Homework 3
CS 5787 Deep Learning
Spring 2020

Ananya Shivaditya - aks298@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.

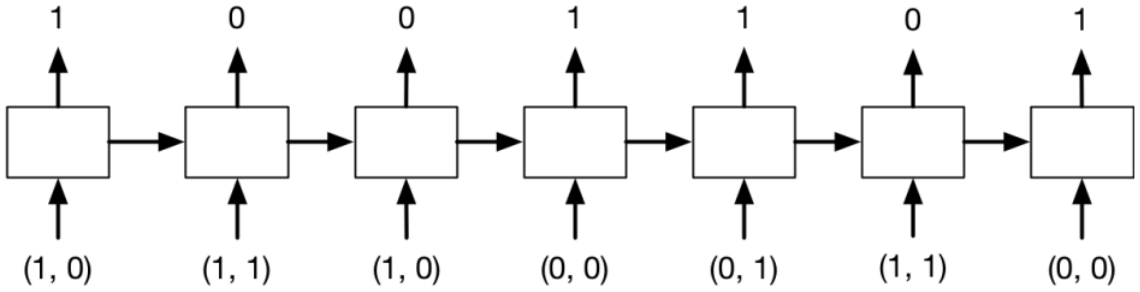
The input to your RNN will be two binary numbers, starting with the *least* significant bit. You will need to pad the largest number with an additional zero on the left side and you should make the other number the same length by padding it with zeros on the left side. For instance, the problem

$$100111 + 110010 = 1011001$$

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:



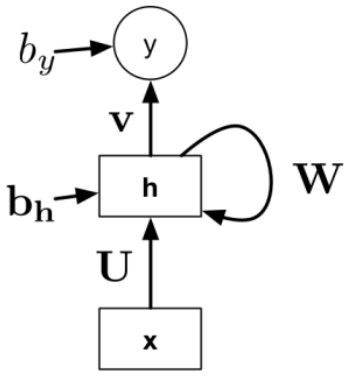
The RNN that implements binary addition has three hidden units, and all of the units use the following non-differentiable hard-threshold activation function

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

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:

$$\mathbf{U} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{pmatrix} \; ; \quad \mathbf{W} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \; ; \quad \mathbf{b_h} = \begin{pmatrix} 0 \\ -1 \\ -2 \end{pmatrix}$$

$$\mathbf{v} = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} \; ; \quad b_y = 0$$

$$\mathbf{h_0} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

These settings of **U** **W** and $\mathbf{b_h}$ allow for the hidden units to activate for each value from 1 to n+1, where n is the base for the addition(in this case 2). The n^{th} column of **W** will be all ones, with all other elements 0. This column represents the sum reaching n in the previous input, and will take the carry forward to the next input from the previous hidden state. **v** allows for representation of the current sum modulo n.

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:

Data preprocessing procedure was implemented in **Appendix 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:

Binary Prediction RNN with GRU was implemented in **Appendix 2**
The accuracy on the test set was 0.8145.

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:

Comparison with MLP for Binary Prediction was implemented in **Appendix 3**
The accuracy on the test set was 0.5032.
In comparison to the GRU model (acc = 0.81 on test set), a feedforward network fails to understand context and performs far less accurately.

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.

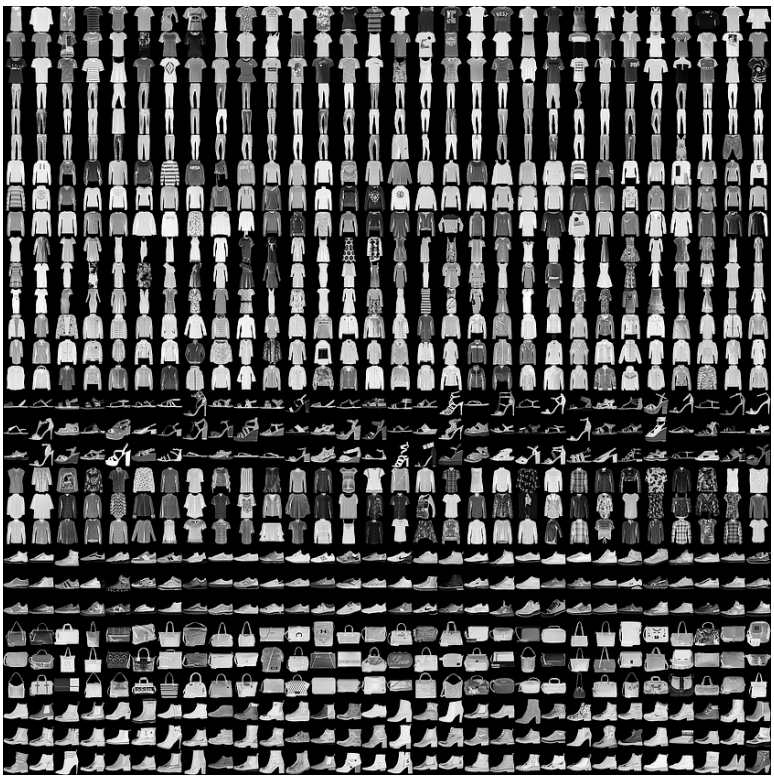


Figure 1: Fashion-Mnist dataset example images. It contains 10 classes of cloths, shoes, and bags.

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:

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

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:

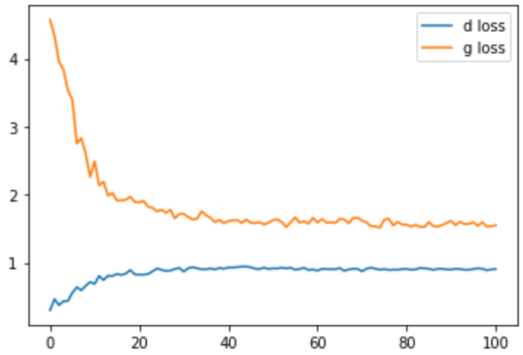
```
For epoch 1:max_epochs
    Train D:
        Get a batch of real images
        Get a batch of fake samples from G
        Optimize D to correctly classify the two batches

    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:
Vanilla GAN was trained with the given architecture. **Appendix 4: Vanilla GAN**
Training loss as a function of epochs:



Images from beginning of training :

Epoch 11:

Epoch 21:

Images from intermediate stage of training :

Epoch 91 Epoch 101

Images after convergence (Epoch 191):

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

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

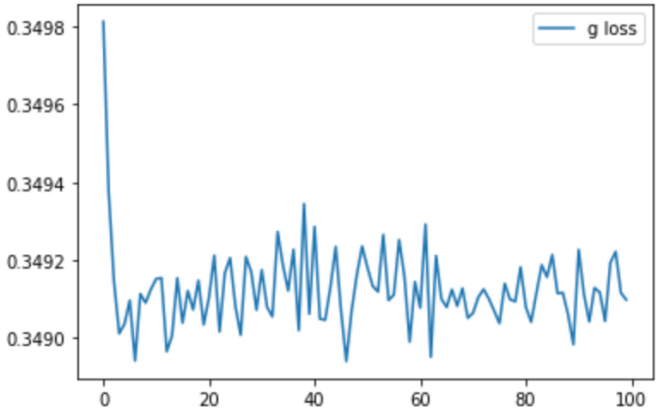
The idea is to provide a smoother loss surface than the original GAN loss.

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.

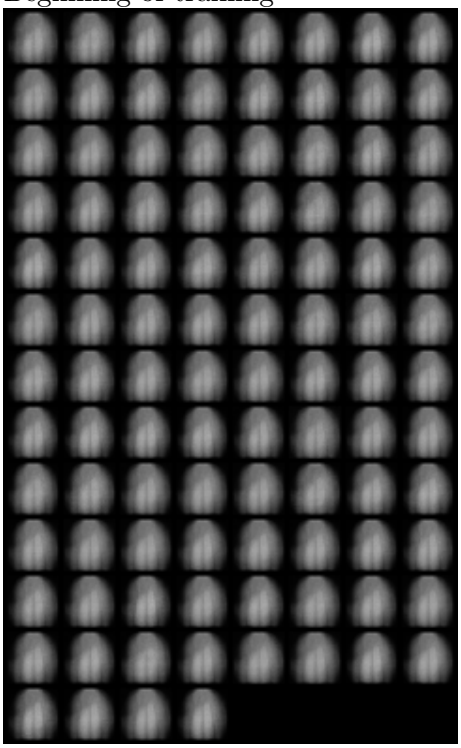
Solution:

Appendix 5: MSE Loss

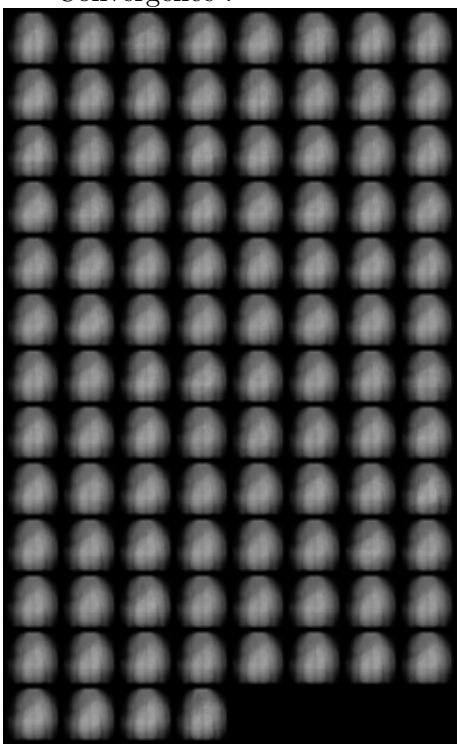
Training loss as a function of epochs:



Beginning of training



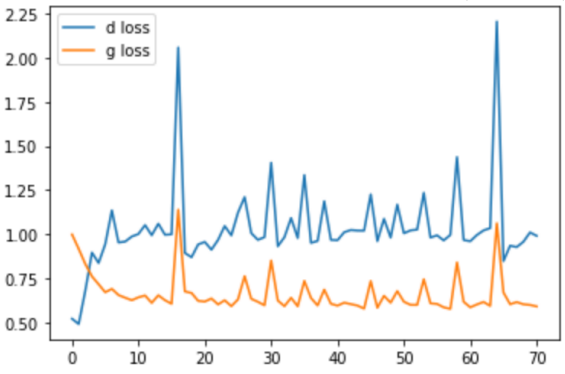
Convergence :



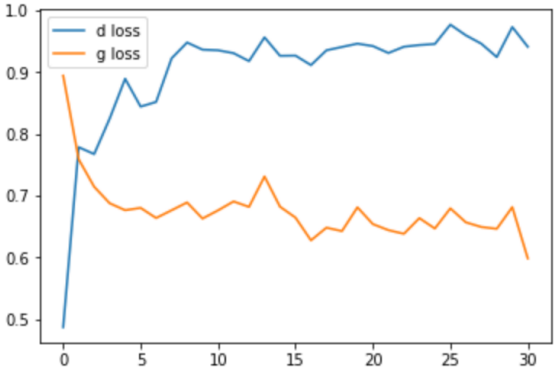
Almost no variation across generated images, unclear and ineffective.

Appendix 6: Wasserstein GAN (WGAN)

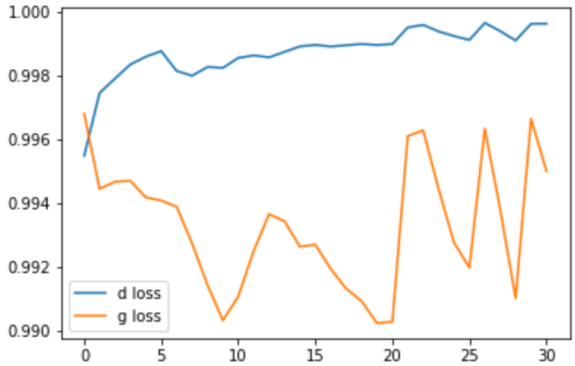
Training loss as a function of epochs(c = 0.1):



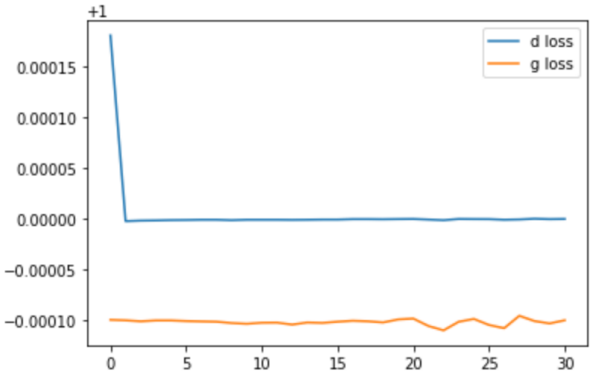
(c = 0.01):



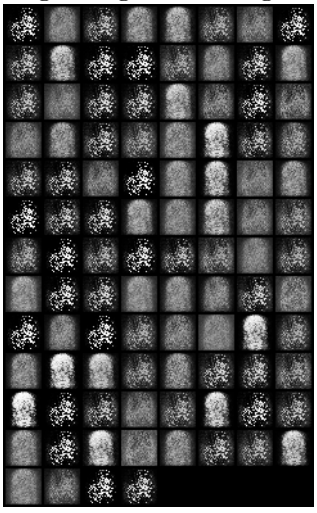
(c = 0.001):



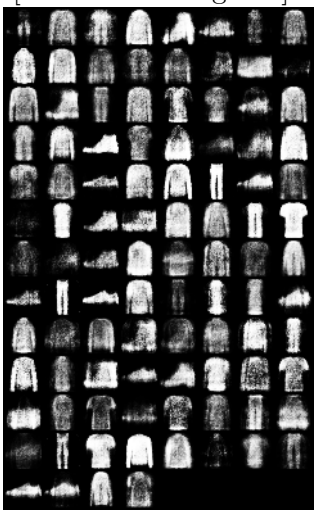
(c = 0.0001):



Beginning of training:



[c = 0.1 Convergence]:



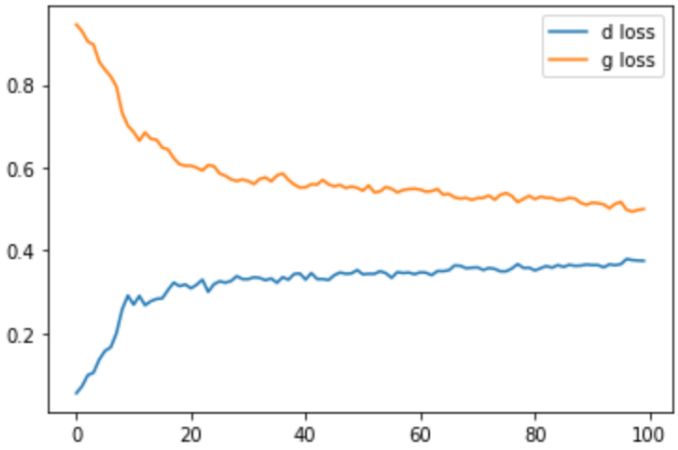
[c = 0.01 Convergence]:

Beginning of training: [c = 0.001 Convergence]: [c = 0.0001 Convergence]:

It appears that WGAN allows faster convergence and higher resolution of generated images. We also find that c = 0.001 produces best results in terms of resolution.

Appendix 7: Least Square GAN

Training loss as a function of epochs:



Beginning of training: [epoch 21]: [epoch 91]:

Least Squared loss produces good resolution with relatively fast convergence across epochs.

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)
```

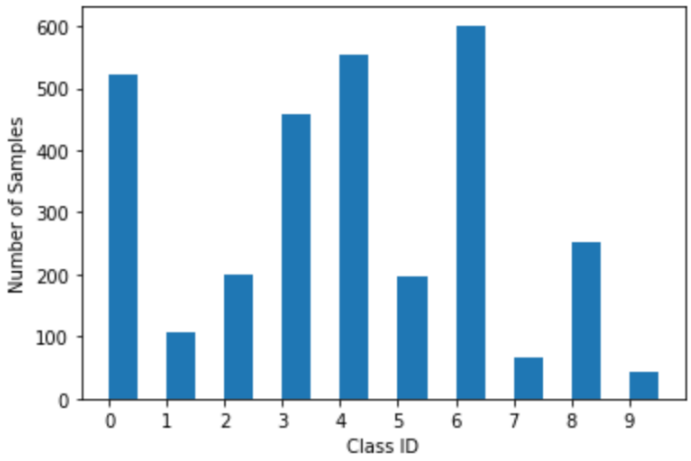
Note that G is trained with a copy of D at each epoch. The original D should not be updated during that part of training.

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.

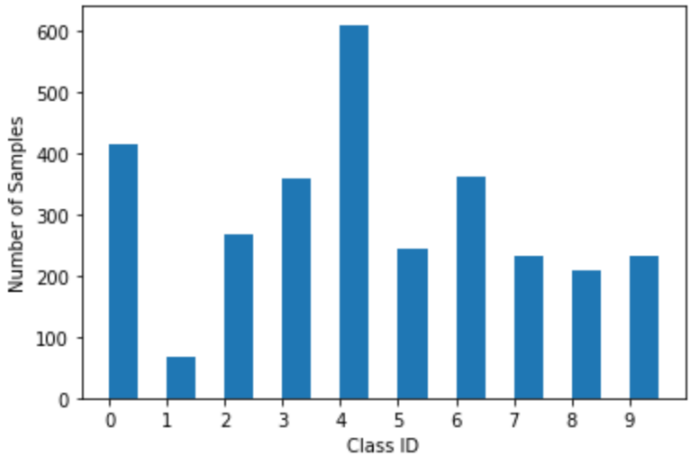
WGAN is claimed to be less affected by mode collapse too. In addition to your vanilla GAN model, use the WGAN model you trained in Part 2 and plot the histogram of class distribution, compare it to unrolled GAN and vanilla GAN.

Solution: Appendix 8: Mode Collapse

Test set accuracy on updated Vanilla GAN discriminator classifier was : 0.872
Histogram of class labels for 3000 generated images from Vanilla GAN:

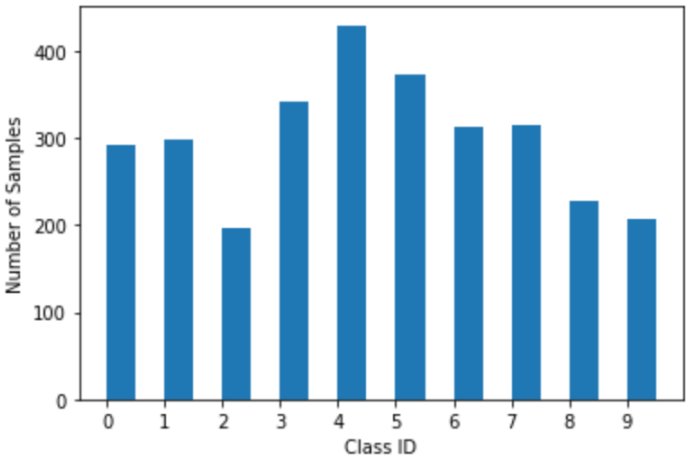


The unrolled GAN was retrained and the histogram was re plotted:



It seems the unrolled GAN does indeed help to reduce the mode collapse problem. Even though there are still minority and majority classes, there are more even counts of classes close to the 300-400 number of samples range.

The WGAN was used to plot the histogram and compared with previous plots:



It's evident that the WGAN is indeed more robust against mode collapse when compared with Vanilla GAN. It also appears better than an unrolled GAN in this aspect.

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.

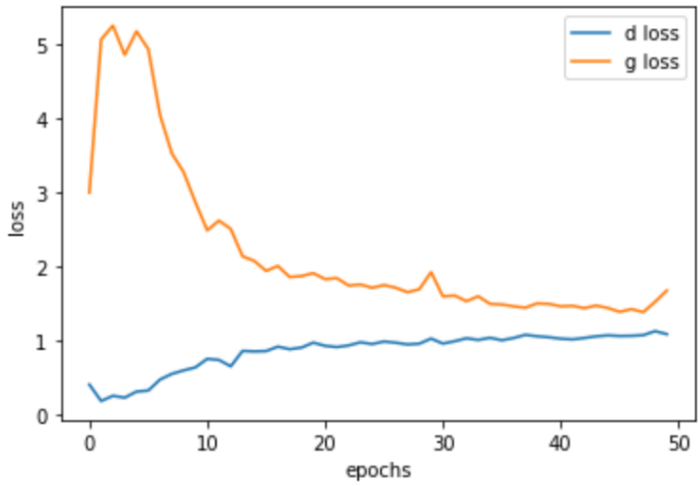
To add the conditional input vector, we need to modify both D and G. First, we need to define the input label vector. We are going to use one-hot encoding vectors for labels: for an image sample with label k of K classes, the vector is K dimensional and has 1 at k-th element and 0 otherwise.

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.

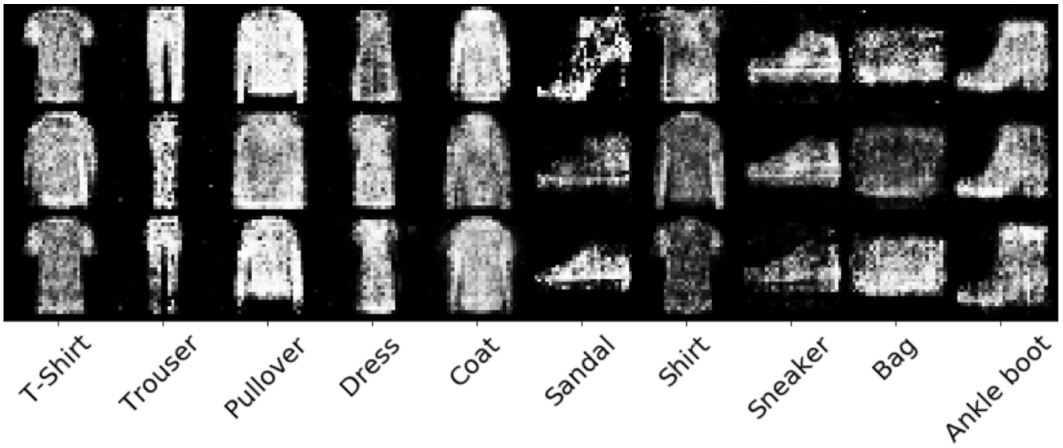
Train a Conditional GAN using the training script from Part 1. Plot training curves for D and G. Generate 3 samples from each of the 10 classes. Discuss differences in the generated images produced compared to the non-conditional models you built.

Solution:

Training curves:



Samples from each class on next page:



These images are lower in resolution and clarity than some of the non conditional GANs. Although this is subject to a specific class label, there is still reliably fast convergence and not many artifacts.

Code Appendix

Appendix 1: GRU for Sentiment Analysis - Preprocessing

```
import numpy as np
import pandas as pd
import nltk
from collections import Counter
import re
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
import torch
from torch.utils.data import DataLoader, TensorDataset
nltk.download('stopwords')
nltk.download('wordnet')

# load data
tneg = pd.read_csv('test_neg_merged.txt',sep='lollll',header=None,engine='python')
tpos = pd.read_csv('test_pos_merged.txt',sep='lollll',header=None,engine='python')
neg = pd.read_csv('train_neg_merged.txt',sep='lollll',header=None,engine='python')
pos = pd.read_csv('train_pos_merged.txt',sep='lollll',header=None,engine='python')

tneg['label'] = 0
neg['label'] = 0
tpos['label'] = 1
pos['label'] = 1


# Rem punct
stop_words = set(stopwords.words("english"))
lemmatizer = WordNetLemmatizer()

def clean_text(text):
    text = re.sub(r'[^\w\s]','',text)
    text = text.lower()
    text = [lemmatizer.lemmatize(token) for token in text.split(" ")]
    text = [lemmatizer.lemmatize(token, "v") for token in text]
    text = [word for word in text if not word in stop_words]
    text = " ".join(text)
    return text
```

```
tneg['Reviews'] = tneg[0].apply(lambda x: clean_text(x))
neg['Reviews'] = neg[0].apply(lambda x: clean_text(x))
pos['Reviews'] = pos[0].apply(lambda x: clean_text(x))
tpos['Reviews'] = tpos[0].apply(lambda x: clean_text(x))

tneg.drop(columns=[0],inplace=True)
pos.drop(columns=[0],inplace=True)
neg.drop(columns=[0],inplace=True)
tpos.drop(columns=[0],inplace=True)


# Build vocab

lol = pd.read_csv('all_merged.txt',sep='lollll',header=None,engine='python')
lol[0] = lol[0].apply(lambda x: clean_text(x))
words = ' '.join(lol[0])
words = words.split()
count_words = Counter(words)
total_words = len(words)
sorted_words = count_words.most_common(total_words)

vocab_to_int = {w:i+1 for i, (w,c) in enumerate(sorted_words)}
train = pd.concat([pos,neg],ignore_index=True)
test = pd.concat([tpos,tneg],ignore_index=True)
train = train.sample(frac=1).reset_index(drop=True)
test = test.sample(frac=1).reset_index(drop=True)

tr, te, j = [], [], 0
for i in [train, test]:
    for review in i['Reviews']:
        r = [vocab_to_int[w] for w in review.split()]
        if j == 0:
            tr.append(r)
        else:
            te.append(r)
    j += 1



# Pad, truc, encode
```

```
tr_encoded_labels = np.array(train['label'])
te_encoded_labels = np.array(test['label'])

def pad_features(reviews_int, seq_length):
    features = np.zeros((len(reviews_int), seq_length), dtype = int)
    for i, review in enumerate(reviews_int):
        review_len = len(review)
        if review_len <= seq_length:
            zeroes = list(np.zeros(seq_length-review_len))
            new = zeroes+review
        elif review_len > seq_length:
            new = review[0:seq_length]
        features[i,:] = np.array(new)
    return features

tr_sents = pad_features(tr,400)
te_sents = pad_features(te,400)



# To dataloader

train_data = TensorDataset(torch.from_numpy(tr_sents), torch.from_numpy(tr_encoded_labels))
test_data = TensorDataset(torch.from_numpy(te_sents), torch.from_numpy(te_encoded_labels))
batch_size = 50
train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
test_loader = DataLoader(test_data, shuffle=True, batch_size=batch_size)
```

Appendix 2: Sentiment Analysis RNN with GRU

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.autograd as autograd
import torch.optim as optim

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

        self.embedding = nn.Embedding(max_features, embedding_dim)
        self.gru = nn.GRU(embedding_dim, 256, 12, dropout=0.5, batch_first=True, \
        bidirectional=True)
        self.dropout = nn.Dropout(p=0.5)
        self.linear = nn.Linear(512, 1)

    def forward(self, x, h):
        x = self.embedding(x)
        x, h = self.gru(x, h)
        x = self.dropout(x[:,-1,:].squeeze())
        x = F.sigmoid(self.linear(x))
        return x, h

    def init_hidden(self):
        return autograd.Variable(torch.randn(24, batch_size, 256)).cuda()

ITERS = 20
max_features =len(vocab_to_int)+1
batch_size = 50

# training data are index for words for the embedding layer,
x_train = tr_sents
y_train = tr_encoded_labels

x_test = te_sents
y_test = te_encoded_labels

loss_fn = nn.BCELoss()
```

```
output_size = 1
embedding_dim = 50
hidden_dim = 256
n_layers = 2

model = BIGRU()
optimizer = optim.Adam(model.parameters(), lr=1e-3)

print ('Putting model on GPU ... ')
model.cuda()
h = model.init_hidden()
print ('Training ...')
model.train()
num_batch_epoch = len(x_train) // batch_size
for e in range(ITERS):
    print ('\n' + 'Epoch {}/{}'.format(e, ITERS))
    print ('-' * 10)
    idx = np.random.permutation(len(x_train))
    iter_loss = 0.
    iter_correct = 0.
    for b in range(num_batch_epoch):
        x_b = torch.from_numpy(x_train[idx[b*batch_size:(b+1)*batch_size]]).long()
        y_b = torch.from_numpy(y_train[idx[b*batch_size:(b+1)*batch_size]]).float()
        x_batch = autograd.Variable(x_b.cuda())
        y_batch = autograd.Variable(y_b.cuda())

        h.detach_()
        y_pred, h = model(x_batch, h)
        optimizer.zero_grad()
        loss = loss_fn(torch.round(y_pred), y_batch)
        loss.backward()
        optimizer.step()

        trn_preds = torch.round(y_pred.data)
        acc = torch.sum(trn_preds == y_batch.data)
        iter_correct += acc
        iter_loss += loss.item()

    print ('Training Loss: {:.3} | Training Acc: {:.3}'.format(iter_loss / \
    (num_batch_epoch*batch_size), float(iter_correct) / (num_batch_epoch*batch_size)))
```

```
model.eval()
running_corrects = 0
for vb in range(len(x_test) // batch_size):
    x_v_b = torch.from_numpy(x_test[vb*batch_size:(vb+1)*batch_size]).long()
    y_v_b = torch.from_numpy(y_test[vb*batch_size:(vb+1)*batch_size]).float()

    x_v_batch = autograd.Variable(x_v_b.cuda(), volatile=True)
    y_v_batch = autograd.Variable(y_v_b.cuda())
    outputs, h = model(x_v_batch, h)
    preds = torch.round(outputs.data)
    running_corrects += torch.sum(preds == y_v_batch.data)

    if vb % 100 == 0:
        print ('Valid batch:', vb, 'done!')

print ('Percent correct:', float(running_corrects) / len(x_test))
```

Appendix 3: Comparison with MLP for Binary Prediction

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.autograd as autograd
import torch.optim as optim

class MLP(torch.nn.Module):
    def __init__(self, input_size, hidden_size):
        super(MLP, self).__init__()
        self.input_size = input_size
        self.hidden_size  = hidden_size
        self.fc1 = torch.nn.Linear(self.input_size, self.hidden_size)
        self.relu = torch.nn.ReLU()
        self.fc2 = torch.nn.Linear(self.hidden_size, 1)
        self.sigmoid = torch.nn.Sigmoid()

    def forward(self, x):
        hidden = self.fc1(x)
        relu = self.relu(hidden)
        output = self.fc2(relu)
        # output = self.sigmoid(output)
        return output

def accu(y_pred, y_test):
    y_pred_tag = torch.round(torch.sigmoid(y_pred))

    correct_results_sum = (y_pred_tag == y_test).sum().float()
    acc = correct_results_sum/y_test.shape[0]
    acc = torch.round(acc * 100)
    return acc

model = MLP(400, 1000)
criterion = torch.nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(model.parameters(),)

epoch = 90
model.cuda()
model.train()
```

```
for epoch_n in range(epoch):
    total_loss = 0
    epoch_acc = 0
    total_batch = 0
    for train_x_b, train_y_b in train_loader:
        # print(train_x_b.shape)
        train_x_b, train_y_b = train_x_b.float().cuda(), train_y_b.float().cuda()

        optimizer.zero_grad()    # Forward pass

        y_pred = model(train_x_b)    # Compute Loss
        # print(y_pred, train_y_b)
        loss = criterion(y_pred.flatten(), train_y_b.flatten())
        # print(y_pred)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        acc = accu(y_pred, train_y_b)
        epoch_acc += acc.item()
        total_batch+=1
    print('Epoch {}: train loss: {} acc: {}'.format(epoch_n, total_loss/total_batch, epoch_

total_loss = 0
epoch_acc = 0
total_batch = 0
for train_x_b, train_y_b in test_loader:
      train_x_b, train_y_b = train_x_b.float().cuda(), train_y_b.float().cuda()

      optimizer.zero_grad()
      y_pred = model(train_x_b)
      loss = criterion(y_pred.flatten(), train_y_b)
      loss.backward()
      optimizer.step()
      total_loss += loss.item()
      acc = accu(y_pred, train_y_b)
      epoch_acc += acc.item()
      total_batch+=1
print('train loss: {} acc: {}'.format(total_loss/total_batch, epoch_acc/total_batch))
```

Appendix 4: Vanilla GAN

```
import matplotlib.pyplot as plt
import torch
from torchvision import datasets, transforms

datasets.FashionMNIST?

transform = transforms.Compose([
                transforms.ToTensor(),
                transforms.Lambda(lambda x : 2 * x - 1) # normalize to be in [-1, 1]
            ])

dataset = datasets.FashionMNIST(
    root='/content/data',
    train=True,
    transform=transform,
    download=True
)

np.unique(dataset.train_labels)

plt.hist(dataset.train_labels,
         bins=np.linspace(0, 9.5, 20))

_ = plt.xticks(range(10))
plt.xlabel('Class ID')
plt.ylabel('Number of Samples')
plt.show()

dataset.train_labels

n_images = 5

f, axarr = plt.subplots(len(np.unique(dataset.train_labels)), n_images, \
    figsize=(10, 10))
f.subplots_adjust(wspace=-0.7)

for n_class in np.unique(dataset.train_labels):
    axarr[n_class, 0].text(-20, 20, str(n_class), size=20, weight='bold')
```

```
    for n_img, image in enumerate(dataset.train_data.numpy()[ \
    dataset.train_labels.numpy() == n_class][:n_images]):
        axarr[n_class, n_img].imshow(image, interpolation='bilinear', cmap='Greys')
        axarr[n_class, n_img].axis('off')

BATCH_SIZE = 100
data_loader = torch.utils.data.DataLoader(dataset=dataset,
                                          batch_size=BATCH_SIZE,
                                          shuffle=True,
                                          num_workers=3)

n_batches = int(np.ceil(len(dataset)/ BATCH_SIZE))
n_batches

from torch import nn

# Discriminator
D = nn.Sequential(
    nn.Linear(28 * 28, 256), #1
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256), #2
    nn.LeakyReLU(0.2),
    nn.Linear(256, 1), #3
    nn.Sigmoid()
)

# Generator
G = nn.Sequential(
    nn.Linear(64, 256), #1
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256), #2
    nn.LeakyReLU(0.2),
    nn.Linear(256, 28 * 28), #3
    nn.Tanh()
)

criterion = nn.BCELoss()
d_optimizer = torch.optim.Adam(D.parameters(), lr=0.0003)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.0003)

is_cuda = torch.cuda.is_available()
```

```
if is_cuda:
    device = torch.device("cuda")
    D.cuda()
    G.cuda()
else:
    device = torch.device("cpu")

from torch.autograd import Variable

def to_var(x):
    return Variable(x).to(device)

import torch.nn.functional as F
from torchvision.utils import save_image

def denorm(x):
    # convert back from [-1, 1] to [0, 1]
    out = (x + 1) / 2
    return out #out.clamp(0, 1)

N_EPOCHS = 200

"""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)
"""

def showimg(img):
    img = img.view(img.size(0), 1, 28, 28)
    image = img[0].detach().permute(2, 1, 0)
    plt.imshow(image.reshape(28,28))
    plt.show()
```

```
def plotLoss(d_loss, g_loss):
    plt.plot(d_loss, label="d loss")
    plt.plot(g_loss, label="g loss")
    plt.legend(loc='best')
    plt.show()

def vanillaGAN(criterion):
    total_D_loss = []
    total_G_loss = []
    try:
        for epoch in range(N_EPOCHS):
            D_loss = []
            G_loss = []
            for batch_number, (images, _) in enumerate(data_loader):
                # Train D
                D.zero_grad()
                batch_size = images.shape[0]
                real_images = to_var(images.view(batch_size, -1))
                z = to_var(torch.randn(batch_size, 64))
                fake_images = G(z)
                # Optimize D to correctly classify the two batches
                real_labels = to_var(torch.ones(batch_size, 1))
                fake_labels = to_var(torch.zeros(batch_size, 1))
                real_image_scores = D(real_images)
                d_loss_real = criterion(real_image_scores, real_labels)
                fake_image_scores = D(fake_images)
                d_loss_fake = criterion(fake_image_scores, fake_labels)
                # Backprop + Optimize the discriminator
                d_loss = d_loss_real + d_loss_fake
                d_loss.backward()
                d_optimizer.step()

                G.zero_grad()
                D.zero_grad()
                # Train G
                z = to_var(torch.randn(batch_size, 64))
                fake_images_g = G(z)
                fake_image_scores_g = D(fake_images_g)
                g_loss = criterion(fake_image_scores_g, real_labels)
```

```
                g_loss.backward()
                g_optimizer.step()
                D_loss.append(d_loss.detach().cpu().numpy())
                G_loss.append(g_loss.detach().cpu().numpy())
            #plotLoss(D_loss, G_loss)
            print("epoch:",epoch, " d loss:", np.mean(D_loss), " g loss:", \
            np.mean(G_loss))
            total_D_loss.append(np.mean(D_loss))
            total_G_loss.append(np.mean(G_loss))
            fake_images_g = fake_images_g.view(fake_images_g.size(0), 1, 28, 28)

            if epoch%10 == 0:
                print("Total losses:"," D loss:", total_D_loss, " G loss:", total_G_loss)
                plotLoss(total_D_loss, total_G_loss)
                save_image(denorm(fake_images_g.data), \
                'My Drive/data/DL_hw3/fake_images-%0.3d.png' %(epoch + 1))
        print("Total losses:"," D loss:", total_D_loss, " G loss:", total_G_loss)
        plotLoss(total_D_loss, total_G_loss)

    except KeyboardInterrupt:
        print('Training ended early.')

vanillaGAN(nn.BCELoss())

# Save the trained parameters
# torch.save(G.state_dict(), \
'My Drive/data/DL_hw3/models/generator-%0.3d.pkl' %(50 + 1))
# torch.save(D.state_dict(), \
'My Drive/data/DL_hw3/models/discriminator-%0.3d.pkl' %(50 + 1))
```

Appendix 5: MSE Loss GAN

```
G2 = nn.Sequential(
    nn.Linear(64, 256), #1
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256), #2
    nn.LeakyReLU(0.2),
    nn.Linear(256, 28 * 28), #3
    nn.Tanh()
)

G2 = G2.to(device)
criterion = nn.MSELoss()
g_optimizer = torch.optim.Adam(G2.parameters(), lr=3e-3)

def plotMSELoss(g_loss):
    plt.plot(g_loss, label="g loss")
    plt.legend(loc='best')
    plt.show()
    plt.savefig("My Drive/data/DL_hw3/mse_loss_png.png")

def trainMSE(criterion):
    total_G_loss = []
    try:
        for epoch in range(N_EPOCHS):
            D_loss = []
            G_loss = []
            for batch_number, (images, _) in enumerate(data_loader):
                # Train D
                G2.zero_grad()
                batch_size = images.shape[0]
                real_images = to_var(images.view(batch_size, -1))
                z = to_var(torch.randn(batch_size, 64))
                fake_images = G2(z)
                g_loss = criterion(fake_images, real_images)
                g_loss.backward()
                g_optimizer.step()
                G_loss.append(g_loss.detach().cpu().numpy())

            z = to_var(torch.randn(batch_size, 64))
```

```
            fake_images_g = G2(z)
            print("epoch:",epoch, " d loss:",\
            np.mean(D_loss), " g loss:", np.mean(G_loss))
            total_G_loss.append(np.mean(G_loss))
            fake_images_g = fake_images_g.view(fake_images_g.size(0), 1, 28, 28)
            if epoch%10 == 0:
                print("Total losses:", " G loss:", total_G_loss)
                plotMSELoss(total_G_loss)
        print("Total losses:", "G loss:", total_G_loss)
        plotMSELoss(total_G_loss)
    except KeyboardInterrupt:
        print("end early")
        plotMSELoss(total_G_loss)

N_EPOCHS=100
trainMSE(criterion)
```

Appendix 6: Wasserstein GAN (WGAN)

```
def initWGAN():
    # Discriminator
    W_D = nn.Sequential(
    nn.Linear(28 * 28, 512),
    nn.LeakyReLU(0.2),
    nn.Linear(512, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 1),
    )
    # Generator
    W_G = nn.Sequential(
    nn.Linear(64, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 28 * 28),
    nn.Tanh()
    )
    d_optimizer = torch.optim.Adam(W_D.parameters(), lr=0.008)
    g_optimizer = torch.optim.Adam(W_G.parameters(), lr=0.0008)
    return W_D.to(device), W_G.to(device), d_optimizer, g_optimizer


def plotWLoss(d_loss, g_loss, clip_value):
    plt.plot(d_loss, label="d loss")
    plt.plot(g_loss, label="g loss")
    plt.legend(loc='best')
    plt.show()


def WGAN(criterion, clip_value, W_D, W_G, d_optimizer, g_optimizer):
    total_D_loss = []
    total_G_loss = []
    try:
        for epoch in range(N_EPOCHS):
            D_loss = []
            G_loss = []
            for batch_number, (images, _) in enumerate(data_loader):
```

```
        # Train D
        batch_size = images.shape[0]
        real_images = to_var(images.view(batch_size, -1))
        z = to_var(torch.randn(batch_size, 64))
        W_D.zero_grad()

        fake_images = W_G(z)
        # Optimize D to correctly classify the two batches
        real_labels = to_var(torch.ones(batch_size, 1))
        fake_labels = to_var(torch.zeros(batch_size, 1))
        real_image_scores = W_D(real_images)
        d_loss_real = criterion(real_image_scores, real_labels)
        fake_image_scores = W_D(fake_images)
        d_loss_fake = criterion(fake_image_scores, fake_labels)

        d_loss = d_loss_real + d_loss_fake
        d_loss.backward()
        d_optimizer.step()
        for p in W_D.parameters():
            p.data.clamp_(-clip_value, clip_value)

        # Train G
        W_G.zero_grad()
        W_D.zero_grad()
        z = to_var(torch.randn(batch_size, 64))
        fake_images_g = W_G(z)
        fake_image_scores_g = W_D(fake_images_g)
        g_loss = criterion(fake_image_scores_g, real_labels)
        g_loss.backward()
        g_optimizer.step()
        D_loss.append(d_loss.detach().cpu().numpy())
        G_loss.append(g_loss.detach().cpu().numpy())

    print("epoch:",epoch, " d loss:", \
        np.mean(D_loss), " g loss:", np.mean(G_loss))
    total_D_loss.append(np.mean(D_loss))
    total_G_loss.append(np.mean(G_loss))
    fake_images_g = fake_images_g.view(fake_images_g.size(0), 1, 28, 28)
    if epoch%10 == 0:
        print(clip_value, " Total losses:"," D loss:",\
        total_D_loss, " G loss:", total_G_loss)
```

```
                plotWLoss(total_D_loss, total_G_loss, str(clip_value))
                save_image(denorm(fake_images_g.data),\
                    'My Drive/data/DL_hw3/WGAN/clip' \
                    +str(clip_value)+'WGAN_fake_images-%0.3d.png' %(epoch + 1))

        print(clip_value," Total losses:"," D loss:",\
        total_D_loss, " G loss:", total_G_loss)
        # plotLoss(total_D_loss, total_G_loss)

    except KeyboardInterrupt:
        print('Training ended early.')


N_EPOCHS = 40
for clip_value in [0.1,0.01,0.001,0.0001]:
    W_D, W_G, d_optimizer, g_optimizer = initWGAN()
    WGAN(nn.L1Loss(),clip_value,W_D, W_G, d_optimizer, g_optimizer)
```

Appendix 7: Least Square GAN

```
def vanillaGAN(criterion):
    total_D_loss = []
    total_G_loss = []
    try:
        for epoch in range(N_EPOCHS):
            D_loss = []
            G_loss = []
            for batch_number, (images, _) in enumerate(data_loader):
                D.zero_grad()
                batch_size = images.shape[0]
                real_images = to_var(images.view(batch_size, -1))
                z = to_var(torch.randn(batch_size, 64))
                fake_images = G(z)
                # Optimize D to correctly classify the two batches
                real_labels = to_var(torch.ones(batch_size, 1))
                fake_labels = to_var(torch.zeros(batch_size, 1))
                real_image_scores = D(real_images)
                d_loss_real = criterion(real_image_scores, real_labels)
                fake_image_scores = D(fake_images)
                d_loss_fake = criterion(fake_image_scores, fake_labels)
                d_loss = d_loss_real + d_loss_fake
                d_loss.backward()
                d_optimizer.step()

                G.zero_grad()
                D.zero_grad()
                z = to_var(torch.randn(batch_size, 64))
                fake_images_g = G(z)
                fake_image_scores_g = D(fake_images_g)
                g_loss = criterion(fake_image_scores_g, real_labels)

                g_loss.backward()
                g_optimizer.step()
                D_loss.append(d_loss.detach().cpu().numpy())
                G_loss.append(g_loss.detach().cpu().numpy())
            print("epoch:",epoch, " d loss:", \
            np.mean(D_loss), " g loss:", np.mean(G_loss))
            total_D_loss.append(np.mean(D_loss))
```

```
            total_G_loss.append(np.mean(G_loss))
            fake_images_g = fake_images_g.view(fake_images_g.size(0), 1, 28, 28)

            if epoch%10 == 0:
                print("Total losses:"," D loss:", total_D_loss, " G loss:", total_G_loss)
                plotLoss(total_D_loss, total_G_loss)
                save_image(denorm(fake_images_g.data),\
                'My Drive/data/DL_hw3/LS/fake_images-%0.3d.png' %(epoch + 1))
        print("Total losses:"," D loss:", total_D_loss, " G loss:", total_G_loss)
        plotLoss(total_D_loss, total_G_loss)

    except KeyboardInterrupt:
        print('Training ended early.')

# Discriminator
D = nn.Sequential(
    nn.Linear(28 * 28, 256), #1
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256), #2
    nn.LeakyReLU(0.2),
    nn.Linear(256, 1), #3
    nn.Sigmoid()
)
D = D.to(device)

# Generator
G = nn.Sequential(
    nn.Linear(64, 256), #1
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256), #2
    nn.LeakyReLU(0.2),
    nn.Linear(256, 28 * 28), #3
    nn.Tanh()
)
G = G.to(device)

d_optimizer = torch.optim.Adam(D.parameters(), lr=0.0003)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.0003)

N_EPOCHS = 100
vanillaGAN(nn.MSELoss())
```

Appendix 8: Mode Collapse

```
import numpy as np
import matplotlib.pyplot as plt
# %matplotlib inline
from torchvision.utils import save_image
from torch import nn
import torch

is_cuda = torch.cuda.is_available()

if is_cuda:
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

import torch
from torchvision import datasets, transforms

transform = transforms.Compose([
                transforms.ToTensor(),
                transforms.Lambda(lambda x : 2 * x - 1) # normalize to be in [-1, 1]
            ])

trainset = datasets.FashionMNIST('F_MNIST_data/', download=True, train=True, \
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=100, shuffle=True)

testset = datasets.FashionMNIST('F_MNIST_data/', download=True, train=False, \
transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=True)

from torch.autograd import Variable

def to_var(x):
    return Variable(x).to(device)

def denorm(x):
    out = (x + 1) / 2
    return out
```

```
labels_map = {0 : 'T-Shirt', 1 : 'Trouser', 2 : 'Pullover', 3 : 'Dress', 4 : 'Coat', \
5 : 'Sandal', 6 : 'Shirt', 7 : 'Sneaker', 8 : 'Bag', 9 : 'Ankle Boot'}

trainset.data.shape

testset.data.shape

trainset.train_data[0].shape

dataset = datasets.FashionMNIST(
    root='./data',
    train=True,
    transform=transform,
    download=True
)

BATCH_SIZE = 100
data_loader = torch.utils.data.DataLoader(dataset=dataset,
                                          batch_size=BATCH_SIZE,
                                          shuffle=True,
                                          num_workers=3)

n_batches = int(np.ceil(len(dataset)/ BATCH_SIZE))
n_batches

# Discriminator
D = nn.Sequential(
    nn.Linear(28 * 28, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 1),
    nn.Sigmoid()
)
D = D.to(device)

# Generator
G = nn.Sequential(
    nn.Linear(64, 256),
    nn.LeakyReLU(0.2),
```

```
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 28 * 28),
    nn.Tanh()
)
G = G.to(device)

criterion = nn.BCELoss()
d_optimizer = torch.optim.Adam(D.parameters(), lr=0.0003)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.0003)

def plotLoss(d_loss, g_loss):
    plt.plot(d_loss, label="d loss")
    plt.plot(g_loss, label="g loss")
    plt.legend(loc='best')
    plt.savefig("loss_png.png")
    plt.show()

def vanillaGAN(criterion):
    total_D_loss = []
    total_G_loss = []
    try:
        for epoch in range(N_EPOCHS):
            D_loss = []
            G_loss = []
            for batch_number, (images, _) in enumerate(data_loader):
                # Train D
                D.zero_grad()
                batch_size = images.shape[0]
                real_images = to_var(images.view(batch_size, -1))
                z = to_var(torch.randn(batch_size, 64))
                fake_images = G(z)
                # Optimize D to correctly classify the two batches
                real_labels = to_var(torch.ones(batch_size, 1))
                fake_labels = to_var(torch.zeros(batch_size, 1))
                real_image_scores = D(real_images)
                d_loss_real = criterion(real_image_scores, real_labels)
                fake_image_scores = D(fake_images)
                d_loss_fake = criterion(fake_image_scores, fake_labels)

                d_loss = d_loss_real + d_loss_fake
```

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

                G.zero_grad()
                D.zero_grad()
                # Train G
                z = to_var(torch.randn(batch_size, 64))
                fake_images_g = G(z)
                fake_image_scores_g = D(fake_images_g)
                g_loss = criterion(fake_image_scores_g, real_labels)


                g_loss.backward()
                g_optimizer.step()
                D_loss.append(d_loss.detach().cpu().numpy())
                G_loss.append(g_loss.detach().cpu().numpy())
            #plotLoss(D_loss, G_loss)
            print("epoch:",epoch, " d loss:", np.mean(D_loss), \
            " g loss:", np.mean(G_loss))
            total_D_loss.append(np.mean(D_loss))
            total_G_loss.append(np.mean(G_loss))
            fake_images_g = fake_images_g.view(fake_images_g.size(0), 1, 28, 28)
            save_image(denorm(fake_images_g.data), 'f-%0.3d.png' %(epoch + 1))
            if epoch%10 == 0:
                print("Total losses:"," D loss:", total_D_loss, " G loss:", total_G_loss)
                plotLoss(total_D_loss, total_G_loss)
        print("Total losses:"," D loss:", total_D_loss, " G loss:", total_G_loss)
        plotLoss(total_D_loss, total_G_loss)

    except KeyboardInterrupt:
        print('Training ended early.')

N_EPOCHS =50
vanillaGAN(nn.BCELoss())

NB_SAMPLES = 3000
z = to_var(torch.randn(NB_SAMPLES, 64))
fake_images = G(z)
fake_images_g = fake_images.view(fake_images.size(0), 1, 28, 28)

# save_image(denorm(fake_images_g.data), './vangan.png' )
```

```
def onehotencoding(labels):
    rows = labels.shape[0]
    onehotlabels = to_var(torch.zeros((rows, 10), dtype=torch.long))
    for i in range(rows):
        val = labels[i].item()
        onehotlabels[i][val] = torch.ones(1, dtype=torch.long)
    return to_var(onehotlabels)

# Discriminator
D_G = nn.Sequential(
    nn.Linear(28 * 28, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 10),
)
D_G = D_G.to(device)

def trainClassifier(criterion, optimizer, epochs):
    running_loss = 0
    for e in range(epochs):
        running_loss = 0
        predictions_list = []
        correct = 0
        total = 0

        for batch_number, (images, labels) in enumerate(trainloader):
            batch_size = images.shape[0]
            D_G.zero_grad()

            real_images = to_var(images.view(batch_size, -1))
            # real_labels = onehotencoding(labels)
            labels = labels.to(device)
            real_image_scores = D_G(real_images)
            # print(labels)
            d_loss = criterion(real_image_scores, labels)
            d_loss.backward()
            optimizer.step()
            running_loss+=d_loss.item()
```

```
            _, predictions = torch.max(real_image_scores, 1)
            predictions_list.append(predictions)
            correct += (predictions == labels).sum().item()
            total += len(labels)
            # print(d_loss.item())

        accuracy = correct * 100 / total
        print("train accuracy:",accuracy)
        print("epoch "+ str(e)+ ":",str(running_loss/600.0), running_loss)
        get_test_set_accuracy()

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(D_G.parameters(), lr=1e-2)

trainClassifier(criterion, optimizer, 1)

def get_test_set_accuracy():
    correct= 0
    accuracy=0
    total=0
    for images, labels in testloader:
      batch_size = images.shape[0]
      labels = labels.to(device)
      real_images = to_var(images.view(batch_size, -1))
      real_image_scores = D_G(real_images)
      predictions = torch.max(real_image_scores, 1)[1]
      correct += (predictions == labels).sum().item()
      total += len(labels)
    accuracy = correct * 100 / total
    print("test accuracy:",accuracy)

get_test_set_accuracy()

from PIL import Image
import glob
t = transforms.ToTensor()
image_list = []
image_list_t = []
for filename in glob.glob('fakes2/fakes2/*.png'):
    im=Image.open(filename)
    im = im.convert(mode='L')
```

```
    image_list.append(im)
    img_t = t(im)
    image_list_t.append(img_t)

prediction_list =[]
real_image_scores = D_G(fake_images)
predictions = torch.max(real_image_scores, 1)[1]

plt.hist(predictions.cpu().numpy(),
         bins=np.linspace(0, 9.5, 20))

_ = plt.xticks(range(10))
plt.xlabel('Class ID')
plt.ylabel('Number of Samples')
plt.savefig('distribution_2.png')
plt.show()

def plotLoss(d_loss, g_loss, xlabel = "epochs", ylabel = "loss"):
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.plot(d_loss, label="d loss")
    plt.plot(g_loss, label="g loss")
    plt.legend(loc='best')
    plt.savefig("unroll_loss.png")
    plt.show()

# Discriminator
D = nn.Sequential(
    nn.Linear(28 * 28, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 1),
    nn.Sigmoid()
)
D = D.to(device)

import copy

D_unroll = nn.Sequential(
    nn.Linear(28 * 28, 256),
```

```
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 1),
    nn.Sigmoid()
)
D_unroll.load_state_dict(D.state_dict()) # get weights
D_unroll = D_unroll.to(device)

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

d_optimizer = torch.optim.Adam(D.parameters(), lr=0.0003)
d_unroll_optimizer = torch.optim.Adam(D_unroll.parameters(), lr=0.0003)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.0003)

def train_unrolled(epochs, criterion, k):
    total_D_loss = []
    total_G_loss = []
    try:
        D_loss = []
        G_loss = []
        for epoch in range(epochs):
            # Train D
            for batch_number, (images, _) in enumerate(data_loader):
                batch_size = images.shape[0]
                real_images = to_var(images.view(batch_size, -1))
                z = to_var(torch.randn(batch_size, 64))
                D.zero_grad()

                fake_images = G(z)
                # Optimize D to correctly classify the two batches
                real_labels = to_var(torch.ones(batch_size, 1))
```

```
        fake_labels = to_var(torch.zeros(batch_size, 1))
        real_image_scores = D(real_images)
        d_loss_real = criterion(real_image_scores, real_labels)
        fake_image_scores = D(fake_images)
        d_loss_fake = criterion(fake_image_scores, fake_labels)
        # Backprop + Optimize the discriminator
        d_loss = d_loss_real + d_loss_fake
        d_loss.backward()
        d_optimizer.step()


        D_unroll.load_state_dict(D.state_dict())

        for step in range(k):
            z = to_var(torch.randn(batch_size, 64))
            D_unroll.zero_grad()
            fake_images = G(z)
            # Optimize D_unroll to correctly classify the two batches
            real_image_scores = D_unroll(real_images)
            d_loss_real = criterion(real_image_scores, real_labels)
            fake_image_scores = D_unroll(fake_images)
            d_loss_fake = criterion(fake_image_scores, fake_labels)
            # Backprop + Optimize the discriminator
            d_loss = d_loss_real + d_loss_fake
            d_loss.backward()
            d_unroll_optimizer.step()
        # Train G

        D_unroll.zero_grad()
        G.zero_grad()

        z = to_var(torch.randn(batch_size, 64))
        fake_images_g = G(z)
        fake_image_scores_g = D_unroll(fake_images_g)
        g_loss = criterion(fake_image_scores_g, real_labels)
        g_loss.backward()
        g_optimizer.step()
        D_loss.append(d_loss.detach().cpu().numpy())
        G_loss.append(g_loss.detach().cpu().numpy())
    print("epoch:",epoch, " d loss:", np.mean(D_loss), " g loss:",\
    np.mean(G_loss))
```

```
            total_D_loss.append(np.mean(D_loss))
            total_G_loss.append(np.mean(G_loss))
            fake_images_g = fake_images_g.view(fake_images_g.size(0), 1, 28, 28)
            save_image(denorm(fake_images_g.data), 'f-%0.3d.png' %(epoch + 1))
            if epoch%2 == 0:
                print("Total losses:"," D loss:", total_D_loss, " G loss:", total_G_loss)
                plotLoss(total_D_loss, total_G_loss)
        print("Total losses:"," D loss:", total_D_loss, " G loss:", total_G_loss)
        plotLoss(total_D_loss, total_G_loss)

    except KeyboardInterrupt:
        print('Training ended early.')

train_unrolled(8, nn.BCELoss(), 7)

NB_SAMPLES = 3000
z = to_var(torch.randn(NB_SAMPLES, 64))
fake_images = G(z)
fake_images_g = fake_images.view(fake_images.size(0), 1, 28, 28)

prediction_list =[]
real_image_scores = D_G(fake_images)
predictions = torch.max(real_image_scores, 1)[1]

plt.hist(predictions.cpu().numpy(),
         bins=np.linspace(0, 9.5, 20))

_ = plt.xticks(range(10))
plt.xlabel('Class ID')
plt.ylabel('Number of Samples')
plt.savefig('distribution_unrolled.png')
plt.show()

W_D = nn.Sequential(
  nn.Linear(28 * 28, 512),
  nn.LeakyReLU(0.2),
  nn.Linear(512, 256),
  nn.LeakyReLU(0.2),
  nn.Linear(256, 1),
)
# Generator
```

```
W_G = nn.Sequential(
  nn.Linear(64, 256),
  nn.LeakyReLU(0.2),
  nn.Linear(256, 256),
  nn.LeakyReLU(0.2),
  nn.Linear(256, 28 * 28),
  nn.Tanh()
)

W_G.load_state_dict(torch.load('./WGAN-g-051.pkl'))
W_D.load_state_dict(torch.load('./WGAN-d-051.pkl'))

W_G = W_G.to(device)

NB_SAMPLES = 3000
z = to_var(torch.randn(NB_SAMPLES, 64))
fake_images = W_G(z)
fake_images_g = fake_images.view(fake_images.size(0), 1, 28, 28)

prediction_list =[]
real_image_scores = D_G(fake_images)
predictions = torch.max(real_image_scores, 1)[1]

plt.hist(predictions.cpu().numpy(),
         bins=np.linspace(0, 9.5, 20))

_ = plt.xticks(range(10))
plt.xlabel('Class ID')
plt.ylabel('Number of Samples')
plt.savefig('Wasserstein_unrolled.png')
plt.show()
```

Appendix 9: Conditional GAN

```
def plotLoss(d_loss, g_loss, xlabel = "epochs", ylabel = "loss"):
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.plot(d_loss, label="d loss")
    plt.plot(g_loss, label="g loss")
    plt.legend(loc='best')
    plt.savefig("conditional_loss.png")
    plt.show()

# Discriminator
D = nn.Sequential(
    nn.Linear((28 * 28)+10, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 1),
    nn.Sigmoid()
)
D = D.to(device)

# Generator
G = nn.Sequential(
    nn.Linear(64 + 10, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 256),
    nn.LeakyReLU(0.2),
    nn.Linear(256, 28 * 28),
    nn.Tanh()
)
G = G.to(device)

criterion = nn.BCELoss()
d_optimizer = torch.optim.Adam(D.parameters(), lr=0.0003)
g_optimizer = torch.optim.Adam(G.parameters(), lr=0.0003)

def ConditionalGAN(criterion):
    total_D_loss = []
    total_G_loss = []
```

```
try:
    for epoch in range(N_EPOCHS):
        D_loss = []
        G_loss = []
        for batch_number, (images, labels) in enumerate(data_loader):
            # Train D
            D.zero_grad()
            labels = onehotencoding(labels).type(torch.FloatTensor).to(device)
            # print(labels.shape)
            batch_size = images.shape[0]
            real_images = to_var(images.view(batch_size, -1))
            real_images = torch.cat((real_images, labels), dim=1)
            z = to_var(torch.randn(batch_size, 64),)
            # print(z.shape)
            z = torch.cat((z, labels), dim=1)
            fake_images = G(z)
            # Optimize D to correctly classify the two batches
            real_labels = to_var(torch.ones(batch_size, 1))
            fake_labels = to_var(torch.zeros(batch_size, 1))

            real_image_scores = D(real_images)

            d_loss_real = criterion(real_image_scores, real_labels)

            fake_images = torch.cat((fake_images, labels), dim=1)
            fake_image_scores = D(fake_images)
            d_loss_fake = criterion(fake_image_scores, fake_labels)
            # Backprop + Optimize the discriminator
            d_loss = d_loss_real + d_loss_fake
            d_loss.backward()
            d_optimizer.step()

            G.zero_grad()
            D.zero_grad()
            # Train G
            z = to_var(torch.randn(batch_size, 64))
            z = torch.cat((z, labels), dim=1)

            fake_images_g = G(z)

            fake_images_g_2 = torch.cat((fake_images_g, labels), dim=1)
```

```
                fake_image_scores_g = D(fake_images_g_2)
                g_loss = criterion(fake_image_scores_g, real_labels)

                g_loss.backward()
                g_optimizer.step()
                D_loss.append(d_loss.detach().cpu().numpy())
                G_loss.append(g_loss.detach().cpu().numpy())

            print("epoch:",epoch, " d loss:", np.mean(D_loss), \
            " g loss:", np.mean(G_loss))
            total_D_loss.append(np.mean(D_loss))
            total_G_loss.append(np.mean(G_loss))
            fake_images_g = fake_images_g.view(fake_images_g.size(0), 1, 28, 28)
            save_image(denorm(fake_images_g.data), 'f-%0.3d.png' %(epoch + 1))
            if epoch%10 == 0:
                print("Total losses:"," D loss:", total_D_loss, \
                    " G loss:", total_G_loss)
                plotLoss(total_D_loss, total_G_loss)
        print("Total losses:"," D loss:", total_D_loss, \
            " G loss:", total_G_loss)
        plotLoss(total_D_loss, total_G_loss)

    except KeyboardInterrupt:
        print('Training ended early.')

ConditionalGAN(nn.BCELoss())

NB_SAMPLES = 3

for i in range(10):
  labels = np.array([i]*NB_SAMPLES)
  labels = onehotencoding(labels).type(torch.FloatTensor).to(device)
  z = to_var(torch.randn(NB_SAMPLES, 64))
  z = torch.cat((z, labels), dim=1)

  fake_images = G(z)
  fake_images_g = fake_images.view(fake_images.size(0), 1, 28, 28)
```