# Homework 3
# CS 5787 Deep Learning
# Spring 2020

**Irene Font Peradejordi**

**Due: See Canvas**

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

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

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

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

## Problem 0 - Recurrent Neural Networks (10 points)

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

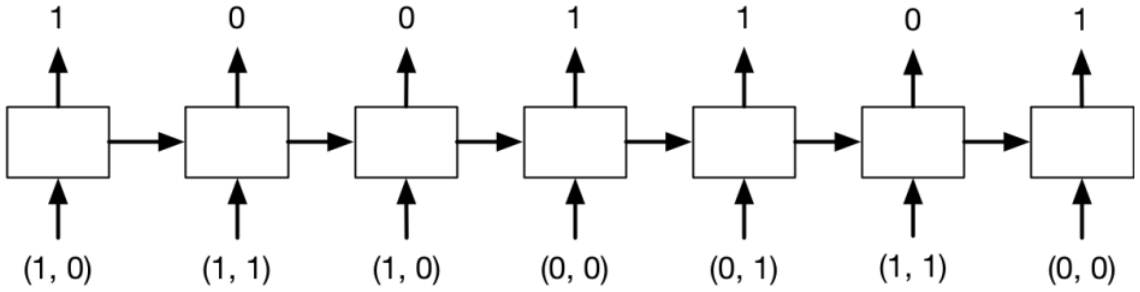
The input to your RNN will be two binary numbers, starting with the *least* significant bit. You will need to pad the largest number with an additional zero on the left side and you should make the other number the same length by padding it with zeros on the left side. For instance, the problem

$$100111 + 110010 = 1011001$$

would be input to your RNN as:

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

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



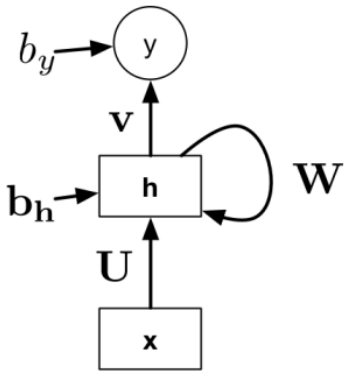

The RNN that implements binary addition has three hidden units, and all of the units use the following non-differentiable hard-threshold activation function

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

The equations for the network are given by

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

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




### Part 1 - Finding Weights

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

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

**Solution:**

$$v = \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} U = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} W = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} bh = \begin{bmatrix} 0 \\ -1 \\ -2 \end{bmatrix} by = 0$$

## Problem 1 - GRU for Sentiment Analysis

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

Put your code in the appendix.

## Part 1 - Preprocessing (5 points)

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

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

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

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

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

Implement the data preprocessing procedure.

**Solution:**

```
train_neg_ = open('data/train_neg_merged.txt', 'r')
train_pos_ = open('data/train_pos_merged.txt', 'r')

test_neg_ = open('data/test_neg_merged.txt', 'r')
test_pos_ = open('data/test_pos_merged.txt', 'r')

all_merged_ = open('data/all_merged.txt', 'r')

train_neg = []
train_pos = []

test_neg = []
test_pos = []

for trn, trp, tsn, tsp in zip(train_neg_.readlines(), train_pos_.readlines(), test_neg_.rea
    train_neg.append(trn)
    train_pos.append(trp)
    test_neg.append(tsn)
    test_pos.append(tsp)
```

```
### DICTIONARY CREATION

all_merged = []

for i in all_merged_.readlines():
    all_merged.append(i)

stop_words = stopwords.words('english')
porter = PorterStemmer()

def cleanhtml(raw_html):
    cleanr = re.compile('<.*?>')
    cleantext = re.sub(cleanr, ' ', raw_html)
    return cleantext

def processing (sentence):
    result = sentence.lower() #Lower case
    result = cleanhtml(result)
    result = re.sub(r'\d+', '', result) #Removing numbers
    result = result.translate(str.maketrans('', '', string.punctuation)) #Remove weird char
    result = result.strip() #Eliminate blanks from begining and end of setences
    result = result.split() #Separate into words
    result = [w for w in result if not w in stop_words] #Eliminate stop_words
    result = [porter.stem(word) for word in result] #Stem Words
    return (result)

vocab = {}

index = 0
for i in range(len(all_merged)):
    word_list = processing(all_merged[i])
    for word in word_list:
        if word not in vocab:
            vocab[word] = index
            index += 1

### VOCAB LEN == 34654

---
```

```
### MATRIX CREATION

def create_mat(data, vocab):
    mat = np.zeros((len(data), 400))
    for i in range(len(data)):
        sentence = processing(data[i])
        if len(sentence) >= 400:
            for w in range(400):
                mat[i][w] = (vocab[sentence[w]])
        else:
            n_zeros = 400 - len(sentence)
            for w in range(len(sentence)):
                mat[i][n_zeros + w] = (vocab[sentence[w]])

    return mat.astype(int)

train_neg_mat = create_mat(train_neg, vocab)
train_pos_mat = create_mat(train_pos, vocab)
test_neg_mat = create_mat(test_neg, vocab)
test_pos_mat = create_mat(test_pos, vocab)

train_x = np.concatenate((train_neg_mat, train_pos_mat), axis=0)
test_x = np.concatenate((test_neg_mat, test_pos_mat), axis=0)

train_y = [0]*len(train_neg_mat)
train_y_pos = [1]*len(train_pos_mat)
test_y = [0]*len(test_neg_mat)
test_y_pos = [1]*len(test_pos_mat)

train_y.extend(train_y_pos)
test_y.extend(test_y_pos)

train_x, dev_x, train_y, dev_y = train_test_split(train_x, train_y, test_size=0.25, random_

batch_size = 64

train_data = TensorDataset (torch.from_numpy(train_x), torch.from_numpy(np.array(train_y)))
dev_data = TensorDataset (torch.from_numpy(dev_x), torch.from_numpy(np.array(dev_y)))
test_data = TensorDataset (torch.from_numpy(test_x), torch.from_numpy(np.array(test_y)))

train_loader = DataLoader (train_data, shuffle=True, batch_size=batch_size)
```

```
dev_loader = DataLoader (dev_data, shuffle=True, batch_size=batch_size)
test_loader = DataLoader (test_data, shuffle=True, batch_size=batch_size)
```

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

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

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

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

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

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

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

**Solution:**

Hyperparameters used:

- embedding dim = 300
- max feat = 400
- batch size = 100
- dim = 300
- h dim = 5
- layers = 1
- out size = 1

- epoch = 11
- learning rate = 0.001

**GRU Accuracy Dev set: 76.8**
**GRU Accuracy Test set: 73.0**

Note: code in the appendix

### Part 3 - Comparison with a MLP (5 points)

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

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

**Solution:**

Using the same hyperparameters as in the GRU, I have obtained significantly lower accuracy.

**MLP Accuracy Dev set: 67.73**
**MLP Accuracy Test set: 62.9**

Note: code in the appendix

## Problem 2 - Generative Adversarial Networks

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

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

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

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

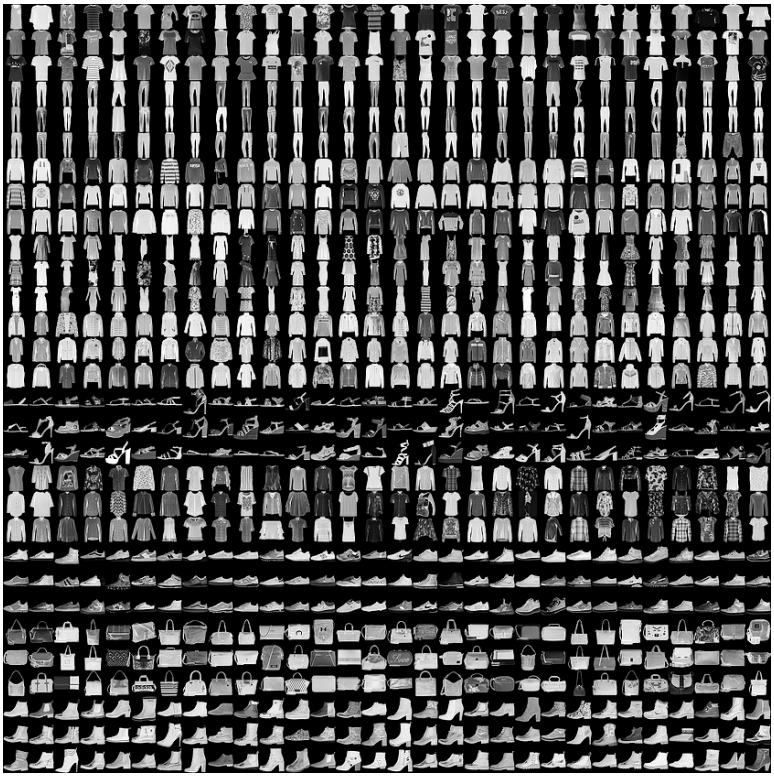


Figure 1: Fashion-Mnist dataset example images. It contains 10 classes of cloths, shoes, and bags.

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

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

Put your code in the appendix.

## Part 1 - Vanilla GAN (10 points)

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

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

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

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

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

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

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

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

**Solution:**

Figure 2: Vanilla GAN - Training Loss curves (250 epoch)

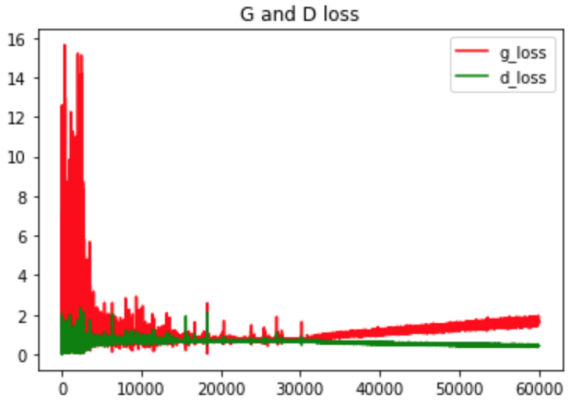



Figure 3: Vanilla GAN - Training Loss curves after each epoch (250 epoch)

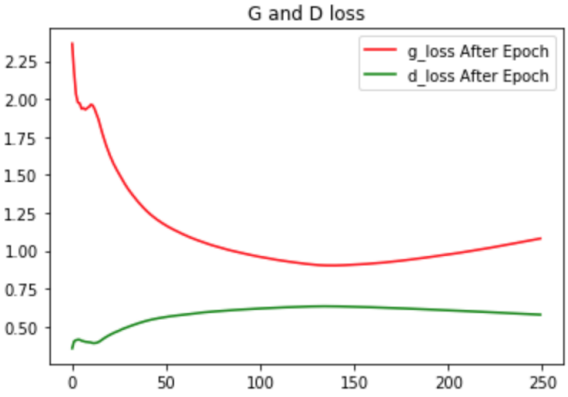


Figure 4: Vanilla GAN - Training Scores (250 epoch)

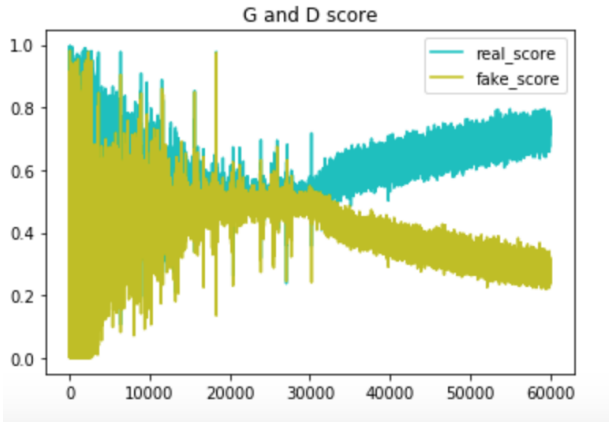



Figure 5: Vanilla GAN - Training Scores after each epoch (250 epoch)

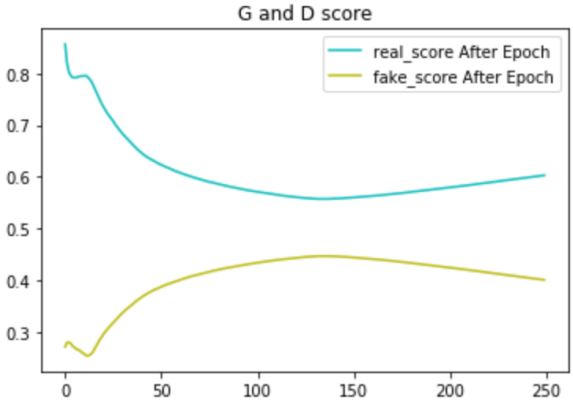


Figure 6: Vanilla GAN - Generated images from G in the beginning of the training (250 epoch)

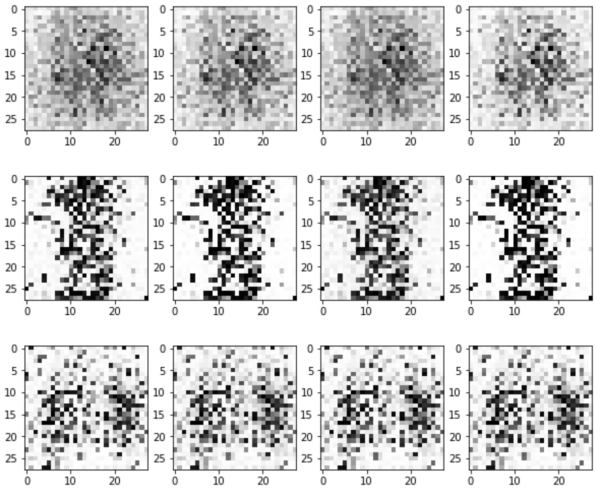


Figure 7: Vanilla GAN - Generated images from G in the intermediate stage of the training (250 epoch)

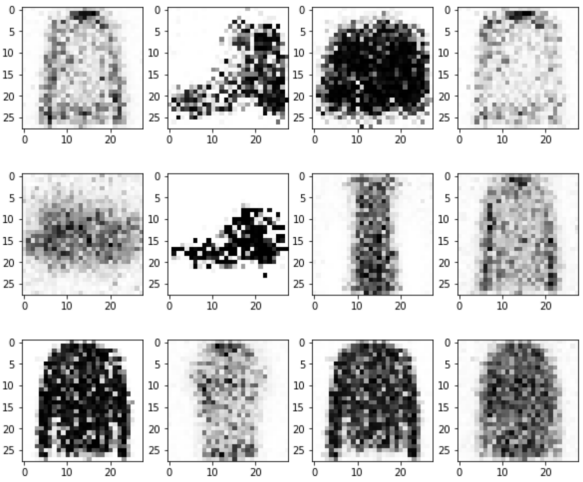


Figure 8: Vanilla GAN - Generated images from G in the intermediate stage of the training (250 epoch)

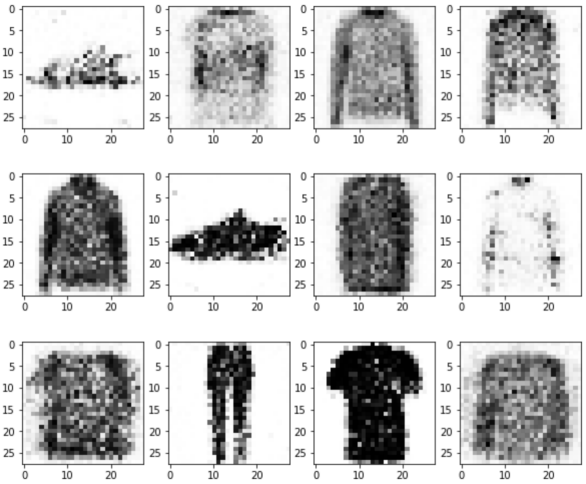
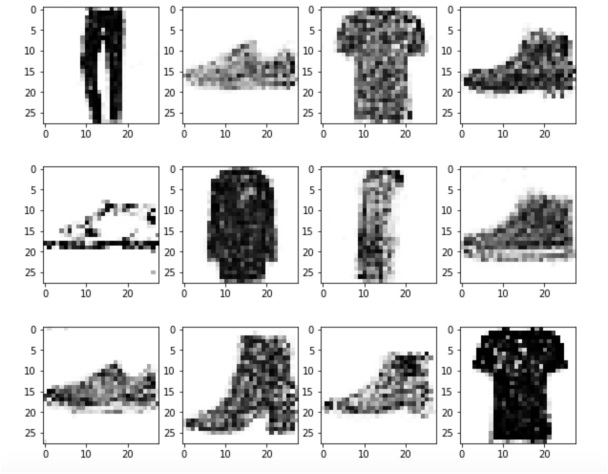


Figure 9: Vanilla GAN - Generated images from G after convergence (250 epoch)



## 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.

Figure 10: MSE GAN - Training Loss curves (batch size = 250)

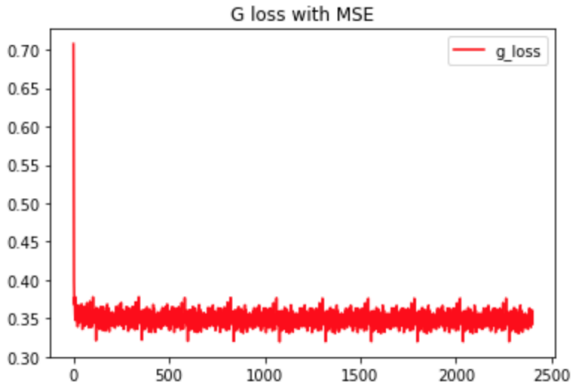
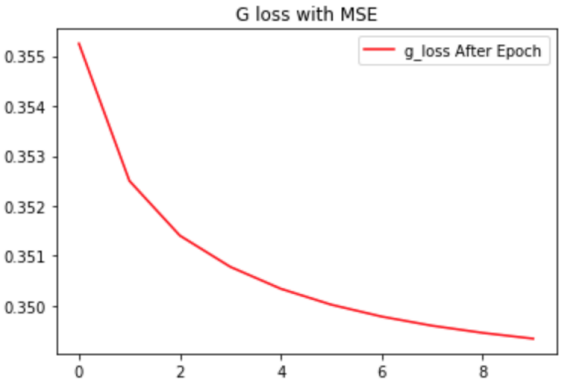



Figure 11: MSE GAN - Training Scores after each epoch (batch size = 250)




As we can see, outputs are not very different. The only difference is that the final examples are smoother, while the examples from the beginning of the training are more pixelated.

Figure 12: MSE GAN - Generated images from G in the beginning of the training (batch size = 250)

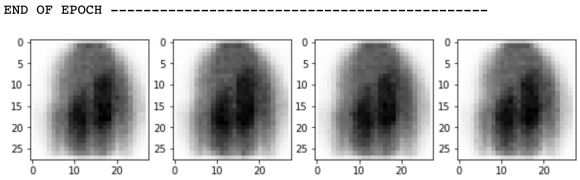
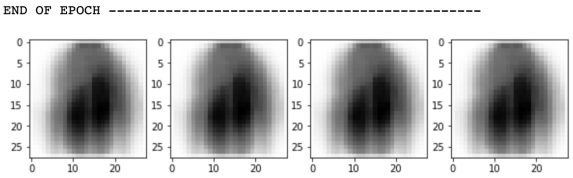



Figure 13: MSE GAN - Generated images from G at the end of the training (batch size = 250)




We can also see that all the images look the same. This is due to the average effect that MSE has among the batch.

Then, what would happen if I reduce the batch size from 250 to 10? Here are the results. However, there is not much change other than having a even more unstable loss.

Figure 14: MSE GAN - Training Loss curves (batch size = 10)

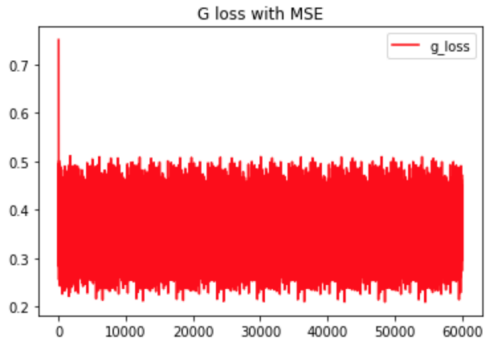


Figure 15: MSE GAN - Training Scores after each epoch (batch size = 10)

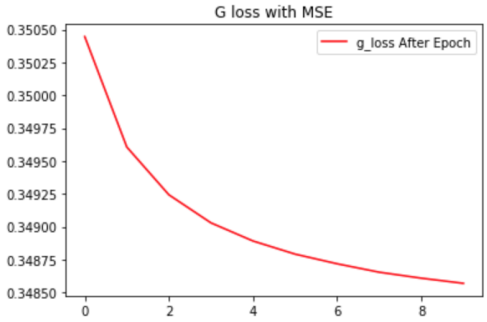



Figure 16: MSE GAN - Generated images from G in the beginning of the training (batch size = 10)

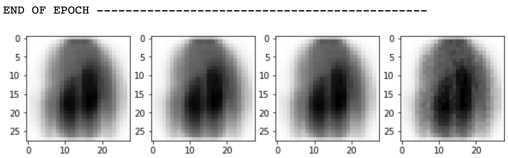
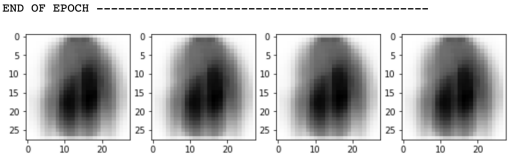



Figure 17: MSE GAN - Generated images from G at the end of the training (batch size = 10)



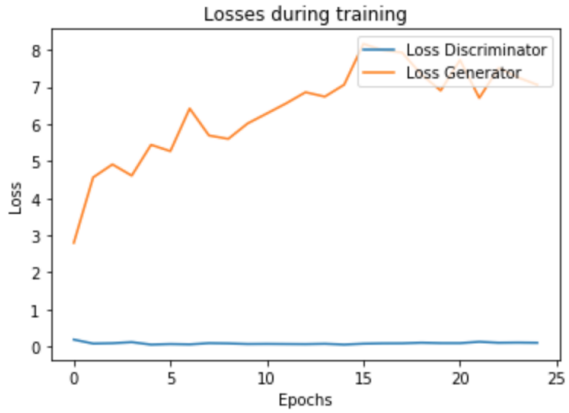
**Wasserstein GAN (WGAN)**

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

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.

Figure 18: WGAN - Training Loss curves




```
class Generator_wgan(nn.Module):

    def __init__(self, z_dim, out_dim):
        super(Generator_wgan, self).__init__()

        self.fc1 = nn.Linear(z_dim, 32)
        self.fc2 = nn.Linear(32, 32)
        self.fc3 = nn.Linear(32, out_dim)

        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()
```

```
        self.fc1.weight = nn.init.xavier_normal_(self.fc1.weight, gain = 1.0)
        self.fc2.weight = nn.init.xavier_normal_(self.fc2.weight, gain = 1.0)
        self.fc3.weight = nn.init.xavier_normal_(self.fc3.weight, gain = 1.0)


    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)

        x = self.fc2(x)
        x = self.relu(x)

        x = self.fc3(x)

        x = self.tanh(x)

        return x

class Discriminator_wgan(nn.Module):

    def __init__(self, out_dim):
        super(Discriminator_wgan, self).__init__()

        self.fc1 = nn.Linear(out_dim, 32)
        self.fc2 = nn.Linear(32, 32)
        self.fc3 = nn.Linear(32, 1)
        self.relu = nn.ReLU()
        #self.sigmoid = nn.Sigmoid()


        self.fc1.weight = nn.init.xavier_normal_(self.fc1.weight, gain = 1.0)
        self.fc2.weight = nn.init.xavier_normal_(self.fc2.weight, gain = 1.0)
        self.fc3.weight = nn.init.xavier_normal_(self.fc3.weight, gain = 1.0)

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)

        x = self.fc2(x)
        x = self.relu(x)
```

```
        x = self.fc3(x)
        #x = self.sigmoid(x)

        return x

class GAN_wgan(pl.LightningModule):

    def __init__(self, hparams):
        super(GAN_wgan, self).__init__()

        self.hparams = hparams

        self.generator = Generator_wgan(z_dim=hparams.z_dim, out_dim=hparams.x_dim)
        self.discriminator = Discriminator_wgan(out_dim=hparams.x_dim)

        self.generated_imgs = None
        self.last_imgs = None

        self.g_loss = []
        self.d_loss = []
        self.g_loss_after_e = []
        self.d_loss_after_e = []

        self.real_score = []
        self.fake_score = []
        self.real_score_after_e = []
        self.fake_score_after_e = []

        self.replay = {'real': None, 'fake': None}

        self.epoch_count = 0

    def forward(self, x):
        return self.generator(x)

    def training_step (self, batch, batch_nb, optimizer_idx):
        imgs, _ = batch
        self.last_imgs = imgs
        imgs = imgs.view(self.hparams.batch_size, -1)
```

```
        if optimizer_idx == 0: # train generator

            z = torch.randn(imgs.size(0), self.hparams.z_dim)
            self.generated_imgs = self(z)

            g_loss = Variable(-torch.mean(self.discriminator(self.generated_imgs.detach())))

            tqdm_dict = {'g_loss': g_loss}
            output = OrderedDict({'loss': g_loss,'progress_bar': tqdm_dict,'log': tqdm_dict

            self.g_loss.append(g_loss.mean().detach().data)

            return output

        if optimizer_idx == 1: # train discriminator

            real_score = self.discriminator(imgs)
            fake_score = self.discriminator(self.generated_imgs.detach())

            d_loss = Variable(-torch.mean(real_score) + torch.mean(fake_score), requires_gr

            tqdm_dict = {'d_loss': d_loss}
            output = OrderedDict({'loss': d_loss,'progress_bar': tqdm_dict,'log': tqdm_dict

            self.d_loss.append(d_loss.mean().detach().data)
            self.real_score.append(real_score.mean().detach().data)
            self.fake_score.append(fake_score.mean().detach().data)

            self.replay['real'] = imgs.clone()
            self.replay['fake'] = self.generated_imgs.detach().clone()

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

            return output

    def configure_optimizers(self):
        lr_g = self.hparams.lr_g
        lr_d = self.hparams.lr_g

        opt_g = torch.optim.RMSprop(self.generator.parameters(), lr=lr_g)
```

```
        opt_d = torch.optim.RMSprop(self.discriminator.parameters(), lr=lr_d)
        return [opt_g, opt_d], []

    def train_dataloader(self):
        transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize(mean=[0
        fmnist = torchvision.datasets.FashionMNIST(root="./", train=True, transform=transfo
        return DataLoader(dataset=fmnist, batch_size=self.hparams.batch_size, num_workers=2

    def on_epoch_end(self):
        z = torch.randn(self.hparams.batch_size, self.hparams.z_dim)
        sample_imgs = self(z)
        sample_imgs = sample_imgs.detach()
        sample_imgs = sample_imgs.reshape(sample_imgs.size(0), 1, 28, 28)
        sample_imgs = (sample_imgs + 0.5)*0.5

        fig, axs = plt.subplots(1, 4, figsize=(10, 8))
        inx = [i for i in range(0,4)]
        for ax, idx in zip(axs, inx):
            ax.imshow(sample_imgs[idx][0], cmap='Greys')
        plt.show()

        self.g_loss_after_e.append(np.mean(self.g_loss))
        self.d_loss_after_e.append(np.mean(self.d_loss))

        self.real_score_after_e.append(np.mean(self.real_score))
        self.fake_score_after_e.append(np.mean(self.fake_score))

c_ = [0.1, 0.01, 0.001, 0.0001]

for c in c_:

    args_wgan = {'max_epochs': 15, 'batch_size': 64, 'lr_g': 0.00005, "lr_d": 0.00005, 'z_d

    hparams_wgan = Namespace(**args_wgan)
    gan_model_wgan = GAN_wgan(hparams_wgan)

    trainer_wgan = pl.Trainer(max_epochs=hparams_wgan.max_epochs, early_stop_callback=False
    trainer_wgan.fit(gan_model_wgan)

    plt.plot(gan_model_wgan.g_loss, c='r')
    plt.plot(gan_model_wgan.d_loss, c='g')
```

```
plt.legend(['g_loss', 'd_loss'])
plt.title('G and D loss with WGAN');plt.show()

plt.plot(gan_model_wgan.g_loss_after_e, c='r')
plt.plot(gan_model_wgan.d_loss_after_e, c='g')
plt.legend(['g_loss After Epoch', 'd_loss After Epoch'])
plt.title('G and D loss with WGAN');plt.show()

plt.plot(gan_model_wgan.real_score, c='c')
plt.plot(gan_model_wgan.fake_score, c='y')
plt.legend(['real_score', 'fake_score'])
plt.title('G and D score with WGAN');plt.show()

plt.plot(gan_model_wgan.real_score_after_e, c='c')
plt.plot(gan_model_wgan.fake_score_after_e, c='y')
plt.legend(['real_score After Epoch', 'fake_score After Epoch'])
plt.title('G and D score with WGAN');plt.show()
```

**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:**

Figure 19: Least Square GAN - Training Loss curves

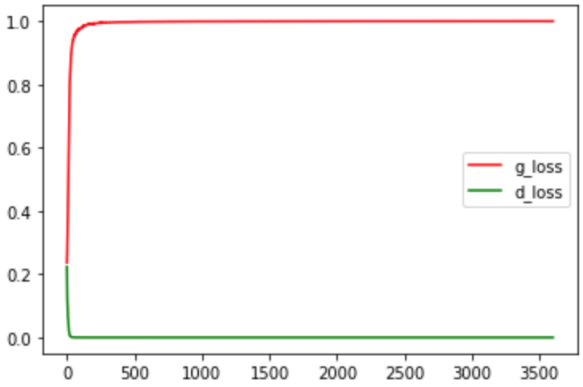
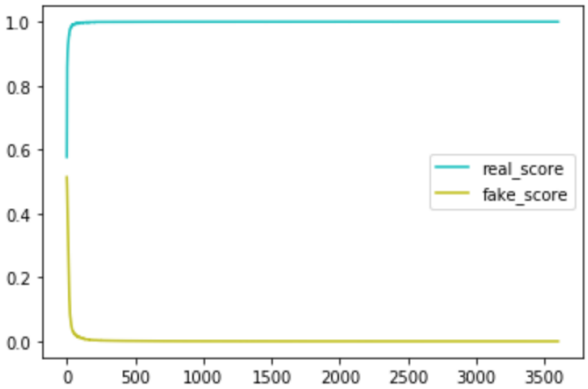



Figure 20: Least Square GAN - Training Scores



```
class GAN_ls(pl.LightningModule):

    def __init__(self, hparams):
        super(GAN_ls, self).__init__()

        self.hparams = hparams

        self.generator = Generator(z_dim=hparams.z_dim, out_dim=hparams.x_dim)
        self.discriminator = Discriminator(out_dim=hparams.x_dim)

        self.generated_imgs = None # cache for generated images
        self.last_imgs = None

        self.g_loss = []
        self.d_loss = []
        self.g_loss_after_e = []
        self.d_loss_after_e = []

        self.real_score = []
        self.fake_score = []
        self.real_score_after_e = []
        self.fake_score_after_e = []

        self.replay = {'real': None, 'fake': None}

        self.epoch_count = 0

        #self.adversarial_loss = torch.nn.MSELoss()

    def forward(self, x):
        return self.generator(x)

    def adversarial_loss(self, y_hat, y):
        return torch.nn.MSELoss()(y_hat, y)


    def training_step(self, batch, batch_nb, optimizer_idx):

        imgs, _ = batch
        self.last_imgs = imgs
        imgs = imgs.view(self.hparams.batch_size, -1)
```

```
        if optimizer_idx == 0: # train generator

            z = torch.randn(imgs.size(0), self.hparams.z_dim) # sample noise
            self.generated_imgs = self(z)

            valid = torch.ones(imgs.size(0), 1) # ground truth result (ie: all fake)
            g_loss = Variable(self.adversarial_loss(self.discriminator(self.generated_imgs.

            tqdm_dict = {'g_loss': g_loss}
            output = OrderedDict({'loss': g_loss,'progress_bar': tqdm_dict,'log': tqdm_dict

            self.g_loss.append(g_loss.mean().detach().data)
            return output

        if optimizer_idx == 1: # train discriminator

            valid = torch.ones(imgs.size(0), 1) # how well can it label as real?
            real_score = self.discriminator(imgs)
            real_loss = self.adversarial_loss(self.discriminator(imgs), valid)

            fake = torch.zeros(imgs.size(0), 1) # how well can it label as fake?
            fake_score = self.discriminator(self.generated_imgs.detach())
            fake_loss = self.adversarial_loss(self.discriminator(self.generated_imgs.detach

            d_loss = (real_loss + fake_loss) / 2 # discriminator loss is the average of the
            tqdm_dict = {'d_loss': d_loss}
            output = OrderedDict({'loss': d_loss,'progress_bar': tqdm_dict,'log': tqdm_dict

            self.d_loss.append(d_loss.mean().detach().data)
            self.real_score.append(real_score.mean().detach().data)
            self.fake_score.append(fake_score.mean().detach().data)

            self.replay['real'] = imgs.clone()
            self.replay['fake'] = self.generated_imgs.detach().clone()

            return output

    def configure_optimizers(self):
        lr_g = self.hparams.lr_g
        lr_d = self.hparams.lr_g
```

```
        b1 = self.hparams.b1
        b2 = self.hparams.b2

        opt_g = torch.optim.Adam(self.generator.parameters(), lr=lr_g, betas=(b1, b2))
        opt_d = torch.optim.Adam(self.discriminator.parameters(), lr=lr_d, betas=(b1, b2))
        return [opt_g, opt_d], []

    def train_dataloader(self):
        transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize(mean=[0
        fmnist = torchvision.datasets.FashionMNIST(root="./", train=True, transform=transfo
        return DataLoader(dataset=fmnist, batch_size=self.hparams.batch_size, num_workers=4

    def on_epoch_end(self):
        z = torch.randn(250, 100)
        sample_imgs = self(z)
        sample_imgs = sample_imgs.detach()
        sample_imgs = sample_imgs.reshape(sample_imgs.size(0), 1, 28, 28)
        sample_imgs = (sample_imgs + 0.5)*0.5

        fig, axs = plt.subplots(1, 4, figsize=(10, 8))
        inx = [i for i in range(0,4)]
        for ax, idx in zip(axs, inx):
            ax.imshow(sample_imgs[idx][0], cmap='Greys')
        plt.show()

        self.g_loss_after_e.append(np.mean(self.g_loss))
        self.d_loss_after_e.append(np.mean(self.d_loss))

        self.real_score_after_e.append(np.mean(self.real_score))
        self.fake_score_after_e.append(np.mean(self.fake_score))

args_ls = {
    'max_epochs': 15,
    'batch_size': 250,
    'lr_g': 0.0002,
    "lr_d": 0.0002,
    'b1': 0.5,
    'b2': 0.999,
    'z_dim': 100,
    'x_dim': 784,
}
```

```
hparams_ls = Namespace(**args_ls)

gan_model_ls = GAN_ls(hparams_ls)


trainer_ls = pl.Trainer(max_epochs=hparams_ls.max_epochs, early_stop_callback=False)
trainer_ls.fit(gan_model_ls)
```

### Part 3 - Mode Collapse in GANs (10 points)

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

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

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

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

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

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

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

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

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

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

**Solution:**

Figure 21: Classification - Accuracy

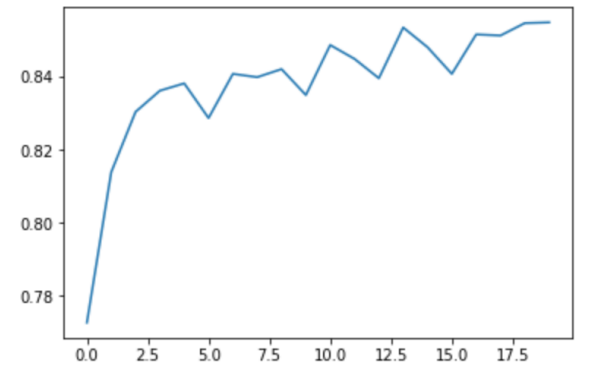
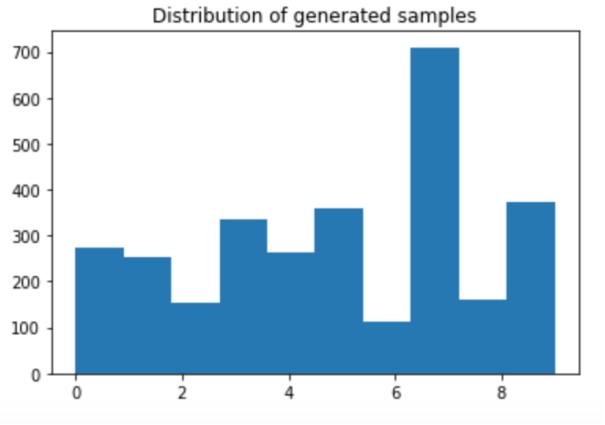
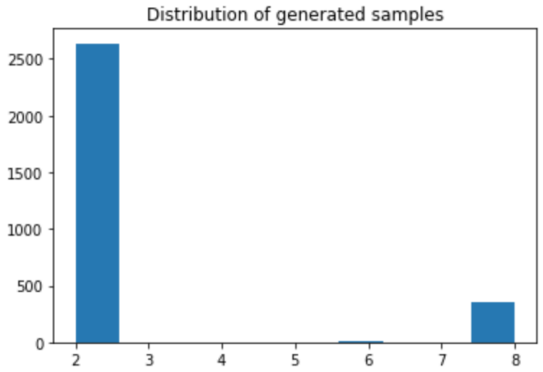



Figure 22: Class distribution - using Vanilla GAN as generator




As it can be seen, in my specific case, the unrolled GAN did not help with a more equally distribution of classes. However, this must be due to the poor Unrolled GAN training. The training happened to be extremely unstable. Judging for the images that it was generating after each epoch, the training was doing well but after the 30 epoch it destabilizes. This happened to me in my Vanilla GAN as well. After researching the best hyperparameters for this kind of task, I believe the ones I have should work quite decently. However, I think

Figure 23: Class distribution - using Unrolled GAN as generator




the model run for at least 100 epoch. Without external GPU and an old computer this task is quite unrealistic to achieve.

When looking at the WGAN we can, indeed, see that it helps to reduce the mode collapse problem in comparison with the Vanilla GAN. However, there is still a long way to go to achieve perfect equal distribution of classes.

Figure 24: Unrolled GAN - Training Loss curves (batch size = 50)

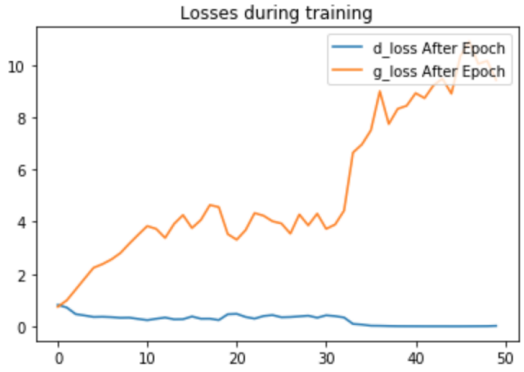
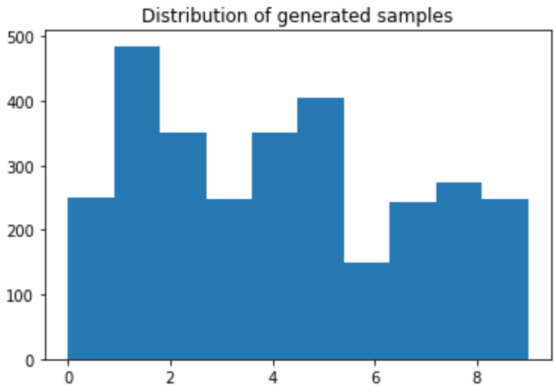


Figure 25: Class distribution - using WGAN as generator



## Part 4 - Conditional GAN (10 points)

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

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

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

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

**Solution:**

With a similar amount of epoch, I found Vanilla GAN to be more effective in generating good quality images. Conditional GAN require higher training time to obtain similar results in all the classes, however, by doing so, we ensure the distribution will be as we desire: we have the power to specify which class we want to generate.

These are the results generated with 50 epoch.

It looks like the generated images at the end of the training are less intuitive than the ones in the middle or beginning. Following, I provide some image samples during all the training process.

Figure 26: Conditional GAN - Training Loss curves

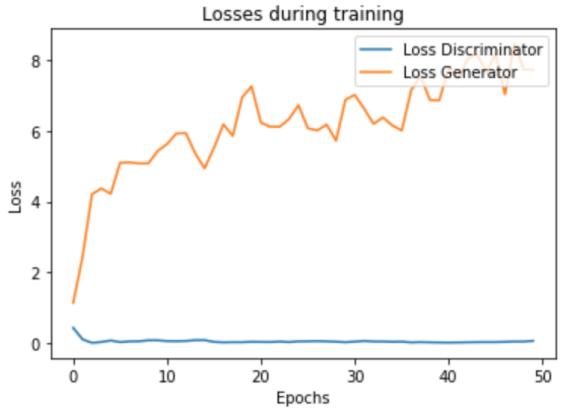



Figure 27: Conditional GAN - 3 Samples from each of the 10 classes

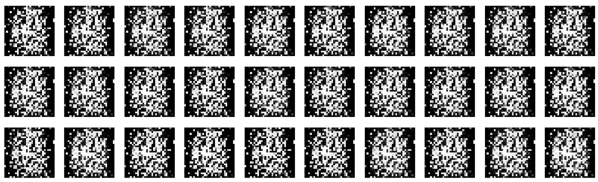


Figure 28: Conditional GAN - Images generated in the beginning of the training

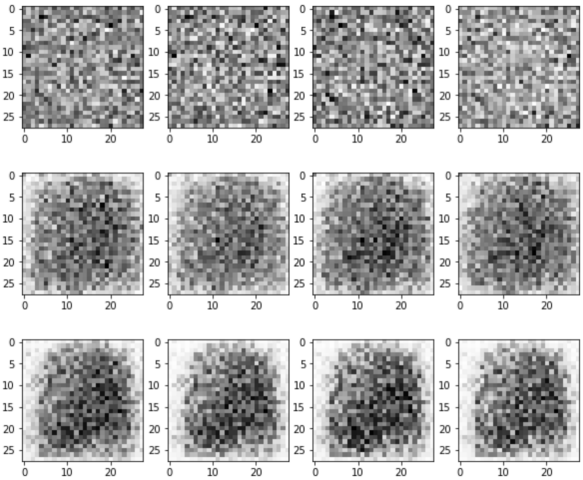


Figure 29: Conditional GAN - Images generated in the middle of the training

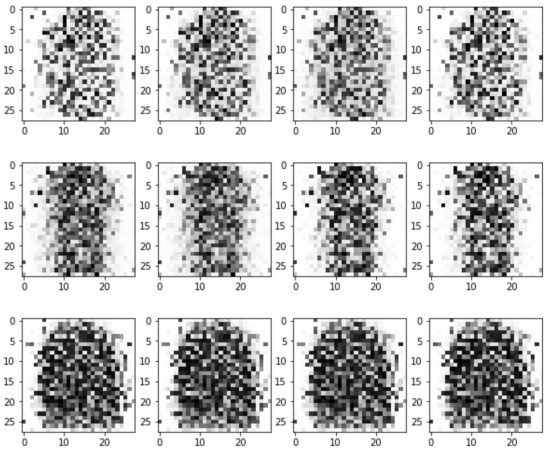


Figure 30: Conditional GAN - Images generated in the middle of the training

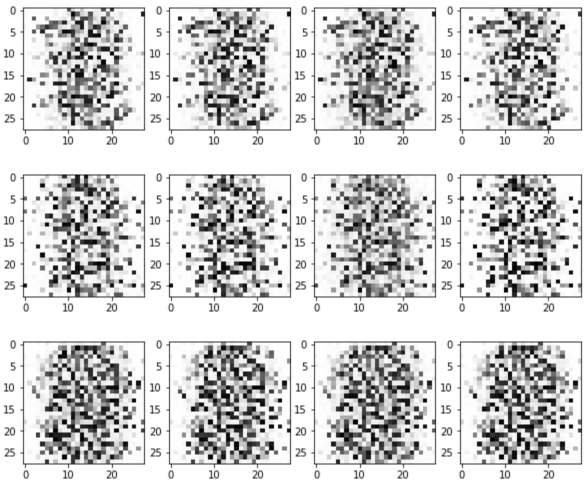


Figure 31: Conditional GAN - Images generated in the end of the training

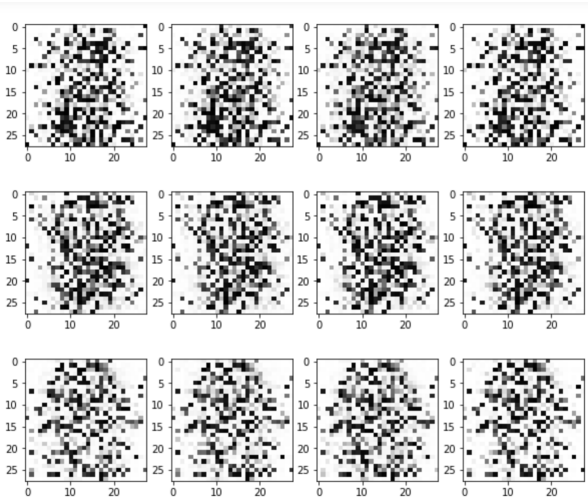


Figure 32: Conditional GAN - Images generated in the end of the training