaky_relu1(self.fc1(z))
        z = self.leaky_relu2(self.BN1(self.fc2(z)))
        z = self.leaky_relu3(self.BN2(self.fc3(z)))
        z = self.leaky_relu4(self.BN3(self.fc4(z)))
        z = torch.tanh(self.fc5(z))
        return z


class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 1)
        self.leaky_relu1 = nn.LeakyReLU(0.2)
        self.leaky_relu2 = nn.LeakyReLU(0.2)

        # Construct distriminator. You should experiment with your model,
        # but the following is a good start:
        #   Linear 784 -> 512
```

```
        #   LeakyReLU(0.2)
        #   Linear 512 -> 256
        #   LeakyReLU(0.2)
        #   Linear 256 -> 1
        #   Output non-linearity

    def forward(self, img):
        # return discriminator score for img
        img = self.leaky_relu1(self.fc1(img))
        img = self.leaky_relu2(self.fc2(img))
        img = torch.sigmoid(self.fc3(img))
        return img
```

In this section, I mainly compare two training strategy (gan_1_d_1_g.ipynb & gan_5_d_2_g_ipynb).

First, I try to train discriminator and generator each one time in one epoch, however, I found that the disciminitor can't distinguish the real and fake data well and the loss of discrimitor is little high, so I increase the training time of dicriminitor in each epoch, and this approach did work out! The analysis of this strategy is shown in the Experiment Result part.

1.2 Dataset

Using MNIST dataset.

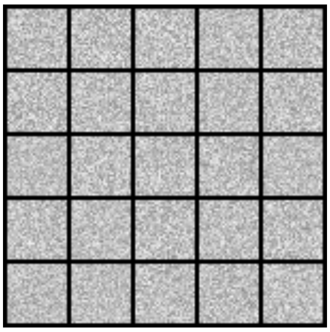
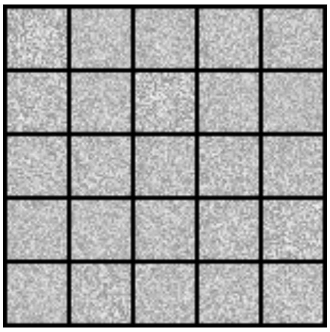
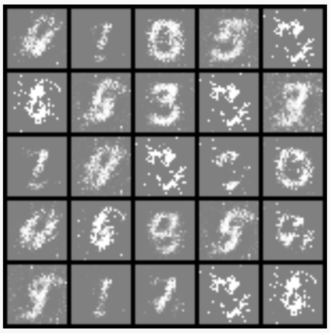
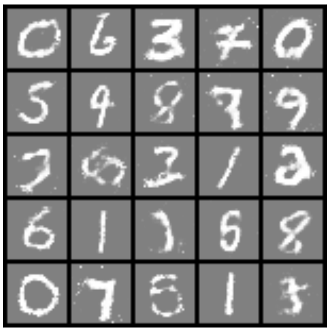
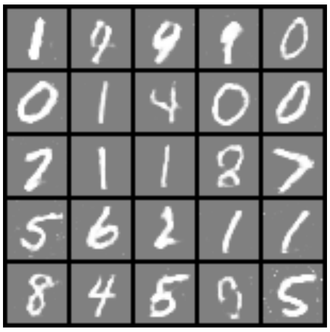
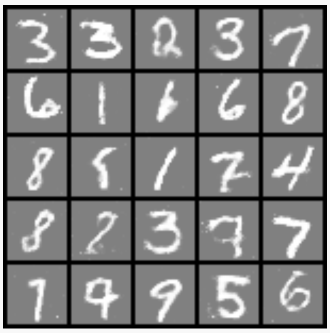
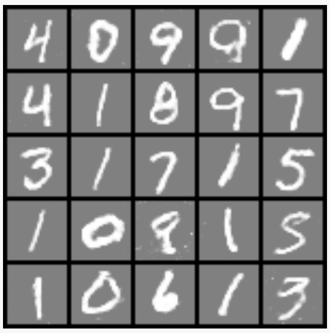
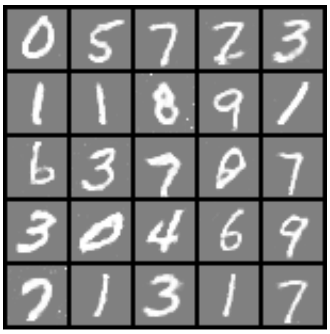
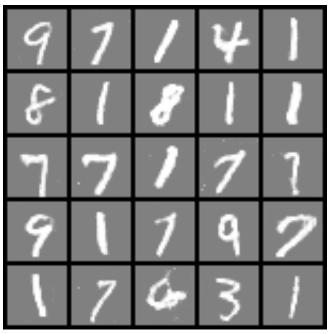
1.3 Hyper-parameter & Environment

```
# hyper-paremeter of GAN
max_epoch = 200
batch_size = 64
learning_rate = 0.0002
latent_dim = 100
save_interval = 500
```

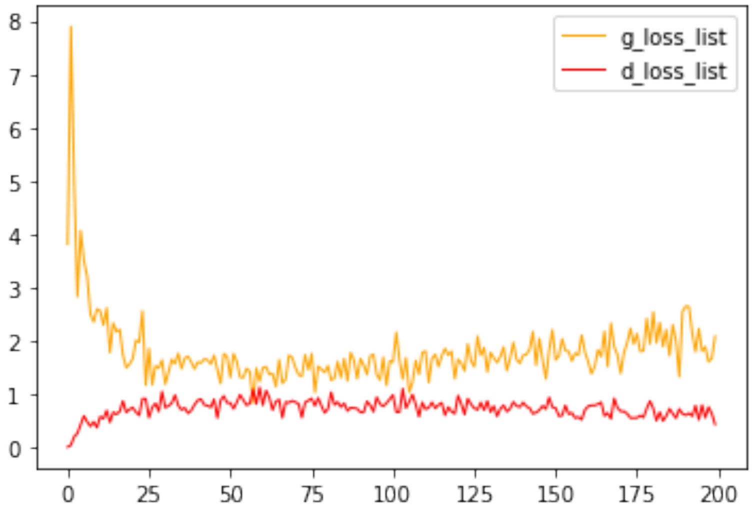
```
# Training & Testing Environment
CPU AMD EPYC 7451
GPU RTX3090 24G

conda 23.5.0
python 3.9.7
torch 2.0.1
CUDA: 11.6
```

2. Experimental Results

2.1 TASK 2

	training strategy 1: 1 d train 1 g train	**training strategy 2: 5 d train 2 g train**
begin id=0		
mid_1 id=10000		
mid_2 id=80000		
mid_3 id=140000		
final id=185500		

The loss curves of G loss and D loss in strategy 2.

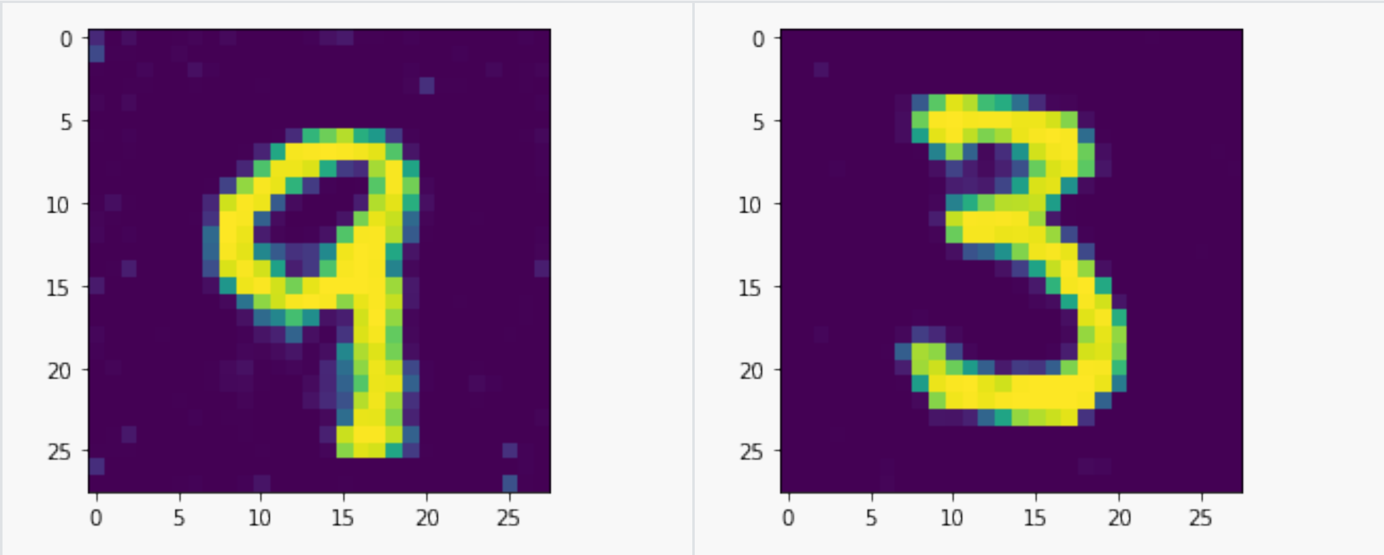


2.2 TASK 3

you can find the implement in GAN.ipynb

generate the images:

take No.37 9 and No.43 8 as an example



interpolated result is:

3. Analysis

Training GAN takes much more time than LSTM, which is consistent with what is learned in the class.

In task 2, some numbers can be generated well in the end, i.e. 1,4,7,9, and some are generated not very good like 5 and 8, this may because the structure of 5 and 8 is littel more complex than other numbers.

And it is found that the generated result of strategy 2 is better than strategy 1 since the discrimitor of strategy 2 is much stronger than the one of strategy 1. And the G loss at first is very large, and it becomes better in each iteration. However, the G loss began to become larger after epoch 100, which may because the discrimitor is training 5 times per epoch, and its training advantage begin to take place, so the generator loss becomes bigger.

The graph of interpolate process of TASK 3 shows the transmission from number 9 to number 3. We can find that at the beginning, the latent space generate number 9, and it changed to number 3 step by step, however, the middle process of this changing is not very Intuitive, in my point of view, this may because the latent space is 100 dim large and the training process of 200 epoch hasing coverage the whole space, so the middle space is little vague. However, we can still figure out the appearance and disappearance of cureves and circle of number 3 and number 9.