Homework 3
CS 5787 Deep Learning
Spring 2020

Due: See Canvas
Name: Kushal Singh
NetID: ks2377
Worked with: Brian Cheang, Gyan Suri

Your homework submission must cite any references used (including articles, books, code, websites, and personal communications). All solutions must be written in your own words, and you must program the algorithms yourself. **If you do work with others, you must list the people you worked with.** Submit your solutions as a PDF to Canvas.

Your homework solution must be typed. We urge you to prepare it in LaTeX. It must be output to PDF format. To use LaTeX, we suggest using `http://overleaf.com`, which is free and can be accessed online.

Your programs must be written in Python. The relevant code to the problem should be in the PDF you turn in. If a problem involves programming, then the code should be shown as part of the solution to that problem. One easy way to do this in LaTeX is to use the verbatim environment, i.e., \begin{verbatim} YOUR CODE \end{verbatim}. For this assignment, you may use the plotting toolbox of your choice, PyTorch, and NumPy.

If told to implement an algorithm, don't use a toolbox, or you will receive no credit.

Problem 0 - Recurrent Neural Networks (10 points)

Recurrent neural networks (RNNs) are universal Turing machines as long as they have enough hidden units. In the next homework assignment we will cover using RNNs for large-scale problems, but in this one you will find the parameters for an RNN that implements binary addition. Rather than using a toolbox, you will find them by hand.

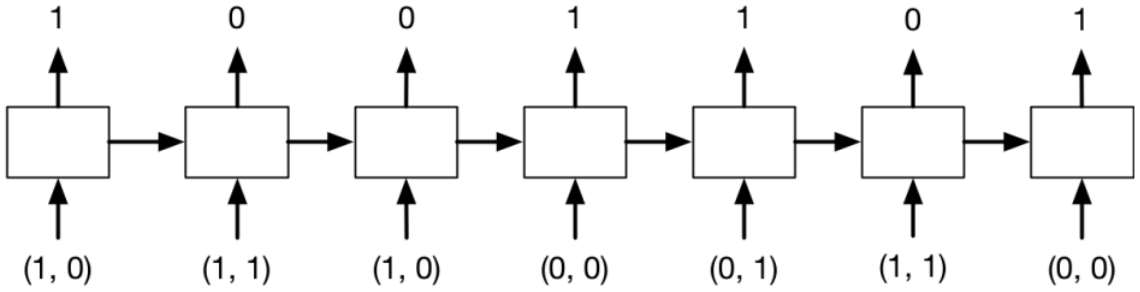
The input to your RNN will be two binary numbers, starting with the *least* significant bit. You will need to pad the largest number with an additional zero on the left side and you should make the other number the same length by padding it with zeros on the left side. For instance, the problem

$$100111 + 110010 = 1011001$$

would be input to your RNN as:

- Input 1: 1, 1, 1, 0, 0, 1, 0
- Input 2: 0, 1, 0, 0, 1, 1, 0
- Correct output: 1, 0, 0, 1, 1, 0, 1

The RNN has two input units and one output unit. In this example, the sequence of inputs and outputs would be:



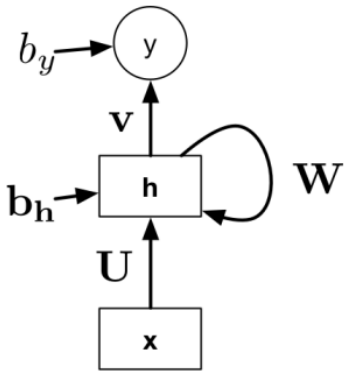
The RNN that implements binary addition has three hidden units, and all of the units use the following non-differentiable hard-threshold activation function

$$\sigma(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases}$$

The equations for the network are given by

$$y_t = \sigma\left(\mathbf{v}^T \mathbf{h}_t + b_y\right)$$
$$\mathbf{h}_t = \sigma\left(\mathbf{U}\mathbf{x}_t + \mathbf{W}\mathbf{h}_{t-1} + \mathbf{b_h}\right)$$

where $\mathbf{x}_t \in \mathbb{R}^2$, $\mathbf{U} \in \mathbb{R}^{3\times 2}$, $\mathbf{W} \in \mathbb{R}^{3\times 3}$, $\mathbf{b_h} \in \mathbb{R}^3$, $\mathbf{v} \in \mathbb{R}^3$, and $b_y \in \mathbb{R}$



Part 1 - Finding Weights

Before backpropagation was invented, neural network researchers using hidden layers would set the weights by hand. Your job is to find the settings for all of the parameters by hand, including the value of $\mathbf{h}_0$. Give the settings for all of the matrices, vectors, and scalars to correctly implement binary addition.

Hint: Have one hidden unit activate if the sum is at least 1, one hidden unit activate if the sum is at least 2, and one hidden unit if it is 3.

Solution:
Using the hint above, we will have the first hidden unit activate if the sum is at least 3, the second hidden unit activate if the sum is at least 2, and the third hidden unit activate if the sum is at most 0.

$$\mathbf{U} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ -1 & -1 \end{bmatrix}, \mathbf{W} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 0 \end{bmatrix}, b_h = \begin{bmatrix} -3 \\ -1 \\ 1 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, b_y = \begin{bmatrix} 1 \end{bmatrix}, h_0 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Problem 1 - GRU for Sentiment Analysis

In this problem you will use a popular RNN model called the Gated Recurrent Units (GRU) to learn to predict the sentiment of a sentence. The dataset we are using is the IMDB review dataset (link). It is a binary sentiment classification (positive or negative) dataset that contains 50,000 movie reviews (50% for training and 50% for testing). We provide four text files for you to download on Canvas: train_pos_reviews.txt, train_neg_reviews.txt, test_pos_reviews.txt, test_neg_reviews.txt. Each line is an independent review for a movie.

Put your code in the appendix.

Part 1 - Preprocessing (5 points)

First you need to do proper preprocessing of the sentences so that each word is represented by a single number index in a vocabulary.

Remove all punctuation from the sentences. Build a vocabulary from the unique words collected from text file so that each word is mapped to a number.

Now you need to convert the data to a matrix where each row is a sentence. Because sentences are of different length, you need to pad or truncate the sentences to make them same length. We are going to use 400 as the fixed length in this problem. That means any sentence that is longer than 400 words will be truncated; any sentence that is shorter than 400 words will be padded with 0s. Please note that your padded 0s should be placed *before* the sentences.

After you prepare the data, you can define a standard PyTorch dataloader directly from numpy arrays (say you have data in train_x and labels in train_y).

```
train_data = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y))
train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
```

Implement the data preprocessing procedure.

Solution: ****Full code segment can be found in the appendix section titled **Problem 1, Part 1******

Part 2 - Build A Binary Prediction RNN with GRU (10 points)

Your RNN module should contain the following modules: a word embedding layer, a GRU, and a prediction unit.

1. You should use nn.Embedding layer to convert an input word to an embedded feature vector.

2. Use nn.GRU module. Feel free to choose your own hidden dimension. It might be good to set the batch_first flag to True so that the GRU unit takes (batch, seq, embedding_dim) as the input shape.

3. The prediction unit should take the output from the GRU and produce a number for this binary prediction problem. Use nn.Linear and nn.Sigmoid for this unit.

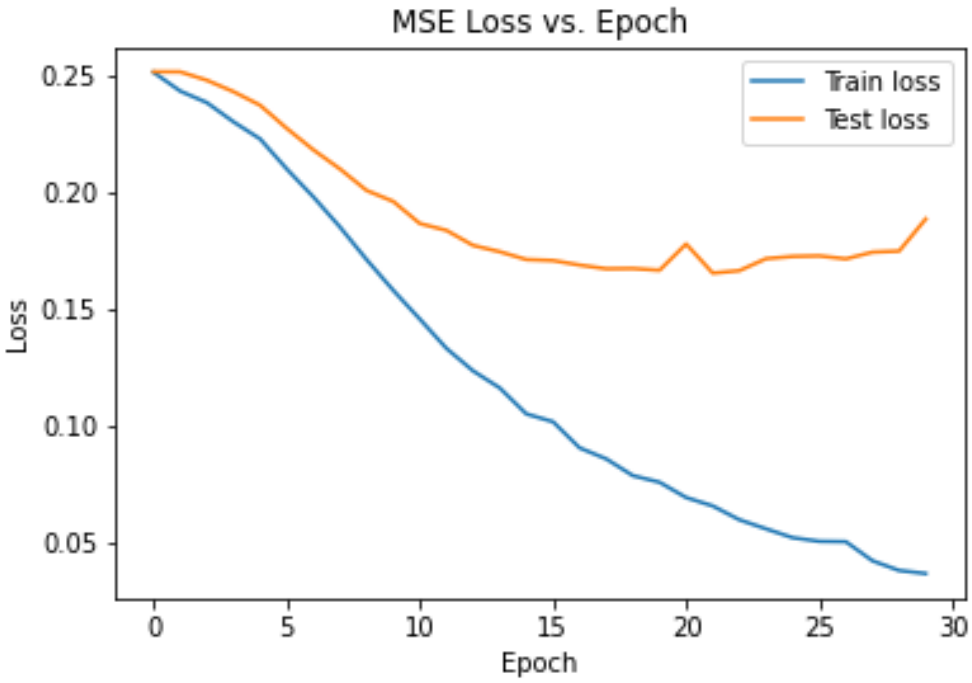
At a high level, the input sequence is fed into the word embedding layer first. Then, the GRU is taking steps through each word embedding in the sequence and return output /

feature at each step. The prediction unit should take the output from the final step of the GRU and make predictions.

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

Solution: ****Full code segment can be found in the appendix section titled **Problem 1, Part 2******

The binary prediction RNN with GRU was trained on 30 epochs and achieved a test accuracy of 74%. The final MSE loss over the train set was close to 0.01, and the final MSE loss over the test set was close to 0.20.



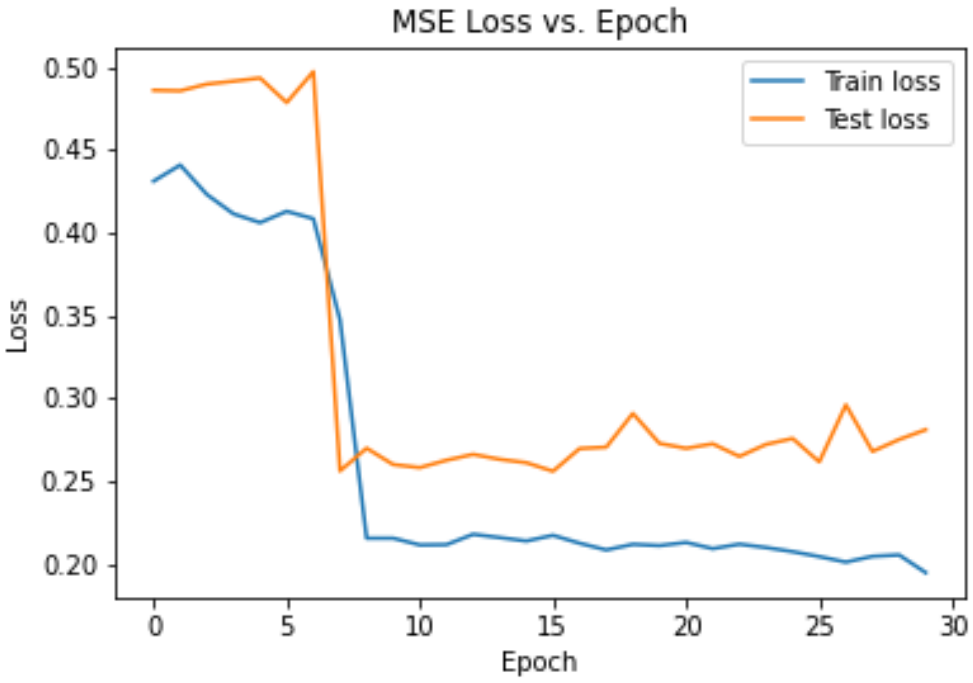
Part 3 - Comparison with a MLP (5 points)

Since each sentence is a fixed length input (with potentially many 0s in some samples), we can also train a standard MLP for this task.

Train a two layer MLP on the training data and report accuracy on the test set. How does it compare with the result from your RNN model?

Solution:
****Full code segment can be found in the appendix section titled **Problem 1, Part 3******

The MLP was trained on 30 epochs and achieved a test accuracy of 50%. The final MSE loss over the train set was close to 0.20, and the final MSE loss over the test set was close to 0.25. Comparing the two models, we can see that the MLP performed worse on the test set than the GRU. However, for the MLP, the train loss more closely mirrored the test loss, whereas for the GRU, the train loss was markedly less than the test loss.



Problem 2 - Generative Adversarial Networks

For this problem, you will be working with Generative Adversarial Networks (GAN) on Fashion-MNIST dataset (Figure 1).

Fashion-MNIST dataset can be loaded directly in PyTorch by the following command:

```
import torchvision
fmnist = torchvision.datasets.FashionMNIST(root="./", train=True,
transform=transform, download=True)
data_loader = torch.utils.data.DataLoader(dataset=fmnist,
batch_size=batch_size, shuffle=True)
```

Similar to the well known MNIST dataset, Fashion-MNIST is designed to be a standard testbed for ML algorithms. It has the same image size and number of classes as MNIST, but is a little bit more difficult.

Figure 1: Fashion-Mnist dataset example images. It contains 10 classes of cloths, shoes, and bags.

We are going to train GANs to generate images that looks like those in Fashion-MNIST dataset. Through the process, you will have a better understanding on GANs and their characteristics.

Training a GAN is notoriously tricky, as we shall see in this problem.

Put your code in the appendix.

Part 1 - Vanilla GAN (10 points)

A GAN is containing a Discriminator model (D) and a Generator model (G). Together they are optimized in a two player minimax game:

$$\begin{aligned}\min_D &= -\mathbb{E}_{x\in p_d} \log D(x) - \mathbb{E}_{z\in p_z} \log(1 - D(G(z)))\\ \min_G &= -\mathbb{E}_{z\in p_z} \log D(G(z))\end{aligned}$$

In practice, a GAN is trained in an iterative fashion where we alternate between training G and training D. In pseudocode, GAN training typically looks like this:

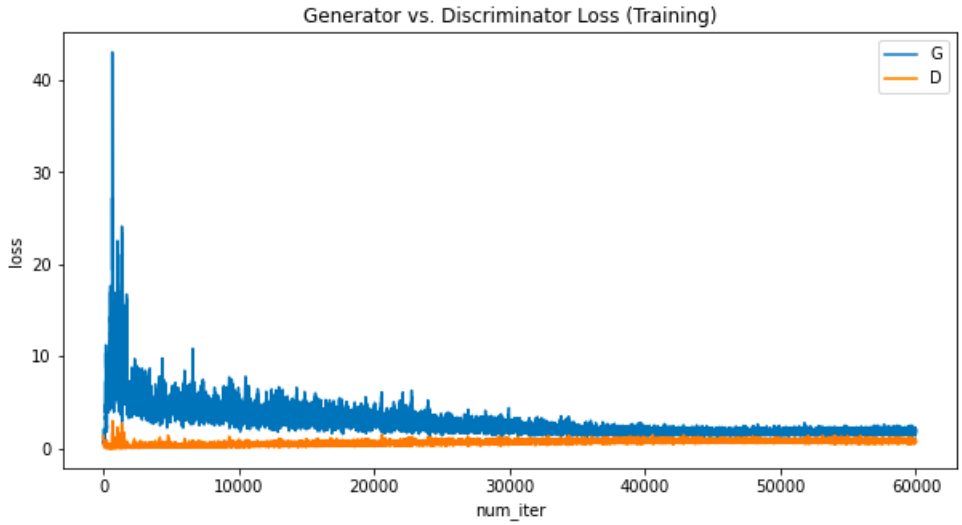
```
For epoch 1:max_epochs
    Train D:
        Get a batch of real images
        Get a batch of fake samples from G
        Optimize D to correctly classify the two batches

    Train G:
        Sample a batch of random noise
        Generate fake samples using the noise
        Feed fake samples to D and get prediction scores
        Optimize G to get the scores close to 1 (means real samples)
```

Choice of G architecture:
Make your generator to be a simple network with three linear hidden layers with ReLU activation functions. For the output layer activation function, you should use hyperbolic tangent (tanh). This is typically used as the output for the generator because ReLU cannot output negative values.

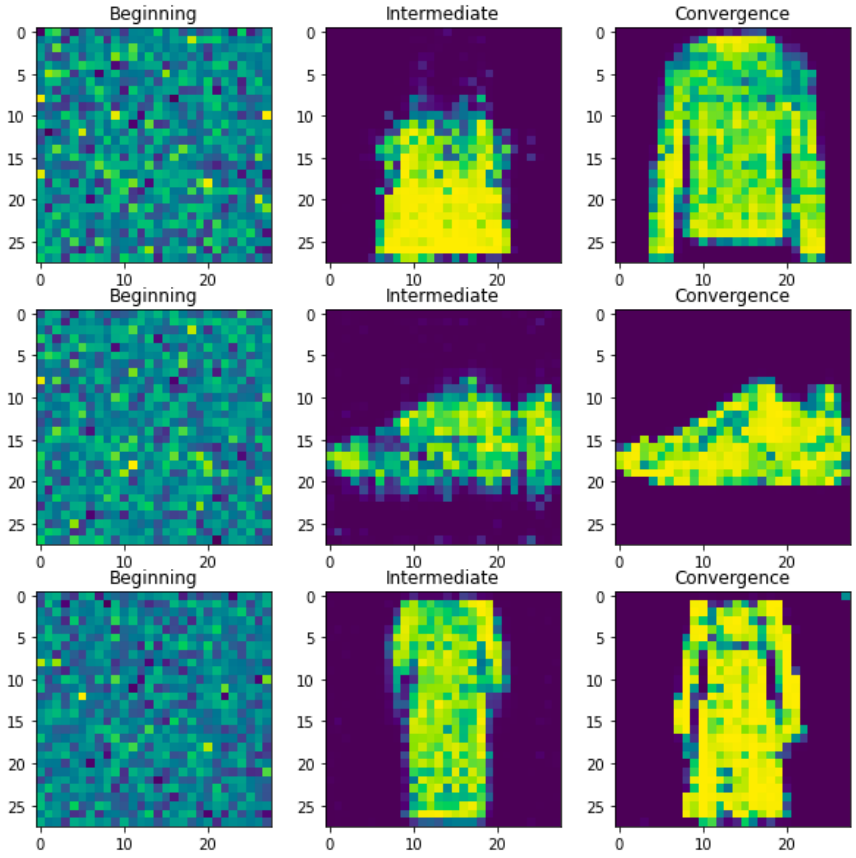
Choice of D architecture:
Make your discriminator to be a similar network with three linear hidden layers using ReLU activation functions, but the last layer should have a logistic sigmoid as its output activation function, since it the discriminator D predicts a score between 0 and 1, where 0 means fake and 1 means real.

Train a basic GAN that can generate images from the Fashion-MNIST dataset. Plot your training loss curves for your G and D. Show the generated samples from G in 1) the beginning of the training; 2) intermediate stage of the training; and 3) after convergence.

Solution:




```
<Figure size 432x288 with 0 Axes>
```



Part 2 - GAN Loss (10 points)

In this part, we are going to modify the model you just created in order to compare different choices of losses in GAN training.

MSE

$$\min_G \quad \mathbb{E}_{z\in p_z, x\in p_d}(x - G(z))^2$$

You can get rid of the discriminator and directly use a MSE loss to train the generator.

Wasserstein GAN (WGAN)

$$\begin{aligned}\min_D \quad & -\mathbb{E}_{x\in p_d} D(x) + \mathbb{E}_{z\in p_z} D(G(z)) \\ \min_G \quad & -\mathbb{E}_{z\in p_z} D(G(z))\end{aligned}$$

WGAN is proposed to address the vanishing gradient problem in the original GAN loss when the discriminator is way ahead of the generator. One thing to change in WGAN is that the output of the discriminator should be now 'unbounded', namely you need to remove the sigmoid function at the output layer. And you need to clip the weights of the discriminator so that their L_1 norm is not bigger than c.

Try c from the set $\{0.1, 0.01, 0.001, 0.0001\}$ and compare their difference.

Least Square GAN

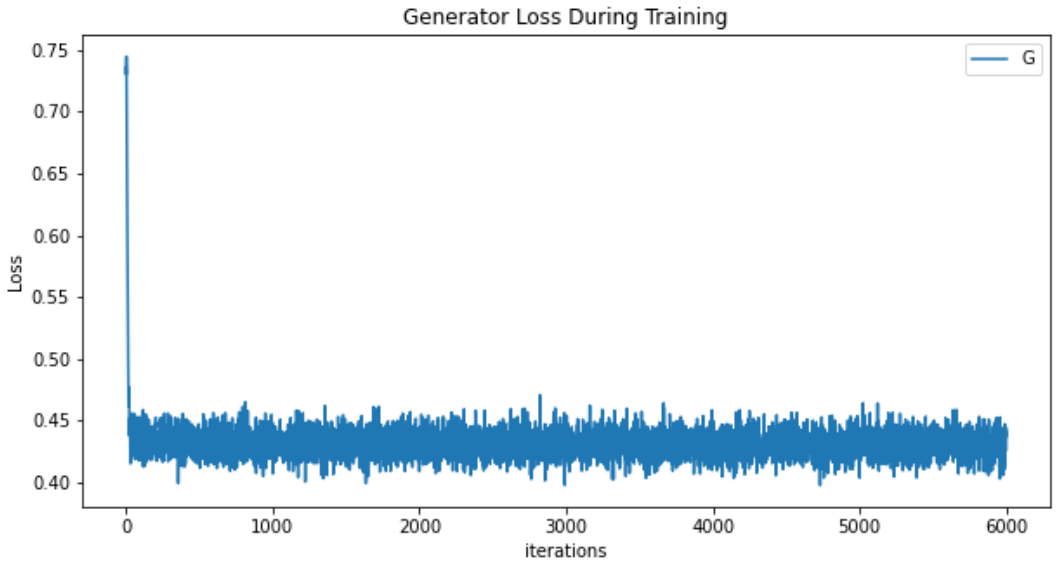
$$\begin{aligned}\min_D \quad & \mathbb{E}_{x\in p_d}(D(x) - 1)^2 + \mathbb{E}_{z\in p_z} D(G(z))^2 \\ \min_G \quad & \mathbb{E}_{z\in p_z}(D(G(z)) - 1)^2\end{aligned}$$

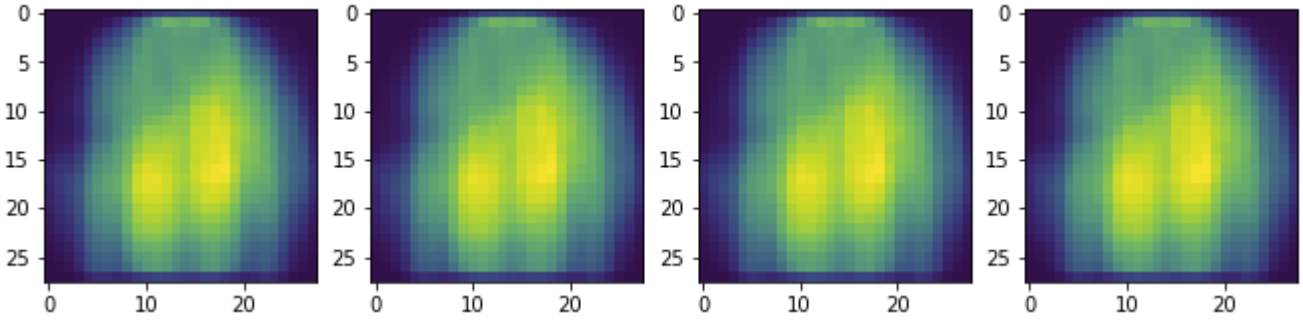
The idea is to provide a smoother loss surface than the original GAN loss.

Plot training curves and show generated samples of the above mentioned losses. Discuss if you find there is any difference in training speed and generated sample's quality.

Solution:
MSE

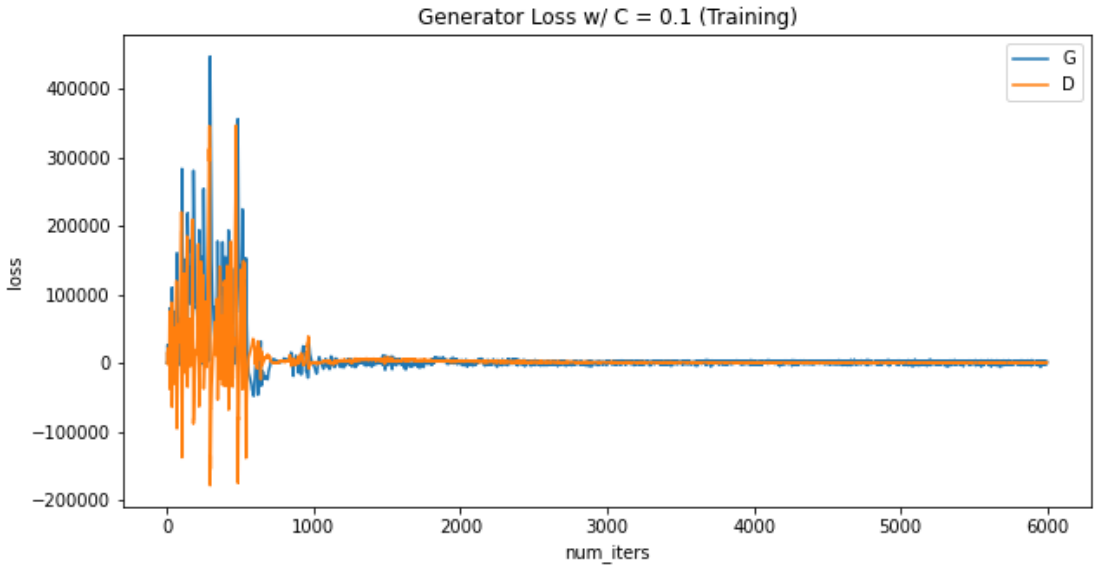
From the graph below, comparing number of iterations with the loss, we can see that the MSE loss converged quickly within 10 epochs. From the images below, however, we can see that the image quality is not as great, since the model only minimized MSE of images against the training data.

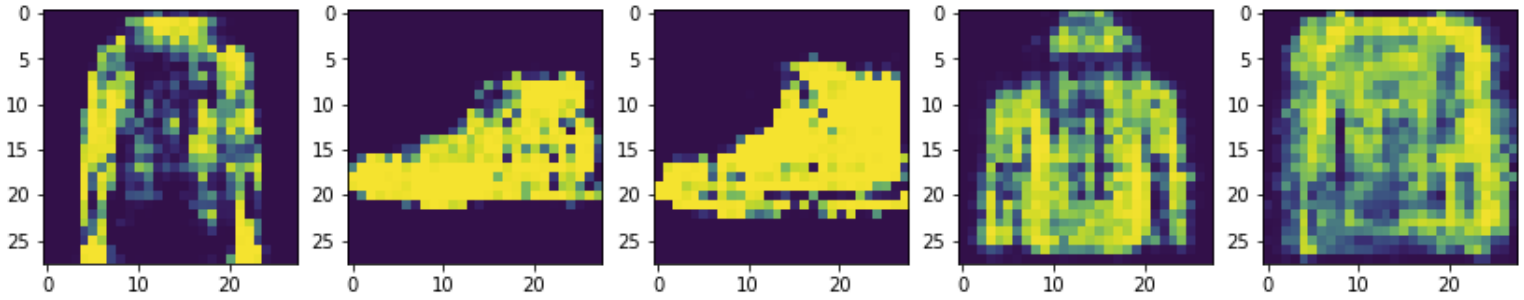


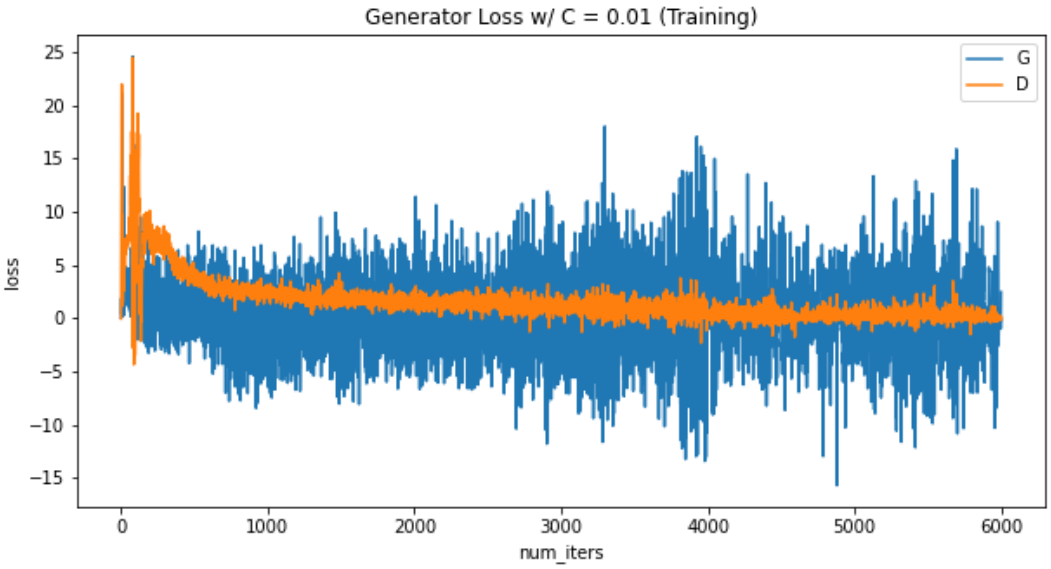


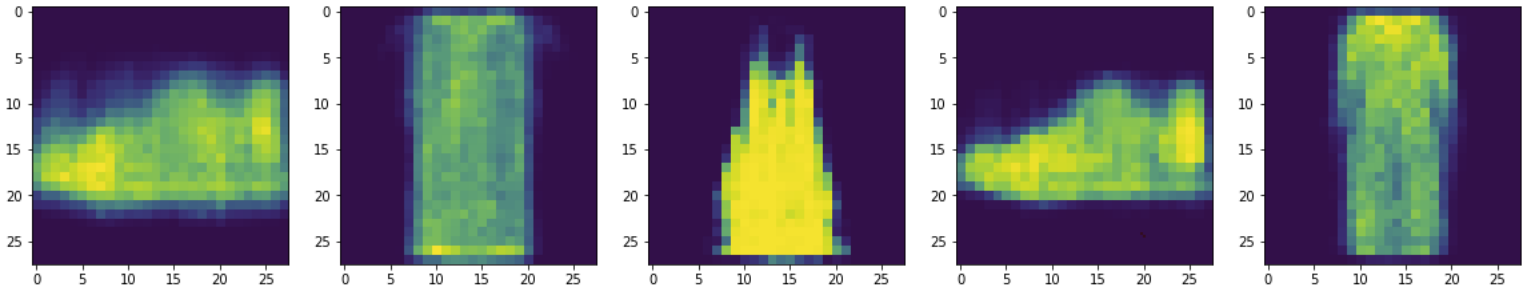
WGAN

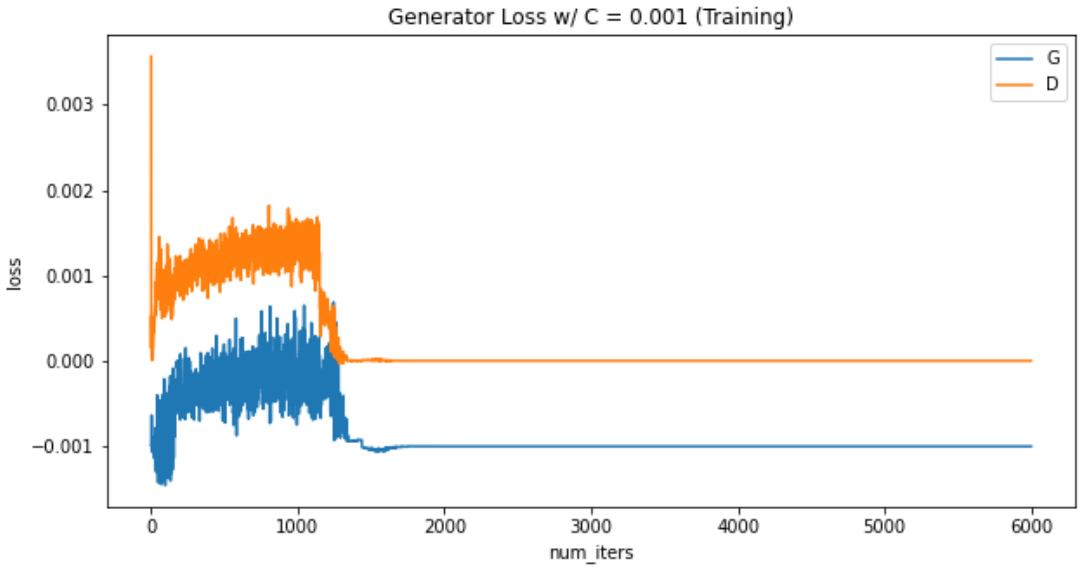
From the graph below, we can see that WGAN converged quickly, but took more epochs than the MSE loss above. Overall, as c decreased in value, the quality and sharpness of the images also decreased.

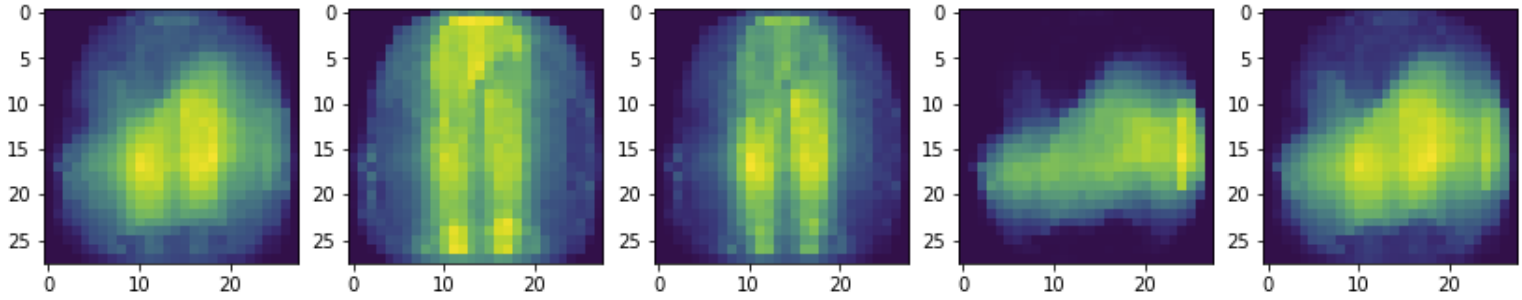


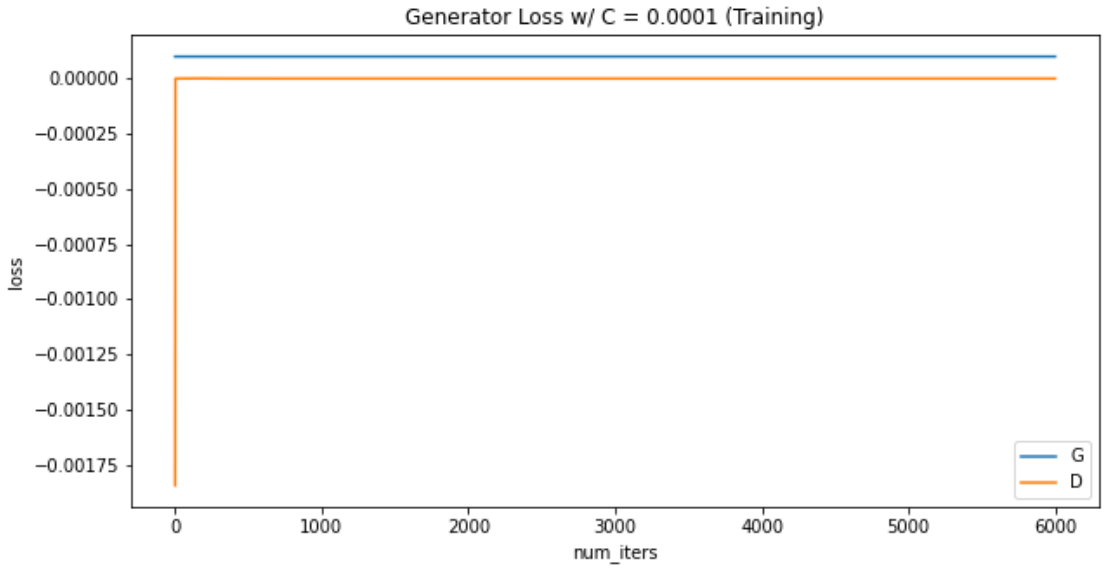


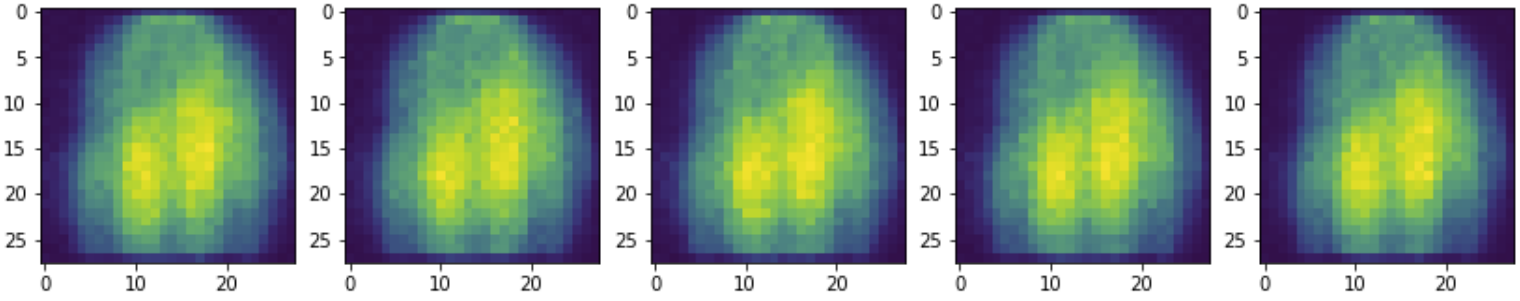

Generator Loss w/ C = 0.01 (Training)
G
D
loss
num_iters
25
20
15
10
5
0
−5
−10
−15
0
1000
2000
3000
4000
5000
6000



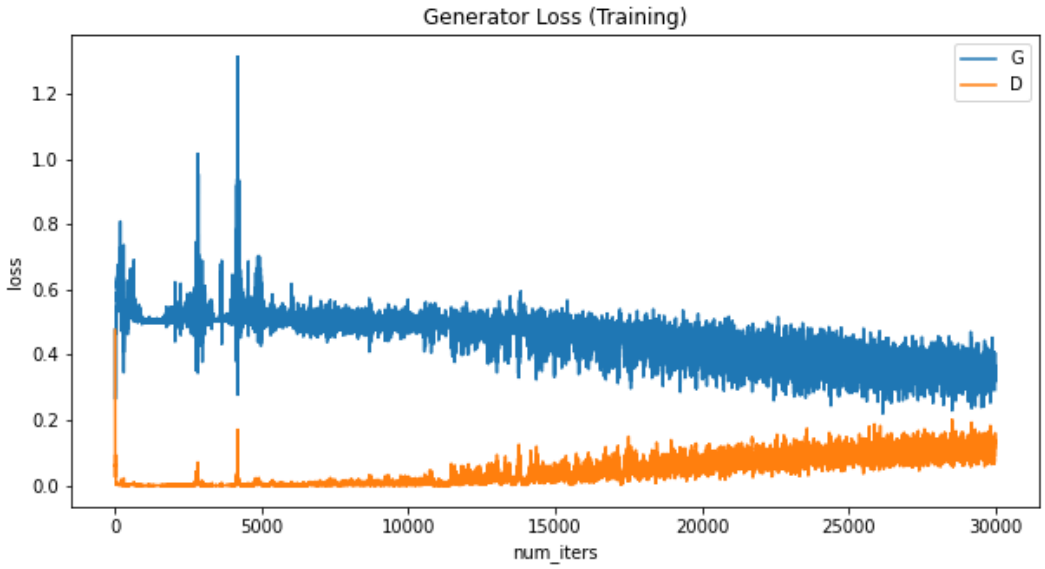

Generator Loss w/ C = 0.001 (Training)
G
D
0.003
0.002
0.001
0.000
−0.001
loss
0
1000
2000
3000
4000
5000
6000
num_iters

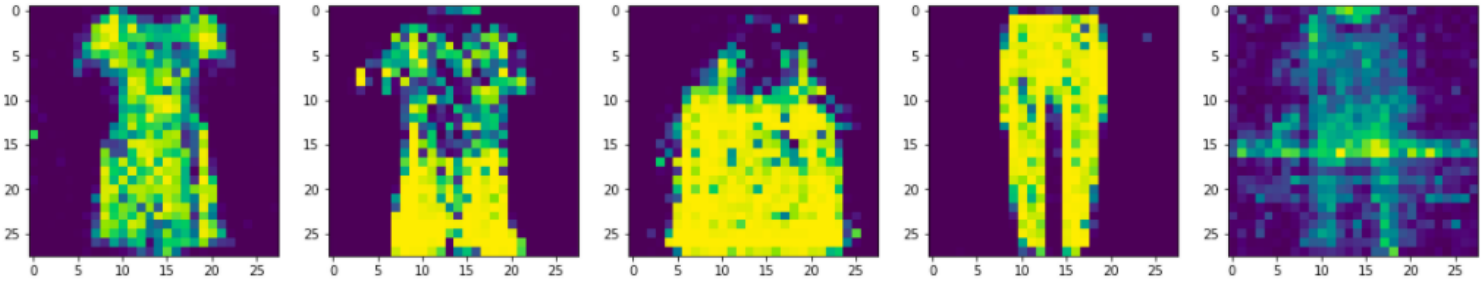

0
5
10
15
20
25
0
10
20


Generator Loss w/ C = 0.0001 (Training)
0.00000
−0.00025
−0.00050
−0.00075
−0.00100
−0.00125
−0.00150
−0.00175
loss
0
1000
2000
3000
4000
5000
6000
num_iters
G
D



Least Squares GAN
The Least Squares GAN took nearly 50 epochs to converge, but the generated images were of fair quality.





Part 3 - Mode Collapse in GANs (10 points)

Take a copy of your vanilla GAN discriminator and change its output channel from 1 output to 10 output units. Fine-tune it as a classifier on the Fashion-MNIST training set. You should easily achieve $\sim 90\%$ accuracy on Fashion-MNIST test set.

Now generate 3000 samples using the generator you trained for Part 1. Use the classifier you just trained to predict the class labels of those samples. Plot the histogram of predicted labels.

Although the original Fashion-MNIST dataset has 10 classes equally distributed, you will find the histogram you just generated is not close to uniform (even if we consider the classifier is not perfect and 3000 samples are not too large). This is a known issue with GAN called Mode Collapse. It means the GAN is often capturing only a subset (mode) of the original data's distribution, not all of them.

Unrolled GAN is proposed to reduce the effect of mode collapse in GAN training. The intuition is that if we let G see ahead how D would change in the next k steps, G can adjust accordingly and hopefully will perform better. Its idea can be summarized in the following modified training scheme:

```
For epoch 1:max_epochs
    Train D:
        Get a batch of real images
        Get a batch of fake samples from G
        Optimize D to correctly classify the two batches

    Make a copy of D into D_unroll
    Train D for k unrolled steps:
        Get a batch of real images
        Get a batch of fake samples from G
        Optimize D_unroll to correctly classify the two batches

    Train G:
        Sample a batch of random noise
        Generate fake samples using the noise
        Feed fake samples to D_unroll and get prediction scores
        Optimize G to get the scores close to 1 (means real samples)
```

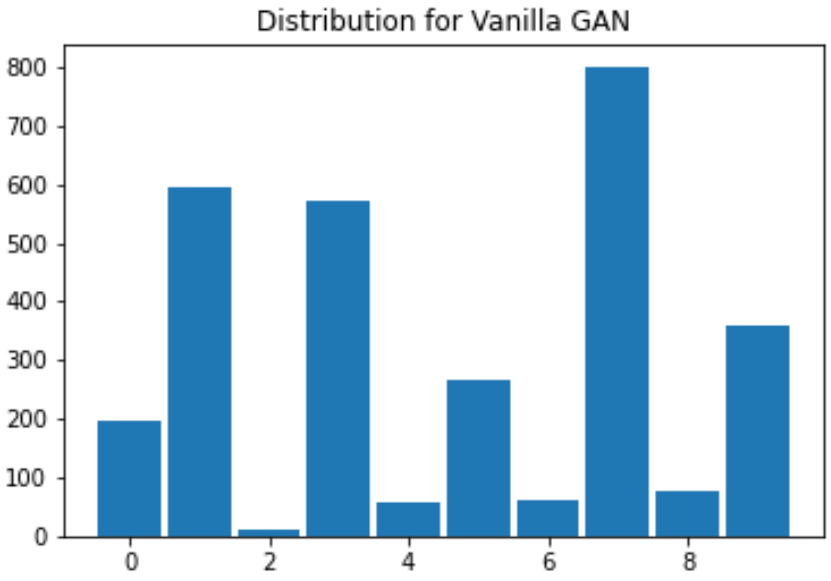
Note that G is trained with a copy of D at each epoch. The original D should not be updated during that part of training.

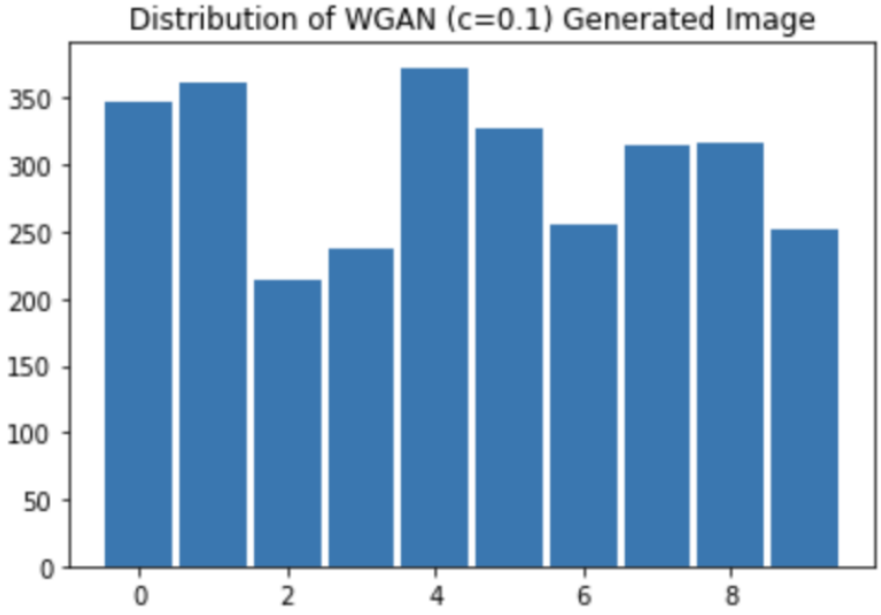
Train an unrolled GAN and re-plot the histogram from 3000 generated samples. Discuss whether unrolled GAN seems to help reduce the mode collapse problem.

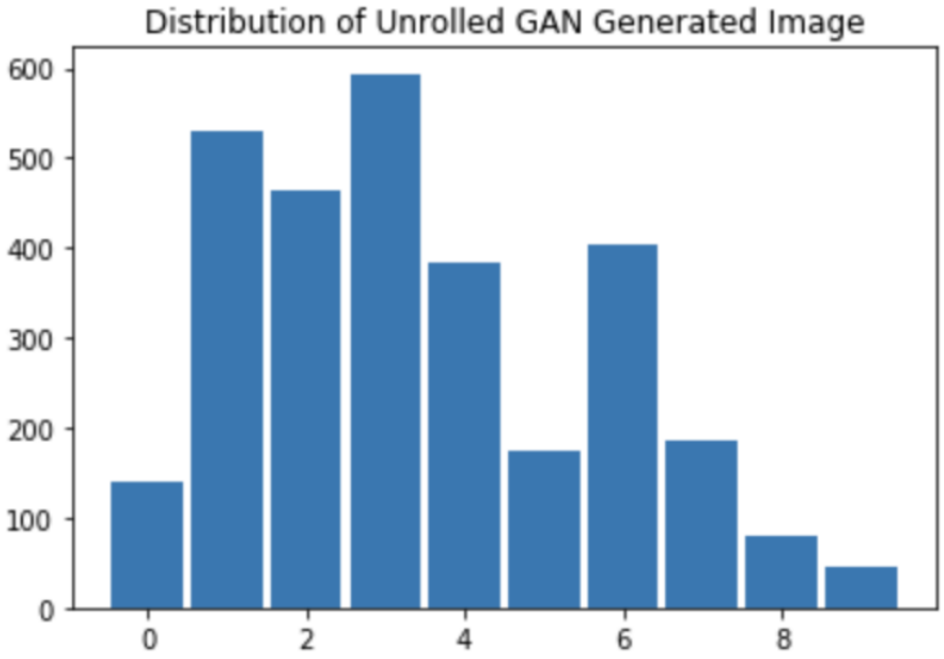
WGAN is claimed to be less affected by mode collapse too. In addition to your vanilla GAN model, use the WGAN model you trained in Part 2 and plot the histogram of class distribution, compare it to unrolled GAN and vanilla GAN.

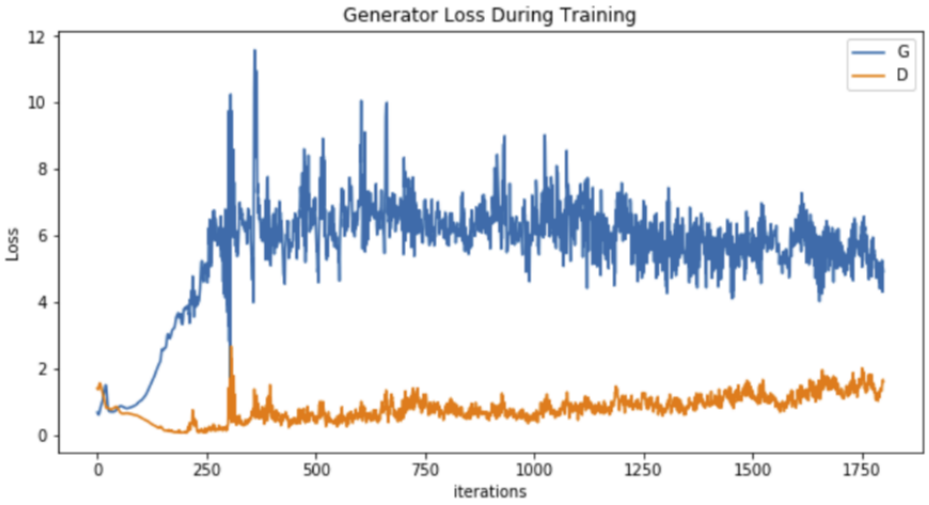
Solution:
The histograms of predicted labels using vanilla GAN, WGAN, and Unrolled GAN are shown below, respectively. Vanilla GAN didn't reduce the mode collapse problem, since the predicted labels were fairly unbalanced. Unrolled GAN reduced the mode collapse problem a bit but was also fairly unbalanced. The WGAN reduced the mode collapse

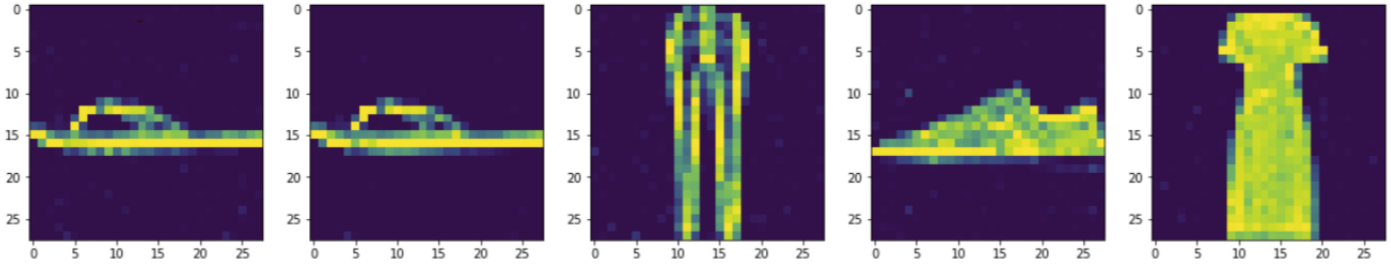
problem the best, as the labels were fairly balanced.






Distribution of Unrolled GAN Generated Image
600
500
400
300
200
100
0
0
2
4
6
8


Generator Loss During Training
G
D
Loss
iterations
12
10
8
6
4
2
0
0
250
500
750
1000
1250
1500
1750



Part 4 - Conditional GAN (10 points)

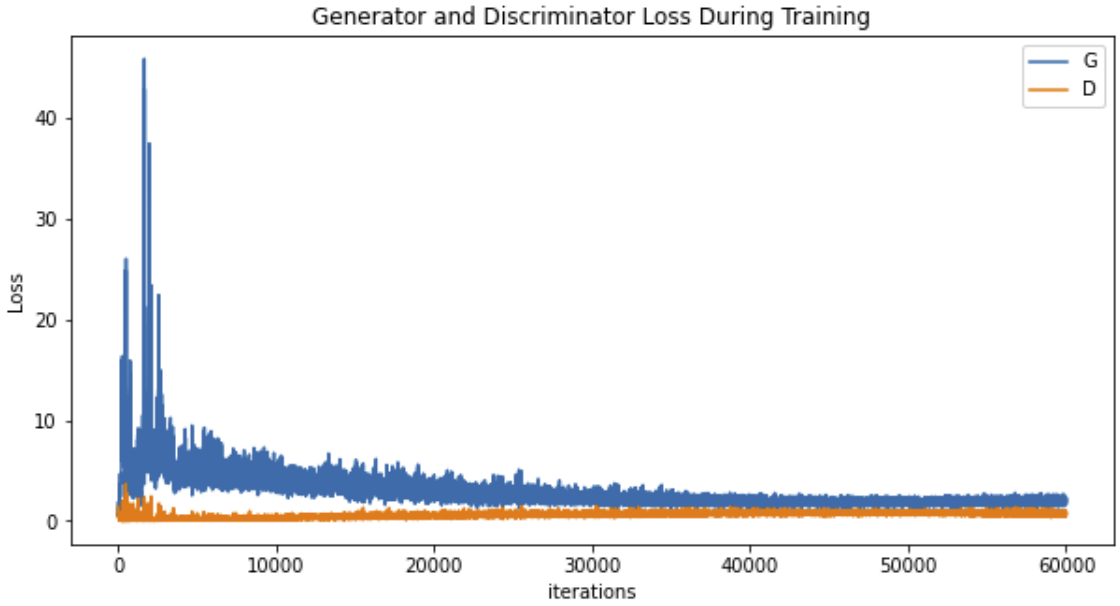
For the GANs we have been playing with, we cannot specify the class we want generated. Now, we explore adding extra information to the GAN to take more control over the generation process. Specifically, we want to generate not just *any* images from Fashion-MNIST data distribution, but images with a particular label such as shoes. This is called the Conditional GAN because now samples are drawn from a conditional distribution given a label as input.

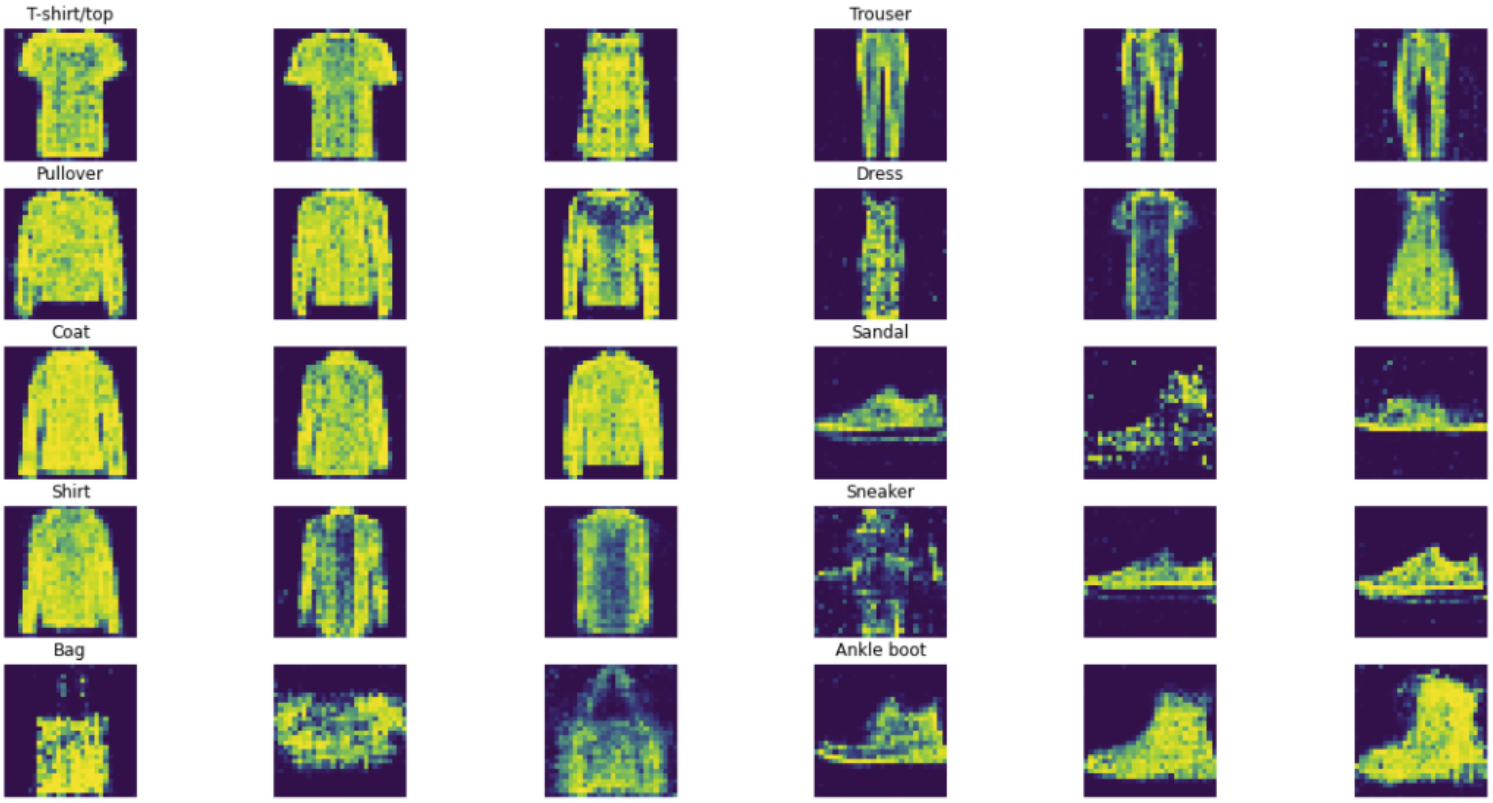
To add the conditional input vector, we need to modify both D and G. First, we need to define the input label vector. We are going to use one-hot encoding vectors for labels: for an image sample with label k of K classes, the vector is K dimensional and has 1 at k-th element and 0 otherwise.

We then concatenate the one-hot encoding of class vector with original image pixels (flattened as a vector) and feed the augmented input to D and G. Note we need to change the number of channels in the first layer accordingly.

Train a Conditional GAN using the training script from Part 1. Plot training curves for D and G. Generate 3 samples from each of the 10 classes. Discuss differences in the generated images produced compared to the non-conditional models you built.

Solution:
The generated images produced by Conditional GAN were fairly good in quality, similar to the quality of the images produced by Vanilla GAN. However, images for the "Shirt" and "Bag" classes were not as clear compared to the other classes. This could be related to the fact that both of these classes were not produced as much by Vanilla GAN and Unrolled GAN. WGAN w/ c=0.1 produced better results for the "Shirt" and "Bag" classes than Conditional GAN, and also did not suffer from as much mode collapse.


Generator and Discriminator Loss During Training
G
D
Loss
iterations
0
10
20
30
40
0
10000
20000
30000
40000
50000
60000


T-shirt/top
Trouser
Pullover
Dress
Coat
Sandal
Shirt
Sneaker
Bag
Ankle boot

Code Appendix

Code for Problem 1, Part 1

```
def preprocess(file):
    with open(file) as f:
      lines = [line.translate(str.maketrans('', '',
string.punctuation)).lower() for line in f]
    return lines

train_pos_reviews = preprocess('drive/My Drive/Deep
Learning/HW3/data/train_pos_merged.txt')
train_neg_reviews = preprocess('drive/My Drive/Deep
Learning/HW3/data/train_neg_merged.txt')
test_pos_reviews = preprocess('drive/My Drive/Deep
Learning/HW3/data/test_pos_merged.txt')
test_neg_reviews = preprocess('drive/My Drive/Deep
Learning/HW3/data/test_neg_merged.txt')

all_train_sentences = train_pos_reviews + train_neg_reviews
all_test_sentences = test_pos_reviews + test_neg_reviews

vocab_to_ind = dict()
i = 1

for sentence in all_train_sentences:
    for word in sentence.split():
        if word not in vocab_to_ind:
            vocab_to_ind[word] = i
            i += 1

def dataMatrix(sentences, vocab):
    matrix = list()
    for sentence in sentences:
        vector, words = np.zeros(400), sentence.split()
        vec_idx = 400 - min(400, len(words))
        for word in words:
            if word in vocab:
                vector[vec_idx] = vocab[word]
            vec_idx += 1
            if vec_idx == 400:
```

```
                break
            matrix.append(vector)
    return np.array(matrix)

pos_reviews_y, neg_reviews_y = np.array([1,0]), np.array([0,1])

X_train = dataMatrix(all_train_sentences, vocab_to_ind)
train_pos_rev_labels = np.tile(pos_reviews_y, (1500,1))
train_neg_rev_labels = np.tile(neg_reviews_y, (1500,1))
y_train = np.append(train_pos_rev_labels, train_neg_rev_labels, axis=0)

X_test = dataMatrix(all_test_sentences, vocab_to_ind)
test_pos_rev_labels = np.tile(pos_reviews_y, (1500,1))
test_neg_rev_labels = np.tile(neg_reviews_y, (1500,1))
y_test = np.append(test_pos_rev_labels, test_neg_rev_labels, axis=0)

print("X_train: ", X_train.shape)
print("y_train: ", y_train.shape)
print("X_test: ", X_test.shape)
print("y_test: ", y_test.shape)
```

```
X_train:  (3000, 400)
y_train:  (3000, 2)
X_test:  (3000, 400)
y_test:  (3000, 2)
```

```
train_data = TensorDataset(torch.from_numpy(X_train).long(),
torch.from_numpy(y_train).float())
train_loader = DataLoader(train_data, shuffle=True, batch_size=50)
test_data = TensorDataset(torch.from_numpy(X_test).long(),
torch.from_numpy(y_test).float())
test_loader = DataLoader(test_data, shuffle=True, batch_size=50)
```

Code for Problem 1, Part 2

```
# Implementation courtesy of: https://github.com/gabrielloye/GRU_Prediction/

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

```
        self.hidden_dim = 32
        self.input_dim = 512
        self.output_dim = 2
        self.n_layers = 2

        self.embedding = nn.Embedding(len(vocab_to_ind)+1, self.input_dim)
        self.gru = nn.GRU(self.input_dim, self.hidden_dim, self.n_layers,
batch_first=True)
        self.fc = nn.Linear(400 * self.hidden_dim, self.output_dim)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x, h):
        x = self.embedding(x)
        out, h = self.gru(x, h)
        out = self.fc(self.sigmoid(out.reshape(50, 400 *
self.hidden_dim)))
        return out, h

    def init_hidden(self, batch_size):
        weight = next(self.parameters()).data
        hidden = weight.new(self.n_layers, batch_size,
self.hidden_dim).zero_().to(device)
        return hidden

gru_model = GRUNet()
gru_model.to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(gru_model.parameters(), lr=0.001)

train_loss, test_loss = list(), list()
gru_model.train()

for epoch in range(30):
    h = gru_model.init_hidden(batch_size)
    avg_train_loss = 0.

    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        h = h.data
        gru_model.zero_grad()
```

```
        outputs, h = gru_model(inputs, h)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        avg_train_loss += loss.item()

    train_loss.append(avg_train_loss / 60)

    avg_test_loss = 0.
    with torch.no_grad():
        for i, data in enumerate(test_loader, 0):
            inputs, labels = data
            h = gru_model.init_hidden(batch_size)

            outputs, h = gru_model(inputs, h)
            loss = criterion(outputs, labels)
            avg_test_loss += loss.item()

    test_loss.append(avg_test_loss / 60)
```

Code for Problem 1, Part 3

```
# Implementation courtesy of: https://medium.com/@galhever/sentiment-analysis-with
-pytorch-part-5-mlp-model-387057f4a06a

class MLP(nn.Module):
        def __init__(self):
            super(MLP, self).__init__()
            self.input_dim = 400
            self.embedding_dim = 512

            self.embedding = nn.Embedding(len(vocab_to_ind) + 1,
self.embedding_dim)
            self.fc1 = nn.Linear(self.input_dim*self.embedding_dim, 2048,
bias=True)
            self.fc2 = nn.Linear(2048, 256, bias=True)
            self.fc3 = nn.Linear(256, 2, bias=True)
            self.sigmoid = nn.Sigmoid()
            self.relu = nn.ReLU()

        def forward(self, x):
            x = self.embedding(x)
```

```
            x = x.view(-1, 400*self.embedding_dim)
            x = F.relu(self.fc2(F.relu(self.fc1(x))))
            return self.sigmoid(self.fc3(x))

criterion = nn.MSELoss()
mlp = MLP()
optimizer = optim.Adam(mlp.parameters(), lr=0.001)

train_loss, test_loss = list(), list()
mlp.train()

for epoch in range(30):
    optimizer.zero_grad()
    avg_train_loss = 0.

    for i, data in enumerate(train_loader, 0):
        inputs, labels = data
        outputs = mlp(inputs)
        loss = criterion(outputs, labels.float())
        loss.backward()
        optimizer.step()
        avg_train_loss += loss.item()

    train_loss.append(avg_train_loss / 60)

    avg_test_loss = 0.
    with torch.no_grad():
        for i, data in enumerate(test_loader, 0):
            inputs, labels = data
            outputs = mlp(inputs)
            loss = criterion(outputs, labels)
            avg_test_loss += loss.item()

    test_loss.append(avg_test_loss / 60)
```

Code for Fashion-MNIST preprocessing

```
def loader(x):
    scaler = MinMaxScaler(feature_range=(-1,1))
    x = scaler.fit_transform(x)
    x = Tensor(x).flatten()
    return x
```

```
fmnist = torchvision.datasets.FashionMNIST(
    root="./",
    train=True,
    transform = preprocess,
    download=True)

batch_size = 100
data_loader = torch.utils.data.DataLoader(dataset=fmnist,
batch_size=batch_size, shuffle=True)
```

Code for Generating Loss Plots

```
    plt.figure(figsize=(10,5))
    plt.title("Generator and Discriminator Loss During Training")
    plt.plot(G_losses,label="G")
    plt.plot(D_losses,label="D")
    plt.xlabel("num_iters")
    plt.ylabel("loss")
    plt.legend()
    plt.savefig("loss_plot")
    plt.show()
```

Code for Generator and Discriminator Classes

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(True),
            nn.Linear(256, 512),
            nn.ReLU(True),
            nn.Linear(512, 1024),
            nn.ReLU(True),
            nn.Linear(1024, 28*28),
            nn.Tanh()
        )

    def forward(self, input_x):
        return self.main(input_x)

class Discriminator(nn.Module):
```

```
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(28*28, 1024),
            nn.ReLU(True),
            nn.Linear(1024, 512),
            nn.ReLU(True),
            nn.Linear(512, 256),
            nn.ReLU(True),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, input_x):
        return self.main(input_x)

class WGANDisc(nn.Module):
    def __init__(self):
        super(WGANDisc, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(28*28, 1024),
            nn.ReLU(True),
            nn.Linear(1024, 512),
            nn.ReLU(True),
            nn.Linear(512, 256),
            nn.ReLU(True),
            nn.Linear(256, 1),
        )

    def forward(self, input_x):
        return self.main(input_x)

class ConditionalGANGenerator(nn.Module):
    def __init__(self):
        super(ConditionalGANGenerator, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(True),
            nn.Linear(256, 512),
            nn.ReLU(True),
            nn.Linear(512, 1024),
```

```
            nn.ReLU(True),
            nn.Linear(1024, 28*28+10),
            nn.Tanh()
        )

    def forward(self, input_x):
        return self.main(input_x)

class ConditionalGANDiscriminator(nn.Module):
    def __init__(self):
        super(ConditionalGANDiscriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(28*28+10, 1024),
            nn.ReLU(True),
            nn.Linear(1024, 512),
            nn.ReLU(True),
            nn.Linear(512, 256),
            nn.ReLU(True),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, input_x):
        return self.main(input_x)

class UnrolledGANDiscriminator(nn.Module):
    def __init__(self):
        super(UnrolledGANDiscriminator, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(28*28+10, 1024),
            nn.ReLU(True),
            nn.Linear(1024, 512),
            nn.ReLU(True),
            nn.Linear(512, 256),
            nn.ReLU(True),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, input_x):
        return self.main(input_x)
```

```
    def load(self, backup):
    for m_from, m_to in zip(backup.modules(), self.modules()):
         if isinstance(m_to, nn.Linear):
            m_to.weight.data = m_from.weight.data.clone()
            if m_to.bias is not None:
                m_to.bias.data = m_from.bias.data.clone()
```

Code for Problem 2, Part 1

```
# Implementation courtesy of: https://pytorch.org/tutorials/beginner/
dcgan_faces_tutorial.html

netD = Discriminator()
netG = Generator()
criterion = nn.BCELoss()
fixed_noise = Variable(torch.randn(batch_size, 100))

real_label = 1
fake_label = 0

lr = 0.0002
optimizerD = optim.Adam(netD.parameters(), lr=lr)
optimizerG = optim.Adam(netG.parameters(), lr=lr)

num_epochs = 100
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
for epoch in range(num_epochs):
    for i, data in enumerate(data_loader, 0):

        ############################
        # UPDATE DISCRIMINATOR
        ###########################
        netD.zero_grad()
        inputs = Variable(data[0])
        labels = Variable(ones(batch_size, 1))
        outputs = netD(inputs)
```

```
        errD_real = criterion(outputs, labels)
        errD_real.backward()

        noise = Variable(torch.randn(batch_size, 100, device=device))
        fake = netG(noise).detach()
        labels = Variable(zeros(batch_size, 1))
        outputs = netD(fake)
        errD_fake = criterion(outputs, labels)
        errD_fake.backward()
        errD = errD_real + errD_fake
        optimizerD.step()

        ############################
        # UPDATE GENERATOR
        ############################
        netG.zero_grad()
        labels = Variable(ones(batch_size, 1))
        noise = Variable(torch.randn(batch_size, 100, device=device))
        fake = netG(noise)
        outputs = netD(fake)
        errG = criterion(outputs, labels)
        errG.backward()
        optimizerG.step()

        G_losses.append(errG.item())
        D_losses.append(errD.item())

        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i ==
len(data_loader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2,
normalize=True))

        iters += 1
```

Code for Problem 2, Part 2 (MSE)

```
generator_MSE = Generator()
loss_MSE = nn.MSELoss()
fixed_noise = Variable(torch.randn(batch_size, 100))
optimizerG = optim.Adam(generator_MSE.parameters(), lr=0.0001)
```

```
num_epochs = 10
img_list = []
G_losses = []
iters = 0

print("Starting Training Loop...")
for epoch in range(num_epochs):
    for i, data in enumerate(data_loader, 0):

        ############################
        # UPDATE GENERATOR
        ###########################
        generator_MSE.zero_grad()
        labels = Variable(ones(batch_size, 1))
        noise = Variable(torch.randn(batch_size, 100, device=device))
        fake = generator_MSE(noise)
        inputs = Variable(data[0])
        errG = loss_MSE(fake, inputs)
        errG.backward()
        optimizerG.step()

        G_losses.append(errG.item())

        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i ==
len(data_loader)-1)):
            with torch.no_grad():
                fake = generator_MSE(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2,
normalize=True))

        iters += 1
```

Code for Problem 2, Part 2 Code (WGAN)

```
netGWGAN = Generator()
netDWGAN = WGANDisc()
optimizerD = optim.RMSprop(netDWGAN.parameters(), lr=0.001)
optimizerG = optim.RMSprop(netGWGAN.parameters(), lr=0.001)
fixed_noise = Variable(torch.randn(batch_size, 100))

img_list = []
```

```
G_losses = []
D_losses = []
D_iters = 0
iters = 0
c_param = 0.1 #0.01, 0.001, 0.0001

print("Starting Training Loop...")
for epoch in range(20):
    for i, data in enumerate(data_loader, 0):

        ############################
        # UPDATE DISCRIMINATOR
        ###########################
        netDWGAN.zero_grad()

        inputs = Variable(data[0])
        outputs = netDWGAN(inputs)
        errD_real = torch.mean(outputs)
        errD_real.backward(torch.tensor(1, dtype=torch.float))

        noise = Variable(torch.randn(batch_size, 100, device=device))
        fake = netGWGAN(noise).detach()
        outputs = netDWGAN(fake)
        errD_fake = torch.mean(outputs)
        errD_fake.backward(-torch.tensor(1, dtype=torch.float))
        errD = errD_fake - errD_real
        optimizerD.step()

        for p in netDWGAN.parameters():
            p.data.clamp_(-c_param, c_param)

        ############################
        # UPDATE GENERATOR
        ###########################
        if iters % 5 == 0:
            netGWGAN.zero_grad()
            noise = Variable(torch.randn(batch_size, 100))
            fake = netGWGAN(noise)
            outputs = netDWGAN(fake)

            errG = torch.mean(outputs)
```

```
            errG.backward(torch.tensor(1, dtype=torch.float))
            optimizerG.step()

            G_losses.append(-errG.item())
            D_losses.append(errD.item())

        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i ==
len(data_loader)-1)):
            with torch.no_grad():
                fake = netGWGAN(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2,
normalize=True))

        iters += 1
```

Code for Problem 2, Part 2 (Least Squares GAN)

```
netDWGAN = WGANDisc()
netG = Generator()
optimizerD = optim.Adam(netDWGAN.parameters(), lr=0.0001)
optimizerG = optim.Adam(netG.parameters(), lr=0.0001)
fixed_noise = Variable(torch.randn(batch_size, 100))

img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
for epoch in range(50):
    for i, data in enumerate(data_loader, 0):

        ############################
        # UPDATE DISCRIMINATOR
        ###########################
        inputs = Variable(data[0])
        labels = Variable(ones(batch_size, 1))
        outputs = netD(inputs)
        errD_real = 0.5*torch.mean((outputs-1)**2)
        errD_real.backward()

        noise = Variable(torch.randn(batch_size, 100))
```

```
        fake = netG(noise)
        labels = -Variable(ones(batch_size, 1))
        outputs = netDWGAN(fake)
        errD_fake = 0.5*torch.mean(outputs**2)
        errD_fake.backward()
        errD = errD_fake + errD_real
        optimizerD.step()

        netD.zero_grad()
        netG.zero_grad()

        ############################
        # UPDATE GENERATOR
        ############################
        labels = Variable(ones(batch_size, 1))
        noise = Variable(torch.randn(batch_size, 100, device=device))
        fake = netG(noise)
        outputs = netDWGAN(fake)

        errG = 0.5*torch.mean((outputs-1)**2)
        errG.backward()
        optimizerG.step()
        netDWGAN.zero_grad()
        netG.zero_grad()

        G_losses.append(errG.item())
        D_losses.append(errD.item())

        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i ==
len(data_loader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2,
normalize=True))

        iters += 1
```

Code for Problem 2, Part 3 (Mode Collapse and Unrolled GANs)

```
# Code implementation based off https://github.com/andrewliao11/unrolled-gans/
blob/master/unrolled_gan.ipynb
```

```
originalD = Discriminator()
netD = UnrolledGANDiscriminator()
netG = Generator()
criterion = nn.BCELoss()
fixed_noise = Variable(torch.randn(batch_size, 100))
real_label, fake_label = 1, 0
optimizerD, optimizerG = optim.Adam(netD.parameters(), lr=0.0001),
optim.Adam(netG.parameters(), lr=0.0001)

def d_unrolled_loop(d_gen_input=None):
    optimizerD.zero_grad()
    d_real_data = list()
    for i in range(batch_size):
        it = iter(fmnist)
        d_real_data.append(np.array(next(it)[0]))

    d_real_data = torch.from_numpy(np.asarray(d_real_data))
    d_real_decision = netD(d_real_data)
    target = Variable(ones(batch_size, 1))
    d_real_error = criterion(d_real_decision, target)

    if d_gen_input is None:
        d_gen_input = Variable(torch.randn(batch_size, 100))

    with torch.no_grad():
        d_fake_data = netG(d_gen_input)

    outputs = netD(d_fake_data)
    target = Variable(zeros(batch_size, 1))

    d_fake_error = criterion(d_real_decision, target)
    d_loss = d_real_error + d_fake_error
    d_loss.backward()
    optimizerD.step()
    return d_real_error.item(), d_fake_error.item()

unrolled_steps = 5
img_list = []
G_losses = []
D_losses = []
iters = 0
```

```
print("Starting Training Loop...")
for epoch in range(20):
    for i, data in enumerate(data_loader, 0):

        ############################
        # UPDATE DISCRIMINATOR
        ###########################
        netD.zero_grad()
        inputs = Variable(data[0])
        target = Variable(ones(batch_size, 1))
        outputs = netD(inputs)
        errD_real = criterion(outputs, target)
        errD_real.backward()

        gen_input = Variable(torch.randn(batch_size, 100, device=device))
        g_fake_data = netG(gen_input).detach()
        target = Variable(zeros(batch_size, 1))
        dg_fake_decision = netD(g_fake_data)
        g_error = criterion(dg_fake_decision, target)
        g_error.backward()
        errD = errD_real + g_error
        optimizerD.step()

        ############################
        # UPDATE GENERATOR
        ###########################
        netG.zero_grad()
        gen_input = Variable(torch.randn(batch_size, 100))

        if unrolled_steps > 0:
            backup = copy.deepcopy(netD)
            for i in range(unrolled_steps):
                d_unrolled_loop(d_gen_input=gen_input)

        g_fake_data = netG(gen_input)
        dg_fake_decision = netD(g_fake_data)
        labels = Variable(ones(batch_size, 1))
        errG = loss(dg_fake_decision, labels)
        errG.backward()
        optimizerG.step()
```

```
        if unrolled_steps > 0:
            netD.load(backup)
            del backup

        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on
fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i ==
len(data_loader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2,
normalize=True))

        iters += 1

    with torch.no_grad():
        noise = Variable(torch.randn(3000, 100))
        fake = netG(noise)
        outputs = originalD(fake)
        _, preds = torch.max(outputs.data, 1)
```

Code for Problem 2, Part 4

```
def one_hot(tar):
    temp = np.zeros(10)
    temp[tar] = 1
    return Tensor(temp).flatten()

conditionalD = ConditionalGANDiscriminator()
conditionalG = ConditionalGANGenerator()
loss = nn.BCELoss()
fixed_noise = Variable(torch.randn(batch_size, 100))
optimizerD = optim.Adam(conditionalD.parameters(), lr=0.0001)
optimizerG = optim.Adam(conditionalG.parameters(), lr=0.0001)

# Same training script as Part 1
# Same code for generating plots as above
```

```
fig = plt.figure(figsize=(10,10))
indices = [30,40,50]
for i in range(len(indices)):
    index = indices[i]
    ax1 = fig.add_subplot(3, 3, 1+3*i)
    ax1.set_title('Beginning')
    ax1.imshow(img_list[0][0,index,:784].reshape(28,28))
    ax2 = fig.add_subplot(3, 3, 2+3*i)
    ax2.imshow(img_list[30][0,index,:784].reshape(28,28))
    ax2.set_title('Intermediate')
    ax3 = fig.add_subplot(3, 3, 3+3*i)
    ax3.set_title('Convergence')
    ax3.imshow(img_list[120][0,index,:784].reshape(28,28))
```