Deep Learning Assignment 3 Report

姓名：邱逸伦

学号：12013006

Note: all the codes are implemented in Jupyter Notebook files, which means they can be run directly if all the packages are installed. I also check the environment in the codes so all the Jupyter Notebook files can be run in both GPU environment (cuda) and CPU environment (very slow).

Part 1

Import the necessary libraries.

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import torch
import argparse
import torchvision
import numpy as np
import torch.nn as nn
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn import model_selection
from torchvision import transforms
from torch.utils.data import DataLoader
```

Implement a `draw` function that can draw two subplots including the loss chart and the accuracy chart in one figure.

```
def draw(steps, accs, losses, model_name):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
    ax1.plot([i for i in range(steps)], losses, '-', color='#4169E1', alpha=0.8, linewidth=1)
    ax1.set_xlabel('Steps')
    ax1.set_ylabel('Loss')
    ax1.set_title(f'{model_name} Loss Chart')
    ax2.plot([i for i in range(steps)], accs, '-', color='#4169E1', alpha=0.8, linewidth=1)
    ax2.set_xlabel('Steps')
    ax2.set_ylabel('Accuracy')
    ax2.set_title(f'{model_name} Accuracy Chart')
    plt.show()
```

Construct the palindrome dataset as the `PalindromeDataset` class. (the codes are given)

```
class PalindromeDataset(data.Dataset):

    def __init__(self, seq_length):
        self.seq_length = seq_length

    def __len__(self):
        return sys.maxsize

    def __getitem__(self, idx):
        full_palindrome = self.generate_palindrome()
        return full_palindrome[0:-1], int(full_palindrome[-1])

    def generate_palindrome(self):
        left = [np.random.randint(0, 10) for _ in range(math.ceil(self.seq_length / 2))]
        left = np.asarray(left, dtype=np.float32)
        right = np.flip(left, 0) if self.seq_length % 2 == 0 else np.flip(left[:-1], 0)
        return np.concatenate((left, right))
```

Implement the `VanillaRNN` class, most of the codes are just copied from Assignment 2 to compare with the `LSTM` class in this assignment, but I just make a few changes here. In Assignment 2, the `batch_size` should be an argument passed to the constructor of the `VanillaRNN`, but it is a bad design because we can not pass batches with different sizes. So in this task, the `batch_size` is computed when the `forward` method is called. In this way, the model can accept batches with different sizes.

```
class VanillaRNN(nn.Module):

    def __init__(self, seq_length, input_dim, hidden_dim, output_dim):
        super(VanillaRNN, self).__init__()
        self.seq_length = seq_length
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.Whx = nn.Parameter(torch.randn(input_dim, hidden_dim))
        self.Whh = nn.Parameter(torch.randn(hidden_dim, hidden_dim))
        self.Wph = nn.Parameter(torch.randn(hidden_dim, output_dim))
        self.bh = nn.Parameter(torch.zeros(1, hidden_dim))
        self.bo = nn.Parameter(torch.zeros(1, output_dim))

    def forward(self, x):
        batch_size = x.shape[0]
        h = torch.zeros(batch_size, self.hidden_dim)
        for t in range(self.seq_length):
            h = torch.tanh(torch.mm(x[:, t].reshape(-1, 1), self.Whx) + torch.mm(h, self.Whh) +
self.bh)
        output = torch.mm(h, self.Wph) + self.bo
        return output
```

Implement the `LSTM` class.

```
class LSTM(nn.Module):

    def __init__(self, seq_length, input_dim, hidden_dim, output_dim):
        super(LSTM, self).__init__()
        self.seq_length = seq_length
        self.input_dim = input_dim
```

```
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.Wgx = nn.Parameter(torch.randn(input_dim, hidden_dim))
        self.Wgh = nn.Parameter(torch.randn(hidden_dim, hidden_dim))
        self.bg = nn.Parameter(torch.randn(1, hidden_dim))
        self.Wix = nn.Parameter(torch.randn(input_dim, hidden_dim))
        self.Wih = nn.Parameter(torch.randn(hidden_dim, hidden_dim))
        self.bi = nn.Parameter(torch.randn(1, hidden_dim))
        self.Wfx = nn.Parameter(torch.randn(input_dim, hidden_dim))
        self.Wfh = nn.Parameter(torch.randn(hidden_dim, hidden_dim))
        self.bf = nn.Parameter(torch.randn(1, hidden_dim))
        self.Wox = nn.Parameter(torch.randn(input_dim, hidden_dim))
        self.Woh = nn.Parameter(torch.randn(hidden_dim, hidden_dim))
        self.bo = nn.Parameter(torch.randn(1, hidden_dim))
        self.Wph = nn.Parameter(torch.randn(hidden_dim, output_dim))
        self.bp = nn.Parameter(torch.randn(1, output_dim))

    def forward(self, x):
        batch_size = x.shape[0]
        h = torch.zeros(batch_size, self.hidden_dim)
        c = torch.zeros(batch_size, self.hidden_dim)
        for t in range(self.seq_length):
            xt = x[:, t].reshape(-1, 1)
            gt = torch.tanh(torch.mm(xt, self.Wgx) + torch.mm(h, self.Wgh) + self.bg)
            it = torch.sigmoid(torch.mm(xt, self.Wix) + torch.mm(h, self.Wih) + self.bi)
            ft = torch.sigmoid(torch.mm(xt, self.Wfx) + torch.mm(h, self.Wfh) + self.bf)
            ot = torch.sigmoid(torch.mm(xt, self.Wox) + torch.mm(h, self.Woh) + self.bo)
            c = gt * it + c * ft
            h = torch.tanh(c) * ot
        return torch.mm(h, self.Wph) + self.bp
```

In the `LSTM` class, all parameters including W_{gx}, W_{gh}, and b_g as the weight and the bias for the cell input activation vector, W_{ix}, W_{ih}, and b_i as the weight and the bias for the input gate's activation vector, W_{fx}, W_{fh}, and b_f as the weight and the bias for the forget gate's activation vector, W_{ox}, W_{oh}, and b_o as the weight and the bias for the output gate's activation vector, are initialized randomly. The remaining parameters W_{ph} and b_p are used in computing the final result. In the `forward` method, the `h` as the hidden state vector and the `c` as the cell state vector are initialized as zero-tensors, both `h` and `c` will change over time.

The compact forms of the equations for the forward pass of an LSTM cell with a forget gate are:

$$
\begin{aligned}
g^t &= \sigma(W_{gx}x^t + W_{gh}h^{t-1} + b_c) \\
i^t &= \sigma(W_{ix}x^t + W_{ih}h^{t-1} + b_i) \\
f^t &= \sigma(W_{fx}x^t + W_{fh}h^{t-1} + b_f) \\
o^t &= \sigma(W_{ox}x^t + W_{oh}h^{t-1} + b_o) \\
c^t &= g^t \odot i^t + c^{t-1} \odot f^t \\
h^t &= tanh(c^t) \odot o^t
\end{aligned}
$$

In this progress, we use a for-loop to traverse all time slices.

Finally, after traversing all the time slices, we use the last h^t to get the final result by using the equations below:

$$
p^t = W_{ph}h^t + b_p
$$

Here we eliminate the `softmax` procedure because it will be applied in `nn.CrossEntropyLoss` to obtain the loss function.

```
steps = 50000
step_limit = 40000
batch_size = 128
input_dim = 1
hidden_dim = 128
output_dim = 10
```

Here we set the maximum number of steps for training as 50000, it is smaller than my experiment in Assignment 2 due to the limit of time. After the number of steps is larger than `step_limit`, the program will record the average accuracy between `step_limit` and `steps` because the performance of the model fluctuates, so the average value can better reflect the model performance. Other parameters are easy to understand from their names.

The `main` method calls the `train` and test methods to train and test the model.

```
def main(model_name, length):
    model = train(model_name, length)
    test(model, model_name, length)
```

The `train` method receives two arguments: `model_name` and `length`. The `model_name` denotes the name of the model to specify a model and the `length` denotes the length of the sequence to be predicted. In this method, we first get the model based on the given `model_name`. After that, we construct the dataset, data loader, optimizer RMSprop using the learning rate 0.001, and the cross entropy loss function. Then we use for-loop to traverse all steps. In each step, we use `torch.nn.utils.clip_grad_norm_` to try to handle exploding gradients. Then we use the model to obtain the predicted output and compute the loss function between the output and the target. Finally, we clear the gradient stored in the optimizer and compute the gradients and update the model parameters by the optimizer. There are also some statements to print related information and draw the charts. The well trained model will be a return argument for testing.

```
def train(model_name, length):
    if model_name == 'VanillaRNN':
        model = VanillaRNN(length-1, input_dim, hidden_dim, output_dim)
    elif model_name == 'LSTM':
        model = LSTM(length-1, input_dim, hidden_dim, output_dim)
    else:
        raise Exception('Model name not supported!')
    losses, accs = [], []
    range_acc = []
    dataset = PalindromeDataset(length)
    data_loader = data.DataLoader(dataset, batch_size)
    optimizer = torch.optim.RMSprop(model.parameters(), lr=0.001)
    criterion = nn.CrossEntropyLoss()
    for step, (batch_inputs, batch_targets) in enumerate(data_loader):
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=10.0)
        batch_outputs = model(batch_inputs)
        loss = criterion(batch_outputs, batch_targets)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        acc = np.mean(np.argmax(batch_outputs.detach().numpy(), axis=1) ==
batch_targets.detach().numpy())
        losses.append(loss.item())
        accs.append(acc)
        if step % 2000 == 0:
            print('Step: ', step, 'Loss: ', loss.item(), 'Accuracy: ', acc)
        if step == steps - 1:
            print('Step: ', step, 'Loss: ', loss.item(), 'Accuracy: ', acc)
```

```
            break
        if step >= step_limit:
            range_acc.append(acc)
    print(f'Average accuracy steps {step_limit} - {steps}: {np.mean(range_acc)}')
    draw(steps, accs, losses, model_name)
    return model
```

The `test` method test the performance of the trained model. In this method, we first construct the dataset and set `model.eval()` for model testing. Here we just test one batch with the size 128 in the data loader.

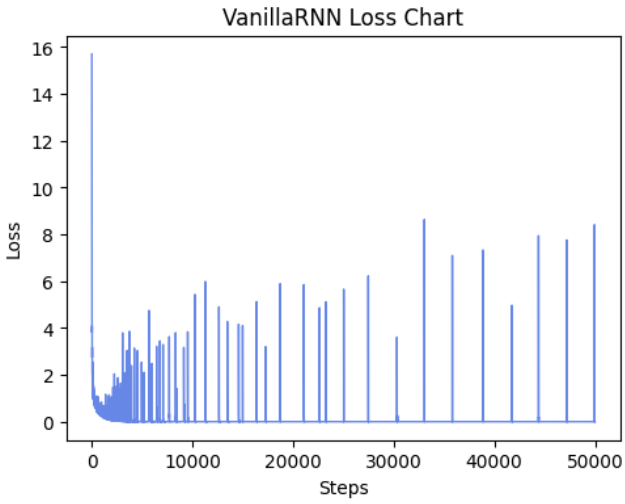
```
def test(model, model_name, length):
    model.eval()
    dataset = PalindromeDataset(length)
    data_loader = data.DataLoader(dataset, batch_size=128)
    for step, (batch_inputs, batch_targets) in enumerate(data_loader):
        batch_outputs = model(batch_inputs)
        acc = np.mean(np.argmax(batch_outputs.detach().numpy(), axis=1) ==
batch_targets.detach().numpy())
        print(f'{model_name} Test Accuracy :', acc)
        break
```

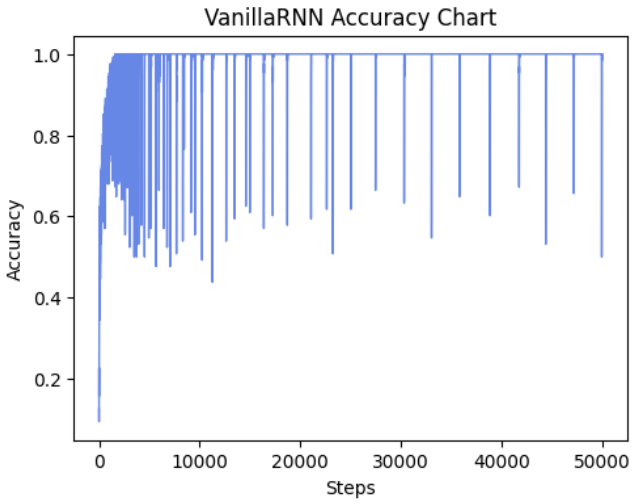
In the following parts, I run both the `VanillaRNN` and `LSTM` class with sequence lengths 5, 10, 15, and 20.

VanillaRNN with length 5

```
length = 5
main("VanillaRNN", length)
```

```
Step:  0 Loss:  15.70534896850586 Accuracy:  0.125
Step:  2000 Loss:  0.06898149847984314 Accuracy:  0.984375
Step:  4000 Loss:  0.021910876035690308 Accuracy:  1.0
Step:  6000 Loss:  0.03117651678621769 Accuracy:  0.9921875
...... # the whole result can be seen in part1.ipynb
Step:  48000 Loss:  5.503869033418596e-06 Accuracy:  1.0
Step:  49999 Loss:  0.001487372093833983 Accuracy:  1.0
Average accuracy steps 40000 - 50000: 0.9992491436643665
```



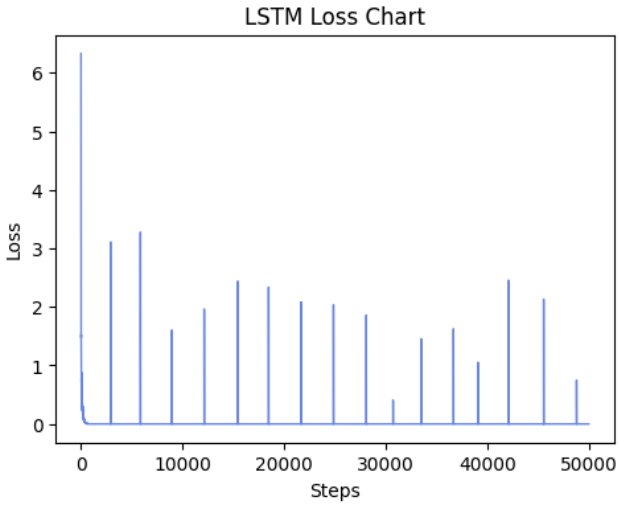


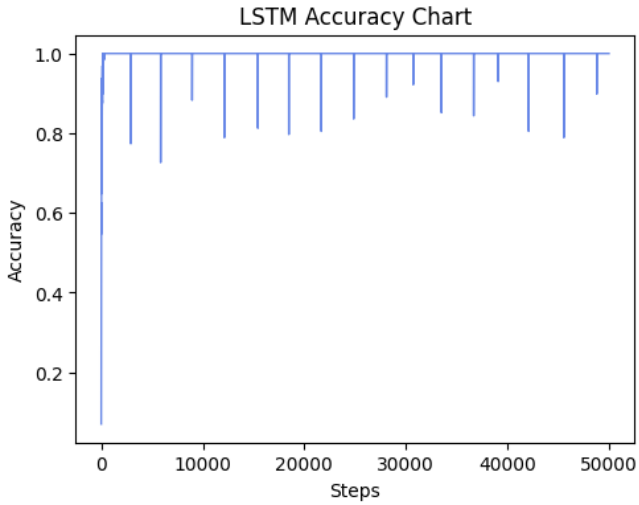

```
VanillaRNN Test Accuracy : 1.0
```

LSTM with length 5

```
length = 5
main("LSTM", length)
```

```
Step:  0 Loss:  6.33057975769043 Accuracy:  0.0703125
Step:  2000 Loss:  1.6065284853539197e-06 Accuracy:  1.0
Step:  4000 Loss:  3.2968776508823794e-07 Accuracy:  1.0
Step:  6000 Loss:  1.8992537661688402e-05 Accuracy:  1.0
...... # the whole result can be seen in part1.ipynb
Step:  48000 Loss:  3.725290076417309e-09 Accuracy:  1.0
Step:  49999 Loss:  2.7101458499600994e-07 Accuracy:  1.0
Average accuracy steps 40000 - 50000: 0.9998788941394139
```



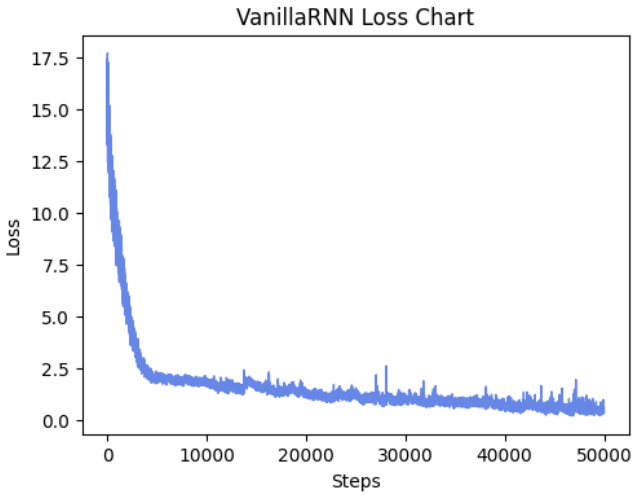


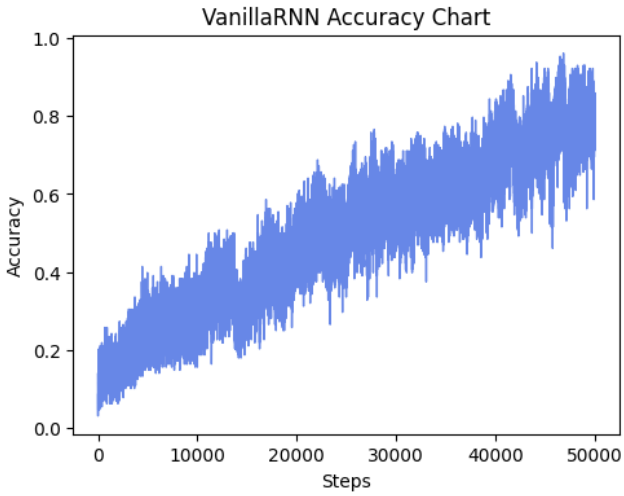

```
LSTM Test Accuracy : 1.0
```

VanillaRNN with length 10

```
length = 10
main("VanillaRNN", length)
```

```
Step:  0 Loss:  16.942338943481445 Accuracy:  0.0859375
Step:  2000 Loss:  5.973904132843018 Accuracy:  0.09375
Step:  4000 Loss:  2.1476986408233643 Accuracy:  0.234375
Step:  6000 Loss:  1.8937902450561523 Accuracy:  0.3125
...... # the whole result can be seen in part1.ipynb
Step:  48000 Loss:  0.3620891571044922 Accuracy:  0.859375
Step:  49999 Loss:  0.3768668472766876 Accuracy:  0.84375
Average accuracy steps 40000 - 50000: 0.7600197519751976
```



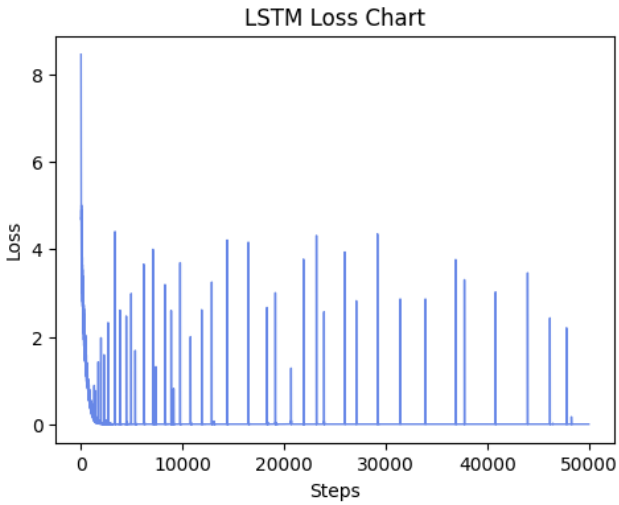


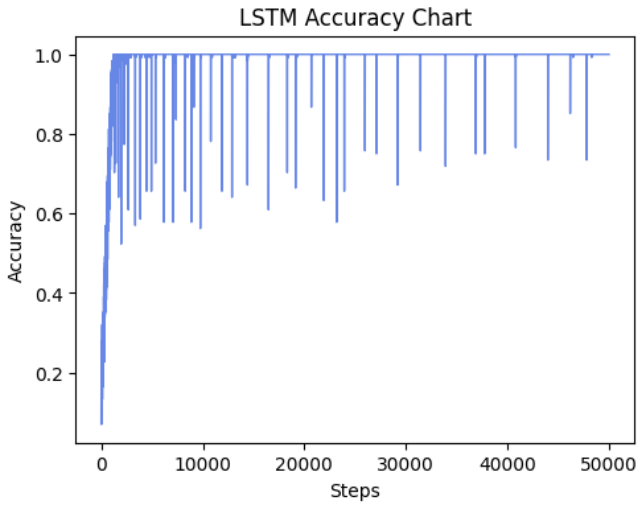

```
VanillaRNN Test Accuracy : 0.8203125
```

LSTM with length 10

```
length = 10
main("LSTM", length)
```

```
Step:  0 Loss:  8.456743240356445 Accuracy:  0.078125
Step:  2000 Loss:  0.021996794268488884 Accuracy:  0.9921875
Step:  4000 Loss:  0.0011693990090861917 Accuracy:  1.0
Step:  6000 Loss:  0.00016551109729334712 Accuracy:  1.0
...... # the whole result can be seen in part1.ipynb
Step:  48000 Loss:  5.718288207390287e-07 Accuracy:  1.0
Step:  49999 Loss:  1.0244547432591844e-08 Accuracy:  1.0
Average accuracy steps 40000 - 50000: 0.9997820094509451
```



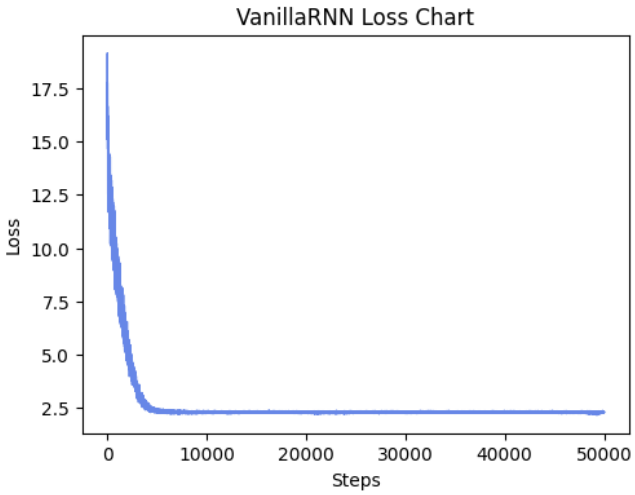


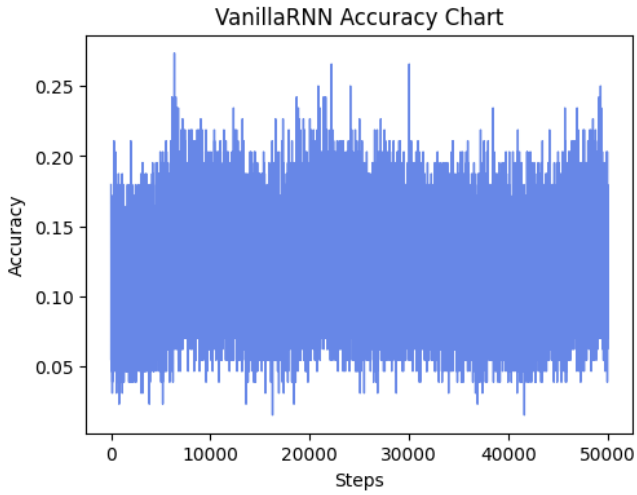

```
LSTM Test Accuracy : 1.0
```

VanillaRNN with length 15

```
length = 15
main("VanillaRNN", length)
```

```
Step:  0 Loss:  15.924747467041016 Accuracy:  0.1796875
Step:  2000 Loss:  4.9273457527160645 Accuracy:  0.125
Step:  4000 Loss:  2.536255121231079 Accuracy:  0.09375
Step:  6000 Loss:  2.2867958545684814 Accuracy:  0.1796875
...... # the whole result can be seen in part1.ipynb
Step:  48000 Loss:  2.3126139640808105 Accuracy:  0.1328125
Step:  49999 Loss:  2.2836196422576904 Accuracy:  0.0859375
Average accuracy steps 40000 - 50000: 0.11861420517051705
```



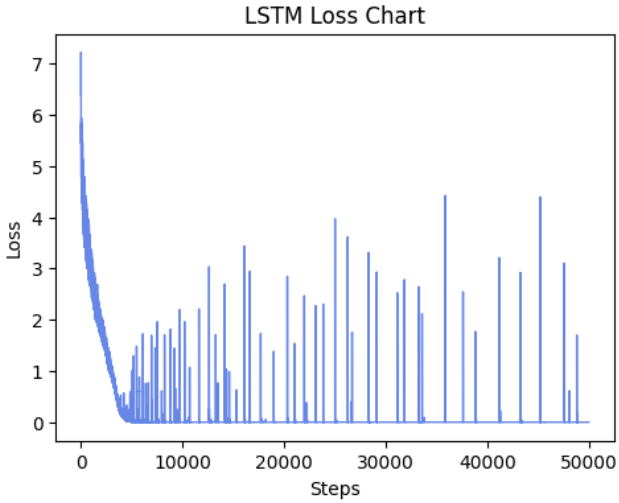


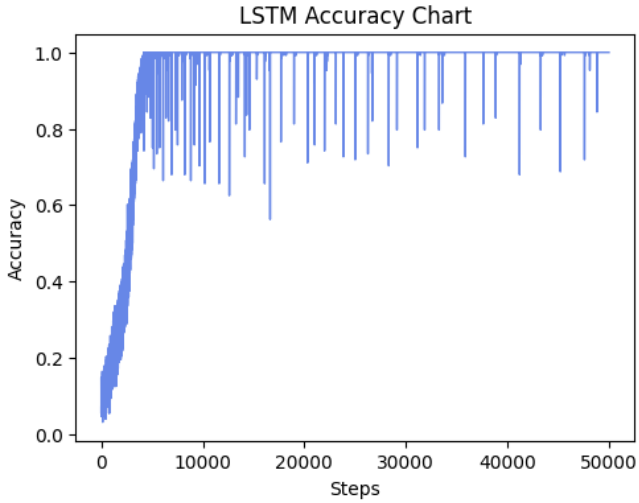

```
VanillaRNN Test Accuracy : 0.140625
```

LSTM with length 15

```
length = 15
main("LSTM", length)
```

```
Step:  0 Loss:  6.3816304206848145 Accuracy:  0.0703125
Step:  2000 Loss:  1.9299362897872925 Accuracy:  0.25
Step:  4000 Loss:  0.1472966969013214 Accuracy:  0.9609375
Step:  6000 Loss:  0.03190506994724274 Accuracy:  0.9921875
...... # the whole result can be seen in part1.ipynb
Step:  48000 Loss:  1.559011934659793e-06 Accuracy:  1.0
Step:  49999 Loss:  4.554157158054295e-07 Accuracy:  1.0
Average accuracy steps 40000 - 50000: 0.9997062206220622
```



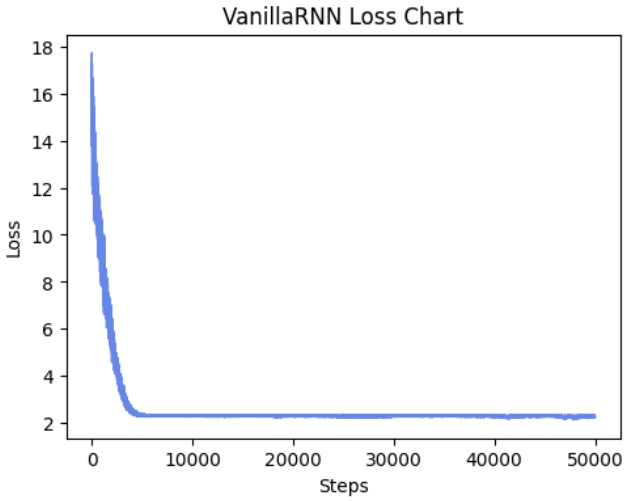


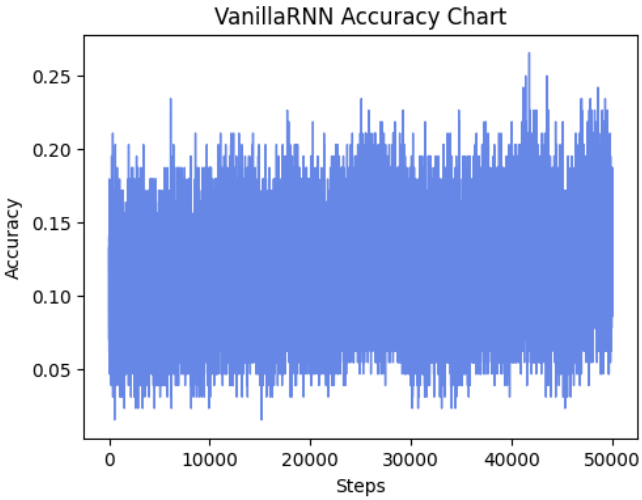

```
LSTM Test Accuracy : 1.0
```

VanillaRNN with length 20

```
length = 20
main("VanillaRNN", length)
```

```
Step:  0 Loss:  15.973852157592773 Accuracy:  0.1328125
Step:  2000 Loss:  5.871718406677246 Accuracy:  0.0625
Step:  4000 Loss:  2.4479520320892334 Accuracy:  0.0859375
Step:  6000 Loss:  2.293243169784546 Accuracy:  0.171875
...... # the whole result can be seen in part1.ipynb
Step:  48000 Loss:  2.2666540145874023 Accuracy:  0.1328125
Step:  49999 Loss:  2.255258560180664 Accuracy:  0.15625
Average accuracy steps 40000 - 50000: 0.12499296804680468
```



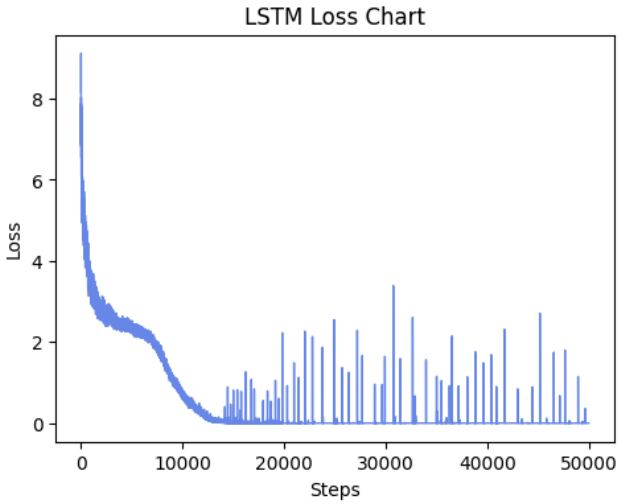


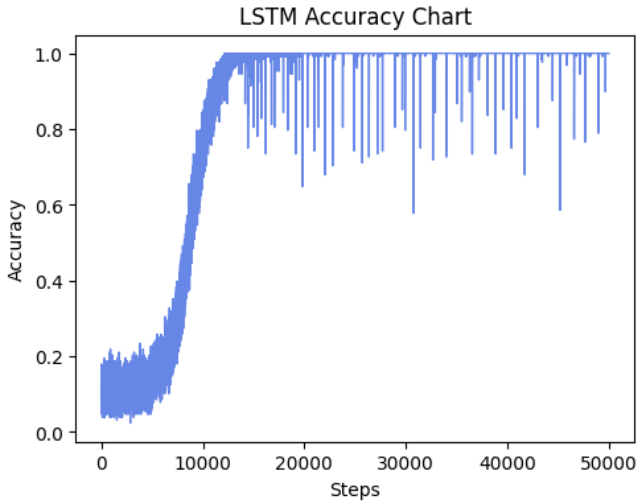

```
VanillaRNN Test Accuracy : 0.1953125
```

LSTM with length 20

```
length = 20
main("LSTM", length)
```

```
Step:  0 Loss:  8.166326522827148 Accuracy:  0.09375
Step:  2000 Loss:  2.680523633956909 Accuracy:  0.1484375
Step:  4000 Loss:  2.4754459857940674 Accuracy:  0.0546875
Step:  6000 Loss:  2.2835092544555664 Accuracy:  0.1796875
Step:  8000 Loss:  1.669158935546875 Accuracy:  0.4140625
Step:  10000 Loss:  0.7012922167778015 Accuracy:  0.75
Step:  12000 Loss:  0.22973191738128662 Accuracy:  0.953125
Step:  14000 Loss:  0.01898360438644886 Accuracy:  1.0
Step:  16000 Loss:  0.030686261132359505 Accuracy:  0.9921875
...... # the whole result can be seen in part1.ipynb
Step:  48000 Loss:  2.1334417397156358e-05 Accuracy:  1.0
Step:  49999 Loss:  2.084018342429772e-05 Accuracy:  1.0
Average accuracy steps 40000 - 50000: 0.9995538616361637
```






```
LSTM Test Accuracy : 1.0
```

Comparison Between VanillaRNN and LSTM:

length\model	VanillaRNN		LSTM	
	40000-5000 AVG	TEST	40000-5000 AVG	TEST
5	0.99924914	1	0.99987889	1
10	0.76002	0.82031	0.99978	1
15	0.11861	0.14063	0.99971	1
20	0.12499	0.19531	0.99955	1

Length 5:

When the sequence length is 5, both VanillaRNN and LSTM can achieve optimal prediction results with a small training step. They both show fluctuations when they converge at accuracy 1 but it can be clearly seen from the graph that LSTM has smaller fluctuations.

Length 10:

When the sequence length is 10, the accuracy of LSTM still converges rapidly to 1 and fluctuates near 1. It can be seen that the fluctuation of the accuracy of LSTM with a sequence length of 10 is greater than that of LSTM with a sequence length of 5, but LSTM still has excellent prediction capabilities. Whereas in the case of VanillaRNN, the loss rapidly decreases at the beginning of training, and when the step is around 5000, the decrease in loss slows down. However, the accuracy of VanillaRNN prediction continues to increase steadily. According to our experiments done in Assignment 2, the accuracy of RNN prediction can still converge to 1 when the training steps reach 100000. From this experiment, it can be seen that LSTM has a faster overall convergence rate than VanillaRNN.

Length 15 and Length 20:

When the sequence lengths are 15 and 20, although the loss function of VanillaRNN decreases rapidly and eventually converges, the accuracy of VanillaRNN predictions remains around 0.12, and the final test results also show that the testing accuracy of VanillaRNN does not exceed 0.2, indicating that VanillaRNN has difficulty retaining complete information of longer sequences. While the loss function of LSTM also drops rapidly, and the prediction accuracy increases accordingly, even though the convergence speed decreases as the sequence length increases. Ultimately, LSTM can almost perfectly predict sequence lengths of 15 and 20, indicating that its gate structures can effectively allow it to remember information about the entire sequence.

Conclusion:

Compared to VanillaRNN, LSTM has the following advantages:

- Can effectively overcome the gradient vanishing problem of VallinaRNN
- Can learn and retain long-term dependencies in sequential data better than VallinaRNN
- Gate structures enable more effective control of information flow in the network
- Overall faster convergence rate
- The predicted results of LSTM are more stable

Part 2

Import the necessary libraries.

```
import os
import tqdm
import torch
import numpy as np
import torch.nn as nn
import matplotlib.pyplot as plt
import torch.utils.data as data
import torchvision.transforms as transforms
from torchvision import datasets
from torchvision.utils import make_grid
```

Construct the generator used in GAN.

```
class Generator(nn.Module):
```

```
    def __init__(self, latent_dim=100):
        super(Generator, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(latent_dim, 128),
            nn.LeakyReLU(0.2),
            nn.Linear(128, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.LeakyReLU(0.2),
            nn.Linear(1024, 784),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.layers(x)
        return x
```

The structure of the generator can be seen below.

```
Generator(
  (layers): Sequential(
    (0): Linear(in_features=100, out_features=128, bias=True)
    (1): LeakyReLU(negative_slope=0.2)
    (2): Linear(in_features=128, out_features=256, bias=True)
    (3): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2)
    (5): Linear(in_features=256, out_features=512, bias=True)
    (6): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2)
    (8): Linear(in_features=512, out_features=1024, bias=True)
    (9): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2)
    (11): Linear(in_features=1024, out_features=784, bias=True)
    (12): Tanh()
  )
)
```

Construct the discriminator used in GAN.

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(784, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
```

```
        x = self.layers(x)
        return x.view(-1)
```

The structure of the discriminator can be seen below.

```
Discriminator(
  (layers): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): LeakyReLU(negative_slope=0.2)
    (2): Linear(in_features=512, out_features=256, bias=True)
    (3): LeakyReLU(negative_slope=0.2)
    (4): Linear(in_features=256, out_features=1, bias=True)
    (5): Sigmoid()
  )
)
```

Define the device used in this assignment. The program will detect whether cuda is available. If we can use cuda, the program will use GPU to accelerate; otherwise, the program just uses CPU to do the work.

```
cuda_id = 0
device_name = "cuda:{}".format(cuda_id) if torch.cuda.is_available() else "cpu"
device = torch.device(device_name)
```

Set the number of epochs as 250, after the experiment, I find out this is a suitable number for models to converge. The size of the batch is 128 and the learning rate is 0.0002. The dimension of noises is defined as 100.

```
epochs = 250
batch_size = 128
lr = 0.0002
latent_dim = 100
```

The dataset used in this part is the MNIST dataset. This is a widely-used dataset in machine learning, consisting of 70,000 labeled images of handwritten digits from 0-9. The dataset contains 60,000 training images and 10,000 testing images, with each image grayscale and its size scaled to 28x28 pixels. Here we just use the training images to do image generation. To achieve better results, we set a normalization transformation and shuffle the dataset. The dataset will be downloaded to the *data* directory if the dataset does not exist. Then the generator and the discriminator will be initialized and moved to the specified device.

Here we use `Adam` optimizers for both the generator and the discriminator with `betas=(0.5, 0.999)` for better performance. The loss function chosen in this part is `nn.BCELoss()`.

```
dataloader = data.DataLoader(
    datasets.MNIST('data', train=True, download=True,
                   transform=transforms.Compose([
                       transforms.ToTensor(),
                       transforms.Normalize(0.5, 0.5)
                   ])),
    batch_size=batch_size, shuffle=True)
generator = Generator().to(device)
discriminator = Discriminator().to(device)
optimizer_g = torch.optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))
optimizer_d = torch.optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))
criterion = nn.BCELoss()
```

The following codes are used in training both the generator and the discriminator. For each iteration, we first reshape the input image. Then we construct `true_labels` as a one-tensor and `fake_labels` as a zero-tensor. Our optimization objective is

$$\min_G \max_D V(D,G) = \min_G \max_D \mathbb{E}_p[\log D(X)] + \mathbb{E}_q[\log(1 - D(G(Z)))]$$

For the discriminator, here the discriminator needs to view all input images as true inputs, so we pass the input image to the discriminator and get a `true_output` result. Then we get the error `error_d_real` by computing the BCE loss between `true_output` and `true_labels`. After that, we generate the `noise` tensor with a latent dimension of 100. Then we use the generator to generate `fake_img` from the constructed `noise`. Then we use the discriminator in `fake_img` to obtain `fake_output`. And we compute the error `error_d_fake` between `fake_output` and `fake_labels`. Then we add two losses `error_d_real` and `error_d_fake` together and we clear the gradient in the model. Finally, we get the gradient of the error through backpropagation and update the model parameters through the optimizer.

For the generator, we generate the `noise` tensor with a latent dimension of 100 again. We use the generator to generate `fake_img` based on `noise` again and use the discriminator to get `fake_output`. Then `fake_output` and `true_labels` are used in calculating the error `error_g`. After that, we clear the gradient in the model, get the gradient of the error through backpropagation, and update the model parameters through the optimizer.

Every 20 epochs, we will try to generate 64 images using the trained generator. After training, we save the trained model.

Note: we use a standard Normal distribution as the noise distribution.

The method `generate` is used to generate images by the trained generator.

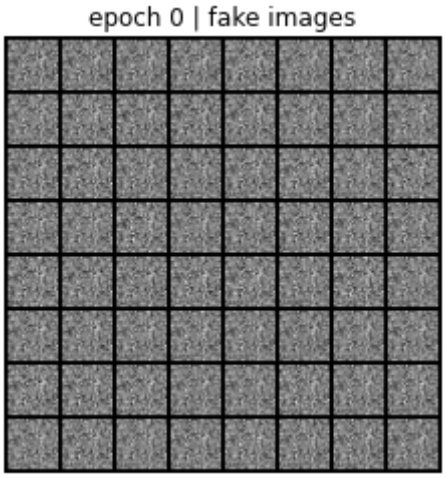
```
def generate(generator, epoch):
    if not os.path.exists('img'):
        os.mkdir('img')
    if not os.path.exists('checkpoints'):
        os.mkdir('checkpoints')
    generator.eval()
    noise = torch.randn(64, latent_dim).to(device)
    with torch.no_grad():
        fake = generator(noise)
    fake = fake.view(-1, 1, 28, 28)
    img_grid_fake = make_grid(fake, normalize=True)
    plt.imshow(np.transpose(img_grid_fake.cpu(), (1, 2, 0)))
    plt.title(f'epoch {epoch} | fake images')
    plt.axis('off')
    plt.show()
```

The training procedure can be seen from the codes below.

```
errors_d = []
errors_g = []
for epoch in tqdm.tqdm(range(epochs)):
    avg_error_d = 0
    avg_error_g = 0
    if epoch == 0 or (epoch + 1) % 20 == 0:
        generate(generator, epoch)
    img = None
    generator.train()
    for i, (img, _) in enumerate(dataloader):
        img = img.view(-1, 784).to(device)
        n_samples = img.shape[0]
```

```
            true_labels = torch.ones(n_samples).to(device)
            fake_labels = torch.zeros(n_samples).to(device)
            true_output = discriminator(img)
            error_d_real = criterion(true_output, true_labels)
            noise = torch.randn(n_samples, latent_dim).to(device)
            fake_img = generator(noise).detach()
            fake_output = discriminator(fake_img)
            error_d_fake = criterion(fake_output, fake_labels)
            error_d = error_d_real + error_d_fake
            avg_error_d += error_d.item()
            optimizer_d.zero_grad()
            error_d.backward()
            optimizer_d.step()
            noise = torch.randn(n_samples, latent_dim).to(device)
            fake_img = generator(noise)
            fake_output = discriminator(fake_img)
            error_g = criterion(fake_output, true_labels)
            avg_error_g += error_g.item()
            optimizer_g.zero_grad()
            error_g.backward()
            optimizer_g.step()
        avg_error_d /= len(dataloader)
        avg_error_g /= len(dataloader)
        errors_d.append(avg_error_d)
        errors_g.append(avg_error_g)
    torch.save(generator.state_dict(), 'checkpoints/generator.pkl')
```

The start of the training:



Halfway through training: (epoch 40)

Halfway through training: (epoch 100)

After the training has terminated: (epoch 500)

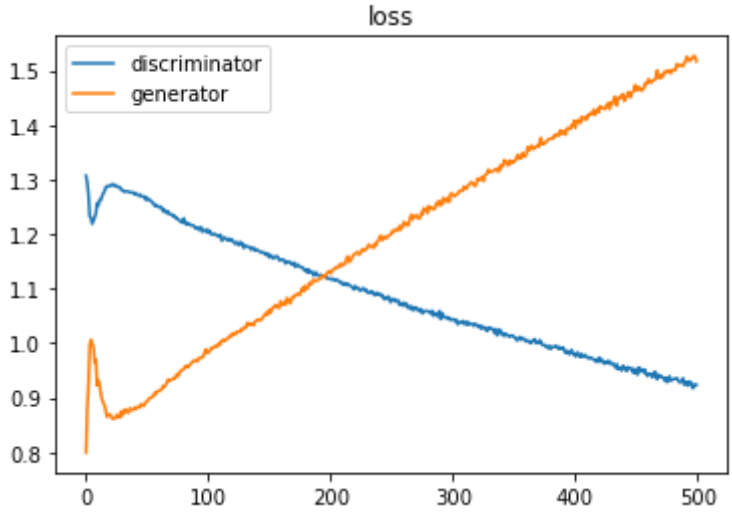
From the figures above, we can find that the figures we generated are better and better when the number of epochs increases. At the start of the training, the generated images are ambiguous. When the epoch is 40, the model can generate rough outlines of handwritten digits that are recognizable to the naked eye as digits, but it does not excel in generating details. When the epoch is 100, the model excels in generating details of the digits, but some parts of the generated images still appear blurry and unclear. When the training has terminated, the generated images by the model are almost indistinguishable from the original dataset, indicating that the image quality generated by the model is already quite high.

After the training procedure, we use the following codes to draw the loss curve for both generator loss and discriminator loss.

```
def draw(errors_d, errors_g):
    plt.plot(errors_d, label='discriminator')
    plt.plot(errors_g, label='generator')
    plt.legend()
    plt.title('loss')
    plt.show()
```

Here `errors_d` and `errors_g` are obtained from the training procedure.

```
draw(errors_d, errors_g)
```

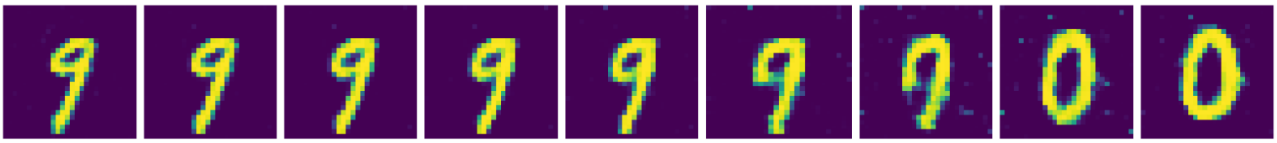


From the loss curve, we can find the discriminator loss drops first and then ascends, while the generator loss ascends first and then drops. Eventually, the discriminator loss exhibits a sustained decline whereas the generator loss displays an ongoing increase. It can be inferred that the curve manifests such a configuration owing to the superior performance of the discriminator as compared to the generator in the context of the zero-sum game. The discriminator consistently outperforms the generator in the late phase, resulting in a downward trend of the discriminator loss. Despite an increasing trend of the generator loss from the curve, the generated images reveal that the generator tends to yield more optimal outputs. Although encountering slight drawbacks during the adversarial process, the generator's performance consistently progresses. It is apparent that the generator and discriminator are continuously engaged in adversarial competition.

Finally, we load the model state saved in training. We sample 2 images from a zero-tensor and a one-tensor, and we use 7 interpolation steps to result in 9 images (including start and end point).

```
model = Generator().to(device)
model.load_state_dict(torch.load('checkpoints/generator.pkl'))
model.eval()
noise0 = torch.zeros(1, latent_dim).to(device)
noise1 = torch.ones(1, latent_dim).to(device)
noises = [noise0]
for i in range(1, 8):
    alpha = i / 8
    noise = alpha * noise1 + (1 - alpha) * noise0
    noises.append(noise)
noises.append(noise1)
for noise in noises:
    with torch.no_grad():
        fake = model(noise)
    fake = fake.view(-1, 1, 28, 28)
    plt.imshow(np.transpose(fake.cpu().numpy()[0], (1, 2, 0)))
    plt.axis('off')
```

```
    plt.show()
```



From the result, we can see that the generated image shows that the entire picture presents a gradual transition effect from the first to the last one. (this result comes from a previous experiment)