Homework 3
CS 5787 Deep Learning
Spring 2021

John Doe - jdoe@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:

example.png

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

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

`rnn.png`

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:

$$\mathrm{U} = \begin{matrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{matrix} ;$$

$$\mathrm{W} = \begin{matrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{matrix} ;$$

$$\mathrm{V} = \begin{matrix} 1 \\ -1 \\ 1 \end{matrix} ;$$

$$\mathrm{Bh} = \begin{matrix} 0 \\ -1 \\ -2 \end{matrix} ;$$

By = 0 ;

$$H0 = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} .$$

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:

Following the instructions, I built a preprocessing finction that takes in the path to the

```
all_merged.txt
```

file and builds a vocabulary which is a dictionary of every unique word (lower case, without punctuation) matched with an index. it also returns the reviews as vectors of length 400 with padding of zeros until the 400 - length of the review where the indices start.

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:

I used a 1-layer GRU model, with 32 hidden layer dimension and 10 input dimension for the embeddings. Accuracy of GRU model: 0.685 on the test set. Train set accuracy under GRU: 0.9477.

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:
I used the embeddings from part 2 and added 2 fully connected layers to the MLP model

of 100 dimensions for the middle linear layer, connecting to the final layer. Yet, I got train accuracy of 0.5030, and test accuracy of 0.4997.

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.

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
```

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

```
        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.

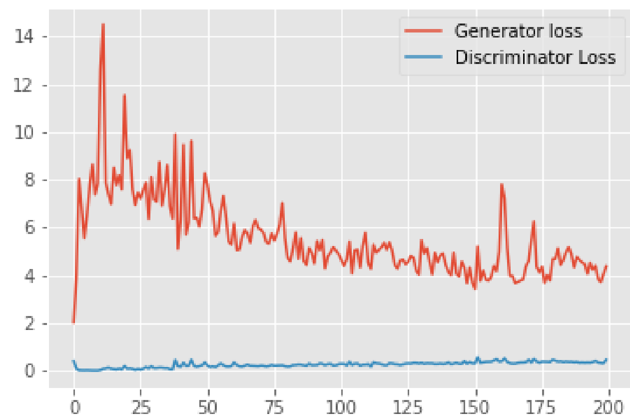
Choice of D architecture:

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.

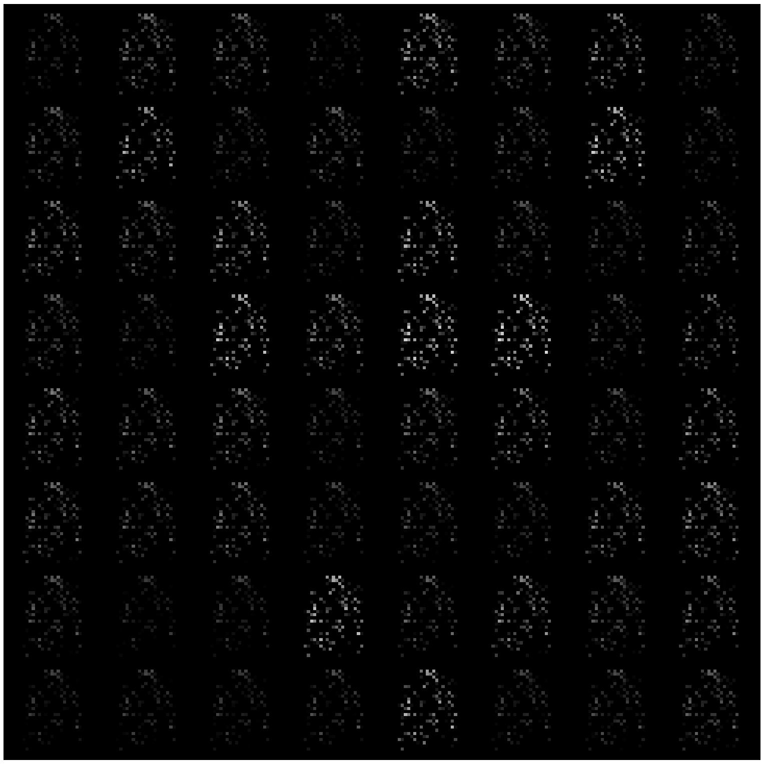
Train a basic GAN that can generate images from the Fashion-MNIST dataset. Plot your training loss curves for your G and D. Show the generated samples from G in 1) the beginning of the training; 2) intermediate stage of the training; and 3) after convergence.

Solution:

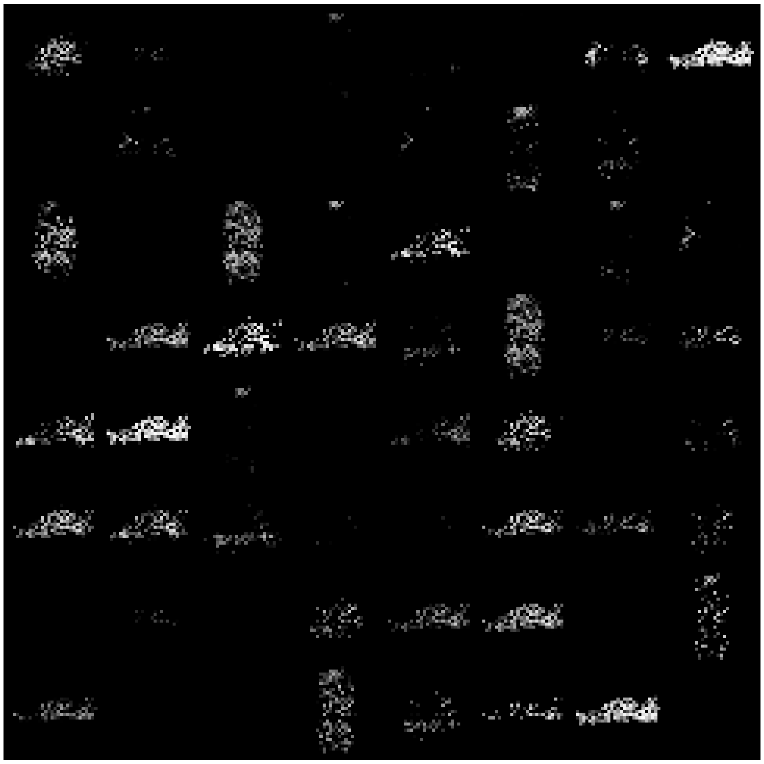
Loss plot:



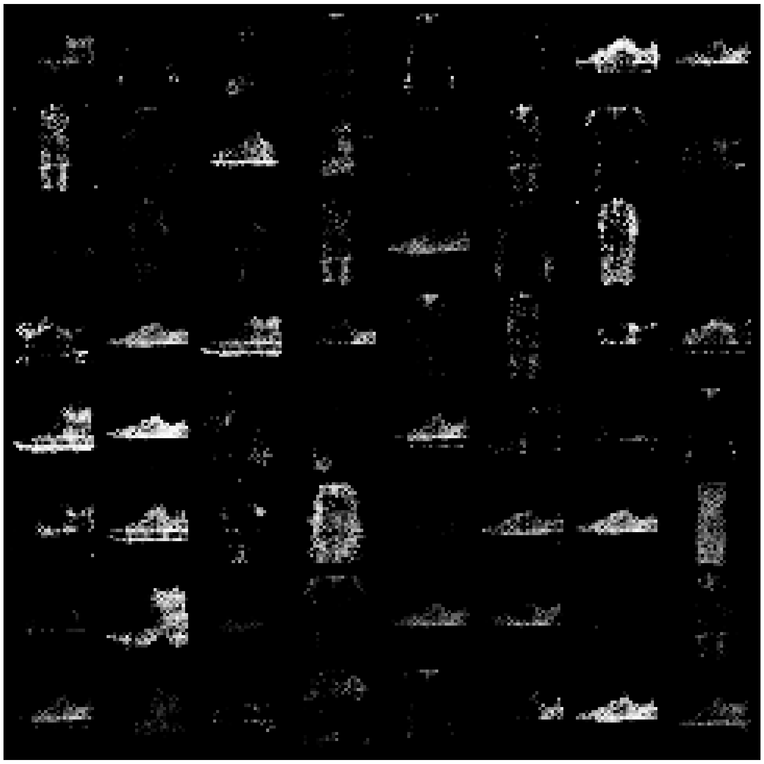
Grid of generated images at epoch 10:



Grid of generated images at epoch 60:



Grid of generated images at epoch 199:



Part 2 - GAN Loss (10 points)

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

MSE

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

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

Wasserstein GAN (WGAN)

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

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

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

Least Square GAN

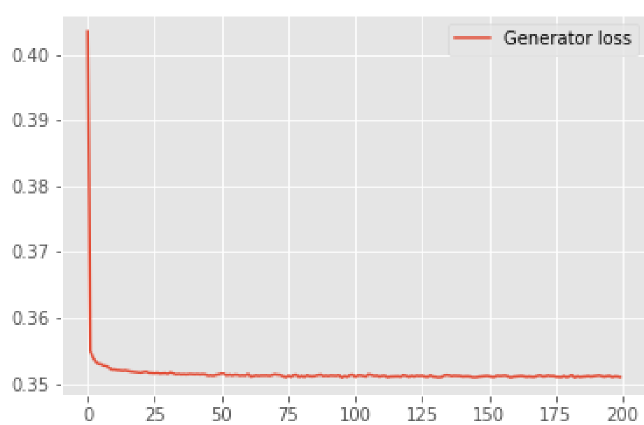
$$\begin{aligned}
\min_D \quad & \mathbb{E}_{x\in p_d}(D(x)-1)^2 + \mathbb{E}_{z\in p_z} D(G(z))^2 \\
\min_G \quad & \mathbb{E}_{z\in p_z}(D(G(z))-1)^2
\end{aligned}$$

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

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

Solution:

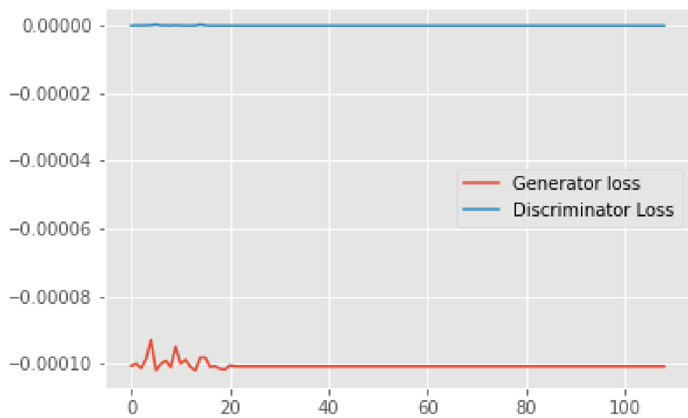
MSE GAN:
Loss plot:



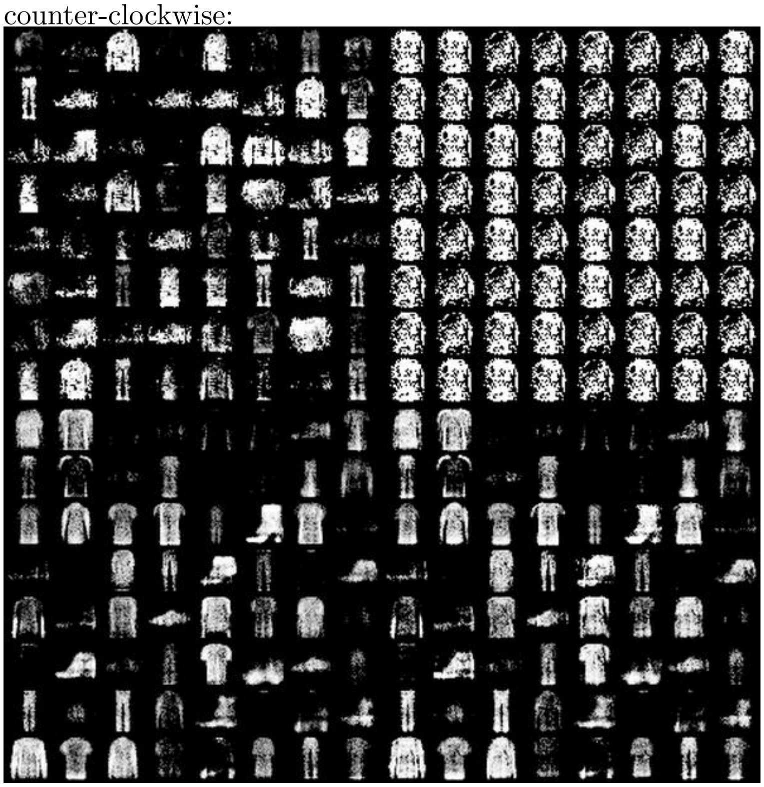
Grid of generated images at epoch 199:

As can be seen from the image, the model doesn't really train to do what it's supposed to do. This is because it only trains to generate something that would look like images from the data-set without the knowledge what parameters do make up a real image. this causes it to find an average of all the images and then minimize the generated images distance from them , without really generating something that looks like an image from the data-set. This obviously took shorter time to train, due to the lack of discriminator training in this model.

Wasserstein GAN:
Loss plot (c=0.0001):

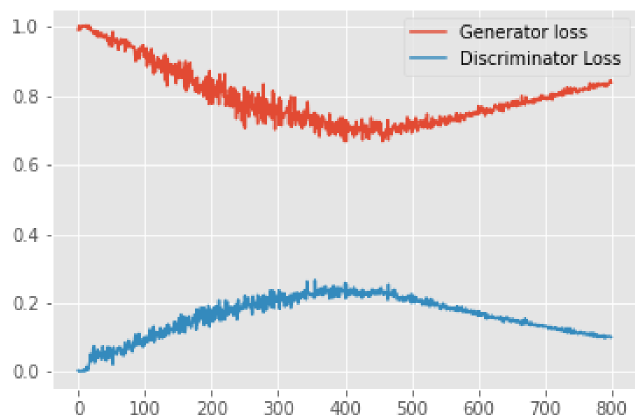


Grid of generated images at epoch 199 with c in {0.1, 0.01, 0.001, 0.0001} from top right counter-clockwise:



This model took more time than the MSE model ti run, however it gave much better results as it trained on both minimizing the generated image from real images AND on making knowing what images make up a "real" image. Also, we observe that the lower the c the higher the quality and resolution of the images.

Least Square GAN:
Loss plot:



Grid of generated images at epoch 799:

This model ran similarly in terms of speed and performance to the WGAN model ,though it was a tiny bit slower (probably due to the extra computation of the squares). the results, however, look very similar, as both are using similar measures of distance.

Part 3 - Mode Collapse in GANs (10 points)

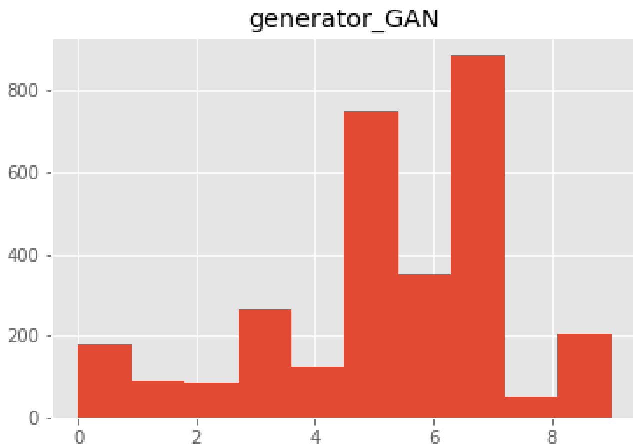
Take a copy of your vanilla GAN discriminator and change its output channel from 1 output to 10 output units. Fine-tune it as a classifier on the Fashion-MNIST training set. You should easily achieve $\sim 90\%$ accuracy on Fashion-MNIST test set.

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:

Predicted labels histogram:



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)
```

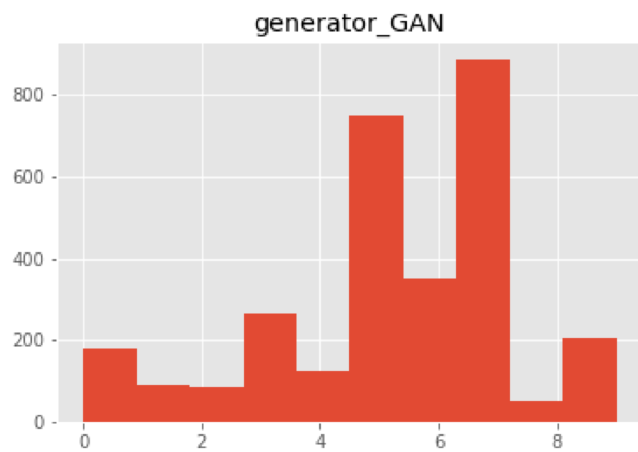
Note that G is trained with a copy of D at each epoch. The original D should not be updated during that part of training. Train an unrolled GAN.

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.

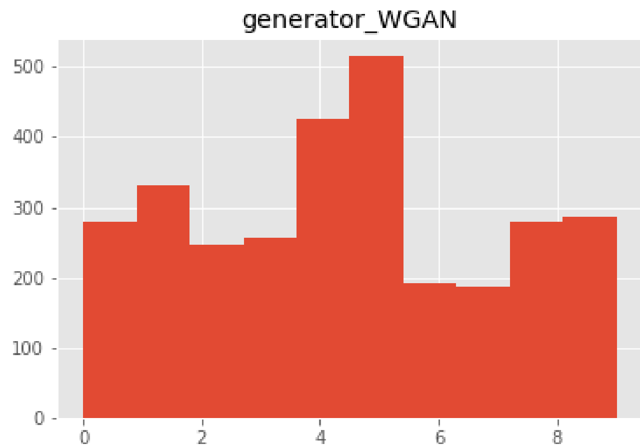
Like unrolled GAN, WGAN is claimed to be less affected by mode collapse. Discuss how each of the three generators performs and which seems to be best at reducing the mode collapse problem.

Solution:

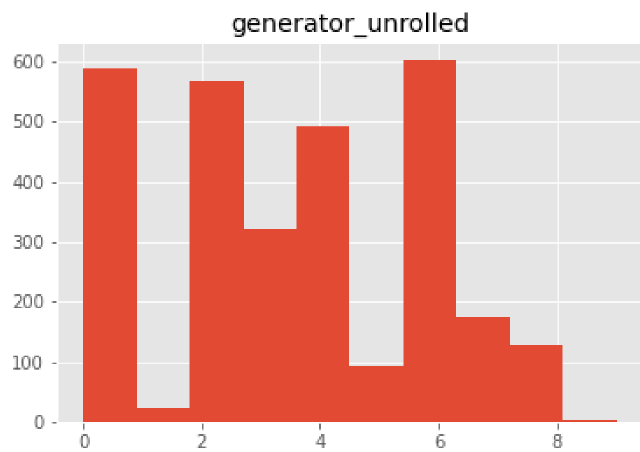
GAN histogram:



Wasserstien GAN histogram:



Unrolled GAN histogram:



The training speed of vanilla gan was the fastes with decent results, the WGAN model was about the same speed with a little better results, but I can't be sure it is due to the model and not to the fine-tuning. In other words, it may be that with better fine-tuning I would get better results for vanilla. (again, I had limited access to gpu which restrained my ability to play with the models). the LSE gave better results after 800 epochs on gpu (comaring to 200 epochs in GAN and WGAN) and gave the best results so far, though it was a little slower. Lastly, the Unrolled was the hardest to fine-tune, but eventually it returned the best results though after the longest training time for 300 epochs. In terms of mode collapse, WGAN definitely performs the best in my experiment. however, it may be a case of undertraining where with more epochs on the generator we could get more uniformity in the unrolled model than that of the WGAN.

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.

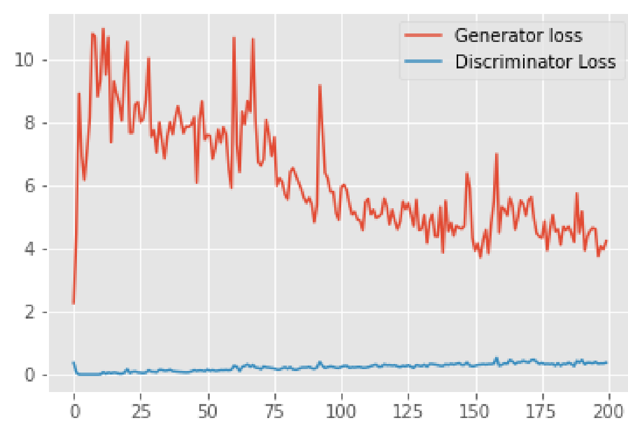
To add the conditional input vector, we need to modify both D and G. First, we need to define the input label vector. We are going to use one-hot encoding vectors for labels: for an image sample with label k of K classes, the vector is K dimensional and has 1 at k-th element and 0 otherwise.

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:

Loss:



3 images of each category generated by the conditional model:

NOTE: Google Colab gave me some trouble and so I trained the model on my personal non-gpu laptop for 200 epochs only and so the results could have been a bit better as can be seen in the loss plot had I run it on for more epochs

First difference is obviously the fact that we control now which categories we want to generate. However, not all categories were generated equally. some are generated with more precision and "depth" (pr volume) than others. The reason may be the relatively low number of epochs that wasn't enough for the generator to learn to mimic the real images. Evidence to this hypothesis is that each category is being generated relatively equally in quality, poor and well alike. it's easy to see that the second row for example is pants, the last row is a bit less obvious but it seems to be boots, but the rest only give us general lines and patterns, though consistent which shows that the model had started to learn them as well.

Code Appendix

Problem 1 Code

```
# To add a new cell, type '# %%'
# To add a new markdown cell, type '# %% [markdown]'
# # %%
import numpy as np
import pandas as pd
import torch
import re
import time
from torch.utils.data import DataLoader, TensorDataset
from torch.nn import Module, GRU, Embedding, Linear, Sigmoid, CrossEntropyLoss
# # Part 1

# from google.colab import drive
# drive.mount('/content/drive')


"data/sentiment_analysis/train_pos_merged.txt"
# function clearing HTML tags from text
def cleanhtml(raw_html):
    cleanr = re.compile('<.*?>')
    cleantext = re.sub(cleanr, '', raw_html)
    return cleantext

# preprocessing
def clean_text(path):
    reviews = []
    all_words = []
    with open(path) as pos:
        lines = pos.readlines()
        for line in lines:
            #clear html tags
            line = cleanhtml(line)
            # lower case and punctuation
            line = re.sub(r'[^a-zA-Z]', ' ', line.lower())
            # split to list of words
            words = line.split()
            # add list to reviews
            reviews.append(words)
            # extend words with new review
            all_words.extend(words)

    return reviews, all_words

def create_vocab(words):
    # create vocabulary with indexes
```

```
    vocab = {}
    id = 1
    for word in words:
        if word not in vocab.keys():
            vocab[word] = id
            id += 1
    return vocab


def vectorize_data(reviews, y, vocab, LENGTH=400):
    y = np.array([y for _ in range(len(reviews))])
    indexed_reviews = np.zeros((len(reviews), LENGTH), dtype = np.int64)
    for i, review in enumerate(reviews):
        indexed_review = []
        for word in review:
            indexed_review.append(vocab[word])
        indexed_reviews[i, max(LENGTH-len(review),0):] = indexed_review[:400]
    return indexed_reviews, y

def preprocessing(path1, path2, y1, y2, vocab, LENGTH=400):
    reviews1, words1 = clean_text(path1)
    reviews2, words2 = clean_text(path2)
    # words1.extend(words2)
    # print(words1)

    del words1, words2

    # vocab = create_vocab(words1)

    x1, y1 = vectorize_data(reviews1, y1, vocab, LENGTH)
    x2, y2 = vectorize_data(reviews2, y2, vocab, LENGTH)

    x = np.concatenate((x1, x2))
    y = np.concatenate((y1, y2))

    return x, y #, vocab






# !ls /content/drive
# path = "/content/drive/MyDrive/Deep_Learning/sentiment_analysis/"
path = "./data/sentiment_analysis/"
reviews, words = clean_text(path + "all_merged.txt")


vocab = create_vocab(words)
```

```python
# vocab


train_x, train_y = preprocessing(path + "train_pos_merged.txt", path + "train_neg_merged.txt", 0, 1, vocab)


# input = torch.from_numpy(train_x[0])
# embedding = Embedding(len(vocab), 3, padding_idx=0)
# embedding(input)


batch_size = 100
train_data =  TensorDataset(torch.from_numpy(train_x), torch.from_numpy(train_y))
train_loader = DataLoader(train_data)

# # Part 2

class GRU_model(Module):

    def __init__(self, vocab_size, input_dim, hidden_dim, n_layers=1, LENGTH=400):

        super(GRU_model, self).__init__()

        self.input_dim = input_dim
        self.n_layers = n_layers
        self.hidden_dim = hidden_dim

        self.embedding = Embedding(vocab_size, input_dim, padding_idx=0)
        self.gru = GRU(input_dim, hidden_dim, n_layers, batch_first=True)
        self.linear = Linear(hidden_dim, 2)
        self.sigmoid = Sigmoid()

    def forward(self, x, h):
        x = self.embedding(x)
        x, h = self.gru(x, h)
        # print(f"shape of x: {x.shape}; shape of h: {h.shape}; shape of x[:,-1]: {x[:,-1].shape}")
        x = self.linear(x[:,-1])
        # print(f"shape of x: {x.shape}")
        x = self.sigmoid(x)
        return x, h

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

```python


# torch.cuda.is_available() checks and returns a Boolean True if a GPU is available, else it'll return False
is_cuda = torch.cuda.is_available()

# If we have a GPU available, we'll set our device to GPU. We'll use this device variable later in our code.
if is_cuda:
    device = torch.device("cuda")
else:
    device = torch.device("cpu")


def gru_train(train_loader, vocab_size, learn_rate, input_dim=10, hidden_dim=16, EPOCHS=5):

    # Setting common hyperparameters
    # input_dim = next(iter(train_loader))[0].shape[1]
    # print(next(iter(train_loader))[0].shape[1])
    output_dim = 1
    n_layers = 1
    # Instantiating the model
    model = GRU_model(vocab_size, input_dim, hidden_dim, output_dim, n_layers)
    model.to(device)

    # Defining loss function and optimizer
    criterion = CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learn_rate)

    model.train()
    print("Starting Training")
    epoch_times = []
    # Start training loop
    for epoch in range(1,EPOCHS+1):
        start_time = time.time()
        h = model.init_hidden(batch_size)
        avg_loss = 0.
        counter = 0
        for x, label in train_loader:
            counter += 1
            h = h.data
            model.zero_grad()

            out, h = model(x.to(device), h)
            # print(f"shape of out.squeeze(): {out.squeeze().shape}; shape of label: {label.shape}")
            loss = criterion(out.squeeze(), label.to(device))
            loss.backward()
```

```
            optimizer.step()
            avg_loss += loss.item()
            if counter%100 == 0:
                print("Epoch {}......Step: {}/{}....... Average Loss for Epoch: {}".format(epoch, counter, len(train_loader), avg_loss/counter))
        current_time = time.time()
        print("Epoch {}/{} Done, Total Loss: {}".format(epoch, EPOCHS, avg_loss/len(train_loader)))
        print("Total Time Elapsed: {} seconds".format(str(current_time-start_time)))
        epoch_times.append(current_time-start_time)
    print("Total Training Time: {} seconds".format(str(sum(epoch_times))))
    return model


def gru_evaluate(model, test_loader): #, label_scalers):
    model.eval()
    outputs = []
    results = []
    start_time = time.time()
    model.eval()
    err = 0
    for x, label in test_loader:
        h = model.init_hidden(test_loader.batch_size).data
        input = x.to(device)
        output, h_out = model(input, h)
        result = torch.argmax(output, dim=1)
        results.append(result)
        err += torch.abs(result.to(device) - label.to(device)).sum()
    accuracy = 1 - err/len(test_loader)
    return accuracy, outputs, results


gru_model = gru_train(
    train_loader,
    vocab_size = len(vocab),
    learn_rate=0.0005,
    hidden_dim=32,
    EPOCHS=300
    )


test_x, test_y = preprocessing(path + "test_pos_merged.txt", path + "test_neg_merged.txt", 0, 1, vocab)


batch_size = 100
test_data =  TensorDataset(torch.from_numpy(test_x), torch.from_numpy(test_y))
test_loader = DataLoader(test_data) #, shuffle=True, batch_size=batch_size)
```

```
gru_model = GRU_model(len(vocab), 10, 32, 1, 1)
gru_model.load_state_dict(torch.load("./models/gru/gru_rnn.pt",
    map_location=device))


test_accuracy, test_outputs, test_results = gru_evaluate(gru_model,
    test_loader)
train_accuracy, train_outputs, train_results = gru_evaluate(gru_model,
    train_loader)


train_accuracy

# ## **Part 3** MLP

class MLP_model(Module):

    def __init__(self, vocab_size, input_dim, hidden_dim, LENGTH = 400):

        super(MLP_model, self).__init__()

        self.embedding = Embedding(vocab_size, input_dim, padding_idx=0)
        self.fc1 = Linear(input_dim*LENGTH, hidden_dim)
        self.fc2 = Linear(hidden_dim, 2)
        self.sigmoid = Sigmoid()

    def forward(self, x):
        x = self.embedding(x)
        x = x.flatten(start_dim=1)
        x = self.fc1(x)
        x = torch.relu(x)
        x = self.fc2(x)
        x = self.sigmoid(x)
        return x


def mlp_train(train_loader, vocab_size, learn_rate, input_dim=10,
    hidden_dim=16, EPOCHS=5):

    # Instantiating the model
    model = MLP_model(vocab_size, input_dim, hidden_dim)
    # print(model)
    model.to(device)

    # Defining loss function and optimizer
    criterion = CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learn_rate)
```

```python

    model.train()
    print("Starting Training")
    epoch_times = []
    # Start training loop
    for epoch in range(1,EPOCHS+1):
        start_time = time.time()
        avg_loss = 0.
        counter = 0
        for x, label in train_loader:
            counter += 1
            model.zero_grad()

            out = model(x.to(device))
            # print(out.shape)
            # print(f"shape of out: {out.shape}; shape of label: {label.shape}")
            # return 0
            loss = criterion(out, label.to(device))
            loss.backward()
            optimizer.step()
            avg_loss += loss.item()
            if counter%100 == 0:
                print("Epoch {}......Step: {}/{}....... Average Loss for Epoch: {}".format(epoch, counter, len(train_loader), avg_loss/counter))
        current_time = time.time()
        print("Epoch {}/{} Done, Total Loss: {}".format(epoch, EPOCHS, avg_loss/len(train_loader)))
        # print("Total Time Elapsed: {} seconds".format(str(current_time-start_time)))
        epoch_times.append(current_time-start_time)
    print("Total Training Time: {} seconds".format(str(sum(epoch_times))))
    return model


def mlp_evaluate(model, test_loader):
    outputs = []
    results = []
    start_time = time.time()
    model.eval()
    err = 0
    for x, label in test_loader:
        input = x.to(device)
        output = model(input)
        result = torch.argmax(output, dim=1)
        results.append(result)
        outputs.append(output)
        err += torch.abs(result.to(device) - label.to(device)).sum()
    accuracy = 1 - err/len(test_loader)
    return accuracy, outputs, results
```

```
mlp_model = mlp_train(train_loader, len(vocab), 0.01, hidden_dim=100 ,
    EPOCHS=20)


train_accuracy, train_outputs, train_results = mlp_evaluate(mlp_model,
    train_loader)
test_accuracy, train_outputs, train_results = mlp_evaluate(mlp_model,
    test_loader)


print(train_accuracy)
print(test_accuracy)
```

Listing 1: Problem 1

Problem 2 vanilla

```
import numpy as np
import pandas as pd
import torch
import torchvision as tv
import re
import time
from torch.utils.data import DataLoader, TensorDataset
from torch.nn import Module, GRU, Embedding, Linear, Sigmoid,
    CrossEntropyLoss, ReLU, Tanh, Sequential
from torch import nn
from torchvision import transforms
import torch.optim as optim
from torchvision.utils import make_grid, save_image
from tqdm import tqdm
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torch.optim as optim
import torchvision.datasets as datasets
import imageio
import numpy as np
import matplotlib
from torchvision.utils import make_grid, save_image
from torch.utils.data import DataLoader
from matplotlib import pyplot as plt
from tqdm import tqdm
matplotlib.style.use('ggplot')

# learning parameters
batch_size = 512
epochs = 200
```

```
sample_size = 64 # fixed sample size
nz = 128 # latent vector size
k = 1 # number of steps to apply to the discriminator
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

transform = transforms.Compose([
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,),(0.5,)),
])
to_pil_image = transforms.ToPILImage()

fmnist = datasets.FashionMNIST(root='./', train=True, download=True,
    transform=transform)
data_loader = DataLoader(fmnist, batch_size=batch_size, shuffle=True)

class Generator(Module):
    def __init__(self, nz):
        super(Generator, self).__init__()
        self.nz = nz
        self.main = Sequential(
            Linear(self.nz, 256),
            ReLU(),

            Linear(256, 512),
            ReLU(),

            Linear(512, 784),
            Tanh(),
        )

    def forward(self, x):
        return self.main(x).view(-1, 1, 28, 28)

class Discriminator(Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.n_input = 784
        self.main = Sequential(
            Linear(self.n_input, 1024),
            ReLU(),
            Linear(1024, 512),
            ReLU(),
            Linear(512, 1),
            nn.Sigmoid(),
        )
    def forward(self, x):
        x = x.view(-1, 784)
        return self.main(x)


```

```python


generator = Generator(nz).to(device)
discriminator = Discriminator().to(device)
print('##### GENERATOR #####')
print(generator)
print('######################')
print('\n##### DISCRIMINATOR #####')
print(discriminator)
print('######################')


# optimizers

optim_g = optim.Adam(generator.parameters(), lr=0.0002)
optim_d = optim.Adam(discriminator.parameters(), lr=0.0002)

# loss function
criterion = nn.BCELoss()


# to create real labels (1s)
def label_real(size):
    data = torch.ones(size, 1)
    return data.to(device)
# to create fake labels (0s)
def label_fake(size):
    data = torch.zeros(size, 1)
    return data.to(device)

def create_noise(sample_size, nz):
    return torch.randn(sample_size, nz).to(device)

# to save the images generated by the generator
def save_generator_image(image, path):
    save_image(image, path)

# function to train the discriminator network
def train_discriminator(optimizer, data_real, data_fake):
    b_size = data_real.size(0)
    real_label = label_real(b_size)
    fake_label = label_fake(b_size)
    optimizer.zero_grad()
    output_real = discriminator(data_real)
    loss_real = criterion(output_real, real_label)
    output_fake = discriminator(data_fake)
    loss_fake = criterion(output_fake, fake_label)
    loss_real.backward()
    loss_fake.backward()
    optimizer.step()
```

```python
    return loss_real + loss_fake

# function to train the generator network
def train_generator(optimizer, data_fake):
    b_size = data_fake.size(0)
    real_label = label_real(b_size)
    optimizer.zero_grad()
    output = discriminator(data_fake)
    loss = criterion(output, real_label)
    loss.backward()
    optimizer.step()
    return loss

# create the noise vector
noise = create_noise(sample_size, nz)
generator.train()
discriminator.train()


# path = "/content/drive/MyDrive/Deep_Learning/HW3/"
path = "./models/vanilla_gan/"
epochs = 200
# k = 10
length = 0.

losses_g = [] # to store generator loss after each epoch
losses_d = [] # to store discriminator loss after each epoch
images = [] # to store images generatd by the generator

for epoch in range(epochs):
    start = time.time()
    loss_g = 0.0
    loss_d = 0.0
    for bi, data in enumerate(data_loader):
        image, _ = data
        image = image.to(device)
        b_size = len(image)
        # run the discriminator for k number of steps
        for step in range(k):
            # print(create_noise(b_size, nz).shape)
            data_fake = generator(create_noise(b_size, nz)).detach()
            data_real = image
            # train the discriminator network
            loss_d += train_discriminator(optim_d, data_real, data_fake)
        data_fake = generator(create_noise(b_size, nz))
        # train the generator network
        loss_g += train_generator(optim_g, data_fake)
    # create the final fake image for the epoch
    generated_img = generator(noise).cpu().detach()
    # make the images as grid
```

```
    generated_img = make_grid(generated_img)
    # save the generated torch tensor models to disk
    save_generator_image(generated_img, path + f"gen_img{epoch}.png")
    images.append(generated_img)
    epoch_loss_g = loss_g / bi # total generator loss for the epoch
    epoch_loss_d = loss_d / bi # total discriminator loss for the epoch
    losses_g.append(epoch_loss_g)
    losses_d.append(epoch_loss_d)
    end = time.time() - start
    length += end
    mean_so_far = length / (epoch+1)
    time_left = (mean_so_far * (epochs - epoch - 1))/60

    print(f"Epoch {epoch} of {epochs}:\t\t{end:.2f} seconds;\ttotal: {
   length:.2f};\tminutes left: {time_left:.2f}")
    print(f"Generator loss: {epoch_loss_g:.8f}, Discriminator loss: {
   epoch_loss_d:.8f}")
```

Listing 2: Problem 2 vanilla

Problem 2 loss

```
from my_functions import *
import numpy as np
import pandas as pd
import torch
from torch._C import device
import torchvision as tv
import re
import time
from torch.utils.data import DataLoader, TensorDataset
from torch.nn import Module, GRU, Embedding, Linear, Sigmoid,
   CrossEntropyLoss, ReLU, Tanh, Sequential
from torch import nn
from torchvision import transforms
import torch.optim as optim
from torchvision.utils import make_grid, save_image
from tqdm import tqdm
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torch.optim as optim
import torchvision.datasets as datasets
import imageio
import numpy as np
import matplotlib
from torchvision.utils import make_grid, save_image
from torch.utils.data import DataLoader
from matplotlib import pyplot as plt
from tqdm import tqdm
```

```
matplotlib.style.use('ggplot')


# learning parameters
batch_size = 512
epochs = 200
sample_size = 64 # fixed sample size
nz = 128 # latent vector size
k = 1 # number of steps to apply to the discriminator
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

transform = transforms.Compose([
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,),(0.5,)),
])
to_pil_image = transforms.ToPILImage()

fmnist = datasets.FashionMNIST(root='./', train=True, download=True,
    transform=transform)
data_loader = DataLoader(fmnist, batch_size=batch_size, shuffle=True)

########### MSE ###########

class Generator(Module):
    def __init__(self, nz):
        super(Generator, self).__init__()
        self.nz = nz
        self.main = Sequential(
            Linear(self.nz, 256),
            ReLU(),

            Linear(256, 512),
            ReLU(),

            Linear(512, 784),
            Tanh(),
        )

    def forward(self, x):
        return self.main(x).view(-1, 1, 28, 28)


nz = 128 # latent vector size
device = 'cude' if torch.cuda.is_available() else 'cpu'

generator = Generator(nz).to(device)
print('##### GENERATOR #####')
print(generator)
print('######################')

```

```python
optim_g = optim.Adam(generator.parameters(), lr=0.0002)

criterion = nn.MSELoss()

losses_g = [] # to store generator loss after each epoch
images = [] # to store images generatd by the generator

# function to train the generator network
def train_generator(optimizer, data_fake, data_real):

    optimizer.zero_grad()
    loss = criterion(data_fake, data_real)
    loss.backward()
    optimizer.step()
    return loss

noise = create_noise(128, nz)

# path = "/content/drive/MyDrive/Deep_Learning/HW3/"
path = "./models/MSE/"
epochs = 200
# k = 10
length = 0.

for epoch in range(epochs):
    start = time.time()
    loss_g = 0.0
    # loss_d = 0.0
    # for bi, data in tqdm(enumerate(data_loader), total=int(len(fmnist)/
    data_loader.batch_size)):
    for bi, data in enumerate(data_loader):
        image, _ = data
        image = image.to(device)
        b_size = len(image)
        data_fake = generator(create_noise(b_size, nz))
        # train the generator network
        loss_g += train_generator(optim_g, data_fake, image)
    # create the final fake image for the epoch
    generated_img = generator(noise).cpu().detach()
    # make the images as grid
    generated_img = make_grid(generated_img)
    # save the generated torch tensor models to disk
    save_generator_image(generated_img, path + f"gen_img{epoch}.png")
    images.append(generated_img)
    epoch_loss_g = loss_g / bi # total generator loss for the epoch
    # epoch_loss_d = loss_d / bi # total discriminator loss for the epoch
    losses_g.append(epoch_loss_g)
    # losses_d.append(epoch_loss_d)
    prev = -9999
    # print(f"")
```

```python
    end = time.time() - start
    length += end
    mean_so_far = length / (epoch+1)
    time_left = (mean_so_far * (epochs - epoch - 1))/60

    print(f"Epoch {epoch} of {epochs}:\t\t{end:.2f} seconds;\ttotal: {
   length:.2f};\tminutes left: {time_left:.2f}")
    print(f"Generator loss: {epoch_loss_g:.8f}")
    prev = epoch_loss_g

########### WGAN ###########

class Generator(Module):
    def __init__(self, nz):
        super(Generator, self).__init__()
        self.nz = nz
        self.main = Sequential(
            Linear(self.nz, 256),
            ReLU(),

            Linear(256, 512),
            ReLU(),

            Linear(512, 784),
            Tanh(),
        )

    def forward(self, x):
        return self.main(x).view(-1, 1, 28, 28)

class Discriminator(Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.n_input = 784
        self.main = Sequential(
            Linear(self.n_input, 1024),
            ReLU(),
            Linear(1024, 512),
            ReLU(),
            Linear(512, 1),
            # nn.Sigmoid(),
        )
    def forward(self, x):
        x = x.view(-1, 784)
        return self.main(x)



generator = Generator(nz).to(device)
discriminator = Discriminator().to(device)
```

```
print('##### GENERATOR #####')
print(generator)
print('######################')
print('\n##### DISCRIMINATOR #####')
print(discriminator)
print('######################')

optim_g = optim.RMSprop(generator.parameters(), lr=0.0001)
optim_d = optim.RMSprop(discriminator.parameters(), lr=0.0001)

# function to train the discriminator network
def train_discriminator(optimizer, data_real, data_fake):
    # b_size = data_real.size(0)
    # real_label = label_real(b_size)
    # fake_label = label_fake(b_size)
    optimizer.zero_grad()
    output_real = discriminator(data_real)
    # loss_real = criterion(output_real, real_label)
    output_fake = discriminator(data_fake)
    loss = -(torch.mean(output_real) - torch.mean(output_fake))
    # loss_fake = criterion(output_fake, fake_label)
    loss.backward()
    optimizer.step()
    return loss

# function to train the generator network
def train_generator(optimizer, data_fake):
    # b_size = data_fake.size(0)
    # real_label = label_real(b_size)
    optimizer.zero_grad()
    output = discriminator(data_fake)
    loss = -torch.mean(output)
    loss.backward()
    optimizer.step()
    return loss

# create the noise vector
noise = create_noise(sample_size, nz)
generator.train()
discriminator.train()

losses_g = [] # to store generator loss after each epoch
losses_d = [] # to store discriminator loss after each epoch
images = [] # to store images generatd by the generator

path = "/content/drive/MyDrive/Deep_Learning/HW3/outputs_wasserstein00001/"
# path = ""
epochs = 200
c = 0.0001
# k = 10
```

```python

for epoch in range(epochs):
    loss_g = 0.0
    loss_d = 0.0
    # for bi, data in tqdm(enumerate(data_loader), total=int(len(fmnist)/
    data_loader.batch_size)):
    for bi, data in enumerate(data_loader):
        image, _ = data
        image = image.to(device)
        b_size = len(image)
        # run the discriminator for k number of steps
        for step in range(k):
            # print(create_noise(b_size, nz).shape)
            data_fake = generator(create_noise(b_size, nz)).detach()
            data_real = image
            # train the discriminator network
            loss_d += train_discriminator(optim_d, data_real, data_fake)
        data_fake = generator(create_noise(b_size, nz))
        # train the generator network
        loss_g += train_generator(optim_g, data_fake)
    # create the final fake image for the epoch
    # noise = create_noise(b_size, nz)
    generated_img = generator(noise).cpu().detach()
    # make the images as grid
    generated_img = make_grid(generated_img)
    # save the generated torch tensor models to disk
    save_generator_image(generated_img, path + f"gen_img{epoch}.png")
    images.append(generated_img)
    epoch_loss_g = loss_g / bi # total generator loss for the epoch
    epoch_loss_d = loss_d / bi # total discriminator loss for the epoch
    losses_g.append(epoch_loss_g)
    losses_d.append(epoch_loss_d)

    for p in discriminator.parameters():
        p.data.clamp_(-c, c)

    print(f"Epoch {epoch} of {epochs}")
    print(f"Generator loss: {epoch_loss_g:.8f}, Discriminator loss: {
    epoch_loss_d:.8f}")



######### LSE #########

class Generator(Module):
    def __init__(self, nz):
        super(Generator, self).__init__()
        self.nz = nz
        self.main = Sequential(
            Linear(self.nz, 256),
```

```
            ReLU(),

            Linear(256, 512),
            ReLU(),

            Linear(512, 784),
            Tanh(),
        )

    def forward(self, x):
        return self.main(x).view(-1, 1, 28, 28)

class Discriminator(Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.n_input = 784
        self.main = Sequential(
            Linear(self.n_input, 1024),
            ReLU(),
            Linear(1024, 512),
            ReLU(),
            Linear(512, 1),
            nn.Sigmoid(),
        )
    def forward(self, x):
        x = x.view(-1, 784)
        return self.main(x)



generator = Generator(nz).to(device)
discriminator = Discriminator().to(device)
print('##### GENERATOR #####')
print(generator)
print('######################')
print('\n##### DISCRIMINATOR #####')
print(discriminator)
print('######################')

optim_g = optim.Adam(generator.parameters(), lr=0.0002)
optim_d = optim.Adam(discriminator.parameters(), lr=0.0002)

# function to train the discriminator network
def train_discriminator(optimizer, data_real, data_fake):
    optimizer.zero_grad()
    output_real = discriminator(data_real)
    output_fake = discriminator(data_fake)
    loss = torch.mean((output_real-1)**2) + torch.mean(output_fake**2)
    loss.backward()
    optimizer.step()
```

```
    return loss

# function to train the generator network
def train_generator(optimizer, data_fake):
    optimizer.zero_grad()
    output = discriminator(data_fake)
    loss = torch.mean((output-1)**2)
    loss.backward()
    optimizer.step()
    return loss


# create the noise vector
noise = create_noise(sample_size, nz)
generator.train()
discriminator.train()


losses_g = [] # to store generator loss after each epoch
losses_d = [] # to store discriminator loss after each epoch
images = [] # to store images generatd by the generator

path = "/content/drive/MyDrive/Deep_Learning/HW3/outputs_LSE/"
# path = ""
epochs = 800
# c = 0.01
# k = 10

for epoch in range(epochs):
    loss_g = 0.0
    loss_d = 0.0
    for bi, data in enumerate(data_loader):
        image, _ = data
        image = image.to(device)
        b_size = len(image)
        # run the discriminator for k number of steps
        for step in range(k):
            # print(create_noise(b_size, nz).shape)
            data_fake = generator(create_noise(b_size, nz)).detach()
            data_real = image
            # train the discriminator network
            loss_d += train_discriminator(optim_d, data_real, data_fake)
        data_fake = generator(create_noise(b_size, nz))
        # train the generator network
        loss_g += train_generator(optim_g, data_fake)
    # create the final fake image for the epoch
    # noise = create_noise(b_size, nz)
    generated_img = generator(noise).cpu().detach()
    # make the images as grid
    generated_img = make_grid(generated_img)
```

```
    # save the generated torch tensor models to disk
    save_generator_image(generated_img, path + f"gen_img{epoch}.png")
    images.append(generated_img)
    epoch_loss_g = loss_g / bi # total generator loss for the epoch
    epoch_loss_d = loss_d / bi # total discriminator loss for the epoch
    losses_g.append(epoch_loss_g)
    losses_d.append(epoch_loss_d)

    # for p in discriminator.parameters():
    #     p.data.clamp_(-c, c)

    print(f"Epoch {epoch} of {epochs}")
    print(f"Generator loss: {epoch_loss_g:.8f}, Discriminator loss: {
    epoch_loss_d:.8f}")
```

Listing 3: Problem 2 loss

Problem 2 mode collapse

```
from my_functions import *
import numpy as np
import pandas as pd
import torch
import torchvision as tv
import re
import time
from torch.utils.data import DataLoader, TensorDataset
from torch.nn import Module, GRU, Embedding, Linear, Sigmoid,
    CrossEntropyLoss, ReLU, Tanh, Sequential
from torch import nn
from torchvision import transforms
import torch.optim as optim
from torchvision.utils import make_grid, save_image
from tqdm import tqdm
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torch.optim as optim
import torchvision.datasets as datasets
import imageio
import numpy as np
import matplotlib
from torchvision.utils import make_grid, save_image
from torch.utils.data import DataLoader
from matplotlib import pyplot as plt
from tqdm import tqdm
matplotlib.style.use('ggplot')


# learning parameters
```

```python
batch_size = 512
epochs = 200
sample_size = 64 # fixed sample size
nz = 128 # latent vector size
k = 1 # number of steps to apply to the discriminator
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

transform = transforms.Compose([
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,),(0.5,)),
])
to_pil_image = transforms.ToPILImage()

fmnist = datasets.FashionMNIST(root='./', train=True, download=True,
    transform=transform)
data_loader = DataLoader(fmnist, batch_size=batch_size, shuffle=True)

import torch.nn as nn
import torch.nn.functional as F

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(28*28, 500)
        self.fc2 = nn.Linear(500, 256)
        self.fc3 = nn.Linear(256, 10)


    def forward(self, x):
        x = x.view(-1, 28*28)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


MLP_network = MLP()


discriminator = MLP()
discriminator.load_state_dict(torch.load("./outputs/fmnist_classifier.pth",
     map_location=device))

optim_d = optim.Adam(discriminator.parameters(), lr=0.00005)

criterion = nn.CrossEntropyLoss()

losses = []
accuracies = []
epochs = 50
```

```

discriminator.train()

# function to train the discriminator network
def train_discriminator(optimizer, data, labels):
    optimizer.zero_grad()
    output = discriminator(data)
    # print(output.shape)
    loss = criterion(output, labels)
    loss.backward()
    optimizer.step()
    return loss, output

for epoch in range(epochs):
    total = 0
    acc_loss = 0
    correct = 0
    for bi, (images, labels) in enumerate(data_loader):
        loss, output = train_discriminator(optim_d, images, labels)
        acc_loss += loss
        b_size = len(labels)
        total += b_size
        predicted = torch.argmax(output, dim=1)
        correct += (predicted==labels).sum()
        accuracy = correct/total
        avg_loss = loss/b_size
        if bi%20==0:
            print(f"Epoch {epoch}/{epochs}; Batch {bi}: Loss = {loss:.5f}\t
\tAccuracy = {accuracy:.5f}")


    losses.append(acc_loss/total)
    accuracies.append(accuracy)


# load model and set to evaluate mode

class Generator(Module):
    def __init__(self, nz):
        super(Generator, self).__init__()
        self.nz = nz
        self.main = Sequential(
            Linear(self.nz, 256),
            ReLU(),

            Linear(256, 512),
            ReLU(),

            Linear(512, 784),
            Tanh(),
```

```
        )

    def forward(self, x):
        return self.main(x).view(-1, 1, 28, 28)


generator_GAN = Generator(nz)
generator_WGAN = Generator(nz)
generator_unrolled = Generator(nz)

generator_GAN.load_state_dict(torch.load("./models/vanilla_gan/generator.
    pth", map_location=torch.device('cpu')))
generator_WGAN.load_state_dict(torch.load("./models/wasserstein/generator.
    pth", map_location=torch.device('cpu')))
generator_unrolled.load_state_dict(torch.load("./models/unrolled/generator.
    pt", map_location=torch.device('cpu')))
generator_GAN.eval()
generator_WGAN.eval()
generator_unrolled.eval()
# print(generator)

# generate 3,000 new images

sample_size = 3000

def create_noise(sample_size, nz):
    return torch.randn(sample_size, nz).to(device)

nz = 128
# create noise

models = {'generator_GAN': generator_GAN, 'generator_WGAN': generator_WGAN,
     'generator_unrolled': generator_unrolled}

for key, generator in models.items():

    noise = create_noise(sample_size, nz)

    # feed noise to generator
    new_images = generator(noise)

    # feed new images to discriminator
    new_softmax = discriminator(new_images)
    new_labels = torch.argmax(new_softmax, dim=1)

    plt.figure()
    plt.hist(new_labels.numpy(), bins=10)
    plt.title(key)
```

```
    plt.savefig(f"./outputs/new_labels_histogram_{key}.png")
```

Listing 4: Problem 2 mode collapse

Problem 2 unrolled

```
from my_functions import *
import numpy as np
import pandas as pd
import torch
import torchvision as tv
import re
import time
from torch.utils.data import DataLoader, TensorDataset
from torch.nn import Module, GRU, Embedding, Linear, Sigmoid,
    CrossEntropyLoss, ReLU, Tanh, Sequential
from torch import nn
from torchvision import transforms
import torch.optim as optim
from torchvision.utils import make_grid, save_image
from tqdm import tqdm
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torch.optim as optim
import torchvision.datasets as datasets
import imageio
import numpy as np
import matplotlib
from torchvision.utils import make_grid, save_image
from torch.utils.data import DataLoader
from matplotlib import pyplot as plt
from tqdm import tqdm
import copy
matplotlib.style.use('ggplot')

# learning parameters
batch_size = 32
epochs = 200
sample_size = 64 # fixed sample size
nz = 128 # latent vector size
k = 1 # number of steps to apply to the discriminator
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

transform = transforms.Compose([
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,),(0.5,)),
])
to_pil_image = transforms.ToPILImage()

```

```python
fmnist = datasets.FashionMNIST(root='./', train=True, download=True,
    transform=transform)
data_loader = DataLoader(fmnist, batch_size=batch_size, shuffle=True)

class Generator(Module):
    def __init__(self, nz):
        super(Generator, self).__init__()
        self.nz = nz
        self.main = Sequential(
            Linear(self.nz, 256),
            ReLU(),

            Linear(256, 512),
            ReLU(),

            Linear(512, 784),
            Tanh(),
        )

    def forward(self, x):
        return self.main(x).view(-1, 1, 28, 28)


class Discriminator(Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.n_input = 784
        self.main = Sequential(
            Linear(self.n_input, 1024),
            ReLU(),
            Linear(1024, 512),
            ReLU(),
            Linear(512, 1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        x = x.view(-1, 784)
        return self.main(x)

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

generator = Generator(nz).to(device)
discriminator = Discriminator().to(device)
print('##### GENERATOR #####')
```

```
print(generator)
print('#####################')
print('\n##### DISCRIMINATOR #####')
print(discriminator)
print('#####################')

# loss function
criterion = nn.BCELoss()

# optimizers
optim_g = optim.Adam(generator.parameters(), lr=0.0002)
optim_d = optim.Adam(discriminator.parameters(), lr=0.0002)


# function to train the discriminator network
def train_discriminator(optimizer, data_real, data_fake, discriminator,
       create_graph=False):
    b_size = data_real.size(0)
    real_label = label_real(b_size)
    fake_label = label_fake(b_size)
    optimizer.zero_grad()
    output_real = discriminator(data_real)
    loss_real = criterion(output_real, real_label)
    output_fake = discriminator(data_fake)
    loss_fake = criterion(output_fake, fake_label)
    # loss_real.backward()
    # loss_fake.backward()
    loss = loss_fake + loss_real
    loss.backward(create_graph=create_graph)
    optimizer.step()
    return loss.item()

# function to train the generator network
def train_generator(optimizer, data_fake, discriminator):
    b_size = data_fake.size(0)
    real_label = label_real(b_size)
    optimizer.zero_grad()
    output = discriminator(data_fake)
    loss = criterion(output, real_label)
    loss.backward()
    optimizer.step()
    return loss.item()

    # create the noise vector
noise = create_noise(sample_size, nz)
generator.train()
discriminator.train()

# path = "/content/drive/MyDrive/Deep_Learning/HW3/outputs_unrolled_gan/"
path = "./models/unrolled/"
```

```
epochs = 200
k = 1

losses_g = [] # to store generator loss after each epoch
losses_d = [] # to store discriminator loss after each epoch
losses_ud = []
images = [] # to store images generatd by the generator
length = 0.

for epoch in range(epochs):
    start = time.time()
    # length = 0.
    loss_g = 0.0
    loss_d = 0.0
    loss_ud = 0.
    for bi, data in enumerate(data_loader):
        image, _ = data
        image = image.to(device)
        b_size = len(image)


        # print(create_noise(b_size, nz).shape)
        data_fake = generator(create_noise(b_size, nz)).detach()
        data_real = image
        # train the discriminator network
        loss_d += train_discriminator(optim_d, data_real, data_fake,
    discriminator)

        # unroll
        backup = copy.deepcopy(discriminator)
        # run the unrolled discriminator for k number of steps
        for step in range(k):
            data_fake = generator(create_noise(b_size, nz)).detach()
            loss_ud += train_discriminator(optim_d, data_real, data_fake,
    discriminator, create_graph=True)

        # data_fake = generator(create_noise(b_size, nz))
        # train the generator network
        data_fake = generator(create_noise(b_size, nz)).detach()
        loss_g += train_generator(optim_g, data_fake, discriminator)
        discriminator.load(backup)
        del backup


    # create the final fake image for the epoch
    if epoch%1==0:
        generated_img = generator(noise).cpu().detach()
        # make the images as grid
        generated_img = make_grid(generated_img)
        # save the generated torch tensor models to disk
```

```
        save_generator_image(generated_img, path + f"gen_img{epoch}.png")
    images.append(generated_img)
    epoch_loss_g = loss_g / bi # total generator loss for the epoch
    epoch_loss_d = loss_d / bi # total discriminator loss for the epoch
    epoch_loss_ud = loss_ud / (bi*k)
    losses_g.append(epoch_loss_g)
    losses_d.append(epoch_loss_d)
    losses_ud.append(epoch_loss_ud)
    end = time.time() - start
    length += end
    mean_so_far = length / (epoch+1)
    time_left = (mean_so_far * (epochs - epoch - 1))/60

    print(f"Epoch {epoch} of {epochs}:\t\t{end:.2f} seconds;\ttotal: {
    length:.2f};\tminutes left: {time_left:.2f}")
    print(f"Generator loss: {epoch_loss_g:.8f}, Discriminator loss: {
    epoch_loss_d:.5f}, Unrolled Discriminator loss: {epoch_loss_ud:.5f}")
```

Listing 5: Problem 2 unrolled

Problem 2 conditional

```
from my_functions import *
import numpy as np
import pandas as pd
import torch
import torchvision as tv
import re
import time
from torch.utils.data import DataLoader, TensorDataset
from torch.nn import Module, GRU, Embedding, Linear, Sigmoid,
    CrossEntropyLoss, ReLU, Tanh, Sequential
from torch import nn
from torchvision import transforms
import torch.optim as optim
from torchvision.utils import make_grid, save_image
from tqdm import tqdm
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import torch.optim as optim
import torchvision.datasets as datasets
import imageio
import numpy as np
import matplotlib
from torchvision.utils import make_grid, save_image
from torch.utils.data import DataLoader
from matplotlib import pyplot as plt
from tqdm import tqdm
matplotlib.style.use('ggplot')
```

```python

# learning parameters
batch_size = 512
epochs = 200
sample_size = 100 # fixed sample size
nz = 128 # latent vector size
k = 1 # number of steps to apply to the discriminator
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

transform = transforms.Compose([
                                transforms.ToTensor(),
                                transforms.Normalize((0.5,),(0.5,)),
])
to_pil_image = transforms.ToPILImage()

fmnist = datasets.FashionMNIST(root='./', train=True, download=True,
    transform=transform)
data_loader = DataLoader(fmnist, batch_size=batch_size, shuffle=True)

class Generator(Module):
    def __init__(self, nz):
        super(Generator, self).__init__()
        self.nz = nz
        self.main = Sequential(
            Linear(self.nz + 10, 256),
            ReLU(),

            Linear(256, 512),
            ReLU(),

            Linear(512, 784),
            Tanh(),
        )

    def forward(self, x, k):
        x = torch.cat((x, k), dim=1)
        # print(x)
        return self.main(x).view(-1, 1, 28, 28)

class Discriminator(Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.n_input = 784
        self.main = Sequential(
            Linear(self.n_input + 10, 1024),
            ReLU(),
            Linear(1024, 512),
            ReLU(),
            Linear(512, 1),
            nn.Sigmoid()
```

```
        )
    def forward(self, x, k):
        x = x.view(-1, 784)
        x = torch.cat((x, k), dim=1)
        return self.main(x)


generator = Generator(nz).to(device)
discriminator = Discriminator().to(device)
print('##### GENERATOR #####')
print(generator)
print('######################')
print('\n##### DISCRIMINATOR #####')
print(discriminator)
print('######################')

# optimizers

optim_g = optim.Adam(generator.parameters(), lr=0.0002)
optim_d = optim.Adam(discriminator.parameters(), lr=0.0002)

criterion = nn.BCELoss()

# function to train the discriminator network
def train_discriminator(optimizer, data_real, data_fake, onehot):
    b_size = data_real.size(0)
    real_label = label_real(b_size)
    fake_label = label_fake(b_size)
    optimizer.zero_grad()
    output_real = discriminator(data_real, onehot)
    loss_real = criterion(output_real, real_label)
    output_fake = discriminator(data_fake, onehot)
    loss_fake = criterion(output_fake, fake_label)
    loss_real.backward()
    loss_fake.backward()
    optimizer.step()
    return loss_real + loss_fake

# function to train the generator network
def train_generator(optimizer, data_fake, onehot):
    b_size = data_fake.size(0)
    real_label = label_real(b_size)
    optimizer.zero_grad()
    output = discriminator(data_fake, onehot)
    loss = criterion(output, real_label)
    loss.backward()
    optimizer.step()
    return loss

# create the noise vector
```

```
noise = create_noise(sample_size, nz)
onehot_out = np.array([[1 if j % 10 == i else 0 for i in range(10)] for j in range(sample_size)])
onehot_out = torch.from_numpy(onehot_out)
generator.train()
discriminator.train()

# path = "/content/drive/MyDrive/Deep_Learning/HW3/"
path = "./models/conditional/"
epochs = 200
# k = 10
length = 0.

losses_g = [] # to store generator loss after each epoch
losses_d = [] # to store discriminator loss after each epoch
images = [] # to store images generatd by the generator

for epoch in range(epochs):
    start = time.time()
    loss_g = 0.0
    loss_d = 0.0
    for bi, data in enumerate(data_loader):
        image, labels = data
        image = image.to(device)
        onehot = torch.zeros((labels.size(0), labels.max() + 1))
        onehot[np.arange(labels.size(0)), labels] = 1

        # print(image.shape, onehot.shape)
        # print(torch.cat())

        b_size = len(image)
        # run the discriminator for k number of steps
        for step in range(k):
            # print(create_noise(b_size, nz).shape)
            data_fake = generator(create_noise(b_size, nz), onehot).detach()
            data_real = image
            # train the discriminator network
            loss_d += train_discriminator(optim_d, data_real, data_fake, onehot)
        data_fake = generator(create_noise(b_size, nz), onehot)
        # train the generator network
        loss_g += train_generator(optim_g, data_fake, onehot)
    # create the final fake image for the epoch
    generated_img = generator(noise, onehot_out).cpu().detach()
    # make the images as grid
    generated_img = make_grid(generated_img, nrow = 10)
    # save the generated torch tensor models to disk
    save_generator_image(generated_img, path + f"gen_img{epoch}.png")
    images.append(generated_img)
```

```
    epoch_loss_g = loss_g / bi # total generator loss for the epoch
    epoch_loss_d = loss_d / bi # total discriminator loss for the epoch
    losses_g.append(epoch_loss_g)
    losses_d.append(epoch_loss_d)
    end = time.time() - start
    length += end
    mean_so_far = length / (epoch+1)
    time_left = (mean_so_far * (epochs - epoch - 1))/60

    print(f"Epoch {epoch} of {epochs}:\t\t{end:.2f} seconds;\ttotal: {
   length:.2f};\tminutes left: {time_left:.2f}")
    print(f"Generator loss: {epoch_loss_g:.8f}, Discriminator loss: {
   epoch_loss_d:.8f}")

n = 10

onehot = torch.zeros((n*3, n))

print(onehot)
onehot[np.arange(n*3), np.array([[i,i,i] for i in range(n)]).flatten()] = 1
print(onehot)


noise = create_noise(n*3, nz)

generated_img = generator(noise, onehot).cpu().detach()
# make the images as grid
generated_img = make_grid(generated_img, nrow = 3)
save_generator_image(generated_img, f"./models/conditional/
   conditional_generated_{n}.png")
```

Listing 6: Problem 2 conditional