CS 5787 Assignment 3

Ran Ji
rj369@

Q1

(1) Processing

Code for this problem is attached in the end.

(2) Implement your RNN module, train the model and report accuracy on the test set.

RNN:

```
class GRU(nn.Module):
    def __init__(self, feature_num):
        super(GRU, self).__init__()
        self.embedding = nn.Embedding(feature_num, 128)
        self.rnn = nn.GRU( input_size=128,
                           hidden_size=64,
                           num_layers=1,
                           dropout=0.5,
                           batch_first=True)
        self.linear1 = nn.Linear(64, 32)
        self.linear2 = nn.Linear(32, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        embedded = self.embedding(x)
        out, _ = self.rnn(embedded)
        out_last = out[:, -1, :] # get last value

        x = self.linear1(out_last.view(-1, out_last.shape[-1]))
        x = self.linear2(F.relu(x))
        x = self.sigmoid(x)
        return x
```

```python
optimizer = None
criterion = None

def get_accuracy(predict, target):
    temp1 = predict >= 0.5
    temp2 = target >= 0.5
    temp3 = (temp1.numpy().reshape(BATCH_SIZE) == temp2.numpy().reshape(BATCH_SIZE))
    return np.sum(temp3)/BATCH_SIZE

def train(model, train_loader, test_loader):
    for time in range(10):
        print(time)
        for i, data in enumerate(train_loader):
            inputs, labels = data

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

    print('Finished Training')

    accuracy_sum = 0.0

    for i, data in enumerate(test_loader):
        inputs, label = data

        output = model(inputs)
        loss = criterion(output, label)

        accuracy = get_accuracy(output, label)
        accuracy_sum = accuracy_sum + accuracy
    print("accuracy is " + str(accuracy_sum*BATCH_SIZE/3000))

```

Accuracy:

```python

model1 = GRU(SIZE+2)
criterion = nn.BCELoss()
optimizer = optim.Adam(model1.parameters(), lr=0.001, betas=(0.9, 0.999))
train(model1, train_loader, test_loader)
```

```
0
1
2
3
4
5
6
7
8
9
Finished Training
accuracy is 0.7470000000000001
```

Part 3 - Comparison with a MLP

```python
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.embedding = nn.Embedding(SIZE+2, 64)

        self.linear1 = nn.Linear(400 * 64, 32)
        self.linear2 = nn.Linear(32, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x)
        x = x.view(-1, 400*64)

        x = self.linear1(x)
        x = self.linear2(x)
        x = self.sigmoid(x)
        return x

model2 = MLP()
criterion = nn.BCELoss()
optimizer = optim.Adam(model2.parameters(), lr=0.0001, betas=(0.9, 0.999))
train(model2, train_loader, test_loader)
```

```
0
1
2
3
4
5
6
7
8
9
Finished Training
accuracy is 0.5706666666666664
```

RNN performs much better on the test data.

Q2

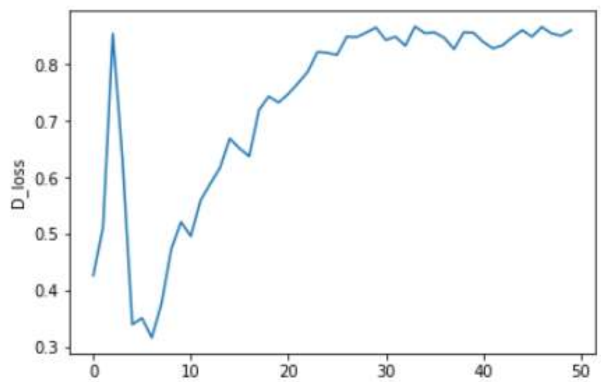
(1)

Train a basic GAN

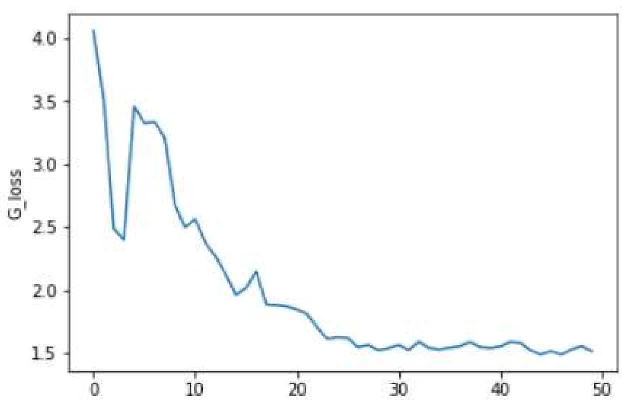
See code attached in the end.

Plot training loss curves for your G and D

```
plt.plot(D_loss)
plt.ylabel("D_loss")
plt.show()
```



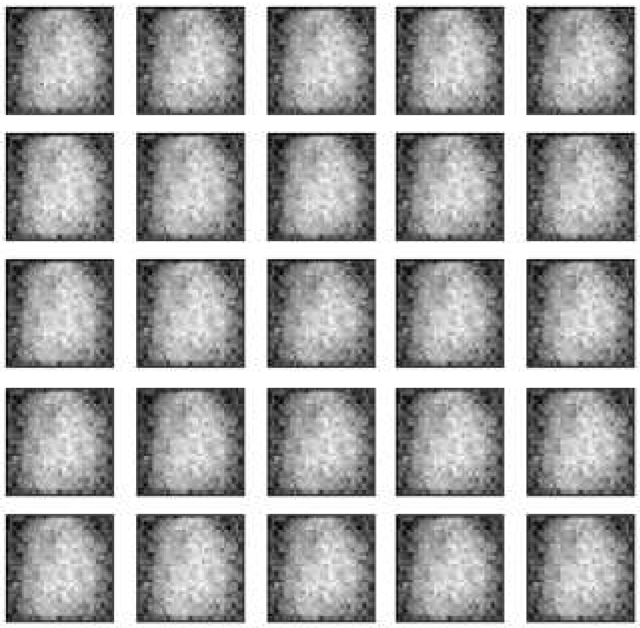

```
plt.plot(G_loss)
plt.ylabel("G_loss")
plt.show()
```



Show the generated samples from G in:

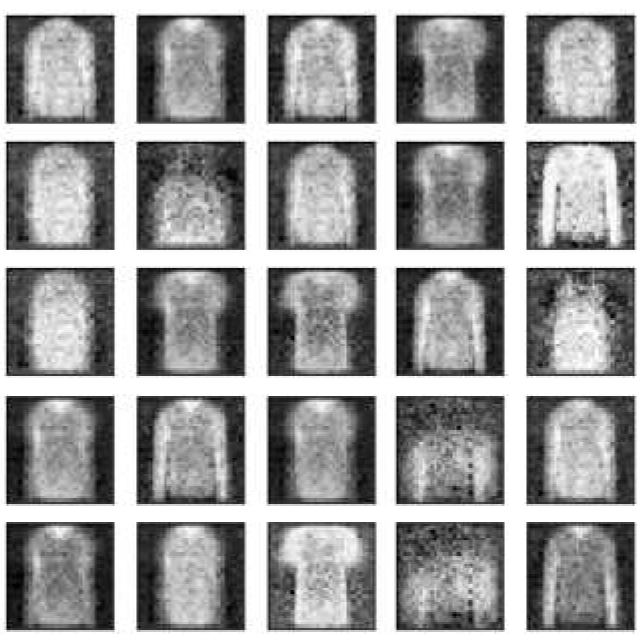
1) the beginning of the training;

```
epoch 2: loss_d: 0.855, loss_g: 2.487
```



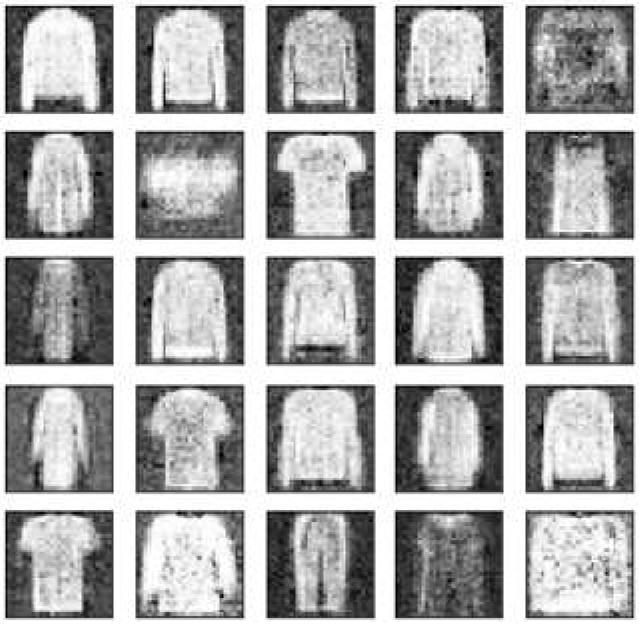
2) intermediate stage of the training;

```
epoch 10: loss_d: 0.496, loss_g: 2.563
```



3) after convergence.

```
epoch 48: loss_d: 0.851, loss_g: 1.556
```



(2) GAN Loss

MSE

Model:

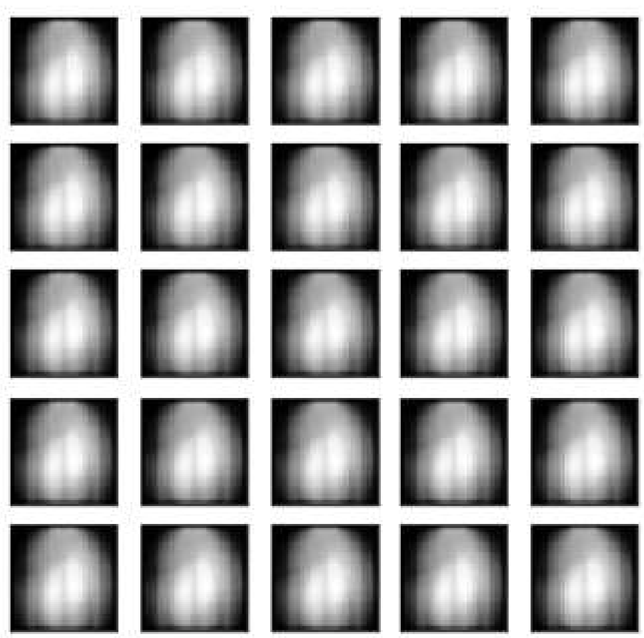
```
class generator(nn.Module):
    def __init__(self):
        super(generator, self).__init__()

        self.layer0 = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(),
        )

        self.layer1 = nn.Sequential(
            nn.Linear(256, 512),
            nn.ReLU(),
        )

        self.layer2 = nn.Sequential(
            nn.Linear(512, 1024),
            nn.ReLU(),
        )

        self.layer3 = nn.Sequential(
            nn.Linear(1024, 784),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.layer0(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        return x
```

Train:

```
G_loss = []

# train
for epoch in range(epoch_number):
    g_epoch_loss = []
    for x, _ in data_loader:
        x = x.view(-1, 28 * 28)
        batch_size = x.size()[0]

        # train generator G
        noise = torch.randn((batch_size, 100))
        y_target = torch.ones(batch_size)
        noise, y_target = Variable(noise.cuda()), Variable(y_target.cuda())

        G.zero_grad()
        G_result = G(noise)
        G_train_loss = criterion(G_result.to('cuda'), x.to('cuda'))
        G_train_loss.backward()
        G_optimizer.step()

        g_epoch_loss.append(G_train_loss.cpu().data.item())

    G_loss.append(sum(g_epoch_loss)/len(g_epoch_loss))

    if (epoch == 0 or epoch == 1 or epoch == 2 or epoch == 10 or epoch == 20 or epoch == 30 or epoch == 48):
        print('epoch %d: loss_g: %.3f' % (epoch, G_loss[epoch]))
        show_result()
```

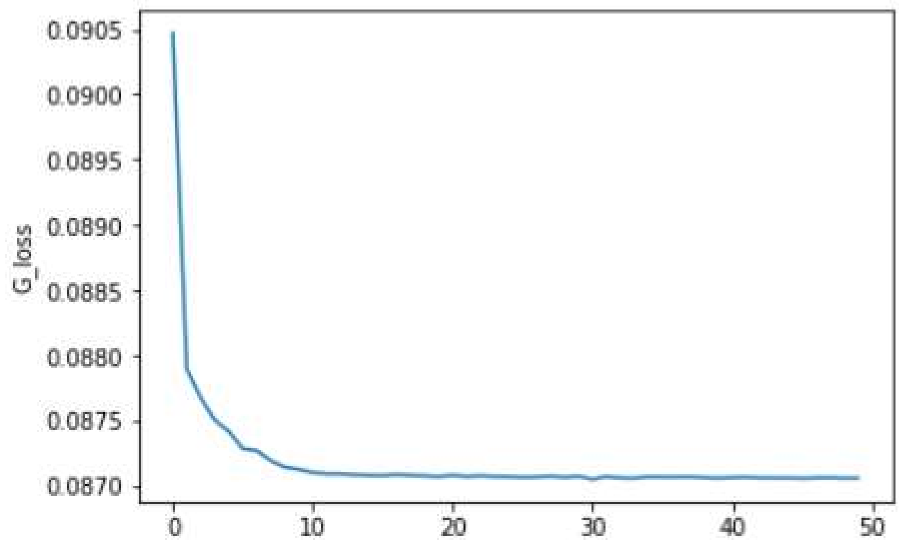
Generated graph:

```
epoch 48: loss_g: 0.087
```



Training loss:

```
plt.plot(G_loss)
plt.ylabel("G_loss")
plt.show()
```



The MSE result is not very good because only the input image is used when training.

Wasserstein GAN (WGAN)

Model (c = 0.1):

```python
class generator(nn.Module):
    def __init__(self):
        super(generator, self).__init__()

        self.layer0 = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(),
        )

        self.layer1 = nn.Sequential(
            nn.Linear(256, 512),
            nn.ReLU(),
        )

        self.layer2 = nn.Sequential(
            nn.Linear(512, 1024),
            nn.ReLU(),
        )

        self.layer3 = nn.Sequential(
            nn.Linear(1024, 784),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.layer0(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        return x
```

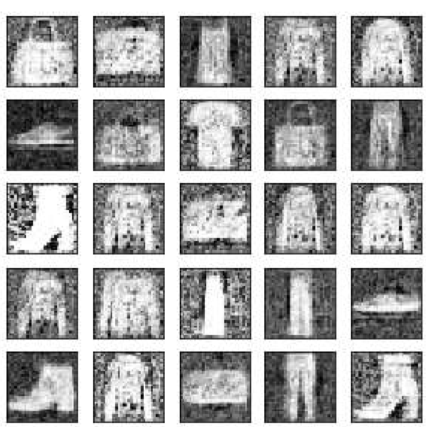
```
class discriminator(nn.Module):
    def __init__(self):
        super(discriminator, self).__init__()

        self.layer0 = nn.Sequential(
            nn.Linear(784, 1024),
            nn.ReLU(),
            nn.Dropout(0.3)
        )

        self.layer1 = nn.Sequential(
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.3)
        )

        self.layer2 = nn.Sequential(
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.3)
        )

        self.layer3 = nn.Sequential(
            nn.Linear(256, 1),
#             nn.Sigmoid()
        )


    def forward(self, x):
        x = self.layer0(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        return x
```

Train:

```python
# train
D_losses = []
G_losses = []
for epoch in range(epoch_number):
    for x, _ in data_loader:
        current_batch_size = x.size()[0]
        x = autograd.Variable(x).detach().cuda()
        x = x.view(x.shape[0], 28*28)

        # Train discriminator
        G_result = G(autograd.Variable(torch.randn(current_batch_size, 100), requires_grad=True).cuda())
        D_optimizer.zero_grad()

        D_result_real = D(x).mean()
        D_result_fake = D(G_result).mean()
        D_total_loss = ((-1)*D_result_real + D_result_fake)
        D_total_loss.backward()
        D_optimizer.step()

        for p in D.parameters():
            p.data.clamp_(-clamp, clamp)



        # Train Generator
        fake_image = G(autograd.Variable(torch.randn(current_batch_size, 100), requires_grad=True).cuda())
        G_optimizer.zero_grad()

        D_result = D(fake_image).mean()
        G_train_loss = (-1) * D_result
        G_train_loss.backward()
        G_optimizer.step()

        for p in G.parameters():
            p.data.clamp_(-clamp, clamp)

    G_losses.append(G_train_loss.item())
    D_losses.append(D_total_loss.item())

    if (epoch == 0 or epoch == 1 or epoch == 2 or epoch == 10 or epoch == 20 or epoch == 30 or epoch == 48):
#     if (True):
        print('epoch %d: loss_d: %f, loss_g: %f' % (epoch, D_losses[epoch], G_losses[epoch]))
        show_result()
```

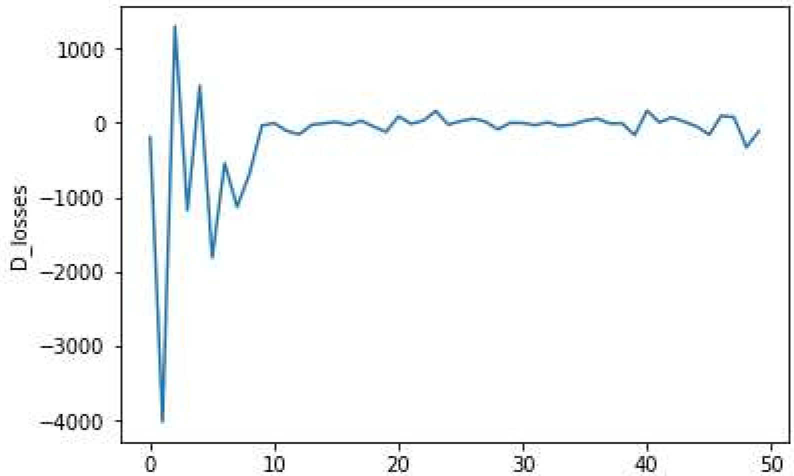
Graph generated:

```
epoch 48: loss_d: -327.651855, loss_g: -402.395721
```

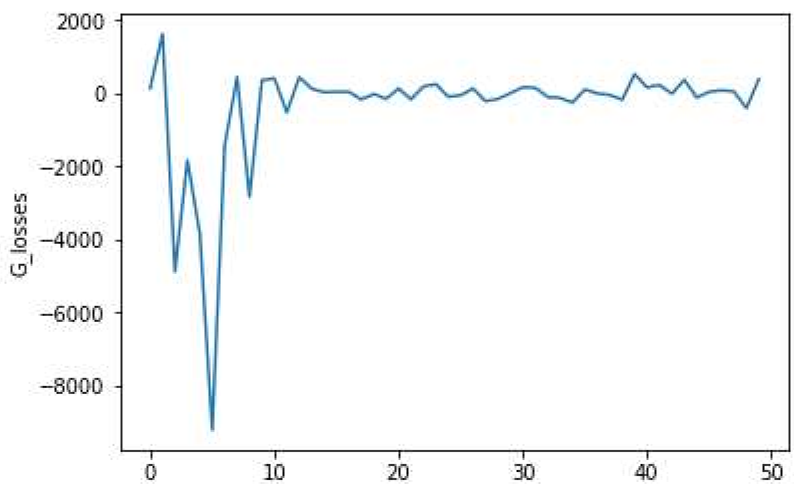


Training loss:

```
In [13]:
plt.plot(D_losses)
plt.ylabel("D_losses")
plt.show()
```




```
In [14]:
plt.plot(G_losses)
plt.ylabel("G_losses")
plt.show()
```



I also tried c with 0:01, 0:001 and 0:0001. The smaller C is, the harder for the model to coverage.

Least Square GAN

Model:

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

        self.layer0 = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(),
        )

        self.layer1 = nn.Sequential(
            nn.Linear(256, 512),
            nn.ReLU(),
        )

        self.layer2 = nn.Sequential(
            nn.Linear(512, 1024),
            nn.ReLU(),
        )

        self.layer3 = nn.Sequential(
            nn.Linear(1024, 784),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.layer0(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        return x
```

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

        self.layer0 = nn.Sequential(
            nn.Linear(784, 1024),
            nn.ReLU(),
            nn.Dropout(0.3)
        )

        self.layer1 = nn.Sequential(
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.3)
        )

        self.layer2 = nn.Sequential(
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.3)
        )

        self.layer3 = nn.Sequential(
            nn.Linear(256, 1),
            nn.Sigmoid()
        )


    def forward(self, x):
        x = self.layer0(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        return x
```

Train:

```
G = generator()
D = discriminator()
G.cuda()
D.cuda()

criterion = nn.BCELoss()

G_optimizer = optim.RMSprop(G.parameters(), lr=lr)
D_optimizer = optim.RMSprop(D.parameters(), lr=lr)
```

```
# train
D_losses = []
G_losses = []
for epoch in range(epoch_number):
    for x, _ in data_loader:
        current_batch_size = x.size()[0]
        x = autograd.Variable(x).detach().cuda()
        x = x.view(x.shape[0], 28*28)

        # Train discriminator
        G_result = G(autograd.Variable(torch.randn(current_batch_size, 100), requires_grad=True).cuda())
        D_optimizer.zero_grad()

        D_result_real = D(x)
        D_result_fake = D(G_result)
        D_total_loss = torch.mean((D_result_real-1)**2) + torch.mean(D_result_fake**2)
        D_total_loss.backward()
        D_optimizer.step()

        # Train Generator
        fake_image = G(autograd.Variable(torch.randn(current_batch_size, 100), requires_grad=True).cuda())
        G_optimizer.zero_grad()

        D_result = D(fake_image)
        G_train_loss = torch.mean((D_result-1)**2)
        G_train_loss.backward()
        G_optimizer.step()

    G_losses.append(G_train_loss.item())
    D_losses.append(D_total_loss.item())

    if (epoch == 0 or epoch == 1 or epoch == 2 or epoch == 10 or epoch == 20 or epoch == 30 or epoch == 48):
#     if (True):
        print('epoch %d: loss_d: %f, loss_g: %f' % (epoch, D_losses[epoch], G_losses[epoch]))
        show_result()
```

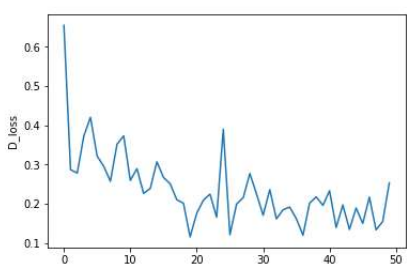
```
epoch 0: loss_d: 0.655025, loss_g: 0.652975
```

Graph generated:

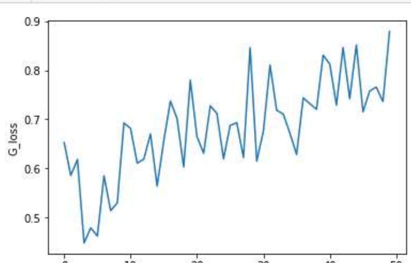
epoch 48: loss_d: 0.154834, loss_g: 0.736266

Training loss:

```
plt.plot(D_losses)
plt.ylabel("D_loss")
plt.show()
```




```
plt.plot(G_losses)
plt.ylabel("G_loss")
plt.show()
```



The MSE takes less time to train, but the result is hard to see. WGAN and Least Square Gan can produce high quantity image from random noise.

Part 3 - Mode Collapse in GANs

To output a classifier, I change the output dimension from 1 to 10 and then train the model using training data and test the result using test data.

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

        self.layer0 = nn.Sequential(
            nn.Linear(784, 1024),
            nn.ReLU(),
            nn.Dropout(0.3)
        )

        self.layer1 = nn.Sequential(
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.3)
        )

        self.layer2 = nn.Sequential(
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.3)
        )

        self.layer3 = nn.Sequential(
            nn.Linear(256, 10),
            nn.Softmax()
        )


    def forward(self, x):
        x = self.layer0(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        return x
```

The accuracy is

```
#test
acc_list = []
for x, label in data_loader2:
    current_batch_size = x.size()[0]

    x = autograd.Variable(x).detach().cuda()
    x = x.view(x.shape[0], 28*28)

    y_real = Variable(label.cuda())
    C_result_real = C(x).detach()
    acc_list.append(get_accuracy(C_result_real, y_real))

acc_list = acc_list[:-1]
print(sum(acc_list)/len(acc_list))

```

```
0.8095953525641025
```

Then I training the GAN network and generate 3000 samples and draw their distributions.

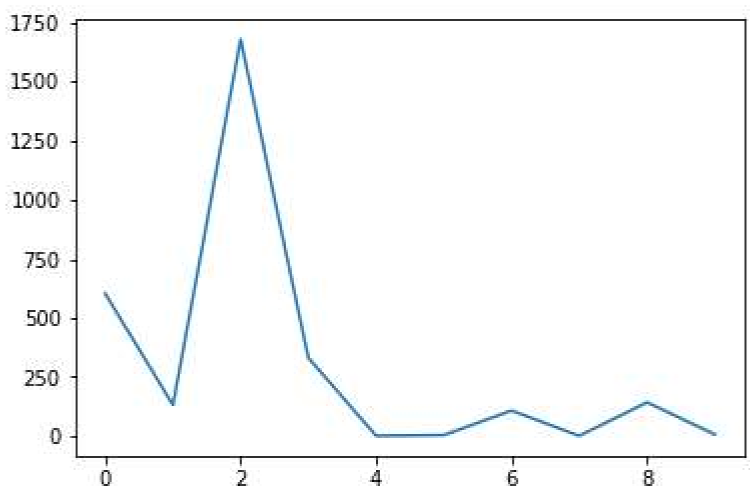
```python
count_map = {}

for i in range(10):
    count_map[i] = 0

for i in range(3000):
    value = generate_image()
    count_map[value] = count_map[value] + 1

print(count_map)
```

```
{0: 603, 1: 131, 2: 1677, 3: 331, 4: 0, 5: 3, 6: 107, 7: 0, 8
```

```python
plot_values = [0]*10
for i in range(10):
    plot_values[i] = count_map[i]

plot_values
```

```
[603, 131, 1677, 331, 0, 3, 107, 0, 142, 6]
```

```python
plt.plot(range(10), plot_values)
plt.show()
```



Q3

(1) Using Discrete States

```python
import numpy as np
import random
from matplotlib import pyplot as plt
```

```python
qtable = np.zeros((81, 4))
```

```python
def load_prob(path, prob_map):
    f = open(path)
    lines = f.readlines()
    f.close
    for line in lines:
        l = line.split()
        key = int(l[0])
        value = int(l[1])
        prob = float(l[2])
        if key in prob_map:
            prob_map[key].append([value, prob])
        else:
            prob_map[key] = [[value, prob]]

def get_state_by_prob(state_prob_s):
    acc = 0.0
    r_prob = random.random()
    for name in state_prob_s:
        acc = acc + name[1]
        if acc >= r_prob:
            return name[0]
    print("None!!!!")
    return None
#     return lofprob[-1][0]

def get_action_state(s, action):
    if action == 0:
        return get_state_by_prob(left_prob[s])
    if action == 1:
        return get_state_by_prob(up_prob[s])
    if action == 2:
        return get_state_by_prob(right_prob[s])
    if action == 3:
        return get_state_by_prob(down_prob[s])
    return None
```

```python
left_prob = {}
up_prob = {}
right_prob = {}
down_prob = {}

load_prob("rl-files/prob-a1.txt", left_prob)
load_prob("rl-files/prob-a2.txt", up_prob)
load_prob("rl-files/prob-a3.txt", right_prob)
load_prob("rl-files/prob-a4.txt", down_prob)
```

```python
def get_state_reward(s):
    if s == 47 or s == 49 or s == 51 or s == 65 or s == 67 or s == 69:
        return -1
    if s == 79:
        return 1
    return 0

def is_playing(s):
    return get_state_reward(s) == 0
```

```python
def processAction(current_s, action):
    playing = is_playing(current_s)
    new_state = get_action_state(current_s, action)
    reward = get_state_reward(current_s)
    return new_state, reward, playing
```

```python
def get_max_action(s):
    max_value = max(qtable[s][0], qtable[s][1], qtable[s][2], qtable[s][3])
    if max_value == qtable[s][0]:
        return 0
    if max_value == qtable[s][1]:
        return 1
    if max_value == qtable[s][2]:
        return 2
    if max_value == qtable[s][3]:
        return 3
    print("None 2!!!!!")
    return None

```

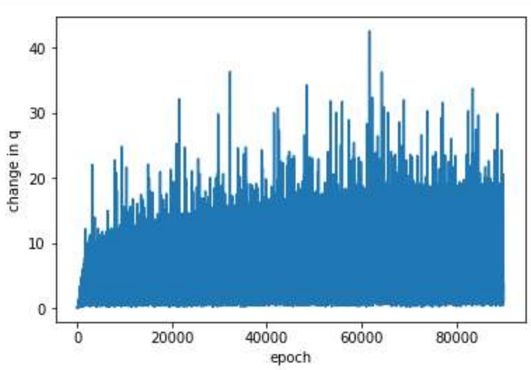
Train

```
eps = 0.8
alpha = 0.1
r = 0.99

delta_q = []

for i in range(90000):
    if (i % 100 == 0):
        eps = eps*0.8
        eps = max(eps, 0.2)
    playing = True
    current_state = 3

    qtable_before = np.copy(qtable)

    while playing:
        action = None
        if random.random() <= eps:
            action = random.randint(0,3)
        else:
            action = get_max_action(current_state)
        new_state, reward, playing = processAction(current_state, action)
#         print("current_state is " + str(current_state))
#         print("new_state is " + str(new_state))
#         print("action is " + str(action))
        new_q_value = (1-alpha)*qtable[current_state][action] + alpha*(reward+r*np.max(qtable[new_state]))
        qtable[current_state][action] = new_q_value
        current_state = new_state

    delta_q.append(np.linalg.norm(qtable-qtable_before))
```

```
qtable[70]
```

Plot Q:

```
plt.plot(range(len(delta_q)), delta_q)
plt.xlabel("epoch")
plt.ylabel("change in q")
plt.show()
```



Plot table:

```python
dir = ['←', '↑', '→', '↓']
special = {47: 'x', 49: 'x', 65: 'x', 67: 'x', 51: 'x', 69: 'x', 79:'o'}

for i in range(9):
    for j in range(9):
        cur = i + j * 9
        if cur in special:
            print(' ', special[cur], end='')
        elif not np.sum(qtable[cur]) == 0:
            print(' ', dir[np.argmax(qtable[cur])], end='')
        else:
            print(' ', '.',end='')
    print('\n')
```

```
  .  .  .  .  .  .  .  .  .

  .  .  .  .  .  .  .  .  .

  .  →  →  ↓  .  x  ↓  x  .

  →  ↑  .  ↓  ←  ←  ↓  ←  .

  .  .  ↓  ←  .  x  ↓  x  .

  .  .  ↓  .  .  .  ↓  .  .

  .  ↓  ←  .  .  x  ↓  x  .

  .  ↓  .  →  →  →  →  →  o

  .  →  →  ↑  .  ↑  →  ↑  .
```

I played around with the epsilon parameter. When it’s small, it’s hard for the agent to learn new states. When it’s big, it’s easier to explore new states but hard to coverage to a good policy. Given this information, I used a reducing epsilon corresponding to epoch number. It will learn fast in the beginning, and do a better job in coverage after exploring enough number of new states.

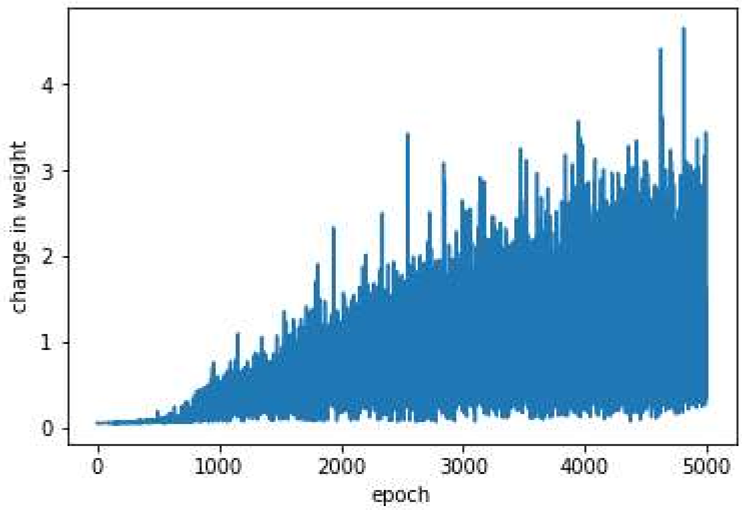
Part 2 - Using One-Hot Vectors

Model (gradient descent update rule and implementation):

```
w = np.zeros((4, 82))
losses = []
delta_w = []

eps = 0.8
r = 0.99
alpha = 0.05

for i in range(5000):
    if (i % 100 == 0):
        print(i)
        eps = eps*0.8
        eps = max(eps, 0.2)
    playing = True
    current_state = 3
    current_state_vector = encode(current_state)
    w_before = np.copy(w)

    while playing:
        action = get_next_move(current_state, w, eps)
        next_state, reward, playing = processAction(current_state, action)

        next_state_vector = encode(next_state)
        max_q = max([w[i].dot(next_state_vector) for i in range(4)])
        qplus = reward + r * max_q

        losses.append(0.5*((np.dot(w[action], current_state_vector) - qplus) ** 2))
        gradient = np.dot(np.dot(w[action], current_state_vector) - qplus, current_state_vector)
        w[action] = w[action] - alpha * gradient

        current_state = next_state
        current_state_vector = next_state_vector

    delta_w.append(np.linalg.norm(w-w_before))
```

Weight change:

```
plt.plot(range(len(delta_w)), delta_w)
plt.xlabel("epoch")
plt.ylabel("change in weight")
plt.show()
```



Plot table:

```
plot_table(q)
```

```
.  .  .  .  .  .  .  .  .

.  →  →  ↓  .  x  ↓  x  .

→  ↑  .  →  →  →  ↓  ←  .

.  .  →  ↑  .  x  ↓  x  .

.  .  ↑  .  .  .  ↓  .  .

.  ↑  ↑  .  .  x  ↓  x  .

.  ↑  .  →  →  →  →  →  o

.  ↑  ↑  ↑  .  ↑  ↑  ←  .

.  .  .  .  .  .  .  .  .
```

In the red circle, the result is different from part 1. In part 1, it actually wants to run away from the path close to many dragons, but in part 2 it takes the path that is close to many dragons.

p1

April 26, 2019

```
In [1]: # P1

In [ ]: import re
        import numpy as np

        import torch
        import torch.nn as nn
        import torch.nn.functional as F
        from torch.utils.data import TensorDataset, DataLoader

In [2]: import torch.optim as optim

In [3]: f_train_pos = open("sentiment_data/train_pos_merged.txt", encoding="utf8")
        f_train_neg = open("sentiment_data/train_neg_merged.txt", encoding="utf8")
        f_test_pos = open("sentiment_data/test_pos_merged.txt", encoding="utf8")
        f_test_neg = open("sentiment_data/test_neg_merged.txt", encoding="utf8")

In [4]: lines_train_pos = f_train_pos.readlines()
        lines_train_neg = f_train_neg.readlines()
        lines_test_pos = f_test_pos.readlines()
        lines_test_neg = f_test_neg.readlines()

        f_train_pos.close()
        f_train_neg.close()
        f_test_pos.close()
        f_test_neg.close()

In [5]: skip_words = set(['br', 'as', 'an', 'and', 'are', 'at', 'by', 'for', 'has', 'in',
                          'it', 'of', 'on', 'that', 'the', 'there', 'to', 'with', 'they',
                          'this', 'is', 's', 'a', 'be', 'their', 'have', 'was', 'were', 'd',
                          'll', 'he','she', 'his','her','i', 'your', 'or', 'them'])

In [6]: def clean_data(lines):
            ret = [None]*len(lines)
            for index in range(len(lines)):
                line = lines[index].lower()
                temp = re.sub('[^0-9a-zA-Z]+', ' ', line)
                join_string_list = []
```

```
            words = temp.split()
            for word in words:
                if not word in skip_words:
                    join_string_list.append(word)
            ret[index] = ' '.join(join_string_list)
        return ret
```

In [7]:
```
lines_train_pos = clean_data(lines_train_pos)
lines_train_neg = clean_data(lines_train_neg)
lines_test_pos = clean_data(lines_test_pos)
lines_test_neg = clean_data(lines_test_neg)
```

In [8]:
```
def load_set(lines, word_num_set):
    for line in lines:
        words = line.split()
        for word in words:
            if word in skip_words:
                continue
            word_num_set.add(word)
```

In [9]:
```
word_num_map = {}
index_acc = 2 # 0 means padding, 1 means unknown

train_word_set = set()
load_set(lines_train_pos, train_word_set)
load_set(lines_train_neg, train_word_set)

test_word_set = set()
load_set(lines_test_pos, test_word_set)
load_set(lines_test_neg, test_word_set)

all_word_set = set()
for word in train_word_set:
    if word in test_word_set:
        all_word_set.add(word)

for word in test_word_set:
    if word in train_word_set:
        all_word_set.add(word)

for word in all_word_set:
    word_num_map[word] = index_acc
    index_acc = index_acc + 1
```

In [10]:
```
def get_word_number(word):
    if not word in word_num_map:
        return 1
    return word_num_map[word]
```

In [11]:
```
def line_to_num_list(line):
    ret = [0]*400
    words = line.split()
    if len(words) >= 400:
        for i in range(400):
            word = words[i]
            ret[i] = get_word_number(word)
    else:
        index_padding = 400-len(words)
        for i in range(len(words)):
            word = words[i]
            ret[i+index_padding] = get_word_number(word)
    return ret

def word_to_num(lines):
    ret = np.zeros((len(lines), 400), dtype=np.int32)
    for index in range(len(lines)):
        ret[index] = line_to_num_list(lines[index])
    return ret
```

In [12]:
```
array_train_pos = word_to_num(lines_train_pos)
array_train_neg = word_to_num(lines_train_neg)
array_test_pos = word_to_num(lines_test_pos)
array_test_neg = word_to_num(lines_test_neg)
```

In [13]:
```
SIZE = len(word_num_map)
BATCH_SIZE = 50
SIZE
```

Out[13]:
```
17752
```

In [15]:
```
train_x = np.concatenate((array_train_pos, array_train_neg), axis=0)
print(train_x.shape)
train_y = np.concatenate((np.ones(array_train_pos.shape[0]), np.zeros(array_train_neg
print(train_y.shape)
```

```
(3000, 400)
(3000,)
```

In [16]:
```
test_x = np.concatenate((array_test_pos, array_test_neg), axis=0)
print(test_x.shape)
test_y = np.concatenate((np.ones(array_test_pos.shape[0]), np.zeros(array_test_neg.sha
print(test_y.shape)
```

```
(3000, 400)
(3000,)
```

In [17]:
```
train_data = TensorDataset(torch.from_numpy(train_x).type(torch.LongTensor), torch.fr
train_loader = DataLoader(train_data, shuffle=True, batch_size=BATCH_SIZE)
test_data = TensorDataset(torch.from_numpy(test_x).type(torch.LongTensor), torch.from_
test_loader = DataLoader(test_data, shuffle=True, batch_size=BATCH_SIZE)
```

In [18]:
```
optimizer = None
criterion = None

def get_accuracy(predict, target):
    temp1 = predict >= 0.5
    temp2 = target >= 0.5
    temp3 = (temp1.numpy().reshape(BATCH_SIZE) == temp2.numpy().reshape(BATCH_SIZE))
    return np.sum(temp3)/BATCH_SIZE

def train(model, train_loader, test_loader):
    for time in range(10):
        print(time)
        for i, data in enumerate(train_loader):
            inputs, labels = data

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

    print('Finished Training')

    accuracy_sum = 0.0

    for i, data in enumerate(test_loader):
        inputs, label = data

        output = model(inputs)
        loss = criterion(output, label)

        accuracy = get_accuracy(output, label)
        accuracy_sum = accuracy_sum + accuracy
    print("accuracy is " + str(accuracy_sum*BATCH_SIZE/3000))
```

In [19]:
```
class GRU(nn.Module):
    def __init__(self, feature_num):
        super(GRU, self).__init__()
        self.embedding = nn.Embedding(feature_num, 128)
        self.rnn = nn.GRU( input_size=128,
                           hidden_size=64,
                           num_layers=1,
                           dropout=0.5,
```

```
                             batch_first=True)
            self.linear1 = nn.Linear(64, 32)
            self.linear2 = nn.Linear(32, 1)
            self.sigmoid = nn.Sigmoid()

        def forward(self, x):
            embedded = self.embedding(x)
            out, _ = self.rnn(embedded)
            out_last = out[:, -1, :] # get last value

            x = self.linear1(out_last.view(-1, out_last.shape[-1]))
            x = self.linear2(F.relu(x))
            x = self.sigmoid(x)
            return x

    model1 = GRU(SIZE+2)
    criterion = nn.BCELoss()
    optimizer = optim.Adam(model1.parameters(), lr=0.001, betas=(0.9, 0.999))
    train(model1, train_loader, test_loader)
```

```
0


C:\Users\rj369\AppData\Local\Continuum\anaconda3\lib\site-packages\torch\nn\modules\rnn.py:46:
  "num_layers={}".format(dropout, num_layers))
C:\Users\rj369\AppData\Local\Continuum\anaconda3\lib\site-packages\torch\nn\functional.py:2016
  "Please ensure they have the same size.".format(target.size(), input.size()))


1
2
3
4
5
6
7
8
9
Finished Training
accuracy is 0.7470000000000001
```

In [23]:
```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.embedding = nn.Embedding(SIZE+2, 64)

        self.linear1 = nn.Linear(400 * 64, 32)
        self.linear2 = nn.Linear(32, 1)
```

```python
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x)
        x = x.view(-1, 400*64)

        x = self.linear1(x)
        x = self.linear2(x)
        x = self.sigmoid(x)
        return x

model2 = MLP()
criterion = nn.BCELoss()
optimizer = optim.Adam(model2.parameters(), lr=0.0001, betas=(0.9, 0.999))
train(model2, train_loader, test_loader)
```

```
0
1
2
3
4
5
6
7
8
9
Finished Training
accuracy is 0.5706666666666664
```

p2_1

April 26, 2019

```
In [1]: import os
        import matplotlib.pyplot as plt
        import itertools
        import pickle
        import imageio
        import torch
        import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim
        from torchvision import datasets, transforms
        from torch.autograd import Variable
        import torchvision
```

```
In [3]: transform = transforms.Compose([transforms.ToTensor()])
```

```
In [4]: BATCH_SIZE = 128
```

```
In [5]: fmnist = torchvision.datasets.FashionMNIST(root="./", train=True, transform=transform,
        data_loader = torch.utils.data.DataLoader(dataset=fmnist, batch_size=BATCH_SIZE, shuffl
```

```
In [6]: torch.cuda.is_available()
```

```
Out[6]: True
```

```
In [7]: class generator(nn.Module):
            def __init__(self):
                super(generator, self).__init__()

                self.layer0 = nn.Sequential(
                    nn.Linear(100, 256),
                    nn.ReLU(),
                )

                self.layer1 = nn.Sequential(
                    nn.Linear(256, 512),
                    nn.ReLU(),
                )
```

```
        self.layer2 = nn.Sequential(
            nn.Linear(512, 1024),
            nn.ReLU(),
        )

        self.layer3 = nn.Sequential(
            nn.Linear(1024, 784),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.layer0(x)
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        return x
```

In [8]:
```
class discriminator(nn.Module):
    def __init__(self):
        super(discriminator, self).__init__()

        self.layer0 = nn.Sequential(
            nn.Linear(784, 1024),
            nn.ReLU(),
            nn.Dropout(0.3)
        )

        self.layer1 = nn.Sequential(
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.3)
        )

        self.layer2 = nn.Sequential(
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.3)
        )

        self.layer3 = nn.Sequential(
            nn.Linear(256, 1),
            nn.Sigmoid()
        )


    def forward(self, x):
        x = self.layer0(x)
        x = self.layer1(x)
```

```
            x = self.layer2(x)
            x = self.layer3(x)
            return x
```

In [9]:
```
def show_result():
    noise = Variable(torch.randn((5*5, 100)).cuda())

    G.eval()
    test_images = G(noise)
    G.train()

    size_figure_grid = 5
    fig, ax = plt.subplots(size_figure_grid, size_figure_grid, figsize=(5, 5))
    for i, j in itertools.product(range(size_figure_grid), range(size_figure_grid)):
        ax[i, j].get_xaxis().set_visible(False)
        ax[i, j].get_yaxis().set_visible(False)

    for k in range(5*5):
        i = k // 5
        j = k % 5
        ax[i, j].cla()
        ax[i, j].imshow(test_images[k, :].cpu().data.view(28, 28).numpy(), cmap='gray'

    label = ""
    fig.text(0.5, 0.04, label, ha='center')
    plt.show()
```

In [10]:
```
batch_size = 128
lr = 0.0002
epoch_number = 50
```

In [11]:
```
G = generator()
D = discriminator()
G.cuda()
D.cuda()

criterion = nn.BCELoss()

G_optimizer = optim.Adam(G.parameters(), lr=lr)
D_optimizer = optim.Adam(D.parameters(), lr=lr)
```

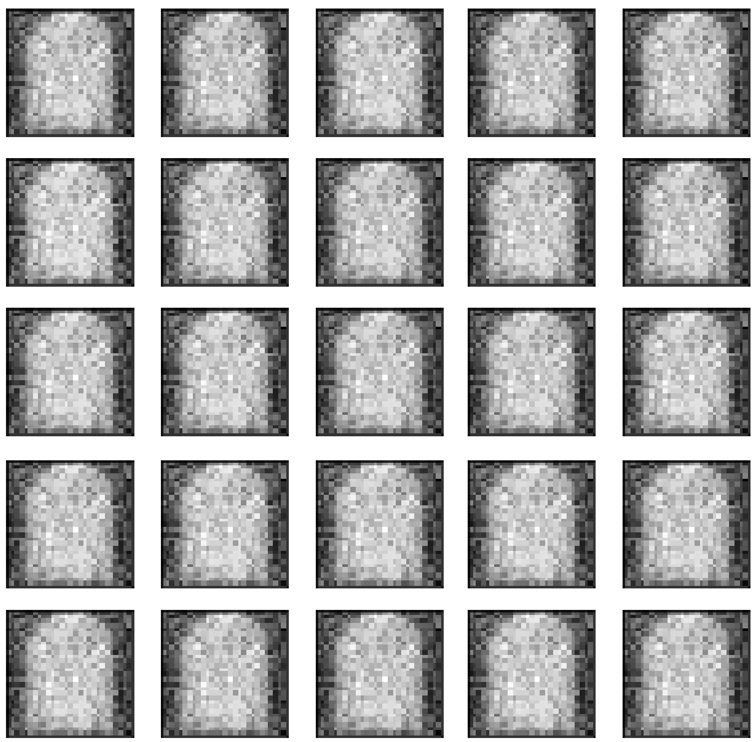
In [12]:
```
temp_index = 0

D_loss = []
G_loss = []

# train
for epoch in range(epoch_number):
    d_epoch_loss = []
```

```
    g_epoch_loss = []
    for x, _ in data_loader:
#         print(temp_index)
#         temp_index = temp_index + 1
#         if (temp_index>= 5000):
#             break

        x = x.view(-1, 28 * 28)
        batch_size = x.size()[0]
        # train D
        y_real = torch.ones(batch_size)
        y_fake = torch.zeros(batch_size)
        x, y_real, y_fake = Variable(x.cuda()), Variable(y_real.cuda()), Variable(y_fa

        # get real image
        D.zero_grad()
        D_result_real = D(x)
        D_real_loss = criterion(D_result_real, y_real)

        # get fake image
        noise = Variable(torch.randn((batch_size, 100)).cuda())
        G_result = G(noise)
        D_result_fake = D(G_result)
        D_fake_loss = criterion(D_result_fake, y_fake)

        D_total_loss = D_real_loss + D_fake_loss

        D_total_loss.backward()
        D_optimizer.step()

        d_epoch_loss.append(D_total_loss.cpu().data.item())

        # train generator G
        noise = torch.randn((batch_size, 100))
        y_target = torch.ones(batch_size)
        noise, y_target = Variable(noise.cuda()), Variable(y_target.cuda())

        G.zero_grad()
        G_result = G(noise)
        D_result = D(G_result)

        G_train_loss = criterion(D_result, y_target)
        G_train_loss.backward()
        G_optimizer.step()

        g_epoch_loss.append(G_train_loss.cpu().data.item())
```

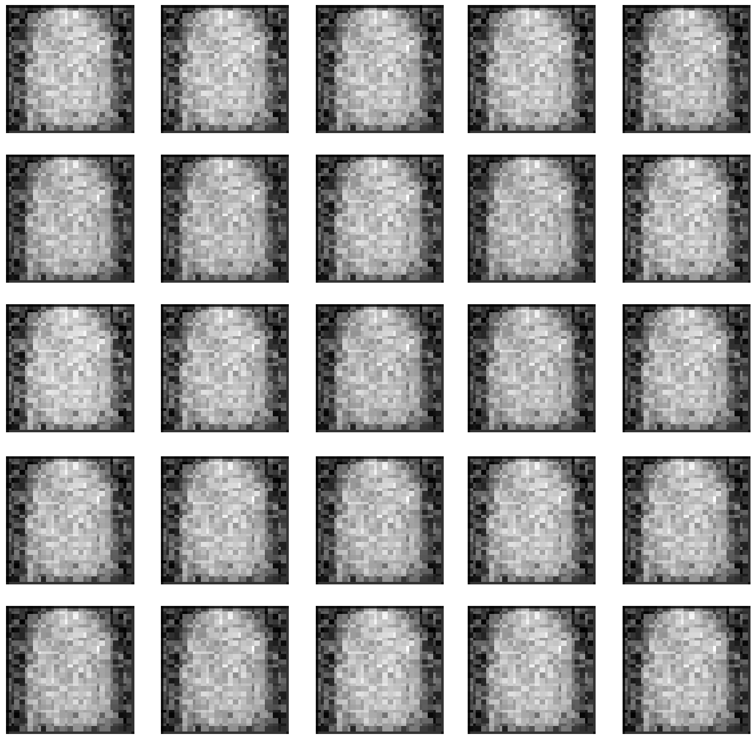
```
            D_loss.append(sum(d_epoch_loss)/len(d_epoch_loss))
            G_loss.append(sum(g_epoch_loss)/len(g_epoch_loss))

            if (epoch == 0 or epoch == 1 or epoch == 2 or epoch == 10 or epoch == 20 or epoch
                print('epoch %d: loss_d: %.3f, loss_g: %.3f' % (
                    epoch, D_loss[epoch], G_loss[epoch]))
                show_result()
```

```
C:\Users\rj369\AppData\Local\Continuum\anaconda3\lib\site-packages\torch\nn\functional.py:2016
  "Please ensure they have the same size.".format(target.size(), input.size()))
C:\Users\rj369\AppData\Local\Continuum\anaconda3\lib\site-packages\torch\nn\functional.py:2016
  "Please ensure they have the same size.".format(target.size(), input.size()))
```

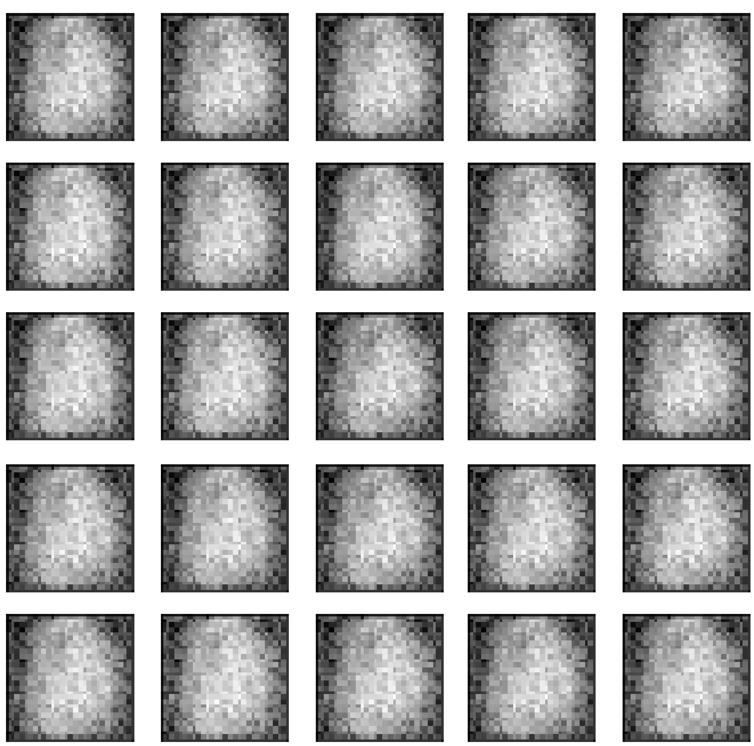
```
epoch 0: loss_d: 0.427, loss_g: 4.058
```



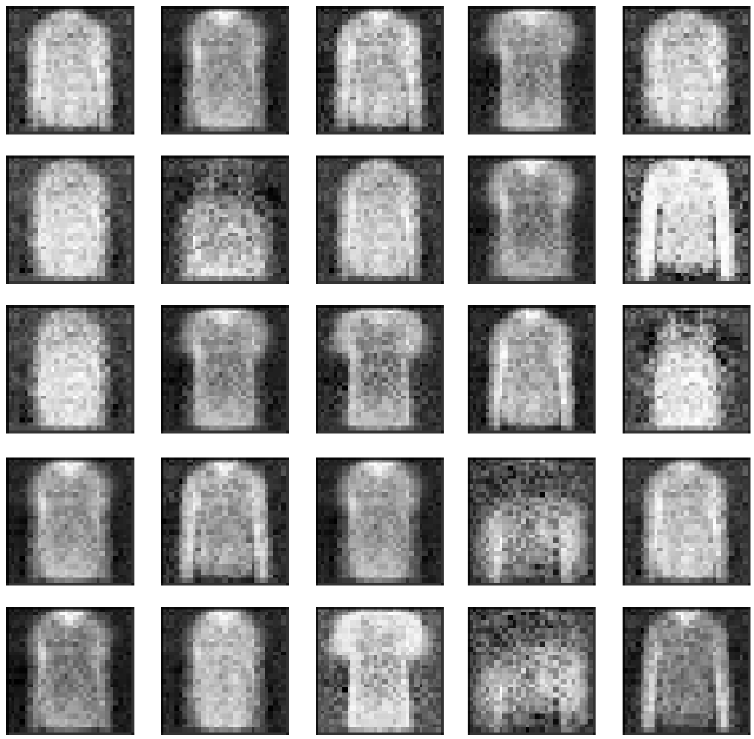
```
epoch 1: loss_d: 0.512, loss_g: 3.512
```



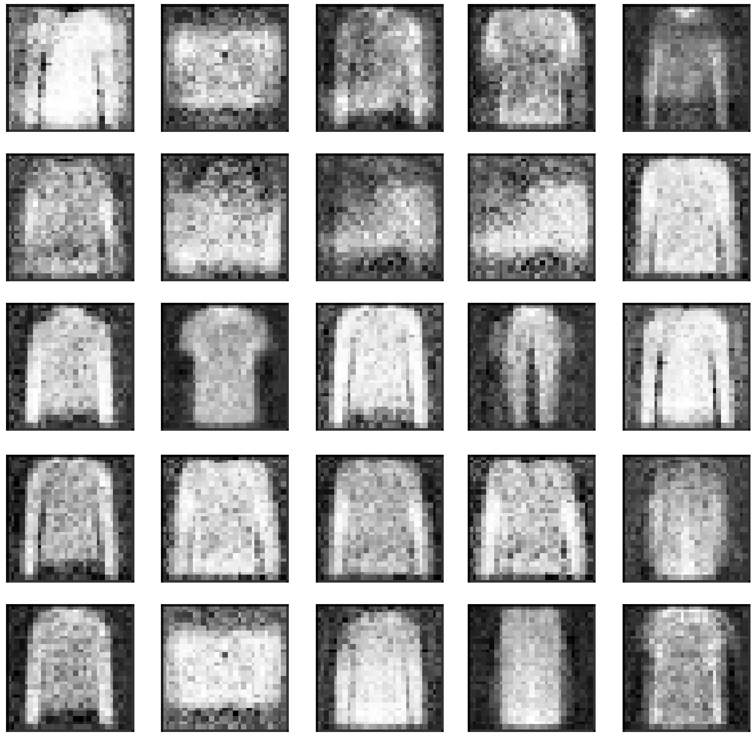
```
epoch 2: loss_d: 0.855, loss_g: 2.487
```



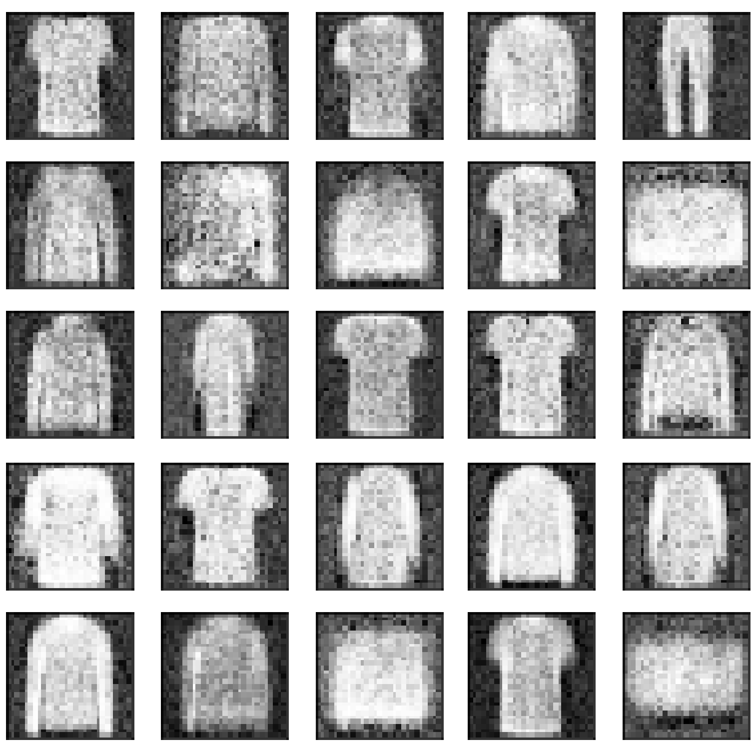
```
epoch 10: loss_d: 0.496, loss_g: 2.563
```



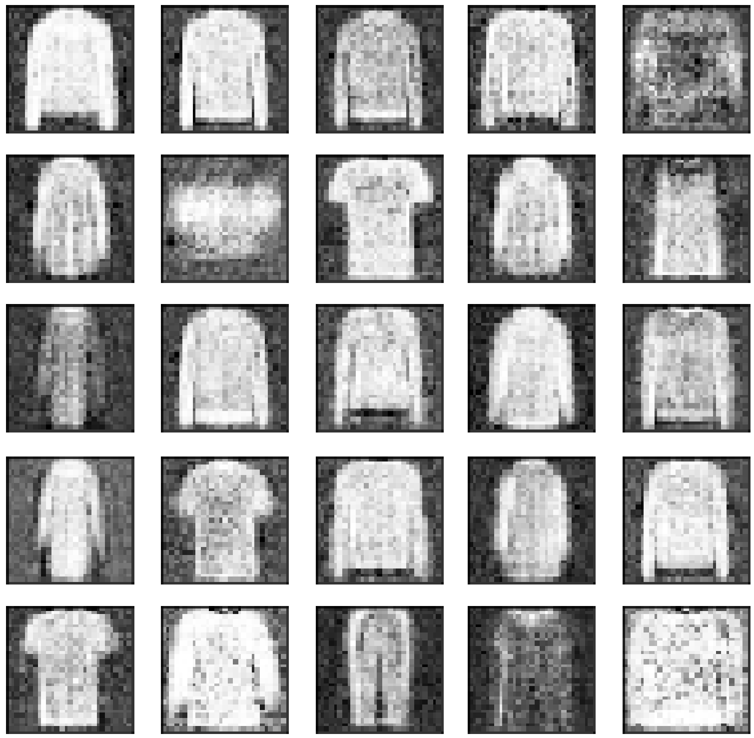
```
epoch 20: loss_d: 0.748, loss_g: 1.845
```



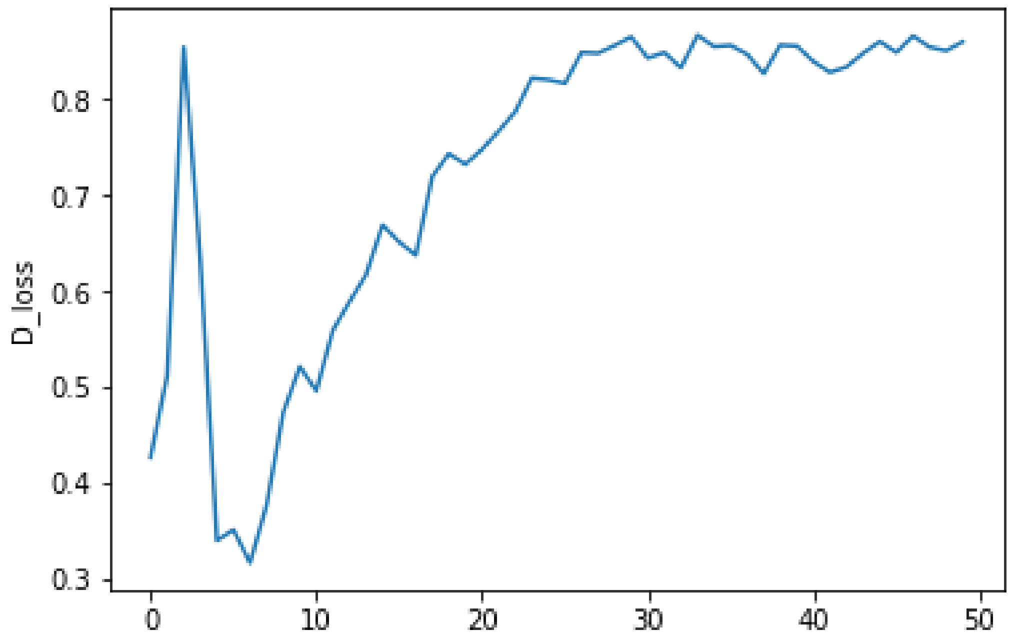
```
epoch 30: loss_d: 0.843, loss_g: 1.565
```



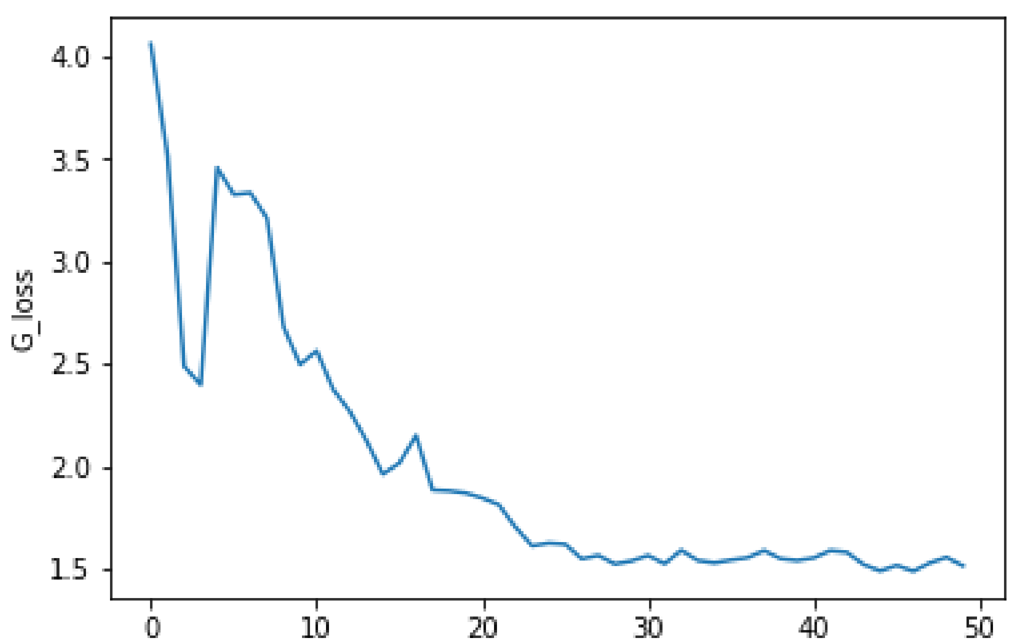
```
epoch 48: loss_d: 0.851, loss_g: 1.556
```



```
In [16]: plt.plot(D_loss)
         plt.ylabel("D_loss")
         plt.show()
```



In [17]:
```
plt.plot(G_loss)
plt.ylabel("G_loss")
plt.show()
```



In [13]: D_loss

```
Out[13]: [0.42667620823279756,
 0.5123352119282111,
 0.8547294700164785,
 0.6313243359327316,
 0.34006550007346853,
 0.35136317059810734,
 0.31668396443446306,
 0.37721978458387256,
 0.47344570131952574,
 0.521587454179711,
 0.4962965416183858,
 0.5593825981243333,
 0.5891309815810434,
 0.617374784466046,
 0.6692556641630526,
 0.6515478943583808,
 0.6376994784071501,
 0.7199377847759962,
 0.7436438177440212,
 0.7327160112766314,
 0.748040123153597,
 0.7669215342129218,
 0.7873478012044293,
 0.8224746010450921,
 0.8207837517327591,
 0.8170455678312509,
 0.8493734579096471,
 0.8485193214436838,
 0.856657028325331,
 0.8655583880095086,
 0.8434241771189643,
 0.8493106560920601,
 0.8333091458786271,
 0.8673819196757986,
 0.8554345772210469,
 0.8567973292712718,
 0.8473844689601011,
 0.8272574177937213,
 0.8570642065900221,
 0.8560907423877513,
 0.8398703744670729,
 0.8285451404321422,
 0.8341935348154893,
 0.8485930802217171,
 0.8608230607850211,
 0.8492864325864992,
```

```
          0.8666080279645126,
          0.8549946855380337,
          0.8510942226533951,
          0.8604613430718623]
```

```
In [14]: G_loss
```

```
Out[14]: [4.057948192680822,
          3.511917046106446,
          2.4871945935509987,
          2.3969935481228046,
          3.455103845738653,
          3.3239485223664405,
          3.332899636042906,
          3.209001996623936,
          2.680175736514744,
          2.4953902281169444,
          2.5629102411046465,
          2.3741609330878837,
          2.2674054733471576,
          2.1256805762553266,
          1.9607102931943783,
          2.0158261739368886,
          2.1490673360539905,
          1.8836762376431464,
          1.879621808462814,
          1.86888105554113,
          1.844722063556663,
          1.8119306061059428,
          1.701764895463549,
          1.6111202989814124,
          1.6247181546713498,
          1.6190725427700767,
          1.5484165171824538,
          1.565328908881653,
          1.5233362430194293,
          1.5384719221830876,
          1.565154656926706,
          1.5246656568828167,
          1.5903300628987456,
          1.5407317116824804,
          1.5289575062326786,
          1.542803963618492,
          1.5541048260894157,
          1.588827984927814,
          1.549421217904162,
          1.5405469515176216,
          1.5532067010143418,
```

```
1.588576265997978,
1.5806053726912053,
1.520864486694336,
1.489760377259651,
1.5167688527849437,
1.4884990268170453,
1.5286537302074148,
1.5562664598290092,
1.5147652374401783]
```