Homework 3
CS 5787 Deep Learning
Spring 2021

Yuxin Zhang - yz2726@cornell.edu

Due: See Canvas

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.

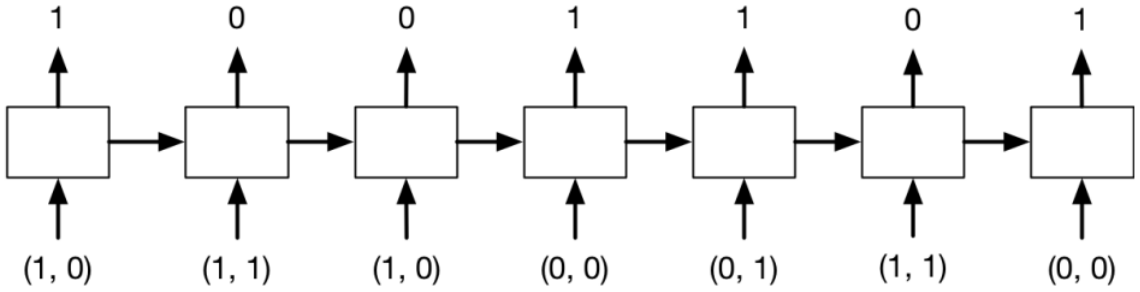
The input to your RNN will be two binary numbers, starting with the *least* significant bit. You will need to pad the largest number with an additional zero on the left side and you should make the other number the same length by padding it with zeros on the left side. For instance, the problem

$$100111 + 110010 = 1011001$$

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:



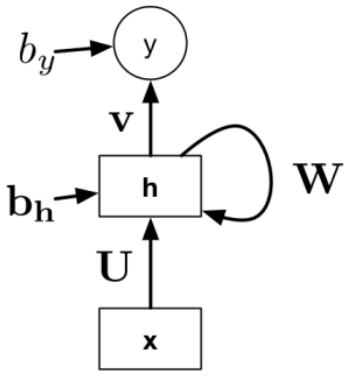
The RNN that implements binary addition has three hidden units, and all of the units use the following non-differentiable hard-threshold activation function

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

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:

$$h_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \; U = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \; W = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 0 & 0.5 & 0.5 \\ 0 & 0.5 & 0.5 \end{bmatrix} \; b_h = \begin{bmatrix} 0 \\ -1 \\ -2 \end{bmatrix} \; V = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix}$$

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 film, television, etc. review. The dataset we are using is the IMDB review dataset (link). It is a binary sentiment classification (positive or negative) dataset. 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 review. Because reviews are of different lengths, you need to pad or truncate the reviews to make them same length. We are going to use 400 as the fixed length in this problem. That means any review that is longer than 400 words will be truncated; any review that is shorter than 400 words will be padded with 0s. Please note that your padded 0s should be placed *before* the review if they are needed.

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:
In the preprocessing part, we perform the following process: removing all the HTML tags; removing non-ASCII characters; converting each string to lower case; removing punctuation; removing stopwords; tokenization. while doing the tokenization of each sentence, we gather collection of tokens(word) and use Counter to directly attain vocabulary dictionary with value being frequency of each word token.
When converting the text data input to a matrix, for the text with token length less than 400, we add 0s in front of the word tokens. Yet for the text with token length more than 400, we select the tokens belonging to the first 400 most frequent words by referring to the vocabulary dictionary. In this way, more common word tokens in the whole input get encoded. And then, we rebuild the vocabulary and word to index reference based on the encoded words. The encoded vocabulary has a total size of 37119.
For further data preparation in training ML models, we split 1000 out of the full 3000 training dataset as the development set, use TensorDataset and DataLoader to set up (data, label) iterators for training, developing, and testing dataset.

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,

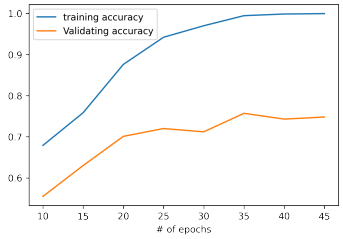
embedding_dim) as the input shape.

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.

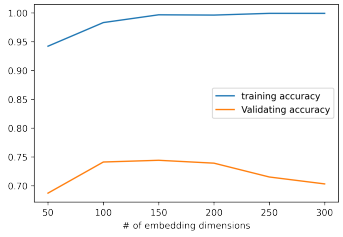
At a high level, the input sequence is fed into the word embedding layer first. Then, the GRU is taking steps through each word embedding in the sequence and return output / feature at each step. The prediction unit should take the output from the final step of the GRU and make predictions.

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

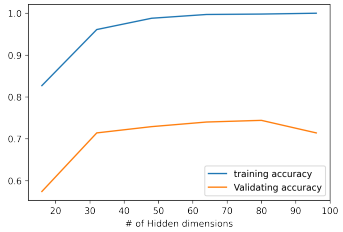
Solution:
In the model construction, we do not consider bidirectional for the GRU unit. The input dimension of the GRU model is the size of the vocabulary we used in encoding the input matrix. Besides We tune three parameters: embedding dimension, hidden dimension, number of epochs.
Firstly, we tune number of epoches in the range of [10,15, 20...45], evaluating the accuracy performance on the training and development set, and the performance graph shows best development performance achieved at epoch = 35.



Secondly, we tune size of embedding dimensions in the range of [50,100, 150...300],evaluating the accuracy performance on the training and development set, and the performance graph shows best development performance achieved at embedding dimensions = 150.



Lastly, we tune size of hidden dimensions in the range of [16,32...96],evaluating the accuracy performance on the training and development set, and the performance graph shows best development performance achieved at hidden dimensions = 80.



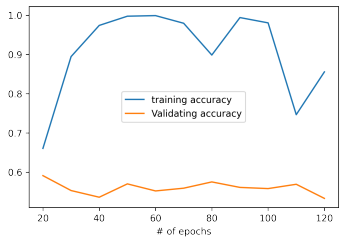
The best combination of hyperparameter is epoch = 35, embedding dimensions = 150, hidden dimensions = 80, which achieved 0.744 accuracy rate on the development set. Applying such hyperparameters to train the GRU modelon the full training input (3000) and predict on the test set, the test set accuracy rate is 0.713.
An interesting observation is that even with 0.5 dropout rate and only 1 hidden layer, the GRU still shows overfitting regardless of the epoch size, embedding dimension, and number of hidden units. This might be due to the size of the training being rather small and the input dimension (vocabulary encoded in input matrix) being too large.

Part 3 - Comparison with a MLP (5 points)

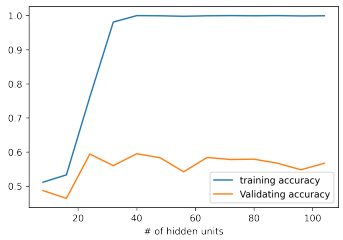
Since each review is a fixed length input (with potentially many 0s in some samples), we can also train a standard MLP for this task.

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 GRU model?

Solution:
Treating the index encoding of the vocabulary and padding in the input matrix as numerical input, we take the problem as a binary classification problem. The MLP has two layers, one hidden layer with input dimension of 400, a relu activation function, and a an output layer with softmax applied in the end. The size of the hidden layer is flexiable for tuning. Two hyperparameters are tuned: the number of training epochs and the number of hidden units.
Firstly, we tune number of epoches in the range of [20,30...120], evaluating the accuracy performance on the training and development set, and the performance graph shows best development performance achieved at epoch = 50.



Secondly, we tune the number of hidden unit in the range of [8, 16...104],evaluating the accuracy performance on the training and development set, and the performance graph shows best development performance achieved at number of hidden unit = 80.



The best combination of hyperparameter is epoch = 50, hidden unit = 180, which achieved 0.579 accuracy rate on the development set. Applying such hyperparameters to train the MLP modelon the full training input (3000) and predict on the test set, the test set accuracy rate is 0.567.
Both GRU and MLP suffer from overfitting, but GRU clearly outperformed MLP (0.713 $>$ 0.567) in that it used contextual information by using an RNN framework.

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.

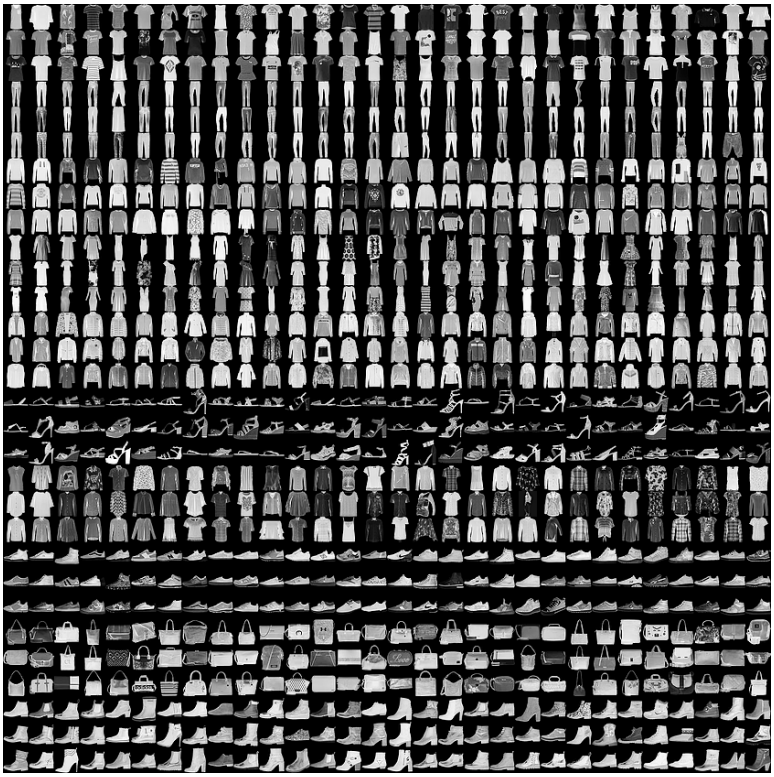


Figure 1: Fashion-Mnist dataset example images. It contains 10 classes of cloths, shoes, and bags.

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.

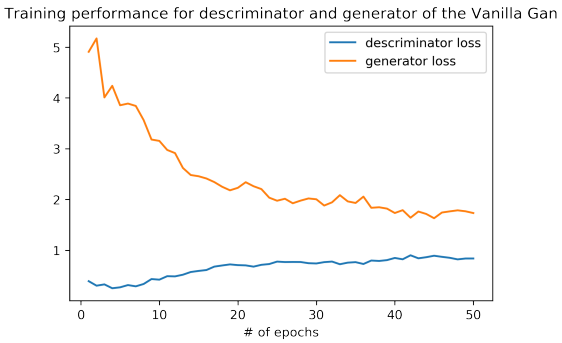
Train a basic GAN that can generate images from the Fashion-MNIST dataset. Plot your training loss curves for your G and D. Show the generated samples from G in 1) the beginning of the training; 2) intermediate stage of the training; and 3) after convergence.

Solution:
The structure of the generator and discriminator is as follows.

```
Generator(
```

```
  (model): Sequential(
    (0): Linear(in_features=64, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=1024, bias=True)
    (5): ReLU()
    (6): Linear(in_features=1024, out_features=784, bias=True)
    (7): Tanh()
  )
)
Discriminator(
  (model): Sequential(
    (0): Linear(in_features=784, out_features=1024, bias=True)
    (1): ReLU()
    (2): Linear(in_features=1024, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=256, bias=True)
    (5): ReLU()
    (6): Linear(in_features=256, out_features=1, bias=True)
    (7): Sigmoid()
  )
)
```

In the training process, the generator is trained by drawing random 64-dimensional noise vectors as inputs to the generator network and evaluating its loss using the prediction from the generated image after putting through the discriminator. On the other hand, the discriminator evaluates its loss from two perspectives, the loss from the fake generated images and that from the real drawn images.
We use BCELoss and Adam optimizer for both the generator and discriminator, with learning rate initialized at 0.0003. Training on 50 epochs and plot the loss on both generator and discriminator, we observed a convergence of both loss and thus attain each generated fake image sample from the first epoch, the 25th epoch and the 49th epoch. The visualization shows the below.



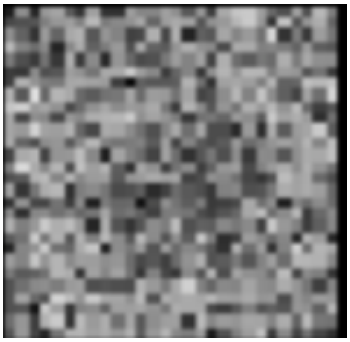


Figure 2: One sample for generated fake image on the first epoch

Figure 3: One sample for generated fake image on the 25th epoch

Figure 4: One sample for generated fake image on the last epoch

Obviously, on earlier training stages, the generator simply produces some random images and performs badly on the discriminator. But as the training goes on, it started to generate images that are very similar to the fashion images, thus tricking the discriminator.

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:
For **MSE loss** that directly applied on the real image and generated image, we only train the generator and calculate its loss by the mean square of the difference between the generated fake image and real image of each batch. Training with 50 epochs with learning rate of 0.0003, we have the following generating loss performance and procedural generated images. The training generator loss did not really converge much. This is expected since without the discriminator gradient backward to update the generator with exact direction compared with the true label, the generator can only update the model by comparing the generated fake image with the true image at each batch, yet each batch has different images, making the update of the model pretty random. This can be also revealed in the generated procedural images.
The training takes 547.867831628 seconds, which is much faster than the following cases since it only computes and updates the generator directly.

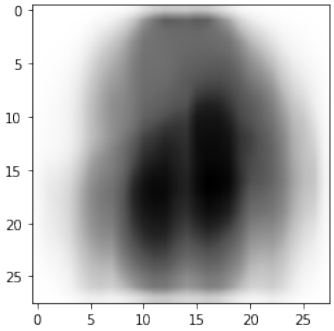


Figure 5: Generated image at epoch = 1

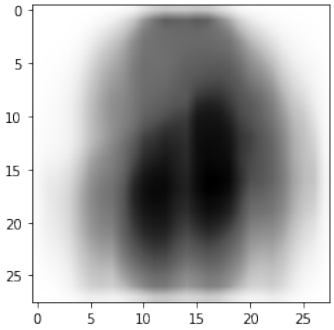


Figure 6: Generated image at epoch = 25

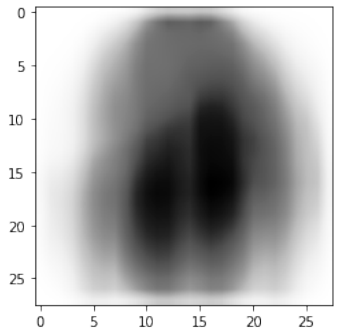
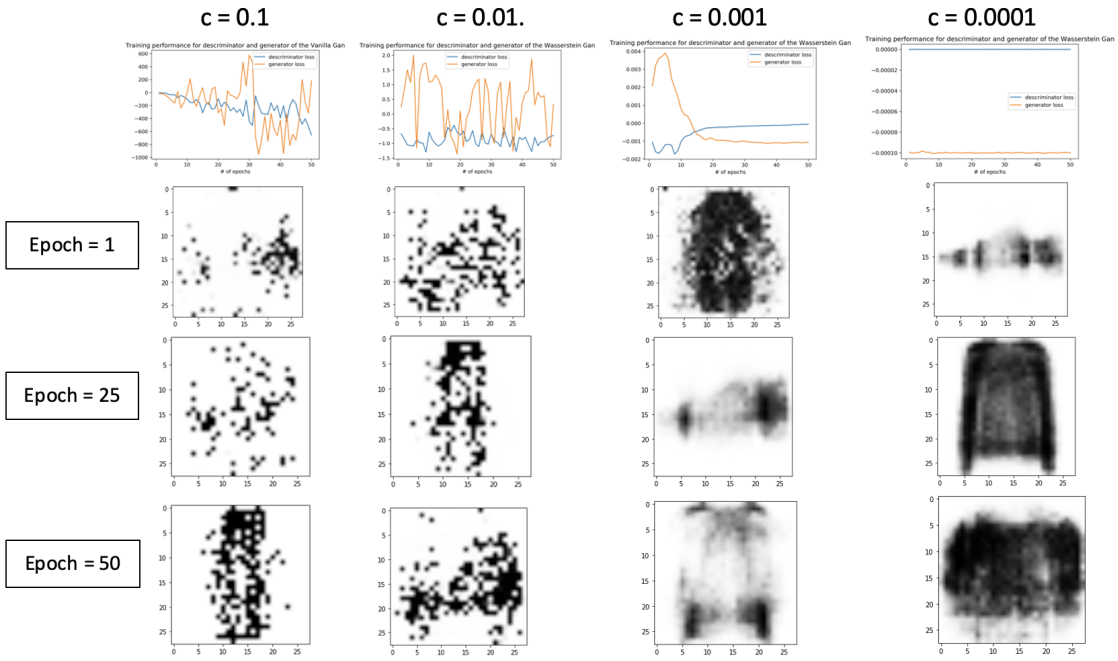
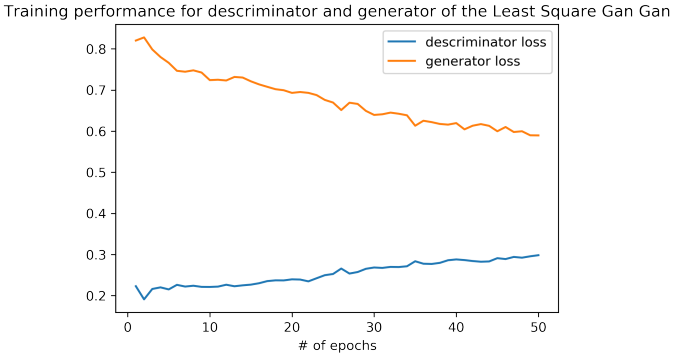


Figure 7: Generated image at epoch = 50

For **Wasserstein Gan**, we remove the nn.sigmoid() layer from the distriminator and for each epoch of distriminator training, we call clamp_(-c, c) to clip the weight of all parameters in the discriminator. The experimenting training with different c from range [0.1, 0.01, 0.001, 0.0001] shows the Gan performance regrading the generator, discriminator loss and procedural generated image as the following (using 50 epochs).
When c is comparatively large (c = 0.1 and c = 0.01), the generator and discriminator loss jumps up and down through epochs since the weights are not finely regulated, thus generating images are generally messy and does not have similar-to-real fashion image shapes. When c is appropriate(neither too large nor too small, like c = 0.001 in this case), the generator and discriminator loss converged with epochs, and the procedural generated images show how the generator has improvised in generating close-to-real fashion images. With very small c (0.0001 in our case), the discriminator and generator loss shows little improvement with time, and the the generator barely learned anything, even if it produced pretty close-to-real fashion images from the very beginning.
The training with c = 0.001 takes 1307.743754863739 seconds.



For **Least Square Gan**, we remove the nn.sigmoid() layer from the discriminator so that we can attain the Least Square loss instead of the Log loss. Also, the discriminator loss is calculated as (torch.mean((outputs_real - real_labels)**2) + torch.mean((outputs_fake-fake_labels)**2)), and the generator loss is calculated as torch.mean((outputs_fake - real_labels)**2). Training the model with 50 epochs with learning rate of 0.00005. We have the generator, discriminator loss and procedural generated images as the follows.
The total training time is 1170.0975952148438 seconds. With the same learning rate and number of epochs, the GAN trained with Least Square loss is faster than that trained with Wasserstein. This can be explained by the extra clipping computation of discriminator at each epoch for the Wasserstein GAN.



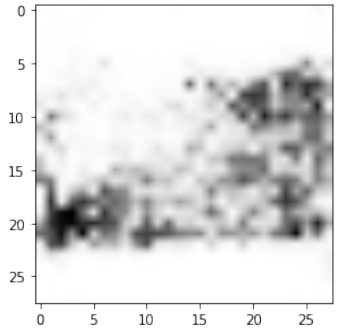


Figure 8: Generated image at epoch = 1

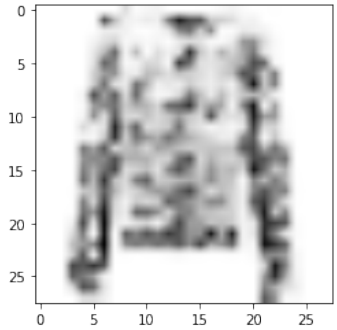


Figure 9: Generated image at epoch = 25

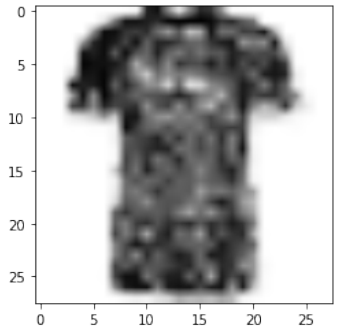


Figure 10: Generated image at epoch = 50

Part 3 - Mode Collapse in GANs (10 points)

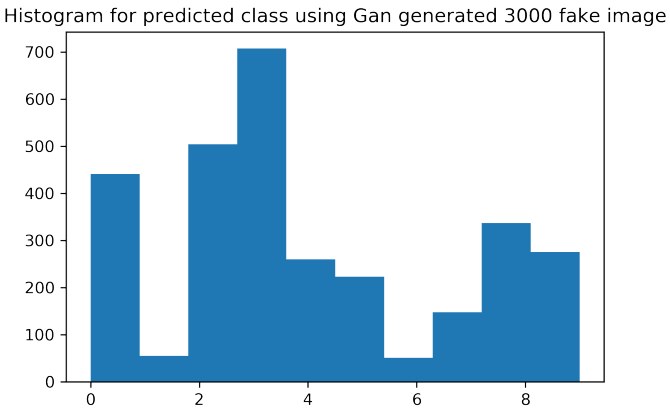
Take a copy of your vanilla GAN discriminator and change its output channel from 1 output to 10 output units. Fine-tune it as a classifier on the Fashion-MNIST training set. You should easily achieve $\sim 90\%$ accuracy on Fashion-MNIST test set.

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. **Solution:**
Build a FashionMNIST_Classifer using the previously defined discriminator in Vanilla GAN and changing the last output linear layer from 1 to 10, which as the following architecture:

```
Discriminator(
  (model): Sequential(
    (0): Linear(in_features=784, out_features=1024, bias=True)
    (1): ReLU()
    (2): Linear(in_features=1024, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=256, bias=True)
    (5): ReLU()
    (6): Linear(in_features=256, out_features=10, bias=True)
    (7): Sigmoid()
  )
)
```

Training with epoch = 50, batch size = 2048, learning rate = 0.0003, we get training accuracy at 0.92163 and test set accuracy at 0.90764. Return the trained discriminate. Also, use the trained Vanilla GAN generator from part 1, which is trained using learning rate of 0.0003 and epoch = 50. Generating 3000 fakes images using the trained generator and classifying each using the trained discriminator, we have the predicted label distribution as the following.



Part 4 - Unrolled GAN (10 points)

Unrolled GAN is a proposed method 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_unroll 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)
```

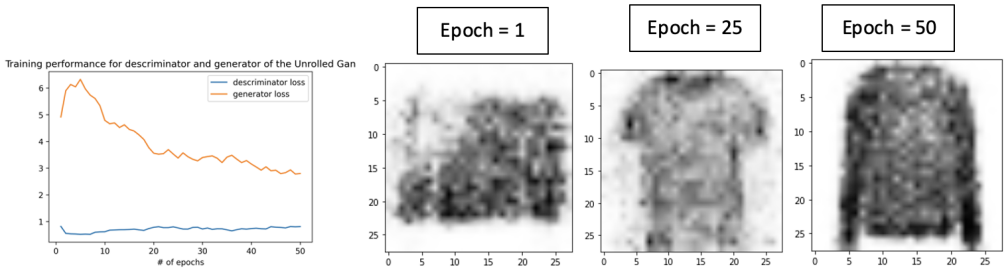
Note that G is trained with a copy of D at each epoch. The original D should not be updated during that part of training. Train an unrolled GAN.

Generate 3000 examples from the vanilla GAN, WGAN, and the unrolled GAN (9000

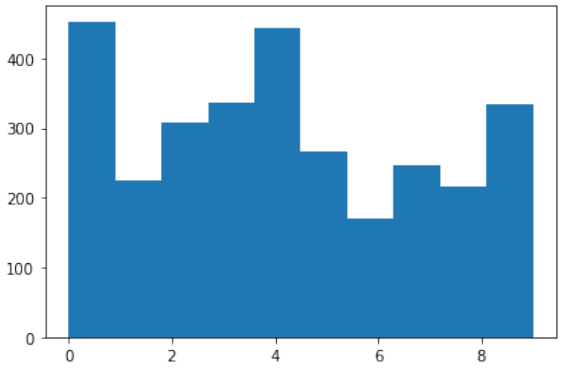
total examples). For each architecture, plot the class distribution histogram from the 3000 generated samples using the classifier you trained in the previous part.

Like unrolled GAN, WGAN is claimed to be less affected by mode collapse. Discuss how each of the three generators performs and which seems to be best at reducing the mode collapse problem.

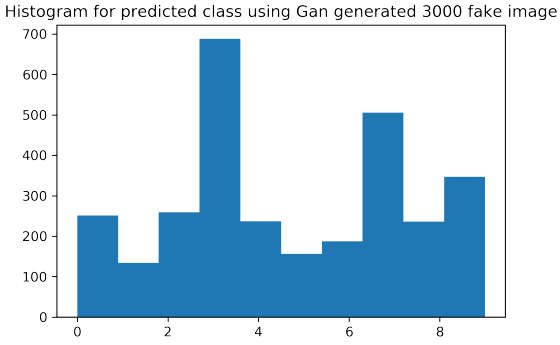
Solution:
The key change made to implement Unrolled GAN compared with the Vanilla GAN is an train_unroll_discriminator function after the discriminator training at each epoch in the training process. This function takes in the a deep copy of the current discriminator, k, current generator, and a shuffled training dataloader. In the function, we use a separate optimizer for the current unrolled discriminator, and for each step of k, we sample a batch of real images and fake generated images, train the unrolled discriminator and return it. Then, using the current epoch trained unrolled discriminator, we train the generator and update the generator optimizer. Training with k = 3, batch size = 100, epoch = 50 on the training dataset, the generator and discriminator loss, as well as the procedural generated fake images are as follows.



Recreate what's done in part 3 with the generator from the trained unrolled GAN, the histogram of the classified image labels are as follows.



Also, for the Wasserstein GAN we trained in part 2, we also retrieve its trained generator and plot the histogram of the classified image labels. And the result is the following.



By analyzing the level of label balance produced by the generators from the three GAN models, we conclude that the improvement from WGAN is not obvious, yet we see a much balanced and uniform distribution of predicted labels produced by the unrolled GAN. So we conclude that unrolled GAN is less affected by mode collapse.

Part 5 - 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.

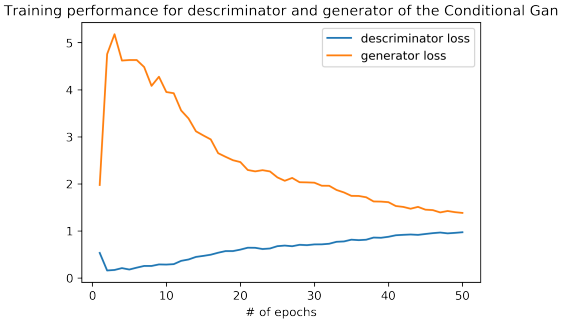
Train a Conditional GAN using the training script from Part 1. Plot training curves for D and G. Generate 3 samples from each of the 10 classes. Discuss differences in the generated images produced compared to the non-conditional models you built.

Solution:

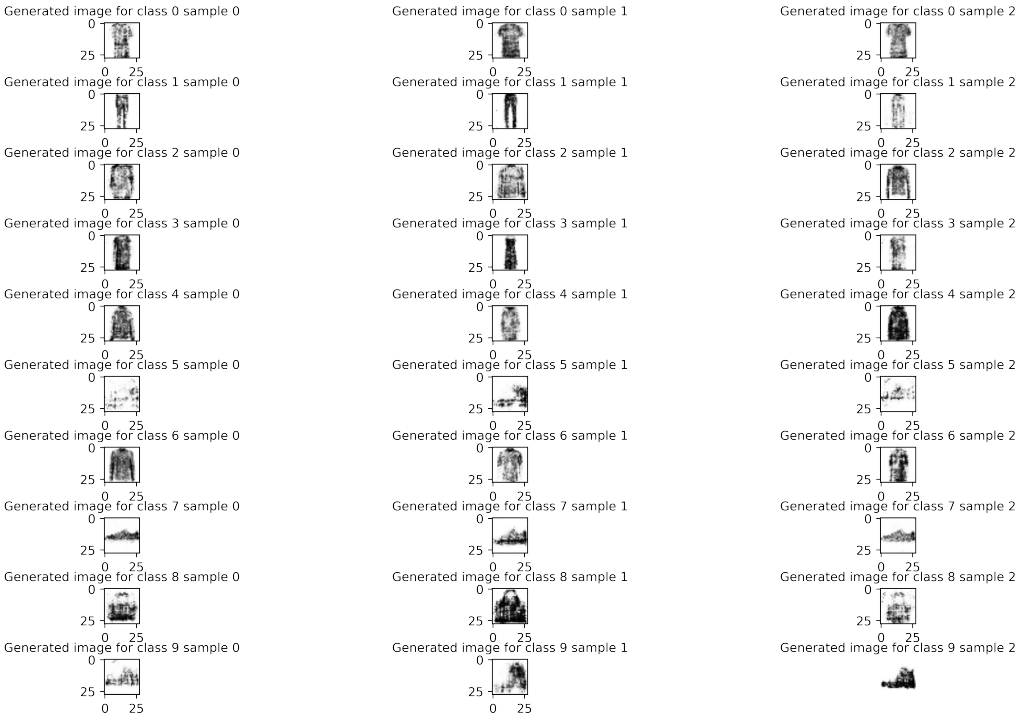
The conditional Gan has its difference in the input dimension of the generator and that of discriminator. With 10 classes in the FashionMINIST dataset, we have the $64 + 10 = 74$ input dimensions in the generator and discriminator. Also, the true label of each batch of data in training is embedded using one-hot-encoding method and concatenated with the flattened image batch to serve as input for the discriminator. A clear structure of how the generator and discriminator looks.

```
Generator(
  (model): Sequential(
    (0): Linear(in_features=74, out_features=256, bias=True)
    (1): ReLU()
    (2): Linear(in_features=256, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=1024, bias=True)
    (5): ReLU()
    (6): Linear(in_features=1024, out_features=784, bias=True)
    (7): Tanh()
  )
)
Discriminator(
  (model): Sequential(
    (0): Linear(in_features=794, out_features=1024, bias=True)
    (1): ReLU()
    (2): Linear(in_features=1024, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=256, bias=True)
    (5): ReLU()
    (6): Linear(in_features=256, out_features=1, bias=True)
    (7): Sigmoid()
  )
)
```

Training with 50 epochs, with learning rate of 0.00005, we have the generator, discriminator loss, and procedural generated images as the following.



Generated 3 samples from each of the 10 classes using trained generator from conditional GAN



The difference in the generated image between the conditional GAN and the previous GANs without label embedding is that we can generate desired class of images instead of random generation, and we ensure that we generate designated class of image with the well-trained generator (not procedural image). Also, observing the 3 examples generated from each of the 10 classes of the FashionMINIST dataset, we see the conditional GAN generator produces pretty stable image outcome of the designated class.

Code Appendix

0.1 Problem 1 - GRU for Sentiment Analysis

0.1.1 Part 1 - Preprocessing

```
import torch
from torch.utils.data import Dataset, TensorDataset, DataLoader
import torchvision
import torchvision.transforms as transforms
import unicodedata
import string
import re
from collections import Counter
from sklearn.model_selection import train_test_split
import operator
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )
def convert_string(s):
    cleanr = re.compile('<.*?>')
    s = re.sub(cleanr, '', s)
    s = unicodeToAscii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    s = re.sub(r'[^\w\s]', '',s)


    return s
def preprocess(string_col):
    v = []
    processed_col = []
    for i in string_col:
        processed_sentence = convert_string(i)
        processed_col.append(processed_sentence)
        v.extend(processed_sentence.split())
    Vocab = Counter(v)
    return processed_col, Vocab
    #return list of processed string, build vocabulary based on both the train and test inp
def load_file(file_name): #return list of strings
```

```
    file = open(file_name,"r")
    data_col = file.readlines()
    file.close()
    return data_col
train_pos = load_file("data/train_pos_merged.txt")
train_neg = load_file("data/train_neg_merged.txt")
test_pos = load_file("data/test_pos_merged.txt")
test_neg = load_file("data/test_neg_merged.txt")
train_pos_len = len(train_pos)
train_neg_len = len(train_neg)
test_pos_len = len(test_pos)
test_neg_len = len(test_neg)
print("training positive size:",train_pos_len)
print("training negative size:",train_neg_len)
print("test positive size:",test_pos_len)
print("test negative size:",test_neg_len)
all_input = []
all_input.extend(train_pos)
all_input.extend(train_neg)
all_input.extend(test_pos)
all_input.extend(test_neg)
processed_col, vocab = preprocess(all_input)
word_to_idx_input = {w: idx+1 for idx, w in enumerate(list(vocab.keys()))}
idx_to_word_input = {idx+1: w for idx, w in enumerate(list(vocab.keys()))}
print("There are %s encoded vocabulary"%len(vocab))
import numpy as np
def convert_to_matrix(string_col, max_length,word_to_idx_input,vocab):
    words_covered_total = []
    output = []
    sorted_vocab = dict( sorted(vocab.items(), key=operator.itemgetter(1),reverse=True))
    sorted_key = list(sorted_vocab.keys())
    for i in string_col:
        tokenized = i.split()
        cur = []
        if len(tokenized) <=max_length:
            cur.extend([0]*(max_length-len(tokenized)))
            for l in range(len(tokenized)):
#                 cur.append(word_to_idx_input[tokenized[l]])
                cur.append(tokenized[l])
                words_covered_total.append(tokenized[l])
        else: #add in the top most frequent 400 tokens
```

```
            #encode the first 400 most frequent
            idx_col = [sorted_key.index(i) for i in tokenized]
            sort_idx = np.argsort(np.array(idx_col))
            cur = []
            for j in range(len(tokenized)):
                if sort_idx[j]<400:
#                     cur.append(word_to_idx_input[tokenized[j]])
                    cur.append(tokenized[j])
                    words_covered_total.append(tokenized[j])

        output.append(cur)
    new_vocab = Counter(words_covered_total)
    word_to_idx_input_new = {w: idx+1 for idx, w in enumerate(list(new_vocab.keys()))}
    idx_to_word_input_new = {idx+1: w for idx, w in enumerate(list(new_vocab.keys()))}
    for i in range(len(output)):
        for j in range(len(output[i])):
            if output[i][j] != 0:
                output[i][j] = word_to_idx_input_new[output[i][j]]
    return output,new_vocab,word_to_idx_input_new,idx_to_word_input_new
input_matrix,new_vocab,word_to_idx_input_new,idx_to_word_input_new2 =
convert_to_matrix(processed_col, 400,word_to_idx_input,vocab)
train_input_matrix = input_matrix[:train_pos_len+train_neg_len]
test_input_matrix = input_matrix[train_pos_len+train_neg_len:]
train_y = [1]* train_pos_len
train_y.extend([0]* train_neg_len)
test_y = [1]* test_pos_len
test_y.extend([0]* test_neg_len)
train_x = np.array(train_input_matrix)
test_x = np.array(test_input_matrix)
train_y = np.array(train_y).reshape(len(train_y),1)
test_y = np.array(test_y).reshape(len(test_y),1)
train_x, dev_x, train_y, dev_y = train_test_split( train_x, train_y,
test_size=1000, random_state=42)
print(train_x.shape, train_y.shape)
print(dev_x.shape, dev_y.shape)
print(test_x.shape, test_y.shape)
train_data = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y))
dev_data = TensorDataset(torch.from_numpy(dev_x), torch.from_numpy(dev_y))
test_data = TensorDataset(torch.from_numpy(test_x), torch.from_numpy(test_y))
train_x_full = np.concatenate((train_x,dev_x))
train_y_full = np.concatenate((train_y,dev_y))
```

```
train_full_data = TensorDataset(torch.from_numpy
(train_x_full), torch.from_numpy(train_y_full))
```

0.1.2 Part 2 - Build A Binary Prediction RNN with GRU

```
import time
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.autograd as autograd
import torch.optim as optim
class GRU(nn.Module):
    def __init__(self,input_dim, embed_dim,hidden_dim, bidirectional):
        super(GRU, self).__init__()
        #embedding unit
        self.embedding = nn.Embedding(input_dim, embed_dim)
        #gru unit
        self.bidirectional = bidirectional
        self.gru = nn.GRU(embed_dim, hidden_dim, num_layers=1,batch_first=True,
        bidirectional=bidirectional)#, bidirectional=True
        self.dropout = nn.Dropout(p=0.5)
        #prediction unit, combined with sigmoid
        if bidirectional:
            self.linear = nn.Linear(2* hidden_dim, 1)
        else:

            self.linear = nn.Linear(hidden_dim, 1)

    def forward(self, x, h):
        x = self.embedding(x)
        x, h = self.gru(x, h)
        x = self.dropout(x[:,-1,:].squeeze()) # just get the last hidden state
        x = torch.sigmoid(self.linear(x)) # sigmoid output for binary classification
        return x, h

    def init_hidden(self):
        if self.bidirectional:
          return autograd.Variable(torch.randn(2, batch_size, hidden_dim))#64
        else:
          return autograd.Variable(torch.randn(1, batch_size, hidden_dim))#64
```

```
def train_test_experiment(epoch, batch_size, embed_dim, hidden_dim,lr ,
vocab, bidirectional):
    epoch = epoch
    batch_size = batch_size
    input_dim, embed_dim, hidden_dim = len(vocab)+1,embed_dim,hidden_dim
    model = GRU(input_dim, embed_dim, hidden_dim,bidirectional)
    model.cuda()
    loss_fn = nn.BCELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    train_data = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y))
    train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
    dev_loader = DataLoader(dev_data, shuffle=True, batch_size=batch_size)
    test_loader = DataLoader(test_data, shuffle=True, batch_size=batch_size)
    h = model.init_hidden()
    h = h.cuda()
    # model.train()
    print("Begin training...")
    for e in range(epoch):  # loop over the dataset multiple times
        model.train()
        print('\n' + 'Epoch {}/{}'.format(e+1, epoch))
        start = time.time()
        iter_loss = 0.
        iter_correct = 0.
        for i, data in enumerate(train_loader, 0):
            inputs, labels = data
            inputs, labels = inputs.cuda(), labels.cuda()
            optimizer.zero_grad()
            h.detach_()
            y_pred, h = model(inputs, h)
            loss = loss_fn(y_pred.to(torch.float32), labels.to(torch.float32))
            loss.backward()
            optimizer.step()
            trn_preds = torch.round(y_pred.data)
            iter_correct += torch.sum(trn_preds == labels.data)
            iter_loss += loss.item()
        print('Training Loss: {:.3} | Training Acc: {:.3}'.
        format(iter_loss / len((train_x)), float(iter_correct) / len((train_x))))
        print('Time: {}'.format(time.time()-start)    )
        print("Evaluting dev...")
        dev_loss = 0
        dev_correct = 0
```

```
        model.eval()
        for i, data in enumerate(dev_loader, 0):
            inputs, labels = data
            inputs, labels = inputs.cuda(), labels.cuda()
            h.detach_()
            y_pred, h = model(inputs, h)
            loss = loss_fn(y_pred.to(torch.float32), labels.to(torch.float32))
            trn_preds = torch.round(y_pred.data)
            dev_correct += torch.sum(trn_preds == labels.data)
            dev_loss += loss.item()
        print('Dev Loss: {:.3} | Dev Acc: {:.3}'.format(dev_loss / len((dev_x)),
        float(dev_correct) / len((dev_x))))
    del model
    torch.cuda.empty_cache()
    return iter_correct/len(train_x),dev_correct/len(dev_x)
import matplotlib.pyplot as plt
def experiment_epoch(epoch_col):
    epoch_train_acc = []
    epoch_test_acc = []
    for i in epoch_col:
        print("For epoch = %s"%(i))
        epoch, batch_size, embed_dim, hidden_dim = i,500,128,64
        train_acc, test_acc =  train_test_experiment(epoch, batch_size, embed_dim,
        hidden_dim,0.001,new_vocab,True)
        epoch_train_acc.append(train_acc)
        epoch_test_acc.append(test_acc)
    plt.figure(dpi = 300)
    plt.plot(epoch_col,epoch_train_acc,label = "training accuracy" )
    plt.plot(epoch_col,epoch_test_acc,label = "Validating accuracy" )
    plt.xlabel("# of epochs")
    plt.legend()
    plt.show()
epoch_col = [5*i for i in range(2,10)]
experiment_epoch(epoch_col)
def experiment_embed(embed_col):
    epoch_train_acc = []
    epoch_test_acc = []
    for i in embed_col:
        print("For embedding_dim = %s"%(i))
        epoch, batch_size, embed_dim, hidden_dim = 35,500,i,64
        train_acc, test_acc =  train_test_experiment(epoch, batch_size, embed_dim,
```

```
        hidden_dim,0.001,new_vocab,True)
        epoch_train_acc.append(train_acc)
        epoch_test_acc.append(test_acc)
    plt.figure(dpi = 300)
    plt.plot(embed_col,epoch_train_acc,label = "training accuracy" )
    plt.plot(embed_col,epoch_test_acc,label = "Validating accuracy" )
    plt.xlabel("# of embedding dimensions")
    plt.legend()
    plt.show()
embed_col = [50* i for i in range(1,7)]
experiment_embed(embed_col)
def experiment_hidden(hidden_col):
    epoch_train_acc = []
    epoch_test_acc = []
    for i in hidden_col:
        print("For hidden dim = %s"%(i))
        epoch, batch_size, embed_dim, hidden_dim = 35,500,150,i
        train_acc, test_acc =  train_test_experiment(epoch, batch_size, embed_dim,
        hidden_dim ,0.001,new_vocab,True)
        epoch_train_acc.append(train_acc)
        epoch_test_acc.append(test_acc)
    plt.figure(dpi = 300)
    plt.plot(embed_col,epoch_train_acc,label = "training accuracy" )
    plt.plot(embed_col,epoch_test_acc,label = "Validating accuracy" )
    plt.xlabel("# of Hidden dimensions")
    plt.legend()
    plt.show()
hidden_col = [16*i for i in range(1,5)]
experiment_hidden(hidden_col)
def test_evaluation(epoch, batch_size,embed_dim,hidden_dim,lr, vocab, bidirectional ):
    epoch = epoch
    batch_size = batch_size
    input_dim, embed_dim, hidden_dim = len(vocab)+1,embed_dim,hidden_dim
    model = GRU(input_dim, embed_dim, hidden_dim,bidirectional)
    model.cuda()
    loss_fn = nn.BCELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

    # train_data = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y))
    train_full_loader = DataLoader(train_full_data, shuffle=True, batch_size=batch_size)
    # dev_loader = DataLoader(dev_data, shuffle=True, batch_size=batch_size)
```

```
test_loader = DataLoader(test_data, shuffle=True, batch_size=batch_size)
h = model.init_hidden()
h = h.cuda()
# model.train()
print("Begin training...")
for e in range(epoch):  # loop over the dataset multiple times
    model.train()
    print('\n' + 'Epoch {}/{}'.format(e+1, epoch))
    start = time.time()
    iter_loss = 0.
    iter_correct = 0.
    for i, data in enumerate(train_full_loader, 0):
        inputs, labels = data
        inputs, labels = inputs.cuda(), labels.cuda()
        optimizer.zero_grad()
        h.detach_()
        y_pred, h = model(inputs, h)
        loss = loss_fn(y_pred.to(torch.float32), labels.to(torch.float32))
        loss.backward()
        optimizer.step()
        trn_preds = torch.round(y_pred.data)
        iter_correct += torch.sum(trn_preds == labels.data)
        iter_loss += loss.item()
    print('Training Loss: {:.3} | Training Acc: {:.3}'.format(iter_loss /
    len((train_x_full)), float(iter_correct) / len((train_x_full))))
    print('Time: {}'.format(time.time()-start)    )

    if e == epoch-1:
      print("Evaluting testing...")
      test_loss = 0
      test_correct = 0
      model.eval()
  #         with torch.no_grad():

      for i, data in enumerate(test_loader, 0):
          inputs, labels = data
          inputs, labels = inputs.cuda(), labels.cuda()
          h.detach_()
      #             inputs, labels =inputs.to(device),labels.to(device)
          y_pred, h = model(inputs, h)
          loss = loss_fn(y_pred.to(torch.float32), labels.to(torch.float32))
```

```
            trn_preds = torch.round(y_pred.data)
            test_correct += torch.sum(trn_preds == labels.data)
            test_loss += loss.item()
        print('Test Loss: {:.3} | Test Acc: {:.3}'.format(test_loss /
        len((test_x)), float(test_correct) / len((test_x))))
    del model
    torch.cuda.empty_cache()
    return iter_correct/len(train_x),test_correct/len(test_x)
epoch, batch_size,embed_dim,hidden_dim,  bidirectional = 35,500,150,80, True
test_evaluation(epoch, batch_size,embed_dim,hidden_dim, 0.001,new_vocab, bidirectional )
```

0.1.3 Part 3 - Comparison with a MLP

```
class MLP(nn.Module):
    def __init__(self,input_dim, hidden_dim):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, 2)
        )

    def forward(self, x):
        # convert tensor (128, 1, 28, 28) --> (128, 1*28*28)
        x = x.view(x.size(0), -1)
        x = self.layers(x)
        return x
import time
def train_mlp_experiment(epoch, batch_size, hidden_dim,lr ):
    epoch = epoch
    batch_size = batch_size
    input_dim, hidden_dim =400,hidden_dim
    model = MLP(input_dim, hidden_dim)
    model.cuda()
    loss_fn = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
    dev_loader = DataLoader(dev_data, shuffle=True, batch_size=batch_size)
    test_loader = DataLoader(test_data, shuffle=True, batch_size=batch_size)
    for e in range(epoch):  # loop over the dataset multiple times
        model.train()
```

```
        start = time.time()
        train_loss = 0.0
        train_correct = 0.
        for data, target in train_loader:
          data, target = data.cuda(), target.cuda()
          data, target = data.to(torch.float32), target.long()
          optimizer.zero_grad()
          output = model(data)
          predicted_label = torch.argmax(output, dim = 1)
          train_correct+= torch.sum(predicted_label == target.data)
          loss = loss_fn(output, target)
          loss.backward()
          optimizer.step()
          train_loss += loss.item()*data.size(0)
        train_loss = train_loss/len(train_loader.dataset)
        dev_loss = 0
        dev_correct = 0
        model.eval()
        for data, target in dev_loader:
          data, target = data.cuda(), target.cuda()
          data, target = data.to(torch.float32), target.long()
          output = model(data)
          predicted_label = torch.argmax(output, dim = 1)
          dev_correct+= torch.sum(predicted_label == target.data)
          loss = loss_fn(output, target)
          dev_loss += loss.item()*data.size(0)
    return train_correct /len(train_loader.dataset) ,float(dev_correct) /
    len(dev_loader.dataset)
def experiment_mlp_epoch(epoch_col):
    epoch_train_acc = []
    epoch_test_acc = []
    for i in epoch_col:
        print("For epoch = %s"%(i))
        epoch, batch_size, hidden_dim = i, 500, 32
        train_acc, test_acc =  train_mlp_experiment(epoch, batch_size, hidden_dim,0.001 )
        epoch_train_acc.append(train_acc)
        epoch_test_acc.append(test_acc)
    plt.figure(dpi = 300)
    plt.plot(epoch_col,epoch_train_acc,label = "training accuracy" )
    plt.plot(epoch_col,epoch_test_acc,label = "Validating accuracy" )
    plt.xlabel("# of epochs")
```

```
    plt.legend()
    plt.show()
epoch_col = [i*10 for i in range(2,13)]
experiment_mlp_epoch(epoch_col)
def experiment_mlp_hidden(hidden_col):
    epoch_train_acc = []
    epoch_test_acc = []
    for i in hidden_col:
        print("For epoch = %s"%(i))
        epoch, batch_size, hidden_dim = 50, 500, i
        train_acc, test_acc =  train_mlp_experiment(epoch, batch_size, hidden_dim,0.001 )
        epoch_train_acc.append(train_acc)
        epoch_test_acc.append(test_acc)
    plt.figure(dpi = 300)
    plt.plot(hidden_col,epoch_train_acc,label = "training accuracy" )
    plt.plot(hidden_col,epoch_test_acc,label = "Validating accuracy" )
    print(epoch_test_acc)
    plt.xlabel("# of hidden units")
    plt.legend()
    plt.show()
hidden_col = [8*i for i in range(1,14)]
experiment_mlp_hidden(hidden_col)
import time
def predict_mlp_test(epoch, batch_size, hidden_dim,lr ):
    epoch = epoch
    batch_size = batch_size
    input_dim, hidden_dim =400,hidden_dim
    model = MLP(input_dim, hidden_dim)
    model.cuda()
    loss_fn = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    train_full_loader = DataLoader(train_full_data, shuffle=True, batch_size=batch_size)
    test_loader = DataLoader(test_data, shuffle=True, batch_size=batch_size)

    for e in range(epoch):
        model.train()
        start = time.time()
        train_loss = 0.0
        train_correct = 0.
        for data, target in train_full_loader :
          data, target = data.cuda(), target.cuda()
```

```
            data, target = data.to(torch.float32), target.long()
            optimizer.zero_grad()
            output = model(data)
            predicted_label = torch.argmax(output, dim = 1)
            train_correct+= torch.sum(predicted_label == target.data)
            loss = loss_fn(output, target)
            loss.backward()
            optimizer.step()
            train_loss += loss.item()*data.size(0)
          train_loss = train_loss/len(train_full_loader.dataset)
          dev_loss = 0
          dev_correct = 0
          model.eval()
          for data, target in test_loader:
            data, target = data.cuda(), target.cuda()
            data, target = data.to(torch.float32), target.long()
            output = model(data)
            predicted_label = torch.argmax(output, dim = 1)
            dev_correct+= torch.sum(predicted_label == target.data)
            loss = loss_fn(output, target)
            dev_loss += loss.item()*data.size(0)
      print('Epoch: {} \tTraining Loss: {:.6f} \t Training Acc: {:.3}'.
      format(e+1, train_loss,train_correct /len(train_full_loader.dataset) ))
      print('Test Loss: {:.3} | Test Acc: {:.3}'.format(dev_loss / len(train_full_loader.data
      float(dev_correct) / len(train_full_loader.dataset)))
      return train_correct /len(train_full_loader.dataset) ,float(dev_correct) /
      len(test_loader.dataset)
predict_mlp_test(50, 500,80,0.001 )
```

0.2 Problem 2 - Generative Adversarial Networks

0.2.1 Part 1 - Vanilla GAN

```
%matplotlib inline
import torch
import torch.nn as nn
import pandas as pd
import numpy as np
from torchvision import transforms
from torch.utils.data import Dataset, DataLoader
from PIL import Image
```

```
from torch import autograd
from torch.autograd import Variable
from torchvision.utils import make_grid
import matplotlib.pyplot as plt
Discriminator = nn.Sequential(
    nn.Linear(28 * 28, 256),
    nn.ReLU(),
    nn.Linear(256, 256),
    nn.ReLU(),
    nn.Linear(256, 256),
    nn.ReLU(),
    nn.Linear(256, 1),
    nn.Sigmoid()
)

# Generator
Generator = nn.Sequential(
    nn.Linear(64, 256),
    nn.ReLU(),
    nn.Linear(256, 256),
    nn.ReLU(),
    nn.Linear(256, 256),
    nn.ReLU(),
    nn.Linear(256, 28 * 28),
    nn.Tanh()
)

if torch.cuda.is_available():
    D.cuda()
    G.cuda()
from torch.autograd import Variable
def to_var(x):
    if torch.cuda.is_available():
        x = x.cuda()
    return Variable(x)
import torch.nn.functional as F
from torchvision.utils import save_image
def denorm(x):
    out = (x + 1) / 2
    return out
BATCH_SIZE = 100
```

```
data_loader = torch.utils.data.DataLoader(dataset=dataset,
                                          batch_size=BATCH_SIZE,
                                          shuffle=False,
                                          num_workers=0)
n_batches = int(np.ceil(len(dataset)/ BATCH_SIZE))
def training(N_EPOCHS,data_loader):
    d_loss_col = []
    g_loss_col = []
    idx_1 = 0
    idx_2 = N_EPOCHS //2
    idx_3 = N_EPOCHS - 1
    img_g_sample = []
    for epoch in range(N_EPOCHS):
        # loop through batches (no need for class labels right now)
        iter_d_loss = 0
        iter_g_loss = 0
        for batch_number, (images, _) in enumerate(data_loader):
            batch_size = images.shape[0]
            images = to_var(images.view(batch_size, -1))
            real_labels = to_var(torch.ones(batch_size, 1))
            fake_labels = to_var(torch.zeros(batch_size, 1))

            # 1) TRAIN DISCRIMINATOR
            outputs = Discriminator(images)
            real_score = outputs
            d_loss_real = criterion(outputs, real_labels)

            # Draw random 64-dimensional noise vectors as inputs to the generator network
            z = to_var(torch.randn(batch_size, 64))
            fake_images = Generator(z)
            # Evaluate the discriminator on the fake images
            outputs = Discriminator(fake_images) # or D(add_instance_noise(fake_images))
            fake_score = outputs
            d_loss_fake = criterion(outputs, fake_labels)
            d_loss = d_loss_real + d_loss_fake
            D.zero_grad()
            d_loss.backward()
            d_optimizer.step()

            # 2) TRAIN GENERATOR
            z = to_var(torch.randn(batch_size, 64))
```

```
            fake_images = Generator(z)
            outputs = Discriminator(fake_images)
            g_loss = criterion(outputs, real_labels)
            Discriminator.zero_grad()
            Generator.zero_grad()
            g_loss.backward()
            g_optimizer.step()
            iter_d_loss +=d_loss.data
            iter_g_loss +=g_loss.data
            if (batch_number + 1) % 300 == 0:
                print('Epoch [%d/%d], Step[%d/%d], d_loss: %.4f, '
                      'g_loss: %.4f, Mean D(x): %.2f, Mean D(G(z)): %.2f'
                      %(epoch,
                        N_EPOCHS,
                        batch_number + 1,
                        n_batches,
                        d_loss.data,
                        g_loss.data,
                        real_score.data.mean(),
                        fake_score.data.mean())
                     )
        if epoch in [idx_1,idx_2,idx_3]:
            fake_images = fake_images.view(fake_images.size(0), 1, 28, 28)
            img_g_sample.append(denorm(fake_images.data))

        d_loss_col.append(iter_d_loss/len(data_loader))
        g_loss_col.append(iter_g_loss/len(data_loader))
    return d_loss_col,g_loss_col,img_g_sample
N_EPOCHS = 150
d_loss_col,g_loss_col,img_g_sample = training(N_EPOCHS,data_loader)
import matplotlib.pyplot as plt
def evaluate_performance(N_EPOCHS,d_loss_col, g_loss_col,img_g_sample):
    plt.figure(dpi = 300)
    plt.title("Training performance for descriminator and generator of the Vanilla Gan")
    plt.plot(np.arange(1,N_EPOCHS + 1),d_loss_col, label = "descriminator loss")
    plt.plot(np.arange(1,N_EPOCHS + 1),g_loss_col, label = "generator loss")
    plt.xlabel("# of epochs")
    plt.legend()
    plt.show()
    for i in range(3):
        plt.imshow(img_g_sample[i][0].reshape(28,28), interpolation='bilinear',
```

```
        cmap='Greys')
        plt.show()
evaluate_performance(N_EPOCHS,d_loss_col, g_loss_col,img_g_sample)
```

0.2.2 Part 2 - GAN Loss

```
#MSE
def training_MSE(N_EPOCHS,data_loader):
    d_optimizer = torch.optim.Adam(Discriminator.parameters(), lr=0.0003)
    g_optimizer = torch.optim.Adam(Generator.parameters(), lr=0.0003)
    d_loss_col = []
    g_loss_col = []
    idx_1 = 0
    idx_2 = N_EPOCHS //2
    idx_3 = N_EPOCHS - 1
    img_g_sample = []
    for epoch in range(N_EPOCHS):
        iter_g_loss = 0
        for batch_number, (images, _) in enumerate(data_loader):
            g_optimizer.zero_grad()
            batch_size = images.shape[0]
            images = to_var(images.view(batch_size, -1))
            real_labels = to_var(torch.ones(batch_size, 1))
            fake_labels = to_var(torch.zeros(batch_size, 1))

             # 2) TRAIN GENERATOR
            z = to_var(torch.randn(batch_size, 64))
            fake_images = Generator(z)
            g_loss =  torch.mean((fake_images - images)**2)
            g_loss.backward()
            g_optimizer.step()

            iter_g_loss +=g_loss.data
            if (batch_number + 1) % 300 == 0:
                print('Epoch [%d/%d], Step[%d/%d], g_loss: %.4f, '
                      %(epoch,
                        N_EPOCHS,
                        batch_number + 1,
                        n_batches,
                        g_loss.data,
```

```
                    ))
        if epoch in [idx_1,idx_2,idx_3]:
            fake_images = fake_images.view(fake_images.size(0), 1, 28, 28)
            img_g_sample.append(denorm(fake_images.data))
        g_loss_col.append(iter_g_loss/len(data_loader))
d_loss_col,g_loss_col,img_g_sample = training_MSE(N_EPOCHS,data_loader)
evaluate_performance(N_EPOCHS,d_loss_col, g_loss_col,img_g_sample, "MSE on generator")
# Wasserstein GAN (WGAN)
critic = nn.Sequential(
    nn.Linear(28 * 28, 512),
    nn.LeakyReLU(0.2),
    nn.Linear(512, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 1)
)

# Generator
generator = nn.Sequential(
    nn.Linear(64, 128),
    nn.LeakyReLU(0.2),
    nn.Linear(128, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 512),
    nn.LeakyReLU(0.2),
    nn.Linear(512, 28 * 28),
    nn.Tanh()
)

N_EPOCHS = 50
def training_Wasserstein(N_EPOCHS,data_loader,CLAMP):
    start_time = time.time()
    critic_optimizer = torch.optim.RMSprop(critic.parameters(), lr=0.00005)
    generator_optimizer = torch.optim.RMSprop(generator.parameters(), lr=0.00005)
    # loop through epochs
    d_loss_col = []
    g_loss_col = []
    idx_col = [0,N_EPOCHS//2, N_EPOCHS-1]
    img_g_sample = []
    for epoch in range(N_EPOCHS):
        crit_steps = 0
        iter_d_loss = 0
```

```
iter_g_loss = 0
for batch_number, (images, _) in enumerate(data_loader):
    batch_size = images.shape[0]
    images = to_var(images.view(batch_size, -1))
    critic.zero_grad()
    generator.zero_grad()
    for p in critic.parameters():
        p.data.clamp_(-CLAMP, CLAMP)
    err_real = torch.mean(critic(images))
    z = to_var(torch.randn(batch_size, 64))
    fake_images = generator(z)
    err_fake = torch.mean(critic(fake_images))
    critic_loss = err_fake - err_real
    critic_loss.backward()
    critic_optimizer.step()
    iter_d_loss += critic_loss.data
    critic.zero_grad()
    generator.zero_grad()
    z = to_var(torch.randn(batch_size, 64))
    fake_images = generator(z)
    outputs = critic(fake_images)
    generator_loss = -torch.mean(outputs)
    generator_loss.backward()
    generator_optimizer.step()
    crit_steps = 0
    if (batch_number + 1) % 300 == 0:
        print('Epoch [%d/%d], Step[%d/%d], d_loss: %.4f, '
                  'g_loss: %.4f, Mean D(x): %.2f, Mean D(G(z)): %.2f'
                  %(epoch,
                    N_EPOCHS,
                    batch_number + 1,
                    n_batches,
                    critic_loss.data,
                    generator_loss.data,
                    err_real.data.mean(),
                    err_fake.data.mean())
              )

    iter_g_loss += generator_loss.data
if epoch in idx_col:
    fake_images = fake_images.view(fake_images.size(0), 1, 28, 28)
```

```
            img_g_sample.append(denorm(fake_images.data))
        d_loss_col.append(iter_d_loss /len(data_loader))
        g_loss_col.append(iter_g_loss /len(data_loader))
    print("Training time %s"%(time.time() - start_time))
    return d_loss_col, g_loss_col,img_g_sample
c_col = [0.1, 0.01, 0.001, 0.0001]
for i in c_col:
    print("Evaluating c = ", i)
    N_EPOCHS,CLAMP = 50,i
    d_loss_col, g_loss_col,img_g_sample = training_Wasserstein(N_EPOCHS,data_loader,CLAMP)
    print("The training loss and sample images from the training generators")
    evaluate_performance(N_EPOCHS,d_loss_col, g_loss_col,img_g_sample, "Wasserstein")
    print("==============================")
#Least Square Gan
Discriminator_LS = nn.Sequential(
    nn.Linear(28 * 28, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 1),
)

# Generator
Generator  = nn.Sequential(
    nn.Linear(64, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 28 * 28),
    nn.Tanh()
def training_vanilla_with_LSLoss(N_EPOCHS,data_loader):
    start_time = time.time()
    d_optimizer = torch.optim.Adam(Discriminator_LS .parameters(), lr=0.00005)
```

```
g_optimizer = torch.optim.Adam(Generator.parameters(), lr=0.00005)
d_loss_col = []
g_loss_col = []
idx_1 = 0
idx_2 = N_EPOCHS //2
idx_3 = N_EPOCHS - 1
img_g_sample = []
for epoch in range(N_EPOCHS):
    iter_d_loss = 0
    iter_g_loss = 0
    for batch_number, (images, _) in enumerate(data_loader):
        d_optimizer.zero_grad()

        batch_size = images.shape[0]
        images = to_var(images.view(batch_size, -1))
        real_labels = to_var(torch.ones(batch_size, 1))
        fake_labels = to_var(torch.zeros(batch_size, 1))

        # 1) TRAIN DISCRIMINATOR
        outputs_real = Discriminator_LS(images)
        z = to_var(torch.randn(batch_size, 64))
        fake_images = Generator(z)
        outputs_fake = Discriminator_LS(fake_images)
        d_loss = (torch.mean((outputs_real -  real_labels)**2) + torch.mean((outputs_fa
        d_loss.backward()
        d_optimizer.step()
        # 2) TRAIN GENERATOR
        g_optimizer.zero_grad()
        fake_images = Generator(z)
        outputs_fake = Discriminator_LS(fake_images)
        g_loss = torch.mean((outputs_fake - real_labels)**2)
        g_loss.backward()
        g_optimizer.step()
        iter_d_loss +=d_loss.data
        iter_g_loss +=g_loss.data
        if (batch_number + 1) % 300 == 0:
            print('Epoch [%d/%d], Step[%d/%d], d_LSloss: %.4f, '
                  'g_LSloss: %.4f,'
                  %(epoch,
                    N_EPOCHS,
                    batch_number + 1,
```

```
                        n_batches,
                        d_loss.data,
                        g_loss.data,
                    ))
        if epoch in [idx_1,idx_2,idx_3]:
            fake_images = fake_images.view(fake_images.size(0), 1, 28, 28)
            img_g_sample.append(denorm(fake_images.data))

        d_loss_col.append(iter_d_loss/len(data_loader))
        g_loss_col.append(iter_g_loss/len(data_loader))
    print("Time used to train %s epcohs is %s"(N_EPOCHS, time.time()-start_time))
    return d_loss_col,g_loss_col,img_g_sample
BATCH_SIZE = 100
data_loader = torch.utils.data.DataLoader(dataset=dataset,
                                          batch_size=BATCH_SIZE,
                                          shuffle=False,
                                          num_workers=0)
n_batches = int(np.ceil(len(dataset)/ BATCH_SIZE)) N_EPOCHS = 150
d_loss_col,g_loss_col,img_g_sample = training_vanilla_with_LSLoss(N_EPOCHS,data_loader)
evaluate_performance(N_EPOCHS,d_loss_col, g_loss_col,img_g_sample, "Least Square Gan")
```

0.2.3 Part 3 - Mode Collapse in GANs

```
import numpy as np
import matplotlib.pyplot as plt
import torch
from torchvision import datasets, transforms
from torch import nn
from torch.autograd import Variable
import torch.nn.functional as F
from torchvision.utils import save_image
%matplotlib inline
Discriminator_part3 = nn.Sequential(
    nn.Linear(28 * 28, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
```

```
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 10),
    nn.Sigmoid()
)
Discriminator = nn.Sequential(
    nn.Linear(28 * 28, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 1),
    nn.Sigmoid()
)

# Generator
Generator  = nn.Sequential(
    nn.Linear(64, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 28 * 28),
    nn.Tanh()
)


def denorm(x):
    # convert back from [-1, 1] to [0, 1]
    out = (x + 1) / 2
    return out #out.clamp(0, 1)
def to_var(x):
    # first move to GPU, if necessary
    if torch.cuda.is_available():
        x = x.cuda()
```

```
    return Variable(x)
def FashionMNIST_Classifer(epoch, batch_size):
    optimizer = torch.optim.Adam(Discriminator_part3.parameters(), lr=0.0003)
    BATCH_SIZE = batch_size
    criterion = nn.CrossEntropyLoss()
    train_data_loader = torch.utils.data.DataLoader(dataset=dataset_train,
                                            batch_size=BATCH_SIZE,
                                            shuffle=False,
                                            num_workers=0)
    test_data_loader = torch.utils.data.DataLoader(dataset=dataset_test,
                                            batch_size=BATCH_SIZE,
                                            shuffle=False,
                                            num_workers=0)
    for i in range(epoch):
        iter_d_loss = 0
        iter_training_acc_count = 0
        per_epoch_count = 0
        for batch_number, (images, labels) in enumerate(train_data_loader):
            batch_size = images.shape[0]
            per_epoch_count += batch_size
            images = to_var(images.view(batch_size, -1))
            labels = to_var(labels )
            preds = Discriminator_part3(images)
            pred_label = torch.argmax(preds, dim = 1)
            iter_training_acc_count +=  (pred_label == labels).sum().item()
#             print((pred_label == labels).sum())
            loss = criterion(preds, labels)
            iter_d_loss += loss.data
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        print("Average Training loss for epoch = %s is %s, training accuracy is %s"%
              (i+1,(iter_d_loss.item()  / per_epoch_count),
              (iter_training_acc_count/per_epoch_count)))
    print("start testing prediction and accuracy calculation...")
    acc_accurate_count = 0
    per_epoch_count = 0
    for batch_number, (images, labels) in enumerate(test_data_loader):
        batch_size = images.shape[0]
        per_epoch_count += batch_size
        images = to_var(images.view(batch_size, -1))
```

```
        labels = to_var(labels )
        preds_label = torch.argmax(Discriminator_part3(images),dim = 1)
        acc_accurate_count += (preds_label == labels).sum().item()
    print("test set accuracy is:",acc_accurate_count / per_epoch_count)
    return Discriminator_part3
epoch = 50
batch_size = 4096
Discriminator_trained = FashionMNIST_Classifer(epoch, batch_size)
def training_vanilla(N_EPOCHS,data_loader):
    d_optimizer = torch.optim.Adam(Discriminator.parameters(), lr=0.0003)
    g_optimizer = torch.optim.Adam(Generator.parameters(), lr=0.0003)
    criterion = nn.BCELoss()
    d_loss_col = []
    g_loss_col = []
    idx_1 = 0
    idx_2 = N_EPOCHS //2
    idx_3 = N_EPOCHS - 1
    img_g_sample = []
    for epoch in range(N_EPOCHS):
        # loop through batches (no need for class labels right now)
        iter_d_loss = 0
        iter_g_loss = 0
        for batch_number, (images, _) in enumerate(data_loader):
            batch_size = images.shape[0]
            images = to_var(images.view(batch_size, -1))
            real_labels = to_var(torch.ones(batch_size, 1))
            fake_labels = to_var(torch.zeros(batch_size, 1))

            # 1) TRAIN DISCRIMINATOR
            outputs = Discriminator(images)
            real_score = outputs
            d_loss_real = criterion(outputs, real_labels)

            # Draw random 64-dimensional noise vectors as inputs to the generator network
            z = to_var(torch.randn(batch_size, 64))
            fake_images = Generator(z)
            # Evaluate the discriminator on the fake images
            outputs = Discriminator(fake_images) # or D(add_instance_noise(fake_images))
            fake_score = outputs
            d_loss_fake = criterion(outputs, fake_labels)
            d_loss = d_loss_real + d_loss_fake
```

```
            Discriminator.zero_grad()
            d_loss.backward()
            d_optimizer.step()

            # 2) TRAIN GENERATOR
            z = to_var(torch.randn(batch_size, 64))
            fake_images = Generator(z)
            outputs = Discriminator(fake_images)
            g_loss = criterion(outputs, real_labels)
            Discriminator.zero_grad()
            Generator.zero_grad()
            g_loss.backward()
            g_optimizer.step()
            iter_d_loss +=d_loss.data
            iter_g_loss +=g_loss.data
            if (batch_number + 1) % 300 == 0:
                print('Epoch [%d/%d], Step[%d/%d], d_loss: %.4f, '
                      'g_loss: %.4f, Mean D(x): %.2f, Mean D(G(z)): %.2f'
                      %(epoch,
                        N_EPOCHS,
                        batch_number + 1,
                        n_batches,
                        d_loss.data,
                        g_loss.data,
                        real_score.data.mean(),
                        fake_score.data.mean())
                     )
        if epoch in [idx_1,idx_2,idx_3]:
            fake_images = fake_images.view(fake_images.size(0), 1, 28, 28)
            img_g_sample.append(denorm(fake_images.data))

        d_loss_col.append(iter_d_loss/len(data_loader))
        g_loss_col.append(iter_g_loss/len(data_loader))
    return d_loss_col,g_loss_col,img_g_sample,Generator, Discriminator
BATCH_SIZE = 100
train_data_loader = torch.utils.data.DataLoader(dataset=dataset_train,
                                                batch_size=BATCH_SIZE,
                                                shuffle=False,
                                                num_workers=0)
test_data_loader = torch.utils.data.DataLoader(dataset=dataset_test,
                                                batch_size=BATCH_SIZE,
```

```
                                            shuffle=False,
                                            num_workers=0)
n_batches = int(np.ceil(len(dataset_train)/ BATCH_SIZE)) # 6000
N_EPOCHS = 50
d_loss_col,g_loss_col,img_g_sample, trainedGenerator, trainedDiscriminator =
training_vanilla(N_EPOCHS,train_data_loader)
import matplotlib.pyplot as plt
def evaluate_performance(N_EPOCHS,d_loss_col, g_loss_col,img_g_sample, model):
    plt.figure(dpi = 300)
    plt.title("Training performance for descriminator and generator of the %s Gan"%(model))
    plt.plot(np.arange(1,N_EPOCHS + 1),d_loss_col, label = "descriminator loss")
    plt.plot(np.arange(1,N_EPOCHS + 1),g_loss_col, label = "generator loss")
    plt.xlabel("# of epochs")
    plt.legend()
    plt.show()
    for i in range(3):
        plt.imshow(img_g_sample[i][0].reshape(28,28), interpolation='bilinear',
        cmap='Greys')
        plt.show()
evaluate_performance(N_EPOCHS,d_loss_col, g_loss_col,img_g_sample, "Vanilla")
def plot_predicted_hist(generate_num,trainedGenerator,Discriminator_trained):
    predicted_label = []
    for i in range(generate_num):
        z = to_var(torch.randn(1, 64))
        fake_images = trainedGenerator(z)
        predicted = Discriminator_trained(fake_images)
        cur_predicted_label = torch.argmax(predicted,dim = 1).item()
        predicted_label.append(cur_predicted_label)
    plt.figure(dpi = 300)
    plt.title("Histogram for predicted class using Gan generated %s fake image"%
    (generate_num))
    plt.hist(predicted_label, bins =10)
    plt.show()
generate_num = 3000
plot_predicted_hist(generate_num,trainedGenerator,Discriminator_trained)
```

0.3 Part 4 - Unrolled GAN

```
Discriminator = nn.Sequential(
    nn.Linear(28 * 28, 256),
    nn.LeakyReLU(0.2),
```

```
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 1),
    nn.Sigmoid()
)

# Generator
Generator  = nn.Sequential(
    nn.Linear(64, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 28 * 28),
    nn.Tanh()
)
def Unrolled_GAN(N_EPOCHS,data_loader, k):
    def train_unroll_discriminator(Discriminator_unroll, k,sample_data_loader,Generator):
        d_unroll_optimizer = torch.optim.Adam(Discriminator_unroll.parameters(), lr=0.0003)
        for batch_number, (images, _) in enumerate(sample_data_loader):
            if batch_number < k:
                batch_size = images.shape[0]
                images = to_var(images.view(batch_size, -1))#real image
                real_labels = to_var(torch.ones(batch_size, 1))
                fake_labels = to_var(torch.zeros(batch_size, 1))
                outputs = Discriminator(images)
                d_loss_real = criterion(outputs, real_labels)
                z = to_var(torch.randn(batch_size, 64))
                fake_images = Generator(z) #fake image
                outputs = Discriminator_unroll(fake_images)
                d_loss_fake = criterion(outputs, fake_labels)
                d_unrolled_loss = d_loss_real + d_loss_fake
                Discriminator_unroll.zero_grad()
                d_unrolled_loss.backward()
```

```
            d_unroll_optimizer.step()
        else:
            break
    return Discriminator_unroll
d_optimizer = torch.optim.Adam(Discriminator.parameters(), lr=0.0003)
g_optimizer = torch.optim.Adam(Generator.parameters(), lr=0.0003)
sample_data_loader = copy.deepcopy(data_loader)
criterion = nn.BCELoss()
d_loss_col = []
g_loss_col = []
idx_1 = 0
idx_2 = N_EPOCHS //2
idx_3 = N_EPOCHS - 1
img_g_sample = []
for epoch in range(N_EPOCHS):
    # loop through batches (no need for class labels right now)
    iter_d_loss = 0
    iter_g_loss = 0
    for batch_number, (images, _) in enumerate(data_loader):
        batch_size = images.shape[0]
        images = to_var(images.view(batch_size, -1))#real image
        real_labels = to_var(torch.ones(batch_size, 1))
        fake_labels = to_var(torch.zeros(batch_size, 1))
        # 1) TRAIN DISCRIMINATOR
        outputs = Discriminator(images)
        real_score = outputs
        d_loss_real = criterion(outputs, real_labels)
        # Draw random 64-dimensional noise vectors as inputs to the generator network
        z = to_var(torch.randn(batch_size, 64))
        fake_images = Generator(z) #fake image
        # Evaluate the discriminator on the fake images
        outputs = Discriminator(fake_images) # or D(add_instance_noise(fake_images))
        fake_score = outputs
        d_loss_fake = criterion(outputs, fake_labels)
        d_loss = d_loss_real + d_loss_fake
        Discriminator.zero_grad()
        d_loss.backward()
        d_optimizer.step()

        #make a copy of D to D_unroll
        Discriminator_unroll = copy.deepcopy(Discriminator)
```

```
            current_Discriminator_unroll = train_unroll_discriminator(Discriminator_unroll,
            k,sample_data_loader,Generator)
            # train the generator
            z = to_var(torch.randn(batch_size, 64))
            fake_images = Generator(z)
            outputs = current_Discriminator_unroll(fake_images)
            g_loss = criterion(outputs, real_labels)
            current_Discriminator_unroll.zero_grad()
            Generator.zero_grad()
            g_loss.backward()
            g_optimizer.step()
            iter_d_loss +=d_loss.data
            iter_g_loss +=g_loss.data
            if (batch_number + 1) % 300 == 0:
                print('Epoch [%d/%d], Step[%d/%d], d_loss: %.4f, '
                      'g_loss: %.4f, Mean D(x): %.2f, Mean D(G(z)): %.2f'
                      %(epoch,
                        N_EPOCHS,
                        batch_number + 1,
                        n_batches,
                        d_loss.data,
                        g_loss.data,
                        real_score.data.mean(),
                        fake_score.data.mean())
                     )
        if epoch in [idx_1,idx_2,idx_3]:
            fake_images = fake_images.view(fake_images.size(0), 1, 28, 28)
            img_g_sample.append(denorm(fake_images.data))

        d_loss_col.append(iter_d_loss/len(data_loader))
        g_loss_col.append(iter_g_loss/len(data_loader))
    return d_loss_col,g_loss_col,img_g_sample,Generator, Discriminator
import copy
BATCH_SIZE = 100
k = 3
train_data_loader = torch.utils.data.DataLoader(dataset=dataset_train,
                                                batch_size=BATCH_SIZE,
                                                shuffle=False,
                                                num_workers=0)
test_data_loader = torch.utils.data.DataLoader(dataset=dataset_test,
                                                batch_size=BATCH_SIZE,
```

```
                                            shuffle=False,
                                            num_workers=0)
n_batches = int(np.ceil(len(dataset_train)/ BATCH_SIZE)) # 6000
N_EPOCHS = 50
d_loss_col,g_loss_col,img_g_sample,Generator, Discriminator = Unrolled_GAN(N_EPOCHS,
train_data_loader, k)
generate_num = 3000
plot_predicted_hist(generate_num,Generator,Discriminator_trained)
```

0.3.1 Part 5 - Conditional GAN

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
class GeneratorModel(nn.Module):
    def __init__(self):
        super(GeneratorModel, self).__init__()
        input_dim = 64 + 10
        output_dim = 784
        self.label_embedding = nn.Embedding(10, 10)
        self.hidden_layer1 = nn.Sequential(
            nn.Linear(input_dim, 256),
            nn.LeakyReLU(0.2)
        )
        self.hidden_layer2 = nn.Sequential(
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2)
        )
        self.hidden_layer3 = nn.Sequential(
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2)
        )
        self.hidden_layer4 = nn.Sequential(
            nn.Linear(1024, output_dim),
            nn.Tanh()
        )
    def forward(self, x, labels):
        c = self.label_embedding(labels)
        x = torch.cat([x,c], 1)
        output = self.hidden_layer1(x)
        output = self.hidden_layer2(output)
        output = self.hidden_layer3(output)
        output = self.hidden_layer4(output)
```

```
        return output.to(device)

class DiscriminatorModel(nn.Module):
    def __init__(self):
        super(DiscriminatorModel, self).__init__()
        input_dim = 784 + 10
        output_dim = 1
        self.label_embedding = nn.Embedding(10, 10)
        self.hidden_layer1 = nn.Sequential(
            nn.Linear(input_dim, 1024),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3)
        )
        self.hidden_layer2 = nn.Sequential(
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3)
        )
        self.hidden_layer3 = nn.Sequential(
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3)
        )
        self.hidden_layer4 = nn.Sequential(
            nn.Linear(256, output_dim),
            nn.Sigmoid()
        )
    def forward(self, x, labels):
        c = self.label_embedding(labels)
        x = torch.cat([x, c], 1)
        output = self.hidden_layer1(x)
        output = self.hidden_layer2(output)
        output = self.hidden_layer3(output)
        output = self.hidden_layer4(output)
        return output.to(device)
Discriminator = DiscriminatorModel()
Generator = GeneratorModel()
Discriminator.to(device)
Generator.to(device)
Generator.to(device)
def training_conditional_vanilla(N_EPOCHS,data_loader):
```

```
d_optimizer = torch.optim.Adam(Discriminator.parameters(), lr=0.00005)
g_optimizer = torch.optim.Adam(Generator.parameters(), lr=0.00005)
criterion = nn.BCELoss()
d_loss_col = []
g_loss_col = []
idx_1 = 0
idx_2 = N_EPOCHS //2
idx_3 = N_EPOCHS - 1
img_g_sample = []
for epoch_idx in range(N_EPOCHS):
    iter_d_loss = 0
    iter_g_loss = 0
    for batch_idx, data_input in enumerate(data_loader):
        batch_size = data_input[0].shape[0]
        z = torch.randn(batch_size,64).to(device)
        fake_labels = torch.randint(0, 10, (batch_size,)).to(device)
        generated_data = Generator(z, fake_labels) # batch_size X 784
        # Discriminator
        true_data = data_input[0].view(batch_size, 784).to(device) # batch_size X 784
        digit_labels = data_input[1].to(device) # batch_size
        true_labels = torch.ones(batch_size).to(device)
        d_optimizer.zero_grad()
        discriminator_output_for_true_data = Discriminator(true_data, digit_labels)
        .view(batch_size)
        real_g_loss = criterion(discriminator_output_for_true_data, true_labels)
        discriminator_output_for_generated_data = Discriminator(generated_data.detach()
        fake_labels).view(batch_size)
        fake_g_loss = criterion(
            discriminator_output_for_generated_data, torch.zeros(batch_size).to(device)
        )
        d_loss = (
            real_g_loss + fake_g_loss
        )
        d_loss.backward()
        d_optimizer.step()
        iter_d_loss+=d_loss.data.item()
        g_optimizer.zero_grad()
        generated_data = Generator(z, fake_labels) # batch_size X 784
        discriminator_output_on_generated_data = Discriminator(generated_data,
        fake_labels).view(batch_size)
        generator_loss = criterion(discriminator_output_on_generated_data, true_labels)
```

```
            generator_loss.backward()
            g_optimizer.step()
            iter_g_loss += generator_loss.data.item()
            if (batch_idx + 1) % 300 == 0:
                print('Epoch [%d/%d], Step[%d/%d], d_loss: %.4f, '
                      'g_loss: %.4f'
                      %(epoch_idx,
                        N_EPOCHS,
                        batch_idx + 1,
                        n_batches,
                        d_loss.data,
                        generator_loss.data))
        if epoch_idx in [idx_1,idx_2,idx_3]:
            fake_images = generated_data.view(generated_data.size(0), 1, 28, 28)
            img_g_sample.append(denorm(fake_images.data))
        d_loss_col.append(iter_d_loss/len(data_loader))
        g_loss_col.append(iter_g_loss/len(data_loader))
    return d_loss_col,g_loss_col,img_g_sample, Generator
BATCH_SIZE = 100
data_loader = torch.utils.data.DataLoader(dataset=dataset,
                                          batch_size=BATCH_SIZE,
                                          shuffle=False,
                                          num_workers=0)
n_batches = int(np.ceil(len(dataset)/ BATCH_SIZE)) # 6000
N_EPOCHS = 50
d_loss_col,g_loss_col,img_g_sample,conditional_Generator = training_conditional_vanilla
(N_EPOCHS,data_loader)
def generate_class_image(conditional_Generator):
    fig, axs = plt.subplots(10,3, figsize=(15,10), dpi=500,gridspec_kw=
    {'hspace': 1, 'wspace': 0.5})
    fig.suptitle("Generated 3 samples from each of the 10 classes using trained generator f
    conditional GAN", fontsize =15)
    for i in range(10):
        for j in range(3):
            z = torch.randn(1,64)
            digit_labels = torch.tensor([i])
            generated_data = conditional_Generator(z, digit_labels)
            generated_image = generated_data.view(generated_data.size(0), 1, 28, 28)
            img_generated_sample = denorm(generated_image.data)
            axs[i, j].imshow(img_generated_sample[0].reshape(28,28),
            interpolation='bilinear', cmap='Greys')
```

```
            axs[i, j].set_title('Generated image for class %s sample %s'%(i,j),
            fontsize = 10)
    plt.axis("off")
    plt.show()
generate_class_image(conditional_Generator)
```