

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.

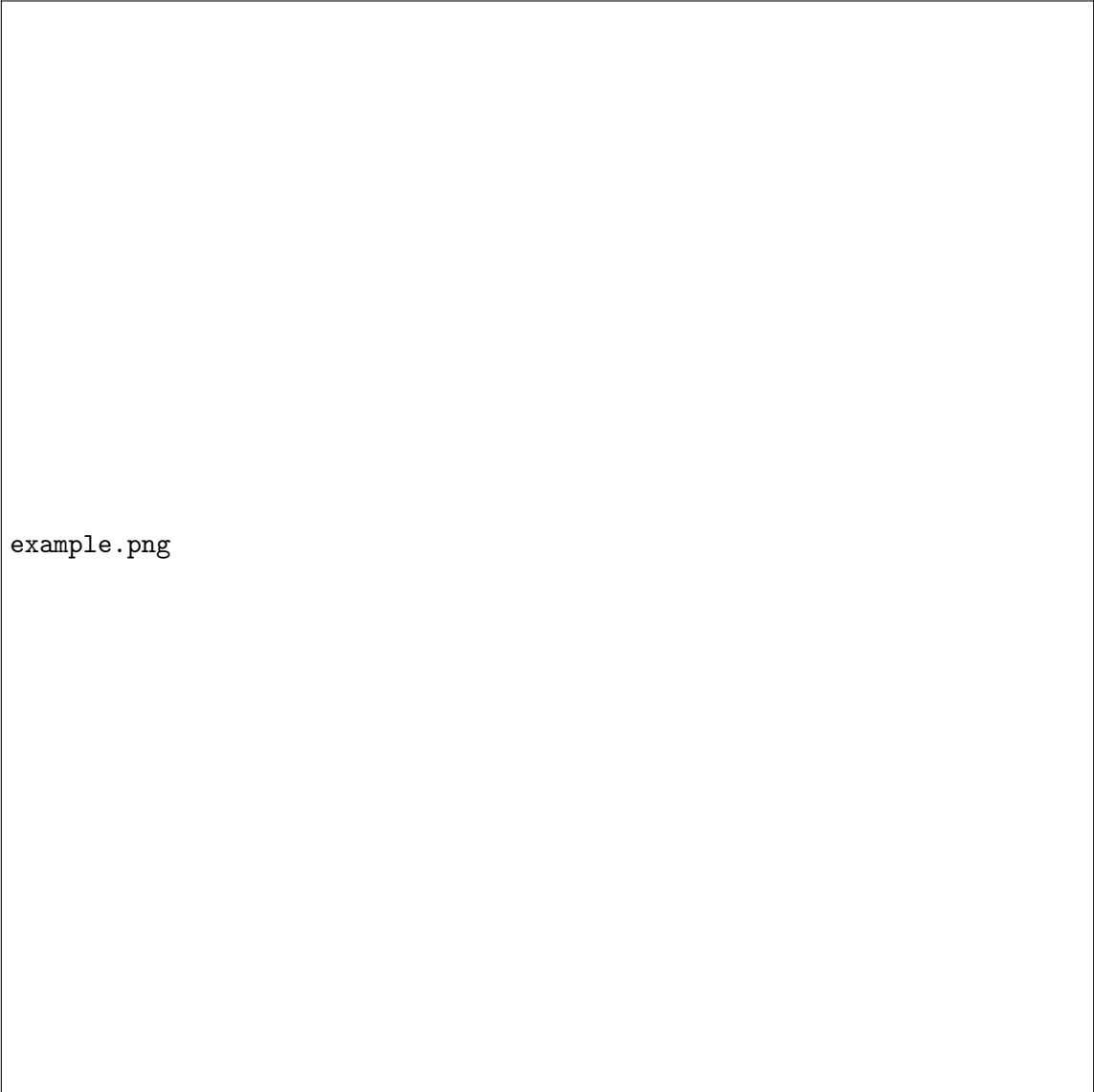
The input to your RNN will be two binary numbers, starting with the *least* significant bit. You will need to pad the largest number with an additional zero on the left side and you should make the other number the same length by padding it with zeros on the left side. For instance, the problem

$$100111 + 110010 = 1011001$$

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:



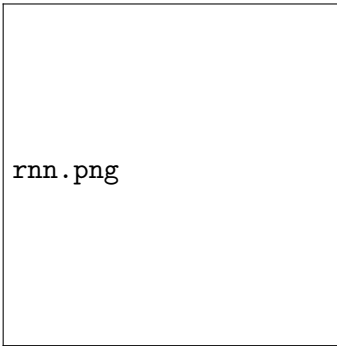
example.png

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

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



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:

$$\mathbf{U} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix};$$

$$\mathbf{W} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix};$$

$$\mathbf{V} = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix};$$

$$\mathbf{B}_h = \begin{bmatrix} 0 \\ -1 \\ -2 \end{bmatrix};$$

$$B_y = 0;$$

$$H_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

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 function 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.

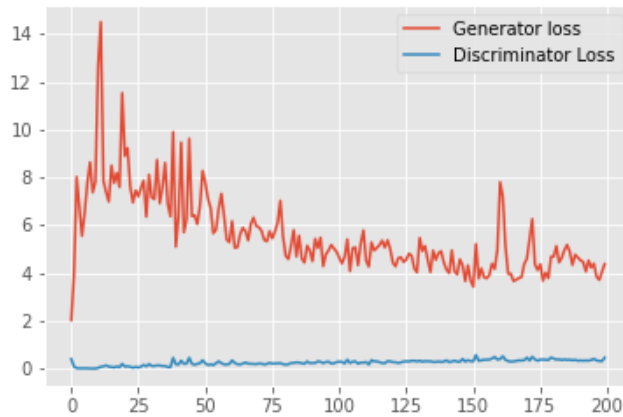
Choice of D architecture:

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:

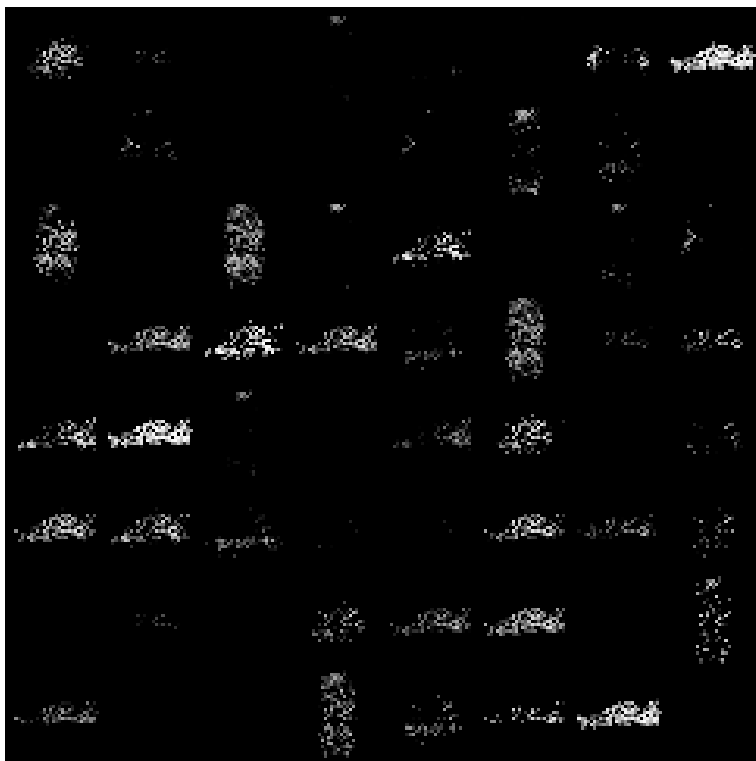
Loss plot:



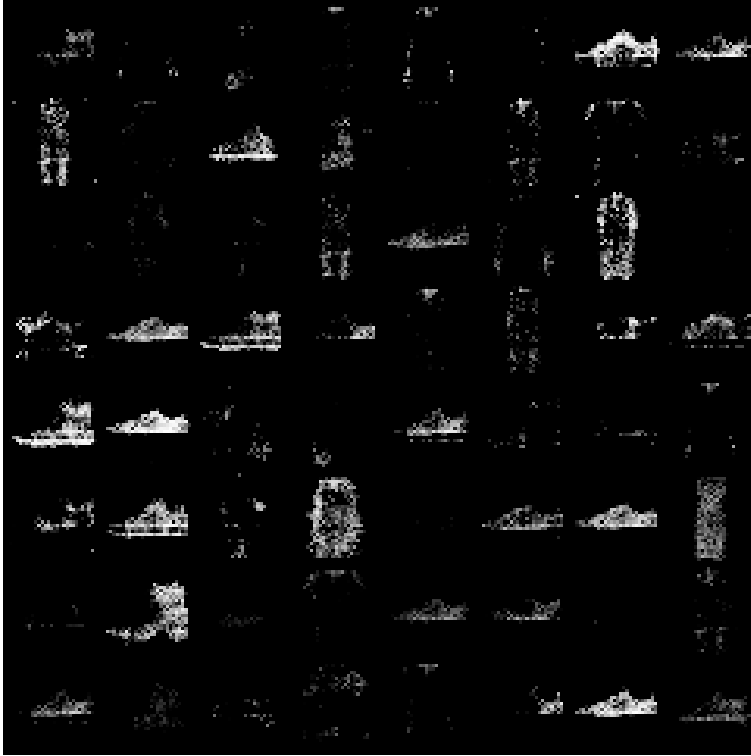
Grid of generated images at epoch 10:



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

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

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

Wasserstein GAN (WGAN)

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

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

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

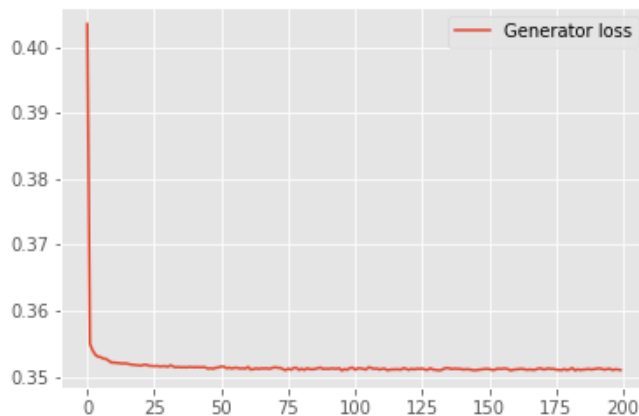
The idea is to provide a smoother loss surface than the original GAN loss.

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.

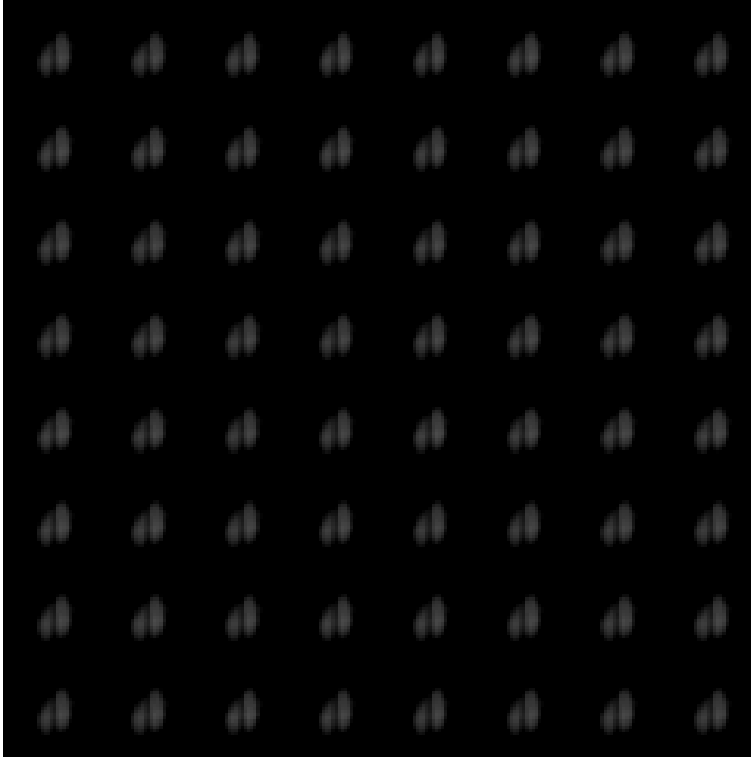
Solution:

MSE GAN:

Loss plot:



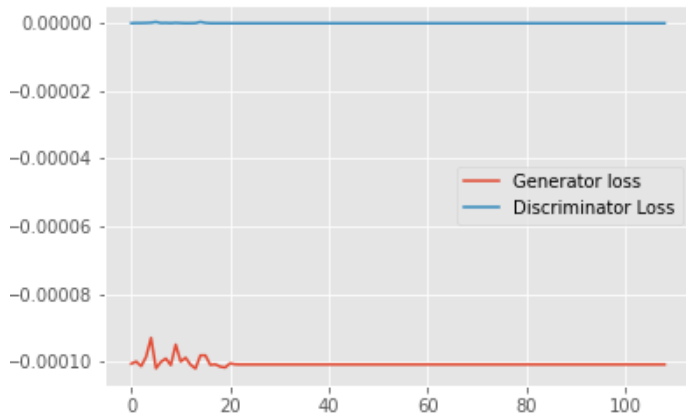
Grid of generated images at epoch 199:



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:

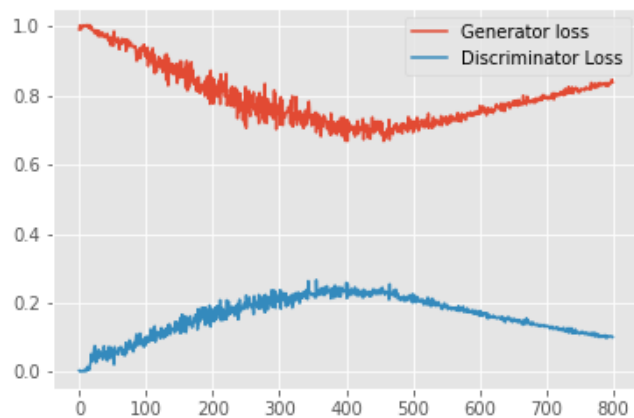


This model took more time than the MSE model to run, however it gave much better results as it trained on both minimizing the generated image from real images AND on making

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)

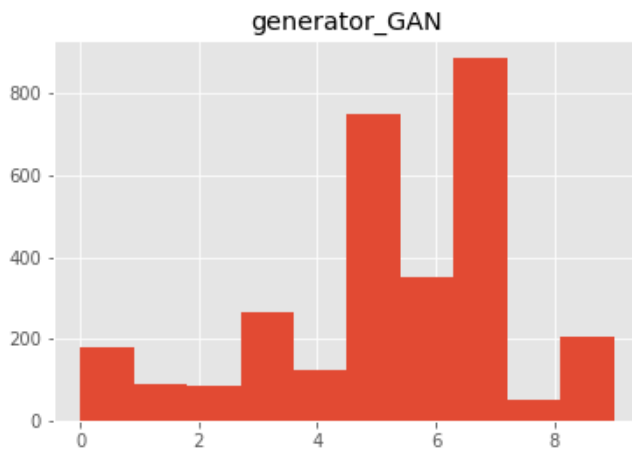
Take a copy of your vanilla GAN discriminator and change its output channel from 1 output to 10 output units. Fine-tune it as a classifier on the Fashion-MNIST training set. You should easily achieve $\sim 90\%$ accuracy on Fashion-MNIST test set.

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

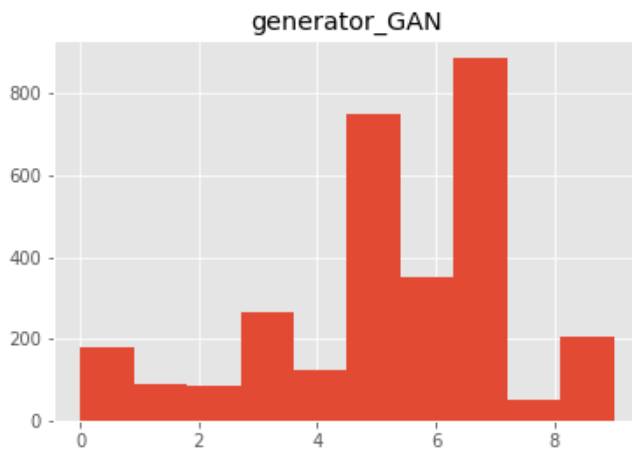
updated during that part of training. Train an unrolled GAN.

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.

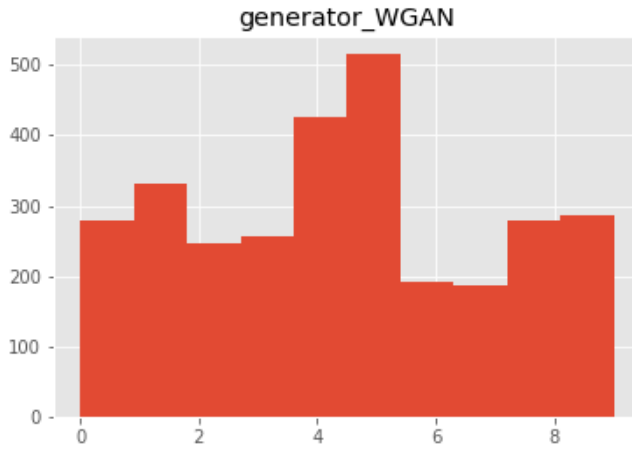
Like unrolled GAN, WGAN is claimed to be less affected by mode collapse. Discuss how each of the three generators performs and which seems to be best at reducing the mode collapse problem.

Solution:

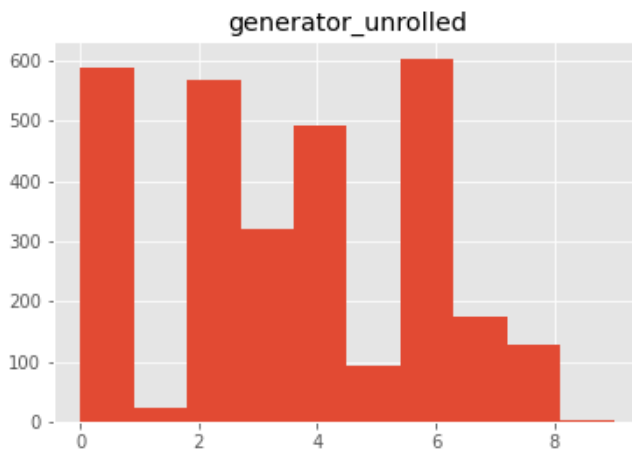
GAN histogram:



Wasserstien GAN histogram:



Unrolled GAN histogram:



The training speed of vanilla gan was the fastest 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.

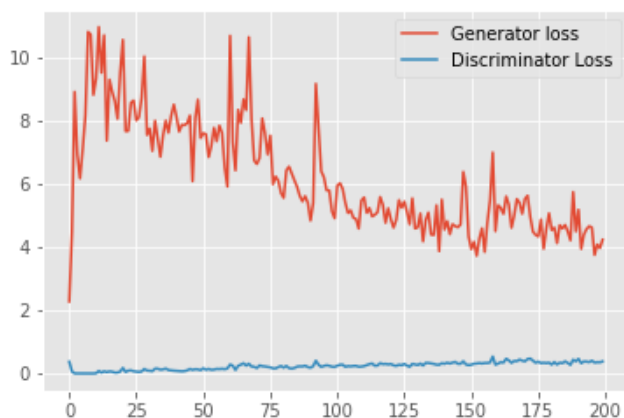
To add the conditional input vector, we need to modify both D and G . First, we need to define the input label vector. We are going to use one-hot encoding vectors for labels: for an image sample with label k of K classes, the vector is K dimensional and has 1 at k -th element and 0 otherwise.

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.

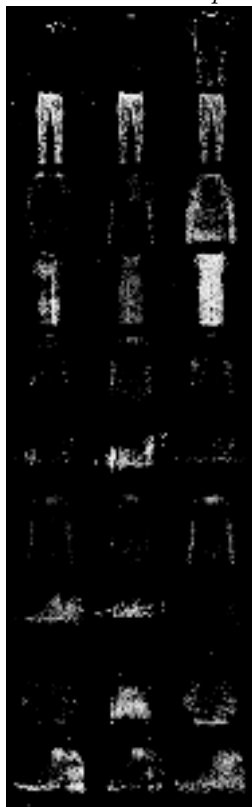
Solution:

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

```
1 # To add a new cell, type '# %%'
2 # To add a new markdown cell, type '# %% [markdown]'
3 # # %%
4 import numpy as np
5 import pandas as pd
6 import torch
7 import re
8 import time
9 from torch.utils.data import DataLoader, TensorDataset
10 from torch.nn import Module, GRU, Embedding, Linear, Sigmoid,
    CrossEntropyLoss
11 # # Part 1
12
13 # from google.colab import drive
14 # drive.mount('/content/drive')
15
16
17 "data/sentiment_analysis/train_pos_merged.txt"
18 # function clearing HTML tags from text
19 def cleanhtml(raw_html):
20     cleanr = re.compile('<.*?>')
21     cleantext = re.sub(cleanr, '', raw_html)
22     return cleantext
23
24 # preprocessing
25 def clean_text(path):
26     reviews = []
27     all_words = []
28     with open(path) as pos:
29         lines = pos.readlines()
30         for line in lines:
31             #clear html tags
32             line = cleanhtml(line)
33             # lower case and punctuation
34             line = re.sub(r'[^a-zA-Z]', ' ', line.lower())
35             # split to list of words
36             words = line.split()
37             # add list to reviews
38             reviews.append(words)
39             # extend words with new review
40             all_words.extend(words)
41
42     return reviews, all_words
43
44 def create_vocab(words):
45     # create vocabulary with indexes
```

```

46     vocab = {}
47     id = 1
48     for word in words:
49         if word not in vocab.keys():
50             vocab[word] = id
51             id += 1
52     return vocab
53
54
55 def vectorize_data(reviews, y, vocab, LENGTH=400):
56     y = np.array([y for _ in range(len(reviews))])
57     indexed_reviews = np.zeros((len(reviews), LENGTH), dtype = np.int64)
58     for i, review in enumerate(reviews):
59         indexed_review = []
60         for word in review:
61             indexed_review.append(vocab[word])
62         indexed_reviews[i, max(LENGTH-len(review),0):] = indexed_review[:400]
63     return indexed_reviews, y
64
65 def preprocessing(path1, path2, y1, y2, vocab, LENGTH=400):
66     reviews1, words1 = clean_text(path1)
67     reviews2, words2 = clean_text(path2)
68     # words1.extend(words2)
69     # print(words1)
70
71     del words1, words2
72
73     # vocab = create_vocab(words1)
74
75     x1, y1 = vectorize_data(reviews1, y1, vocab, LENGTH)
76     x2, y2 = vectorize_data(reviews2, y2, vocab, LENGTH)
77
78     x = np.concatenate((x1, x2))
79     y = np.concatenate((y1, y2))
80
81     return x, y #, vocab
82
83
84
85
86
87
88 # !ls /content/drive
89 # path = "/content/drive/MyDrive/Deep_Learning/sentiment_analysis/"
90 path = "./data/sentiment_analysis/"
91 reviews, words = clean_text(path + "all_merged.txt")
92
93
94 vocab = create_vocab(words)

```



```

95 # vocab
96
97
98 train_x, train_y = preprocessing(path + "train_pos_merged.txt", path + "
    train_neg_merged.txt", 0, 1, vocab)
99
100
101 # input = torch.from_numpy(train_x[0])
102 # embedding = Embedding(len(vocab), 3, padding_idx=0)
103 # embedding(input)
104
105
106 batch_size = 100
107 train_data = TensorDataset(torch.from_numpy(train_x), torch.from_numpy(
    train_y))
108 train_loader = DataLoader(train_data)
109
110 # # Part 2
111
112 class GRU_model(Module):
113
114     def __init__(self, vocab_size, input_dim, hidden_dim, n_layers=1,
        LENGTH=400):
115
116         super(GRU_model, self).__init__()
117
118         self.input_dim = input_dim
119         self.n_layers = n_layers
120         self.hidden_dim = hidden_dim
121
122         self.embedding = Embedding(vocab_size, input_dim, padding_idx=0)
123         self.gru = GRU(input_dim, hidden_dim, n_layers, batch_first=True)
124         self.linear = Linear(hidden_dim, 2)
125         self.sigmoid = Sigmoid()
126
127     def forward(self, x, h):
128         x = self.embedding(x)
129         x, h = self.gru(x, h)
130         # print(f"shape of x: {x.shape}; shape of h: {h.shape}; shape of x
        [:,-1]: {x[:,-1].shape}")
131         x = self.linear(x[:,-1])
132         # print(f"shape of x: {x.shape}")
133         x = self.sigmoid(x)
134         return x, h
135
136     def init_hidden(self, batch_size):
137         weight = next(self.parameters()).data
138         hidden = weight.new(self.n_layers, batch_size, self.hidden_dim).
        zero_().to(device)
139         return hidden

```

```

140
141
142 # torch.cuda.is_available() checks and returns a Boolean True if a GPU is
    available, else it'll return False
143 is_cuda = torch.cuda.is_available()
144
145 # If we have a GPU available, we'll set our device to GPU. We'll use this
    device variable later in our code.
146 if is_cuda:
147     device = torch.device("cuda")
148 else:
149     device = torch.device("cpu")
150
151
152 def gru_train(train_loader, vocab_size, learn_rate, input_dim=10,
    hidden_dim=16, EPOCHS=5):
153
154     # Setting common hyperparameters
155     # input_dim = next(iter(train_loader))[0].shape[1]
156     # print(next(iter(train_loader))[0].shape[1])
157     output_dim = 1
158     n_layers = 1
159     # Instantiating the model
160     model = GRU_model(vocab_size, input_dim, hidden_dim, output_dim,
    n_layers)
161     model.to(device)
162
163     # Defining loss function and optimizer
164     criterion = CrossEntropyLoss()
165     optimizer = torch.optim.Adam(model.parameters(), lr=learn_rate)
166
167     model.train()
168     print("Starting Training")
169     epoch_times = []
170     # Start training loop
171     for epoch in range(1, EPOCHS+1):
172         start_time = time.time()
173         h = model.init_hidden(batch_size)
174         avg_loss = 0.
175         counter = 0
176         for x, label in train_loader:
177             counter += 1
178             h = h.data
179             model.zero_grad()
180
181             out, h = model(x.to(device), h)
182             # print(f"shape of out.squeeze(): {out.squeeze().shape}; shape
    of label: {label.shape}")
183             loss = criterion(out.squeeze(), label.to(device))
184             loss.backward()

```

```

185         optimizer.step()
186         avg_loss += loss.item()
187         if counter%100 == 0:
188             print("Epoch {}.....Step: {}/{}..... Average Loss for
Epoch: {}".format(epoch, counter, len(train_loader), avg_loss/counter))
189             current_time = time.time()
190             print("Epoch {}/{} Done, Total Loss: {}".format(epoch, EPOCHS,
avg_loss/len(train_loader)))
191             print("Total Time Elapsed: {} seconds".format(str(current_time-
start_time)))
192             epoch_times.append(current_time-start_time)
193             print("Total Training Time: {} seconds".format(str(sum(epoch_times))))
194             return model
195
196
197 def gru_evaluate(model, test_loader): #, label_scalers):
198     model.eval()
199     outputs = []
200     results = []
201     start_time = time.time()
202     model.eval()
203     err = 0
204     for x, label in test_loader:
205         h = model.init_hidden(test_loader.batch_size).data
206         input = x.to(device)
207         output, h_out = model(input, h)
208         result = torch.argmax(output, dim=1)
209         results.append(result)
210         err += torch.abs(result.to(device) - label.to(device)).sum()
211     accuracy = 1 - err/len(test_loader)
212     return accuracy, outputs, results
213
214
215 gru_model = gru_train(
216     train_loader,
217     vocab_size = len(vocab),
218     learn_rate=0.0005,
219     hidden_dim=32,
220     EPOCHS=300
221 )
222
223
224 test_x, test_y = preprocessing(path + "test_pos_merged.txt", path + "
test_neg_merged.txt", 0, 1, vocab)
225
226
227 batch_size = 100
228 test_data = TensorDataset(torch.from_numpy(test_x), torch.from_numpy(
test_y))
229 test_loader = DataLoader(test_data) #, shuffle=True, batch_size=batch_size)

```

```

230
231
232
233 gru_model = GRU_model(len(vocab), 10, 32, 1, 1)
234 gru_model.load_state_dict(torch.load("./models/gru/gru_rnn.pt",
    map_location=device))
235
236
237 test_accuracy, test_outputs, test_results = gru_evaluate(gru_model,
    test_loader)
238 train_accuracy, train_outputs, train_results = gru_evaluate(gru_model,
    train_loader)
239
240
241 train_accuracy
242
243 # ## **Part 3** MLP
244
245 class MLP_model(Module):
246
247     def __init__(self, vocab_size, input_dim, hidden_dim, LENGTH = 400):
248
249         super(MLP_model, self).__init__()
250
251         self.embedding = Embedding(vocab_size, input_dim, padding_idx=0)
252         self.fc1 = Linear(input_dim*LENGTH, hidden_dim)
253         self.fc2 = Linear(hidden_dim, 2)
254         self.sigmoid = Sigmoid()
255
256     def forward(self, x):
257         x = self.embedding(x)
258         x = x.flatten(start_dim=1)
259         x = self.fc1(x)
260         x = torch.relu(x)
261         x = self.fc2(x)
262         x = self.sigmoid(x)
263         return x
264
265
266 def mlp_train(train_loader, vocab_size, learn_rate, input_dim=10,
    hidden_dim=16, EPOCHS=5):
267
268     # Instantiating the model
269     model = MLP_model(vocab_size, input_dim, hidden_dim)
270     # print(model)
271     model.to(device)
272
273     # Defining loss function and optimizer
274     criterion = CrossEntropyLoss()
275     optimizer = torch.optim.Adam(model.parameters(), lr=learn_rate)

```

```

276 model.train()
277 print("Starting Training")
278 epoch_times = []
279 # Start training loop
280 for epoch in range(1,EPOCHS+1):
281     start_time = time.time()
282     avg_loss = 0.
283     counter = 0
284     for x, label in train_loader:
285         counter += 1
286         model.zero_grad()
287
288         out = model(x.to(device))
289         # print(out.shape)
290         # print(f"shape of out: {out.shape}; shape of label: {label.
291 shape}")
292         # return 0
293         loss = criterion(out, label.to(device))
294         loss.backward()
295         optimizer.step()
296         avg_loss += loss.item()
297         if counter%100 == 0:
298             print("Epoch {}.....Step: {}/{ }..... Average Loss for
299 Epoch: {}".format(epoch, counter, len(train_loader), avg_loss/counter))
300             current_time = time.time()
301             print("Epoch {}/{ } Done, Total Loss: {}".format(epoch, EPOCHS,
302 avg_loss/len(train_loader)))
303             # print("Total Time Elapsed: { } seconds".format(str(current_time-
304 start_time)))
305             epoch_times.append(current_time-start_time)
306         print("Total Training Time: { } seconds".format(str(sum(epoch_times))))
307         return model
308
309 def mlp_evaluate(model, test_loader):
310     outputs = []
311     results = []
312     start_time = time.time()
313     model.eval()
314     err = 0
315     for x, label in test_loader:
316         input = x.to(device)
317         output = model(input)
318         result = torch.argmax(output, dim=1)
319         results.append(result)
320         outputs.append(output)
321         err += torch.abs(result.to(device) - label.to(device)).sum()
322     accuracy = 1 - err/len(test_loader)
323     return accuracy, outputs, results

```

```

322
323
324 mlp_model = mlp_train(train_loader, len(vocab), 0.01, hidden_dim=100 ,
    EPOCHS=20)
325
326
327 train_accuracy, train_outputs, train_results = mlp_evaluate(mlp_model,
    train_loader)
328 test_accuracy, train_outputs, train_results = mlp_evaluate(mlp_model,
    test_loader)
329
330
331 print(train_accuracy)
332 print(test_accuracy)

```

Listing 1: Problem 1

Problem 2 vanilla

```

1 import numpy as np
2 import pandas as pd
3 import torch
4 import torchvision as tv
5 import re
6 import time
7 from torch.utils.data import DataLoader, TensorDataset
8 from torch.nn import Module, GRU, Embedding, Linear, Sigmoid,
    CrossEntropyLoss, ReLU, Tanh, Sequential
9 from torch import nn
10 from torchvision import transforms
11 import torch.optim as optim
12 from torchvision.utils import make_grid, save_image
13 from tqdm import tqdm
14 import torch
15 import torch.nn as nn
16 import torchvision.transforms as transforms
17 import torch.optim as optim
18 import torchvision.datasets as datasets
19 import imageio
20 import numpy as np
21 import matplotlib
22 from torchvision.utils import make_grid, save_image
23 from torch.utils.data import DataLoader
24 from matplotlib import pyplot as plt
25 from tqdm import tqdm
26 matplotlib.style.use('ggplot')
27
28 # learning parameters
29 batch_size = 512
30 epochs = 200

```

```

31 sample_size = 64 # fixed sample size
32 nz = 128 # latent vector size
33 k = 1 # number of steps to apply to the discriminator
34 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
35
36 transform = transforms.Compose([
37     transforms.ToTensor(),
38     transforms.Normalize((0.5,),(0.5,)),
39 ])
40 to_pil_image = transforms.ToPILImage()
41
42 fmnist = datasets.FashionMNIST(root='.', train=True, download=True,
43     transform=transform)
44 data_loader = DataLoader(fmnist, batch_size=batch_size, shuffle=True)
45
46 class Generator(Module):
47     def __init__(self, nz):
48         super(Generator, self).__init__()
49         self.nz = nz
50         self.main = Sequential(
51             Linear(self.nz, 256),
52             ReLU(),
53             Linear(256, 512),
54             ReLU(),
55             Linear(512, 784),
56             Tanh(),
57         )
58
59     def forward(self, x):
60         return self.main(x).view(-1, 1, 28, 28)
61
62 class Discriminator(Module):
63     def __init__(self):
64         super(Discriminator, self).__init__()
65         self.n_input = 784
66         self.main = Sequential(
67             Linear(self.n_input, 1024),
68             ReLU(),
69             Linear(1024, 512),
70             ReLU(),
71             Linear(512, 1),
72             nn.Sigmoid(),
73         )
74
75     def forward(self, x):
76         x = x.view(-1, 784)
77         return self.main(x)
78
79

```

```

80
81
82 generator = Generator(nz).to(device)
83 discriminator = Discriminator().to(device)
84 print('##### GENERATOR #####')
85 print(generator)
86 print('#####')
87 print('\n##### DISCRIMINATOR #####')
88 print(discriminator)
89 print('#####')
90
91
92 # optimizers
93
94 optim_g = optim.Adam(generator.parameters(), lr=0.0002)
95 optim_d = optim.Adam(discriminator.parameters(), lr=0.0002)
96
97 # loss function
98 criterion = nn.BCELoss()
99
100
101 # to create real labels (1s)
102 def label_real(size):
103     data = torch.ones(size, 1)
104     return data.to(device)
105 # to create fake labels (0s)
106 def label_fake(size):
107     data = torch.zeros(size, 1)
108     return data.to(device)
109
110 def create_noise(sample_size, nz):
111     return torch.randn(sample_size, nz).to(device)
112
113 # to save the images generated by the generator
114 def save_generator_image(image, path):
115     save_image(image, path)
116
117 # function to train the discriminator network
118 def train_discriminator(optimizer, data_real, data_fake):
119     b_size = data_real.size(0)
120     real_label = label_real(b_size)
121     fake_label = label_fake(b_size)
122     optimizer.zero_grad()
123     output_real = discriminator(data_real)
124     loss_real = criterion(output_real, real_label)
125     output_fake = discriminator(data_fake)
126     loss_fake = criterion(output_fake, fake_label)
127     loss_real.backward()
128     loss_fake.backward()
129     optimizer.step()

```



```

130     return loss_real + loss_fake
131
132 # function to train the generator network
133 def train_generator(optimizer, data_fake):
134     b_size = data_fake.size(0)
135     real_label = label_real(b_size)
136     optimizer.zero_grad()
137     output = discriminator(data_fake)
138     loss = criterion(output, real_label)
139     loss.backward()
140     optimizer.step()
141     return loss
142
143 # create the noise vector
144 noise = create_noise(sample_size, nz)
145 generator.train()
146 discriminator.train()
147
148
149 # path = "/content/drive/MyDrive/Deep_Learning/HW3/"
150 path = "./models/vanilla_gan/"
151 epochs = 200
152 # k = 10
153 length = 0.
154
155 losses_g = [] # to store generator loss after each epoch
156 losses_d = [] # to store discriminator loss after each epoch
157 images = [] # to store images generated by the generator
158
159 for epoch in range(epochs):
160     start = time.time()
161     loss_g = 0.0
162     loss_d = 0.0
163     for bi, data in enumerate(data_loader):
164         image, _ = data
165         image = image.to(device)
166         b_size = len(image)
167         # run the discriminator for k number of steps
168         for step in range(k):
169             # print(create_noise(b_size, nz).shape)
170             data_fake = generator(create_noise(b_size, nz)).detach()
171             data_real = image
172             # train the discriminator network
173             loss_d += train_discriminator(optim_d, data_real, data_fake)
174             data_fake = generator(create_noise(b_size, nz))
175             # train the generator network
176             loss_g += train_generator(optim_g, data_fake)
177         # create the final fake image for the epoch
178         generated_img = generator(noise).cpu().detach()
179         # make the images as grid

```

```

180     generated_img = make_grid(generated_img)
181     # save the generated torch tensor models to disk
182     save_generator_image(generated_img, path + f"gen_img{epoch}.png")
183     images.append(generated_img)
184     epoch_loss_g = loss_g / bi # total generator loss for the epoch
185     epoch_loss_d = loss_d / bi # total discriminator loss for the epoch
186     losses_g.append(epoch_loss_g)
187     losses_d.append(epoch_loss_d)
188     end = time.time() - start
189     length += end
190     mean_so_far = length / (epoch+1)
191     time_left = (mean_so_far * (epochs - epoch - 1))/60
192
193     print(f"Epoch {epoch} of {epochs}:\t\t{end:.2f} seconds;\tttotal: {
length:.2f};\tminutes left: {time_left:.2f}")
194     print(f"Generator loss: {epoch_loss_g:.8f}, Discriminator loss: {
epoch_loss_d:.8f}")

```

Listing 2: Problem 2 vanilla

Problem 2 loss

```

1  from my_functions import *
2  import numpy as np
3  import pandas as pd
4  import torch
5  from torch._C import device
6  import torchvision as tv
7  import re
8  import time
9  from torch.utils.data import DataLoader, TensorDataset
10 from torch.nn import Module, GRU, Embedding, Linear, Sigmoid,
    CrossEntropyLoss, ReLU, Tanh, Sequential
11 from torch import nn
12 from torchvision import transforms
13 import torch.optim as optim
14 from torchvision.utils import make_grid, save_image
15 from tqdm import tqdm
16 import torch
17 import torch.nn as nn
18 import torchvision.transforms as transforms
19 import torch.optim as optim
20 import torchvision.datasets as datasets
21 import imageio
22 import numpy as np
23 import matplotlib
24 from torchvision.utils import make_grid, save_image
25 from torch.utils.data import DataLoader
26 from matplotlib import pyplot as plt
27 from tqdm import tqdm

```

```

28 matplotlib.style.use('ggplot')
29
30
31 # learning parameters
32 batch_size = 512
33 epochs = 200
34 sample_size = 64 # fixed sample size
35 nz = 128 # latent vector size
36 k = 1 # number of steps to apply to the discriminator
37 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
38
39 transform = transforms.Compose([
40     transforms.ToTensor(),
41     transforms.Normalize((0.5,),(0.5,)),
42 ])
43 to_pil_image = transforms.ToPILImage()
44
45 fmnist = datasets.FashionMNIST(root='./', train=True, download=True,
46     transform=transform)
47 data_loader = DataLoader(fmnist, batch_size=batch_size, shuffle=True)
48 ##### MSE #####
49
50 class Generator(Module):
51     def __init__(self, nz):
52         super(Generator, self).__init__()
53         self.nz = nz
54         self.main = Sequential(
55             Linear(self.nz, 256),
56             ReLU(),
57
58             Linear(256, 512),
59             ReLU(),
60
61             Linear(512, 784),
62             Tanh(),
63         )
64
65     def forward(self, x):
66         return self.main(x).view(-1, 1, 28, 28)
67
68
69 nz = 128 # latent vector size
70 device = 'cude' if torch.cuda.is_available() else 'cpu'
71
72 generator = Generator(nz).to(device)
73 print('##### GENERATOR #####')
74 print(generator)
75 print('#####')
76

```

```

77 optim_g = optim.Adam(generator.parameters(), lr=0.0002)
78
79 criterion = nn.MSELoss()
80
81 losses_g = [] # to store generator loss after each epoch
82 images = [] # to store images generated by the generator
83
84 # function to train the generator network
85 def train_generator(optimizer, data_fake, data_real):
86
87     optimizer.zero_grad()
88     loss = criterion(data_fake, data_real)
89     loss.backward()
90     optimizer.step()
91     return loss
92
93 noise = create_noise(128, nz)
94
95 # path = "/content/drive/MyDrive/Deep_Learning/HW3/"
96 path = "./models/MSE/"
97 epochs = 200
98 # k = 10
99 length = 0.
100
101 for epoch in range(epochs):
102     start = time.time()
103     loss_g = 0.0
104     # loss_d = 0.0
105     # for bi, data in tqdm(enumerate(data_loader), total=int(len(fmnist)/
106     data_loader.batch_size)):
107         for bi, data in enumerate(data_loader):
108             image, _ = data
109             image = image.to(device)
110             b_size = len(image)
111             data_fake = generator(create_noise(b_size, nz))
112             # train the generator network
113             loss_g += train_generator(optim_g, data_fake, image)
114         # create the final fake image for the epoch
115         generated_img = generator(noise).cpu().detach()
116         # make the images as grid
117         generated_img = make_grid(generated_img)
118         # save the generated torch tensor models to disk
119         save_generator_image(generated_img, path + f"gen_img{epoch}.png")
120         images.append(generated_img)
121         epoch_loss_g = loss_g / bi # total generator loss for the epoch
122         # epoch_loss_d = loss_d / bi # total discriminator loss for the epoch
123         losses_g.append(epoch_loss_g)
124         # losses_d.append(epoch_loss_d)
125         prev = -9999
126         # print(f"")

```

```

126     end = time.time() - start
127     length += end
128     mean_so_far = length / (epoch+1)
129     time_left = (mean_so_far * (epochs - epoch - 1))/60
130
131     print(f"Epoch {epoch} of {epochs}:\t\t{end:.2f} seconds;\ttotal: {
length:.2f};\tminutes left: {time_left:.2f}")
132     print(f"Generator loss: {epoch_loss_g:.8f}")
133     prev = epoch_loss_g
134
135     ##### WGAN #####
136
137     class Generator(Module):
138         def __init__(self, nz):
139             super(Generator, self).__init__()
140             self.nz = nz
141             self.main = Sequential(
142                 Linear(self.nz, 256),
143                 ReLU(),
144
145                 Linear(256, 512),
146                 ReLU(),
147
148                 Linear(512, 784),
149                 Tanh(),
150             )
151
152         def forward(self, x):
153             return self.main(x).view(-1, 1, 28, 28)
154
155     class Discriminator(Module):
156         def __init__(self):
157             super(Discriminator, self).__init__()
158             self.n_input = 784
159             self.main = Sequential(
160                 Linear(self.n_input, 1024),
161                 ReLU(),
162                 Linear(1024, 512),
163                 ReLU(),
164                 Linear(512, 1),
165                 # nn.Sigmoid(),
166             )
167         def forward(self, x):
168             x = x.view(-1, 784)
169             return self.main(x)
170
171
172
173     generator = Generator(nz).to(device)
174     discriminator = Discriminator().to(device)

```

```

175 print('#### GENERATOR ####')
176 print(generator)
177 print('#####')
178 print('\n#### DISCRIMINATOR ####')
179 print(discriminator)
180 print('#####')
181
182 optim_g = optim.RMSprop(generator.parameters(), lr=0.0001)
183 optim_d = optim.RMSprop(discriminator.parameters(), lr=0.0001)
184
185 # function to train the discriminator network
186 def train_discriminator(optimizer, data_real, data_fake):
187     # b_size = data_real.size(0)
188     # real_label = label_real(b_size)
189     # fake_label = label_fake(b_size)
190     optimizer.zero_grad()
191     output_real = discriminator(data_real)
192     # loss_real = criterion(output_real, real_label)
193     output_fake = discriminator(data_fake)
194     loss = -(torch.mean(output_real) - torch.mean(output_fake))
195     # loss_fake = criterion(output_fake, fake_label)
196     loss.backward()
197     optimizer.step()
198     return loss
199
200 # function to train the generator network
201 def train_generator(optimizer, data_fake):
202     # b_size = data_fake.size(0)
203     # real_label = label_real(b_size)
204     optimizer.zero_grad()
205     output = discriminator(data_fake)
206     loss = -torch.mean(output)
207     loss.backward()
208     optimizer.step()
209     return loss
210
211 # create the noise vector
212 noise = create_noise(sample_size, nz)
213 generator.train()
214 discriminator.train()
215
216 losses_g = [] # to store generator loss after each epoch
217 losses_d = [] # to store discriminator loss after each epoch
218 images = [] # to store images generated by the generator
219
220 path = "/content/drive/MyDrive/Deep_Learning/HW3/outputs_wasserstein00001/"
221 # path = ""
222 epochs = 200
223 c = 0.0001
224 # k = 10

```

```

225
226 for epoch in range(epochs):
227     loss_g = 0.0
228     loss_d = 0.0
229     # for bi, data in tqdm(enumerate(data_loader), total=int(len(fmnist)/
data_loader.batch_size)):
230     for bi, data in enumerate(data_loader):
231         image, _ = data
232         image = image.to(device)
233         b_size = len(image)
234         # run the discriminator for k number of steps
235         for step in range(k):
236             # print(create_noise(b_size, nz).shape)
237             data_fake = generator(create_noise(b_size, nz)).detach()
238             data_real = image
239             # train the discriminator network
240             loss_d += train_discriminator(optim_d, data_real, data_fake)
241             data_fake = generator(create_noise(b_size, nz))
242             # train the generator network
243             loss_g += train_generator(optim_g, data_fake)
244         # create the final fake image for the epoch
245         noise = create_noise(b_size, nz)
246         generated_img = generator(noise).cpu().detach()
247         # make the images as grid
248         generated_img = make_grid(generated_img)
249         # save the generated torch tensor models to disk
250         save_generator_image(generated_img, path + f"gen_img{epoch}.png")
251         images.append(generated_img)
252         epoch_loss_g = loss_g / bi # total generator loss for the epoch
253         epoch_loss_d = loss_d / bi # total discriminator loss for the epoch
254         losses_g.append(epoch_loss_g)
255         losses_d.append(epoch_loss_d)
256
257     for p in discriminator.parameters():
258         p.data.clamp_(-c, c)
259
260     print(f"Epoch {epoch} of {epochs}")
261     print(f"Generator loss: {epoch_loss_g:.8f}, Discriminator loss: {
epoch_loss_d:.8f}")
262
263
264
265 ##### LSE #####
266
267 class Generator(Module):
268     def __init__(self, nz):
269         super(Generator, self).__init__()
270         self.nz = nz
271         self.main = Sequential(
272             Linear(self.nz, 256),

```

```

273         ReLU(),
274         Linear(256, 512),
275         ReLU(),
276         Linear(512, 784),
277         Tanh(),
278     )
279
280
281
282     def forward(self, x):
283         return self.main(x).view(-1, 1, 28, 28)
284
285 class Discriminator(Module):
286     def __init__(self):
287         super(Discriminator, self).__init__()
288         self.n_input = 784
289         self.main = Sequential(
290             Linear(self.n_input, 1024),
291             ReLU(),
292             Linear(1024, 512),
293             ReLU(),
294             Linear(512, 1),
295             nn.Sigmoid(),
296         )
297     def forward(self, x):
298         x = x.view(-1, 784)
299         return self.main(x)
300
301
302
303 generator = Generator(nz).to(device)
304 discriminator = Discriminator().to(device)
305 print('##### GENERATOR #####')
306 print(generator)
307 print('#####')
308 print('\n##### DISCRIMINATOR #####')
309 print(discriminator)
310 print('#####')
311
312 optim_g = optim.Adam(generator.parameters(), lr=0.0002)
313 optim_d = optim.Adam(discriminator.parameters(), lr=0.0002)
314
315 # function to train the discriminator network
316 def train_discriminator(optimizer, data_real, data_fake):
317     optimizer.zero_grad()
318     output_real = discriminator(data_real)
319     output_fake = discriminator(data_fake)
320     loss = torch.mean((output_real-1)**2) + torch.mean(output_fake**2)
321     loss.backward()
322     optimizer.step()

```



```

323     return loss
324
325 # function to train the generator network
326 def train_generator(optimizer, data_fake):
327     optimizer.zero_grad()
328     output = discriminator(data_fake)
329     loss = torch.mean((output-1)**2)
330     loss.backward()
331     optimizer.step()
332     return loss
333
334
335 # create the noise vector
336 noise = create_noise(sample_size, nz)
337 generator.train()
338 discriminator.train()
339
340
341 losses_g = [] # to store generator loss after each epoch
342 losses_d = [] # to store discriminator loss after each epoch
343 images = [] # to store images generatd by the generator
344
345 path = "/content/drive/MyDrive/Deep_Learning/HW3/outputs_LSE/"
346 # path = ""
347 epochs = 800
348 # c = 0.01
349 # k = 10
350
351 for epoch in range(epochs):
352     loss_g = 0.0
353     loss_d = 0.0
354     for bi, data in enumerate(data_loader):
355         image, _ = data
356         image = image.to(device)
357         b_size = len(image)
358         # run the discriminator for k number of steps
359         for step in range(k):
360             # print(create_noise(b_size, nz).shape)
361             data_fake = generator(create_noise(b_size, nz)).detach()
362             data_real = image
363             # train the discriminator network
364             loss_d += train_discriminator(optim_d, data_real, data_fake)
365             data_fake = generator(create_noise(b_size, nz))
366             # train the generator network
367             loss_g += train_generator(optim_g, data_fake)
368         # create the final fake image for the epoch
369         # noise = create_noise(b_size, nz)
370         generated_img = generator(noise).cpu().detach()
371         # make the images as grid
372         generated_img = make_grid(generated_img)

```

```

373     # save the generated torch tensor models to disk
374     save_generator_image(generated_img, path + f"gen_img{epoch}.png")
375     images.append(generated_img)
376     epoch_loss_g = loss_g / bi # total generator loss for the epoch
377     epoch_loss_d = loss_d / bi # total discriminator loss for the epoch
378     losses_g.append(epoch_loss_g)
379     losses_d.append(epoch_loss_d)
380
381     # for p in discriminator.parameters():
382     #     p.data.clamp_(-c, c)
383
384     print(f"Epoch {epoch} of {epochs}")
385     print(f"Generator loss: {epoch_loss_g:.8f}, Discriminator loss: {
epoch_loss_d:.8f}")

```

Listing 3: Problem 2 loss

Problem 2 mode collapse

```

1  from my_functions import *
2  import numpy as np
3  import pandas as pd
4  import torch
5  import torchvision as tv
6  import re
7  import time
8  from torch.utils.data import DataLoader, TensorDataset
9  from torch.nn import Module, GRU, Embedding, Linear, Sigmoid,
    CrossEntropyLoss, ReLU, Tanh, Sequential
10 from torch import nn
11 from torchvision import transforms
12 import torch.optim as optim
13 from torchvision.utils import make_grid, save_image
14 from tqdm import tqdm
15 import torch
16 import torch.nn as nn
17 import torchvision.transforms as transforms
18 import torch.optim as optim
19 import torchvision.datasets as datasets
20 import imageio
21 import numpy as np
22 import matplotlib
23 from torchvision.utils import make_grid, save_image
24 from torch.utils.data import DataLoader
25 from matplotlib import pyplot as plt
26 from tqdm import tqdm
27 matplotlib.style.use('ggplot')
28
29
30 # learning parameters

```

```

31 batch_size = 512
32 epochs = 200
33 sample_size = 64 # fixed sample size
34 nz = 128 # latent vector size
35 k = 1 # number of steps to apply to the discriminator
36 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
37
38 transform = transforms.Compose([
39     transforms.ToTensor(),
40     transforms.Normalize((0.5,),(0.5,)),
41 ])
42 to_pil_image = transforms.ToPILImage()
43
44 fmnist = datasets.FashionMNIST(root='.', train=True, download=True,
45     transform=transform)
46 data_loader = DataLoader(fmnist, batch_size=batch_size, shuffle=True)
47
48 import torch.nn as nn
49 import torch.nn.functional as F
50
51 class MLP(nn.Module):
52     def __init__(self):
53         super(MLP, self).__init__()
54         self.fc1 = nn.Linear(28*28, 500)
55         self.fc2 = nn.Linear(500, 256)
56         self.fc3 = nn.Linear(256, 10)
57
58     def forward(self, x):
59         x = x.view(-1, 28*28)
60         x = F.relu(self.fc1(x))
61         x = F.relu(self.fc2(x))
62         x = self.fc3(x)
63         return x
64
65 MLP_network = MLP()
66
67
68 discriminator = MLP()
69 discriminator.load_state_dict(torch.load("./outputs/fmnist_classifier.pth",
70     map_location=device))
71
72 optim_d = optim.Adam(discriminator.parameters(), lr=0.00005)
73
74 criterion = nn.CrossEntropyLoss()
75
76 losses = []
77 accuracies = []
78 epochs = 50

```

```

79
80 discriminator.train()
81
82 # function to train the discriminator network
83 def train_discriminator(optimizer, data, labels):
84     optimizer.zero_grad()
85     output = discriminator(data)
86     # print(output.shape)
87     loss = criterion(output, labels)
88     loss.backward()
89     optimizer.step()
90     return loss, output
91
92 for epoch in range(epochs):
93     total = 0
94     acc_loss = 0
95     correct = 0
96     for bi, (images, labels) in enumerate(data_loader):
97         loss, output = train_discriminator(optim_d, images, labels)
98         acc_loss += loss
99         b_size = len(labels)
100         total += b_size
101         predicted = torch.argmax(output, dim=1)
102         correct += (predicted==labels).sum()
103         accuracy = correct/total
104         avg_loss = loss/b_size
105         if bi%20==0:
106             print(f"Epoch {epoch}/{epochs}; Batch {bi}: Loss = {loss:.5f}\t
\tAccuracy = {accuracy:.5f}")
107
108
109     losses.append(acc_loss/total)
110     accuracies.append(accuracy)
111
112
113 # load model and set to evaluate mode
114
115 class Generator(Module):
116     def __init__(self, nz):
117         super(Generator, self).__init__()
118         self.nz = nz
119         self.main = Sequential(
120             Linear(self.nz, 256),
121             ReLU(),
122
123             Linear(256, 512),
124             ReLU(),
125
126             Linear(512, 784),
127             Tanh(),

```

```

128         )
129
130     def forward(self, x):
131         return self.main(x).view(-1, 1, 28, 28)
132
133
134 generator_GAN = Generator(nz)
135 generator_WGAN = Generator(nz)
136 generator_unrolled = Generator(nz)
137
138 generator_GAN.load_state_dict(torch.load("./models/vanilla_gan/generator.
139    .pth", map_location=torch.device('cpu')))
140 generator_WGAN.load_state_dict(torch.load("./models/wasserstein/generator.
141    .pth", map_location=torch.device('cpu')))
142 generator_unrolled.load_state_dict(torch.load("./models/unrolled/generator.
143     pt", map_location=torch.device('cpu')))
144 generator_GAN.eval()
145 generator_WGAN.eval()
146 generator_unrolled.eval()
147 # print(generator)
148
149 # generate 3,000 new images
150
151 sample_size = 3000
152
153 def create_noise(sample_size, nz):
154     return torch.randn(sample_size, nz).to(device)
155
156 nz = 128
157 # create noise
158
159 models = {'generator_GAN': generator_GAN, 'generator_WGAN': generator_WGAN,
160     'generator_unrolled': generator_unrolled}
161
162 for key, generator in models.items():
163
164     noise = create_noise(sample_size, nz)
165
166     # feed noise to generator
167     new_images = generator(noise)
168
169     # feed new images to discriminator
170     new_softmax = discriminator(new_images)
171     new_labels = torch.argmax(new_softmax, dim=1)
172
173     plt.figure()
174     plt.hist(new_labels.numpy(), bins=10)
175     plt.title(key)

```

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

Listing 4: Problem 2 mode collapse

Problem 2 unrolled

```
1 from my_functions import *
2 import numpy as np
3 import pandas as pd
4 import torch
5 import torchvision as tv
6 import re
7 import time
8 from torch.utils.data import DataLoader, TensorDataset
9 from torch.nn import Module, GRU, Embedding, Linear, Sigmoid,
   CrossEntropyLoss, ReLU, Tanh, Sequential
10 from torch import nn
11 from torchvision import transforms
12 import torch.optim as optim
13 from torchvision.utils import make_grid, save_image
14 from tqdm import tqdm
15 import torch
16 import torch.nn as nn
17 import torchvision.transforms as transforms
18 import torch.optim as optim
19 import torchvision.datasets as datasets
20 import imageio
21 import numpy as np
22 import matplotlib
23 from torchvision.utils import make_grid, save_image
24 from torch.utils.data import DataLoader
25 from matplotlib import pyplot as plt
26 from tqdm import tqdm
27 import copy
28 matplotlib.style.use('ggplot')
29
30 # learning parameters
31 batch_size = 32
32 epochs = 200
33 sample_size = 64 # fixed sample size
34 nz = 128 # latent vector size
35 k = 1 # number of steps to apply to the discriminator
36 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
37
38 transform = transforms.Compose([
39     transforms.ToTensor(),
40     transforms.Normalize((0.5,),(0.5,)),
41 ])
42 to_pil_image = transforms.ToPILImage()
43
```

```

44 fmnist = datasets.FashionMNIST(root='./', train=True, download=True,
    transform=transform)
45 data_loader = DataLoader(fmnist, batch_size=batch_size, shuffle=True)
46
47 class Generator(Module):
48     def __init__(self, nz):
49         super(Generator, self).__init__()
50         self.nz = nz
51         self.main = Sequential(
52             Linear(self.nz, 256),
53             ReLU(),
54
55             Linear(256, 512),
56             ReLU(),
57
58             Linear(512, 784),
59             Tanh(),
60         )
61
62     def forward(self, x):
63         return self.main(x).view(-1, 1, 28, 28)
64
65 class Discriminator(Module):
66     def __init__(self):
67         super(Discriminator, self).__init__()
68         self.n_input = 784
69         self.main = Sequential(
70             Linear(self.n_input, 1024),
71             ReLU(),
72             Linear(1024, 512),
73             ReLU(),
74             Linear(512, 1),
75             nn.Sigmoid(),
76         )
77
78     def forward(self, x):
79         x = x.view(-1, 784)
80         return self.main(x)
81
82     def load(self, backup):
83         for m_from, m_to in zip(backup.modules(), self.modules()):
84             if isinstance(m_to, nn.Linear):
85                 m_to.weight.data = m_from.weight.data.clone()
86                 if m_to.bias is not None:
87                     m_to.bias.data = m_from.bias.data.clone()
88
89 generator = Generator(nz).to(device)
90 discriminator = Discriminator().to(device)
91 print('#### GENERATOR ####')

```

```

93 print(generator)
94 print('#####')
95 print('\n#### DISCRIMINATOR ####')
96 print(discriminator)
97 print('#####')
98
99 # loss function
100 criterion = nn.BCELoss()
101
102 # optimizers
103 optim_g = optim.Adam(generator.parameters(), lr=0.0002)
104 optim_d = optim.Adam(discriminator.parameters(), lr=0.0002)
105
106
107 # function to train the discriminator network
108 def train_discriminator(optimizer, data_real, data_fake, discriminator,
109                        create_graph=False):
110     b_size = data_real.size(0)
111     real_label = label_real(b_size)
112     fake_label = label_fake(b_size)
113     optimizer.zero_grad()
114     output_real = discriminator(data_real)
115     loss_real = criterion(output_real, real_label)
116     output_fake = discriminator(data_fake)
117     loss_fake = criterion(output_fake, fake_label)
118     # loss_real.backward()
119     # loss_fake.backward()
120     loss = loss_fake + loss_real
121     loss.backward(create_graph=create_graph)
122     optimizer.step()
123     return loss.item()
124
125 # function to train the generator network
126 def train_generator(optimizer, data_fake, discriminator):
127     b_size = data_fake.size(0)
128     real_label = label_real(b_size)
129     optimizer.zero_grad()
130     output = discriminator(data_fake)
131     loss = criterion(output, real_label)
132     loss.backward()
133     optimizer.step()
134     return loss.item()
135
136 # create the noise vector
137 noise = create_noise(sample_size, nz)
138 generator.train()
139 discriminator.train()
140
141 # path = "/content/drive/MyDrive/Deep_Learning/HW3/outputs_unrolled_gan/"
142 path = "./models/unrolled/"

```



```

142 epochs = 200
143 k = 1
144
145 losses_g = [] # to store generator loss after each epoch
146 losses_d = [] # to store discriminator loss after each epoch
147 losses_ud = []
148 images = [] # to store images generated by the generator
149 length = 0.
150
151 for epoch in range(epochs):
152     start = time.time()
153     # length = 0.
154     loss_g = 0.0
155     loss_d = 0.0
156     loss_ud = 0.
157     for bi, data in enumerate(data_loader):
158         image, _ = data
159         image = image.to(device)
160         b_size = len(image)
161
162
163         # print(create_noise(b_size, nz).shape)
164         data_fake = generator(create_noise(b_size, nz)).detach()
165         data_real = image
166         # train the discriminator network
167         loss_d += train_discriminator(optim_d, data_real, data_fake,
discriminator)
168
169         # unroll
170         backup = copy.deepcopy(discriminator)
171         # run the unrolled discriminator for k number of steps
172         for step in range(k):
173             data_fake = generator(create_noise(b_size, nz)).detach()
174             loss_ud += train_discriminator(optim_d, data_real, data_fake,
discriminator, create_graph=True)
175
176         # data_fake = generator(create_noise(b_size, nz))
177         # train the generator network
178         data_fake = generator(create_noise(b_size, nz)).detach()
179         loss_g += train_generator(optim_g, data_fake, discriminator)
180         discriminator.load(backup)
181         del backup
182
183
184     # create the final fake image for the epoch
185     if epoch%1==0:
186         generated_img = generator(noise).cpu().detach()
187         # make the images as grid
188         generated_img = make_grid(generated_img)
189         # save the generated torch tensor models to disk

```

```

190     save_generator_image(generated_img, path + f"gen_img{epoch}.png")
191     images.append(generated_img)
192     epoch_loss_g = loss_g / bi # total generator loss for the epoch
193     epoch_loss_d = loss_d / bi # total discriminator loss for the epoch
194     epoch_loss_ud = loss_ud / (bi*k)
195     losses_g.append(epoch_loss_g)
196     losses_d.append(epoch_loss_d)
197     losses_ud.append(epoch_loss_ud)
198     end = time.time() - start
199     length += end
200     mean_so_far = length / (epoch+1)
201     time_left = (mean_so_far * (epochs - epoch - 1))/60
202
203     print(f"Epoch {epoch} of {epochs}:\t\t{end:.2f} seconds;\tttotal: {
length:.2f};\tminutes left: {time_left:.2f}")
204     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

```

1  from my_functions import *
2  import numpy as np
3  import pandas as pd
4  import torch
5  import torchvision as tv
6  import re
7  import time
8  from torch.utils.data import DataLoader, TensorDataset
9  from torch.nn import Module, GRU, Embedding, Linear, Sigmoid,
    CrossEntropyLoss, ReLU, Tanh, Sequential
10 from torch import nn
11 from torchvision import transforms
12 import torch.optim as optim
13 from torchvision.utils import make_grid, save_image
14 from tqdm import tqdm
15 import torch
16 import torch.nn as nn
17 import torchvision.transforms as transforms
18 import torch.optim as optim
19 import torchvision.datasets as datasets
20 import imageio
21 import numpy as np
22 import matplotlib
23 from torchvision.utils import make_grid, save_image
24 from torch.utils.data import DataLoader
25 from matplotlib import pyplot as plt
26 from tqdm import tqdm
27 matplotlib.style.use('ggplot')

```

```

28
29 # learning parameters
30 batch_size = 512
31 epochs = 200
32 sample_size = 100 # fixed sample size
33 nz = 128 # latent vector size
34 k = 1 # number of steps to apply to the discriminator
35 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
36
37 transform = transforms.Compose([
38     transforms.ToTensor(),
39     transforms.Normalize((0.5,),(0.5,)),
40 ])
41 to_pil_image = transforms.ToPILImage()
42
43 fmnist = datasets.FashionMNIST(root='.', train=True, download=True,
44     transform=transform)
45 data_loader = DataLoader(fmnist, batch_size=batch_size, shuffle=True)
46
47 class Generator(Module):
48     def __init__(self, nz):
49         super(Generator, self).__init__()
50         self.nz = nz
51         self.main = Sequential(
52             Linear(self.nz + 10, 256),
53             ReLU(),
54             Linear(256, 512),
55             ReLU(),
56             Linear(512, 784),
57             Tanh(),
58         )
59
60     def forward(self, x, k):
61         x = torch.cat((x, k), dim=1)
62         # print(x)
63         return self.main(x).view(-1, 1, 28, 28)
64
65 class Discriminator(Module):
66     def __init__(self):
67         super(Discriminator, self).__init__()
68         self.n_input = 784
69         self.main = Sequential(
70             Linear(self.n_input + 10, 1024),
71             ReLU(),
72             Linear(1024, 512),
73             ReLU(),
74             Linear(512, 1),
75             nn.Sigmoid()
76

```

```

77     )
78     def forward(self, x, k):
79         x = x.view(-1, 784)
80         x = torch.cat((x, k), dim=1)
81         return self.main(x)
82
83
84 generator = Generator(nz).to(device)
85 discriminator = Discriminator().to(device)
86 print('##### GENERATOR #####')
87 print(generator)
88 print('#####')
89 print('\n##### DISCRIMINATOR #####')
90 print(discriminator)
91 print('#####')
92
93 # optimizers
94
95 optim_g = optim.Adam(generator.parameters(), lr=0.0002)
96 optim_d = optim.Adam(discriminator.parameters(), lr=0.0002)
97
98 criterion = nn.BCELoss()
99
100 # function to train the discriminator network
101 def train_discriminator(optimizer, data_real, data_fake, onehot):
102     b_size = data_real.size(0)
103     real_label = label_real(b_size)
104     fake_label = label_fake(b_size)
105     optimizer.zero_grad()
106     output_real = discriminator(data_real, onehot)
107     loss_real = criterion(output_real, real_label)
108     output_fake = discriminator(data_fake, onehot)
109     loss_fake = criterion(output_fake, fake_label)
110     loss_real.backward()
111     loss_fake.backward()
112     optimizer.step()
113     return loss_real + loss_fake
114
115 # function to train the generator network
116 def train_generator(optimizer, data_fake, onehot):
117     b_size = data_fake.size(0)
118     real_label = label_real(b_size)
119     optimizer.zero_grad()
120     output = discriminator(data_fake, onehot)
121     loss = criterion(output, real_label)
122     loss.backward()
123     optimizer.step()
124     return loss
125
126 # create the noise vector

```

```

127 noise = create_noise(sample_size, nz)
128 onehot_out = np.array([[1 if j % 10 == i else 0 for i in range(10)] for j
    in range(sample_size)])
129 onehot_out = torch.from_numpy(onehot_out)
130 generator.train()
131 discriminator.train()
132
133 # path = "/content/drive/MyDrive/Deep_Learning/HW3/"
134 path = "./models/conditional/"
135 epochs = 200
136 # k = 10
137 length = 0.
138
139 losses_g = [] # to store generator loss after each epoch
140 losses_d = [] # to store discriminator loss after each epoch
141 images = [] # to store images generated by the generator
142
143 for epoch in range(epochs):
144     start = time.time()
145     loss_g = 0.0
146     loss_d = 0.0
147     for bi, data in enumerate(data_loader):
148         image, labels = data
149         image = image.to(device)
150         onehot = torch.zeros((labels.size(0), labels.max() + 1))
151         onehot[np.arange(labels.size(0)), labels] = 1
152
153         # print(image.shape, onehot.shape)
154         # print(torch.cat())
155
156         b_size = len(image)
157         # run the discriminator for k number of steps
158         for step in range(k):
159             # print(create_noise(b_size, nz).shape)
160             data_fake = generator(create_noise(b_size, nz), onehot).detach
161             ()
162             data_real = image
163             # train the discriminator network
164             loss_d += train_discriminator(optim_d, data_real, data_fake,
165             onehot)
166             data_fake = generator(create_noise(b_size, nz), onehot)
167             # train the generator network
168             loss_g += train_generator(optim_g, data_fake, onehot)
169             # create the final fake image for the epoch
170             generated_img = generator(noise, onehot_out).cpu().detach()
171             # make the images as grid
172             generated_img = make_grid(generated_img, nrow = 10)
173             # save the generated torch tensor models to disk
174             save_generator_image(generated_img, path + f"gen_img{epoch}.png")
175             images.append(generated_img)

```

```

174     epoch_loss_g = loss_g / bi # total generator loss for the epoch
175     epoch_loss_d = loss_d / bi # total discriminator loss for the epoch
176     losses_g.append(epoch_loss_g)
177     losses_d.append(epoch_loss_d)
178     end = time.time() - start
179     length += end
180     mean_so_far = length / (epoch+1)
181     time_left = (mean_so_far * (epochs - epoch - 1))/60
182
183     print(f"Epoch {epoch} of {epochs}:\t\t{end:.2f} seconds;\ttotal: {
length:.2f};\tminutes left: {time_left:.2f}")
184     print(f"Generator loss: {epoch_loss_g:.8f}, Discriminator loss: {
epoch_loss_d:.8f}")
185
186 n = 10
187
188 onehot = torch.zeros((n*3, n))
189
190 print(onehot)
191 onehot[np.arange(n*3), np.array([[i,i,i] for i in range(n)]).flatten()] = 1
192 print(onehot)
193
194
195 noise = create_noise(n*3, nz)
196
197 generated_img = generator(noise, onehot).cpu().detach()
198 # make the images as grid
199 generated_img = make_grid(generated_img, nrow = 3)
200 save_generator_image(generated_img, f"./models/conditional/
conditional_generated_{n}.png")

```

Listing 6: Problem 2 conditional