HW 3: GANs

The homework is broken up as follows:

1. Setup
2. Question 1: Adversarial Examples & GANs (three points total)
3. Question 1.1: Generating Adversarial Examples (one point)
4. Question 2: Vanilla Collapse (two points total)
5. Question 2.1: Latent Interpolation (two points)

How to Submit This Homework

The Homework has been provided to you as an `.ipynb` file. I expect you to complete the notebook and submit the following on Gradescope:

- This completed notebook (with your answers filled in, including any source code used to answer the questions)
- A `.pdf` file of the notebook (for ease of grading). This will be a printing-to-pdf version of the notebook.

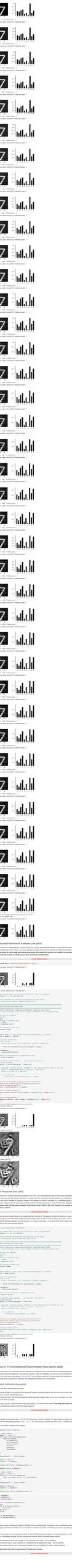
If you cannot only use at Colab, you will lose points. Yet Colab homework will be graded. No substitutions nor excuses of other media will be accepted.

Finally, ask questions regarding anything which should be accessible to your version. For example, if a cell seems to hangs/crash, let me know. Your submitted notebook (and PDF) should not contain any additional content or cells beyond what is requested.

Setup

If you do not have access to a CUDA-compatible NVIDIA GPU, it is recommended that you run this notebook in Google Colab. There, you will have the option to enable GPU acceleration with: `Runtime` -> `Change Runtime Type` -> `Hardware accelerator` -> `GPU` -> `Save`. Note that you may have to re-connect or restart the notebook after changing runtime types.

```
!pip install -q torchvision==0.10.0 torchaudio==0.9.0 torchtext==0.10.0 torch==1.9.0
!pip install -q jupyter-tensorboard
!pip install -q tensorboard==2.4.1 tensorboard-plugin-wit==1.8.0 werkzeug==1.0.1
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_storage_v1beta1/proto/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/protobuf/compiler/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/protobuf/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/protobuf/internal/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/api/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/api/auth/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/api/cache/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/api/client/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/api/contrib/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_storage_v1beta1/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_storage_v1beta1/gapic/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_storage_v1beta1/gapic/transports/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_storage_v1beta1/proto/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_storage_v1beta1/proto/transports/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_storage_v1beta1/types/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery/dbapi/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery/dbapi/sqlalchemy_bigquery/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery/magics/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery/schema/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/gapic/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/gapic/transports/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/proto/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/proto/transports/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/types/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/utilities/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/utilities/bigquery_ipywidgets/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/utilities/bigquery_jupyterlab/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/utilities/bigquery_jupyterlab/labextension/static/*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/utilities/bigquery_jupyterlab/labextension/static/vendors~lib_index_js.*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/utilities/bigquery_jupyterlab/labextension/static/lib_index_js.*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/utilities/bigquery_jupyterlab/labextension/static/remoteEntry.*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/utilities/bigquery_jupyterlab/labextension/static/style_index_js.*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/utilities/bigquery_jupyterlab/labextension/static/vendors~node_modules_jupyterlab_coreutils_lib_index_js.*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/utilities/bigquery_jupyterlab/labextension/static/vendors~node_modules_jupyterlab_services_lib_index_js.*.pyc
!rm -rf /usr/local/lib/python3.7/dist-packages/google/cloud/bigquery_v2/utilities/bigquery_jupyterlab/labextension/static/vendors~node_modules_webpack_sharing_consume_default_react_react-dom_react_jsx-runtime
```



1.2 Unbounded Adversarial Examples (one point)

Let's continue to study the case of a single pixel. In this case, we can plot the decision boundary in a large box for a pre-trained image model (the architecture of the weight-clipped, 10-layer ConvNet in question is in large print below) to see how the decision boundary looks like. **Mostly the boundary looks very far away, and the image looks very robust to adversarial perturbations. However, there are a number of exceptions. In fact, there are even cases very close to the image, where the boundary is very close, but the classifier is still very robust to adversarial perturbations!**

[CODE BELOW HERE]

Starting from image, again (Trained pixel Value=0.75, Label=7):


```
# ------------------------------------------------------------------------
target = torch.tensor(7).cuboid
# ------------------------------------------------------------------------
# Let's begin, again, by an empty bounds to track the gradients
# ------------------------------------------------------------------------
x = x.detach() + (0, 0, 0)
# ------------------------------------------------------------------------
# Iterations optimize an additive perturbation to our input
# ------------------------------------------------------------------------
for iteration in range(1000):
# ------------------------------------------------------------------------
# Forward pass, backward pass, update bounds, zero gradients
# ------------------------------------------------------------------------
outputs, lb, ub = model(x, method_name='CROWN-IBP', eps=eps)[-1].flatten()
loss = (-lb[target] + ub[target] + ub.sum()).sum()
loss.backward()
# ------------------------------------------------------------------------
# Report stuff
# ------------------------------------------------------------------------
print(f'Iteration {iteration:3} Prediction:')
for pixel in range(784):
if (x.grad[0,0,0,0,pixel] != 0):
print(f'True pixel Value={x[0,0,0,0,pixel]:5.2f}, Predicted pixel =')
# ------------------------------------------------------------------------
# Gradient descent step
# ------------------------------------------------------------------------
lr = 0.1
x = x - lr * x.grad
x = torch.clamp(x, 0, 1)
# ------------------------------------------------------------------------
# Zero out the gradients
# ------------------------------------------------------------------------
x.grad = None
# ------------------------------------------------------------------------
# Report stuff
# ------------------------------------------------------------------------
if (iteration % 10 == 0):
print('Iteration: {0:3} Prediction:'.format(iteration))
for pixel in range(784):
if (x.grad[0,0,0,0,pixel] != 0):
print('True pixel Value={0:5.2f}, Predicted pixel ='.format(x[0,0,0,0,pixel]))
# ------------------------------------------------------------------------
# Projection step
# ------------------------------------------------------------------------
x = torch.max(x - eps, x.detach())
x = torch.min(x + eps, x.detach())
# ------------------------------------------------------------------------
# Clamping to valid pixel values
# ------------------------------------------------------------------------
x = torch.clamp(x, 0, 1)
# ------------------------------------------------------------------------
# Report stuff
# ------------------------------------------------------------------------
print('Prediction: {outputs.argmax()}')
# ------------------------------------------------------------------------
```

So, above we process (random) inputs and iteratively find (not fast) the 1 Pixel golden material will optimize a clever solution to a bounds-based, exact, and differentiable (but not fast) loss. Let's see what does it look like for the top-to-bottom, color-coded by pixel-to-pixel distance, 1000 steps of optimization. Note that the image is very large in size, and that many pixels seem to detect some representation of the CNN principal digits 0.

```
target = torch.tensor(7).cuboid
# ------------------------------------------------------------------------
# Let's begin, again, by an empty bounds to track the gradients
# ------------------------------------------------------------------------
x = x.detach() + (0, 0, 0)
# ------------------------------------------------------------------------
# Iterations optimize an additive perturbation to our input
# ------------------------------------------------------------------------
for iteration in range(1000):
# ------------------------------------------------------------------------
# Forward pass, backward pass, update bounds, zero gradients
# ------------------------------------------------------------------------
outputs, lb, ub = model(x, method_name='CROWN-IBP', eps=eps)[-1].flatten()
loss = (-lb[target] + ub[target] + ub.sum()).sum()
loss.backward()
# ------------------------------------------------------------------------
# Report stuff
# ------------------------------------------------------------------------
print(f'Iteration {iteration:3} Prediction:')
for pixel in range(784):
if (x.grad[0,0,0,0,pixel] != 0):
print(f'True pixel Value={x[0,0,0,0,pixel]:5.2f}, Predicted pixel =')
# ------------------------------------------------------------------------
# Gradient descent step
# ------------------------------------------------------------------------
lr = 0.1
x = x - lr * x.grad
x = torch.clamp(x, 0, 1)
# ------------------------------------------------------------------------
# Zero out the gradients
# ------------------------------------------------------------------------
x.grad = None
# ------------------------------------------------------------------------
# Report stuff
# ------------------------------------------------------------------------
if (iteration % 10 == 0):
print('Iteration: {0:3} Prediction:'.format(iteration))
for pixel in range(784):
if (x.grad[0,0,0,0,pixel] != 0):
print('True pixel Value={0:5.2f}, Predicted pixel =')
# ------------------------------------------------------------------------
# Projection step
# ------------------------------------------------------------------------
x = torch.max(x - eps, x.detach())
x = torch.min(x + eps, x.detach())
# ------------------------------------------------------------------------
# Clamping to valid pixel values
# ------------------------------------------------------------------------
x = torch.clamp(x, 0, 1)
# ------------------------------------------------------------------------
# Report stuff
# ------------------------------------------------------------------------
print('Prediction: {outputs.argmax()}')
# ------------------------------------------------------------------------
```


1.3 Free Response (one point)

A large proportion in existing adversarial examples are CNNs, and notes that both ways models using the patterns of a much network is it will cause the the network classifier is a certain way. The researcher therefore point that a simple alternative is CNNs, which are not affected by the pixel-to-pixel distance, but are affected by the overall pattern. Finally, the researcher point that a simple alternative to CNNs would be to train a large model to be robust to adversarial examples. **Do you think this approach will yield images that look like clear examples of the desired class? Why or why not? Support your answer with your results from question 1.2.**

[CODE BELOW HERE]

No, above we process (random) inputs and iteratively find (not fast) the 1 Pixel golden material will optimize a clever solution to a bounds-based, exact, and differentiable (but not fast) loss. Let's see what does it look like for the top-to-bottom, color-coded by pixel-to-pixel distance, 1000 steps of optimization. Note that the image is very large in size, that many pixels seem to detect some representation of the CNN principal digits 0.

```
target = torch.tensor(7).cuboid
# ------------------------------------------------------------------------
# Let's begin, again, by an empty bounds to track the gradients
# ------------------------------------------------------------------------
x = x.detach() + (0, 0, 0)
# ------------------------------------------------------------------------
# Iterations optimize an additive perturbation to our input
# ------------------------------------------------------------------------
for iteration in range(1000):
# ------------------------------------------------------------------------
# Forward pass, backward pass, update bounds, zero gradients
# ------------------------------------------------------------------------
outputs, lb, ub = model(x, method_name='CROWN-IBP', eps=eps)[-1].flatten()
loss = (-lb[target] + ub[target] + ub.sum()).sum()
loss.backward()
# ------------------------------------------------------------------------
# Report stuff
# ------------------------------------------------------------------------
print(f'Iteration {iteration:3} Prediction:')
for pixel in range(784):
if (x.grad[0,0,0,0,pixel] != 0):
print(f'True pixel Value={x[0,0,0,0,pixel]:5.2f}, Predicted pixel =')
# ------------------------------------------------------------------------
# Gradient descent step
# ------------------------------------------------------------------------
lr = 0.1
x = x - lr * x.grad
x = torch.clamp(x, 0, 1)
# ------------------------------------------------------------------------
# Zero out the gradients
# ------------------------------------------------------------------------
x.grad = None
# ------------------------------------------------------------------------
# Report stuff
# ------------------------------------------------------------------------
if (iteration % 10 == 0):
print('Iteration: {0:3} Prediction:'.format(iteration))
for pixel in range(784):
if (x.grad[0,0,0,0,pixel] != 0):
print('True pixel Value={0:5.2f}, Predicted pixel =')
# ------------------------------------------------------------------------
# Projection step
# ------------------------------------------------------------------------
x = torch.max(x - eps, x.detach())
x = torch.min(x + eps, x.detach())
# ------------------------------------------------------------------------
# Clamping to valid pixel values
# ------------------------------------------------------------------------
x = torch.clamp(x, 0, 1)
# ------------------------------------------------------------------------
# Report stuff
# ------------------------------------------------------------------------
print('Prediction: {outputs.argmax()}')
# ------------------------------------------------------------------------
```


Question 2: A Convolutional Discriminator (four points total)

While previously seen from conventional neural networks can outperform that fully connected counterparts by leveraging the local structure, convolutional neural networks can outperform that fully connected counterparts by leveraging the local structure. The hypothesis is the very well is one reason, THEREFORE, that convolutions of stacks of nothing rather than transformer layer. We'll start by considering a convolutional discriminator, and we'll end by considering a convolutional generator.

Convolution Arithmetic (one point)

Assume we are given the following two layers:

- A convolutional layer that accepts a 3D tensor with 3 input channels, 64 output channels, and a kernel size of 5, with a stride of 2, and no padding, resulting in a 3D tensor feature map.
- A convolutional layer that accepts a 3D tensor with 64 input channels, 128 output channels, and a kernel size of 5, with a stride of 2, and no padding, resulting in a single 3D tensor as output.

Thus, both layers accept a 3D dimensional input and produce a 3 D dimensional output. **Assuming that neither layer uses biases, how many trainable parameters does each layer require?**

[CODE BELOW HERE]

Answer

Number of weights in convolution layer is $K^2 \cdot C \cdot K$ where K is kernel size and C is channels. In our case number of weights are 2.5 in layer number 1 and 2.5 in layer number 2 (ignoring biases). Thus, the total number of weights is $2.5 + 2.5 = 5$. Or, ignoring biases, we have 5 weights.

Provide your code below:

```
class TwoLayerConv2d(nn.Module):
def __init__(self):
super().__init__()
self.conv1 = nn.Conv2d(in_channels=3,
out_channels=64,
kernel_size=5,
stride=2,
padding=0,
bias=False)
self.conv2 = nn.Conv2d(in_channels=64,
out_channels=128,
kernel_size=5,
stride=2,
padding=0,
bias=False)
def forward(self, x, torch.Tensor):
output = self.conv1(x)
output = self.conv2(output)
return output
class TwoLayerConv2dLinear(nn.Module):
def __init__(self):
super().__init__()
self.conv1 = nn.Conv2d(in_channels=3,
out_channels=64,
kernel_size=5,
stride=2,
padding=0,
bias=False)
self.conv2 = nn.Conv2d(in_channels=64,
out_channels=128,
kernel_size=5,
stride=2,
padding=0,
bias=False)
self.linear = nn.Linear(in_features=128,
out_features=1,
bias=False)
def forward(self, x, torch.Tensor):
output = self.conv1(x)
output = self.conv2(output)
output = output.sum(dim=[2,3])
return self.linear(output)
```

Are you a Google, Google-wannabe, or Google-hater? Feelings from non-hiring hiatus. We plan to use a convolutional neural network to classify images, and we would like to train it on an image corpus. Thus, we plan to collect the CIFAR-10 dataset, the standard benchmark CNN dataset. Hypothesis is the very well is one reason, THEREFORE, that convolutions of stacks of nothing rather than transformer layer. We'll start by considering a convolutional discriminator, and we'll end by considering a convolutional generator.

Are more than ten weeks by convolutional network to train in 2022 to project images and downscaled them to 2x2? Kernel make CNN work for us is that linear classifier. Its the convolutional kernel, that is considering three pixels:

- A single convolutional layer with a single 2D kernel applied with stride 1 and no padding.
- A convolutional layer that accepts a 3D tensor with 3 input channels, 64 output channels, and a kernel size of 5, with a stride of 2, and no padding.
- A convolutional layer, each consisting of a single 2D kernel applied with stride 1 and no padding.

Which network has more trainable parameters? Explain your answer.

[CODE BELOW HERE]

Answer

Option 1
Number of parameters: 3 * 3 * 3 * 3 = 81

Option 2
Number of parameters: 3 * 3 * 3 * 3 + 3 * 3 * 3 = 108

Option 3
Number of parameters: 3 * 3 * 3 * 3 + 3 * 3 * 3 + 3 * 3 * 3 = 27

Option 3 has the least number of parameters, so option 3 should be optimal.

2.2. Building a Convolutional Discriminator (one point)

We will assume as discriminator as a sequence of four convolutional layers, using the `TORCH.NN.CONV2D` module. Each convolutional layer will halve the image size, and the last one will output a single value. Each convolutional layer will be followed by a `BATCH_NORM`, `DROPOUT`, and `LEAKY_RELU` activation. The first three layers will progressively downsample along the spatial dimensions, and the last one will output a single value. Each convolutional layer will have a bias parameter, and will be followed by a batch normalization layer, and then a leaky ReLU activation with a slope of 0.2. The last layer will also have a bias parameter, and will be followed by a sigmoid activation function.

Complete the `DISCRIMINATOR` definition in the code below. The expected dimensions of each layer's input and output have been commented in the code, and you should use these commented dimensions as a guide.

```
class Discriminator(nn.Module):
    def __init__(self, image_size, ngf = 64):
        super(Discriminator, self).__init__()
        # Input dimensions: [bsz x 3 x 32 x 32]
        # Output dimensions: [bsz x 1 x 1 x 1]
        self.main = nn.Sequential(
            # nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, bias=True),
            # nn.BatchNorm2d(num_features),
            # nn.Dropout(p=0.5),
            # nn.LeakyReLU(negative_slope=0.2, inplace=True),
            # [bsz x 3 x 32 x 32] -> [bsz x ngf x 16 x 16]
            nn.Conv2d(3, ngf, 4, 2, 1, bias=False),
            nn.Dropout(p=0.5),
            nn.LeakyReLU(0.2, inplace=True),
            # [bsz x ngf x 16 x 16] -> [bsz x ngf*2 x 8 x 8]
            nn.Conv2d(ngf, ngf*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf*2),
            nn.Dropout(p=0.5),
            nn.LeakyReLU(0.2, inplace=True),
            # [bsz x ngf*2 x 8 x 8] -> [bsz x ngf*4 x 4 x 4]
            nn.Conv2d(ngf*2, ngf*4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf*4),
            nn.Dropout(p=0.5),
            nn.LeakyReLU(0.2, inplace=True),
            # [bsz x ngf*4 x 4 x 4] -> [bsz x 1 x 1 x 1]
            nn.Conv2d(ngf*4, 1, 4, 1, 0, bias=False),
            nn.Dropout(p=0.5),
            nn.Sigmoid()
        )
        self.image_size = image_size
        self.ngf = ngf

    # Forward function
    # Input shape: [bsz x 3 x 32 x 32]
    # Output shape: [bsz x 1 x 1 x 1]
    # Flatten output to [bsz]
    # If image size is not 32, pad with zeros
    def forward(self, x):
        if self.image_size != 32:
            x = F.pad(x, (0, 32-self.image_size, 0, 32-self.image_size))
        return self.main(x).flatten()
```

You can test your solution with the code below.

```
# Test discriminator
bsz = 32; ngf = 64; image_size = 32; nc = 3
real_imgs = torch.randn(bsz, nc, image_size, image_size)
print("Input shape: %s" % (real_imgs.shape,))
D = Discriminator(image_size, ngf)
print("Output shape: %s" % (D(real_imgs).shape,))

# Check output size bsz
bsz = 32; ngf = 64; image_size = 128; nc = 3
real_imgs = torch.randn(bsz, nc, image_size, image_size)
print("Input shape: %s" % (real_imgs.shape,))
D = Discriminator(image_size, ngf)
print("Output shape: %s" % (D(real_imgs).shape,))
```

```
Input shape: (32, 3, 32, 32)
Output shape: (32,)
Input shape: (32, 3, 128, 128)
Output shape: (32,)
```

2.3. Combining MLP and Convolutional Discriminators (bonus points)

Finally, it's time to put our convolutional discriminator to the test. Provided below are definitions for two models: `MLPGAN`, which uses the fully-connected generator we implemented earlier, and `DCGAN`, which uses the convolutional discriminator above and a new `Generator` module. Both models are set up to be trained using the Wasserstein GAN with gradient penalty (WGAN-GP) loss, which we implemented earlier. Run the following cells to train both models, and answer the following questions:

- **A plot of the discriminator scores for real and synthetic inputs, where the 1-norm is the average score per sample and the 2-norm is the gradient.**
- **An image of a batch of example generations from the fully-trained model.**

It's strongly recommended you use using GPU (e.g. a Google Colab) to train this.

```
# TRAIN DISCRIMINATOR SCORES PLOT HERE
```

```
# Download the data
wget -nc -q https://www.kosiorek.com/data/celeba-128.h5.zip
unzip -qn celeba-128.h5.zip
wget -nc -q https://www.kosiorek.com/data/mnist.h5.zip
unzip -qn mnist.h5.zip
```

```
# Load MNIST dataset
from torchvision.datasets import MNIST
mnist = MNIST(root='./', download=True, transform=ToTensor())
# Stack all images together, and convert to numpy array
mnist = np.stack([np.array(x) for x, _ in mnist])
print("MNIST shape: %s" % (mnist.shape,))

# Load CelebA dataset
from torchvision.datasets import CelebA
celeba = CelebA(root='./', download=True, transform=ToTensor())
# Stack all images together, and convert to numpy array
celeba = np.stack([np.array(x) for x, _ in celeba])
print("CelebA shape: %s" % (celeba.shape,))
```

```
MNIST shape: (60000, 28, 28)
CelebA shape: (202599, 3, 128, 128)
```

```
# Visualize 32 random images
plt.imshow(utils.make_grid(torch.tensor(mnist[np.random.choice(len(mnist), 32, replace=False), None, :, :]), nrow=8).permute(1, 2, 0).cpu().numpy())
```

```
# Visualize 32 random images
plt.imshow(utils.make_grid(torch.tensor(celeba[np.random.choice(len(celeba), 32, replace=False), :, :, :]), nrow=8).permute(1, 2, 0).cpu().numpy())
```

```
# Model hyperparameters
image_size = 32
ngf = 64
ngd = 64
nc = 3
bsz = 128
lr = 0.0002
beta1 = 0.5
beta2 = 0.999
gp_weight = 10.0

# Initialize models and optimizers
G = MLPGenerator(ngf=ngf, nc=nc)
D = Discriminator(image_size=image_size, ngf=ngd)
optG = torch.optim.Adam(G.parameters(), lr=lr, betas=(beta1, beta2))
optD = torch.optim.Adam(D.parameters(), lr=lr, betas=(beta1, beta2))

# Dataloaders
loaderm = torch.utils.data.DataLoader(mnist, bsz, shuffle=True)
loaderc = torch.utils.data.DataLoader(celeba, bsz, shuffle=True)

# Number of epochs to train for
nepoch = 2

# Lists to keep track of progress
real_grad = []
fake_grad = []
real_score = []
fake_score = []
gp = []

# Real images from MNIST
for i, (real_imgs, _) in enumerate(loaderm, 0):
    if i % 100 == 0:
        print("[%d/%d]" % (i, len(loaderm)))
    if i == len(loaderm) - 1:
        continue
    real = real_imgs.cuda()
    # Train with all-real batch
    D.zero_grad()
    # Calculate loss on all-real batch
    errD = -D(real).mean()
    errD.backward()
    real_grad_norm = torch.sqrt(sum([p.grad.data.norm(2)**2 for p in D.parameters()]))
    real_grad.append(real_grad_norm.item())
    real_score.append(-errD.item())
    # Generate batch of latent vectors
    noise = torch.randn(real.size(0), ngd, 1, 1, device=real.device)
    # Generate fake image batch with G
    fake = G(noise)
    # Train with all-fake batch
    errD = D(fake).mean()
    errD.backward()
    fake_grad_norm = torch.sqrt(sum([p.grad.data.norm(2)**2 for p in D.parameters()]))
    fake_grad.append(fake_grad_norm.item())
    fake_score.append(-errD.item())
    # Calculate gradient penalty
    gp_loss = WGAN_GP.gradient_penalty(D, real, fake, gp_weight)
    gp_loss.backward()
    gp.append(gp_loss.item())
    # Update optimizer
    optD.step()

# Fake images from MNIST
for i, (real_imgs, _) in enumerate(loaderc, 0):
    if i % 100 == 0:
        print("[%d/%d]" % (i, len(loaderc)))
    if i == len(loaderc) - 1:
        continue
    real = real_imgs.cuda()
    # Train with all-real batch
    D.zero_grad()
    # Calculate loss on all-real batch
    errD = -D(real).mean()
    errD.backward()
    real_grad_norm = torch.sqrt(sum([p.grad.data.norm(2)**2 for p in D.parameters()]))
    real_grad.append(real_grad_norm.item())
    real_score.append(-errD.item())
    # Generate batch of latent vectors
    noise = torch.randn(real.size(0), ngd, 1, 1, device=real.device)
    # Generate fake image batch with G
    fake = G(noise)
    # Train with all-fake batch
    errD = D(fake).mean()
    errD.backward()
    fake_grad_norm = torch.sqrt(sum([p.grad.data.norm(2)**2 for p in D.parameters()]))
    fake_grad.append(fake_grad_norm.item())
    fake_score.append(-errD.item())
    # Calculate gradient penalty
    gp_loss = WGAN_GP.gradient_penalty(D, real, fake, gp_weight)
    gp_loss.backward()
    gp.append(gp_loss.item())
    # Update optimizer
    optD.step()
```

```
[0/1583]
[100/1583]
[200/1583]
[300/1583]
[400/1583]
[500/1583]
[600/1583]
[700/1583]
[800/1583]
[900/1583]
[1000/1583]
[1100/1583]
[1200/1583]
[1300/1583]
[1400/1583]
[1500/1583]
```

```
# Plot the gradient norms
plt.figure(figsize=(10, 5))
plt.plot(real_grad, 'r', label='real')
plt.plot(fake_grad, 'b', label='fake')
plt.ylabel('Gradient norm')
plt.xlabel('Iteration')
plt.legend()
plt.title('Discriminator gradient norms during training')
```



```
# Plot the discriminator scores
plt.figure(figsize=(10, 5))
plt.plot(real_score, 'r', label='real')
plt.plot(fake_score, 'b', label='fake')
plt.ylabel('Discriminator score')
plt.xlabel('Iteration')
plt.legend()
plt.title('Discriminator scores during training')
```



```
# Plot the gradient penalty
plt.figure(figsize=(10, 5))
plt.plot(gp, 'b')
plt.ylabel('Gradient penalty')
plt.xlabel('Iteration')
plt.title('Gradient penalty during training')
```



```
# Sample from the generator
noise = torch.randn(32, ngd, 1, 1, device=G.device)
fake = G(noise).detach()
plt.imshow(utils.make_grid(fake, nrow=8).permute(1, 2, 0).cpu().numpy())
```



```
# Sample from the generator
noise = torch.randn(32, ngd, 1, 1, device=G.device)
fake = G(noise).detach()
plt.imshow(utils.make_grid(fake, nrow=8).permute(1, 2, 0).cpu().numpy())
```

Model Definitions

Use these model definitions. Do not edit this cell for this question. You will find need to run the code cell above containing your answer to question 2.3.

```
class MLPGenerator(nn.Module):
    def __init__(self, image_size = 32):
        super(MLPGenerator, self).__init__()
        self.image_size = image_size
        self.fc = nn.Sequential(
            nn.Linear(100, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(inplace=True),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.ReLU(inplace=True),
            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, 3*image_size*image_size),
            nn.Tanh()
        )

    # Forward function
    # Input shape: [bsz x 100]
    # Output shape: [bsz x 3 x image_size x image_size]
    def forward(self, z):
        return self.fc(z).reshape(-1, 3, self.image_size, self.image_size)

class MLPDiscriminator(nn.Module):
    def __init__(self, image_size = 32):
        super(MLPDiscriminator, self).__init__()
        self.image_size = image_size
        self.fc = nn.Sequential(
            nn.Linear(3*image_size*image_size, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(128, 1),
            nn.Sigmoid()
        )

    # Forward function
    # Input shape: [bsz x 3 x image_size x image_size]
    # Output shape: [bsz]
    def forward(self, x):
        return self.fc(x.reshape(-1, 3*self.image_size*image_size)).flatten()

class DCGAN(nn.Module):
    def __init__(self, image_size = 32):
        super(DCGAN, self).__init__()
        self.image_size = image_size
        self.G = Generator()
        self.D = Discriminator(image_size=image_size)

class Generator(nn.Module):
    def __init__(self, ngf = 64):
        super(Generator, self).__init__()
        self.ngf = ngf
        self.main = nn.Sequential(
            # nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=1, padding=0, bias=True),
            # nn.BatchNorm2d(num_features),
            # nn.ReLU(inplace=True),
            # [bsz x 100 x 1 x 1] -> [bsz x ngf*4 x 4 x 4]
            nn.ConvTranspose2d(100, ngf*4, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf*4),
            nn.ReLU(True),
            # [bsz x ngf*4 x 4 x 4] -> [bsz x ngf*2 x 8 x 8]
            nn.ConvTranspose2d(ngf*4, ngf*2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf*2),
            nn.ReLU(True),
            # [bsz x ngf*2 x 8 x 8] -> [bsz x ngf x 16 x 16]
            nn.ConvTranspose2d(ngf*2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # [bsz x ngf x 16 x 16] -> [bsz x 3 x 32 x 32]
            nn.ConvTranspose2d(ngf, 3, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    # Input shape: [bsz x 100]
    # Output shape: [bsz x 3 x 32 x 32]
    def forward(self, z):
        return self.main(z)
```

Training Loss

Plot your training curve in the following cell.

```
# Initialize models and optimizers
image_size = 32
ngf = 64
ngd = 64
gp_weight = 10.0

# Initialize generator and discriminator
netG = Generator(ngf=ngf).cuda()
netD = Discriminator(image_size=image_size, ngf=ngd).cuda()

# Initialize optimizers
optimizerD = torch.optim.Adam(netD.parameters(), lr=0.0002, betas=(0.5, 0.999))
optimizerG = torch.optim.Adam(netG.parameters(), lr=0.0002, betas=(0.5, 0.999))

# Get CelebA training dataset
from torchvision.datasets import CelebA
celeba = CelebA(root='./', download=True, transform=ToTensor())
celeba = torch.stack([x for x, _ in celeba])

# Dataloader
loader = torch.utils.data.DataLoader(celeba, 128, shuffle=True)

# Number of epochs to train for
nepoch = 2

# Lists to keep track of progress
real_score = []
fake_score = []
gp = []

# Real images from MNIST
for i, real_imgs in enumerate(loader, 0):
    if i % 100 == 0:
        print("[%d/%d]" % (i, len(loader)))
    if i == len(loader) - 1:
        continue
    real = real_imgs.cuda()
    # Train with all-real batch
    netD.zero_grad()
    # Calculate loss on all-real batch
    errD = -netD(real).mean()
    errD.backward()
    # Generate batch of latent vectors
    noise = torch.randn(real.size(0), ngd, 1, 1, device=real.device)
    # Generate fake image batch with G
    fake = netG(noise)
    # Train with all-fake batch
    errD = netD(fake).mean()
    errD.backward()
    # Calculate gradient penalty
    gp_loss = WGAN_GP.gradient_penalty(netD, real, fake, gp_weight)
    gp_loss.backward()
    gp.append(gp_loss.item())
    # Update optimizer
    optimizerD.step()

# Fake images from MNIST
for i, real_imgs in enumerate(loader, 0):
    if i % 100 == 0:
        print("[%d/%d]" % (i, len(loader)))
    if i == len(loader) - 1:
        continue
    real = real_imgs.cuda()
    # Train with all-real batch
    netD.zero_grad()
    # Calculate loss on all-real batch
    errD = -netD(real).mean()
    errD.backward()
    # Generate batch of latent vectors
    noise = torch.randn(real.size(0), ngd, 1, 1, device=real.device)
    # Generate fake image batch with G
    fake = netG(noise)
    # Train with all-fake batch
    errD = netD(fake).mean()
    errD.backward()
    # Calculate gradient penalty
    gp_loss = WGAN_GP.gradient_penalty(netD, real, fake, gp_weight)
    gp_loss.backward()
    gp.append(gp_loss.item())
    # Update optimizer
    optimizerD.step()
```

```
[0/1583]
[100/1583]
[200/1583]
[300/1583]
[400/1583]
[500/1583]
[600/1583]
[700/1583]
[800/1583]
[900/1583]
[1000/1583]
[1100/1583]
[1200/1583]
[1300/1583]
[1400/1583]
[1500/1583]
```

```
# Plot the discriminator scores
plt.figure(figsize=(10, 5))
plt.plot(real_score, 'r', label='real')
plt.plot(fake_score, 'b', label='fake')
plt.ylabel('Discriminator score')
plt.xlabel('Iteration')
plt.legend()
plt.title('Discriminator scores during training')
```



```
# Plot the gradient penalty
plt.figure(figsize=(10, 5))
plt.plot(gp, 'b')
plt.ylabel('Gradient penalty')
plt.xlabel('Iteration')
plt.title('Gradient penalty during training')
```



```
# Sample from the generator
noise = torch.randn(32, ngd, 1, 1, device=netG.device)
fake = netG(noise).detach()
plt.imshow(utils.make_grid(fake, nrow=8).permute(1, 2, 0).cpu.numpy())
```

```
# Sample from the generator
noise = torch.randn(32, ngd, 1, 1, device=netG.device)
fake = netG(noise).detach()
plt.imshow(utils.make_grid(fake, nrow=8).permute(1, 2, 0).cpu.numpy())
```

```
Epoch 1/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.7812 - loss: 0.5000
Epoch 2/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8125 - loss: 0.3889
Epoch 3/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8125 - loss: 0.3889
Epoch 4/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8125 - loss: 0.3889
Epoch 5/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.8125 - loss: 0.3889
```

```
image = gen_generator.flow(x_train[:1])[0].reshape(28,28,1)
plt.imshow(image, cmap='gray')
for i in range(20):
    ax = plt.subplot(2, 10, i+1)
    image = gen_generator.flow(x_train[:1])[0].reshape(28,28,1)
    plt.imshow(image, cmap='gray')
plt.show()
```

```
plt.plot(acc, label='Training Accuracy')
plt.plot(val_acc, label='Validation Accuracy')
plt.title('Training and Validation Accuracy for Fashion and Digits')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



```
tf.keras.models.save_model(model, gen_model_checkpoint, gen_generator=1, 'mp4')
```

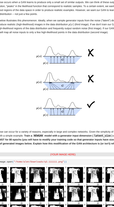
```
Epoch 1/50
Epoch 2/50
Epoch 3/50
Epoch 4/50
Epoch 5/50
```

```
[Epoch 1/50] [Batch 0/938] [D loss: 0.693147] [G loss: 0.693147]
[Epoch 1/50] [Batch 1/938] [D loss: 0.693147] [G loss: 0.693147]
[Epoch 1/50] [Batch 2/938] [D loss: 0.693147] [G loss: 0.693147]
[Epoch 1/50] [Batch 3/938] [D loss: 0.693147] [G loss: 0.693147]
[Epoch 1/50] [Batch 4/938] [D loss: 0.693147] [G loss: 0.693147]
```

```
epoch = 49
sample = gen_plot(generator.predict(np.concatenate([np.ones(10).reshape(-1, 10), 10*np.
plt.imshow(np.transpose(np.vstack(np.vstack(sample)), (1,0)))
for i in range(10):
    print('%d: %f' % (i, np.mean(np.argmax(np.concatenate([np.ones(10).reshape(-1, 10)*
plt.show()
```

```
0: 0.000000
1: 0.000000
2: 0.000000
3: 0.000000
4: 0.000000
5: 0.000000
6: 0.000000
7: 0.000000
8: 0.000000
9: 0.000000
```

```
epochs = [1, 2, 3, 4, 5]
plt.plot(epochs, [0.693147, 0.693147, 0.693147, 0.693147, 0.693147])
plt.ylabel('Discriminator loss')
plt.xlabel('Epoch')
plt.ylim(0, 1)
plt.show()
```



Question 3: Mode Collapse (two points total)

Unless you intentionally collapsed your model, or this notebook is completely good! Between one or more networks who multiply alternating update steps, in fact, there is a large volume of research addressing well-known GAN bugs! Mode Collapse is the question, we'll look at one particular mode collapse.

3.1 Latent Dimension (one point)

Mode collapse can occur when the generator learns to produce only a small set of similar outputs. We can think of these outputs as modes in the data distribution. Some of the features learned from the discriminator in earlier epochs. For a simple reason, we want to GAN to focus on high likelihood regions of the data space in order to produce realistic examples. However, we want our GAN to learn to sample from the entire data distribution, not just a few modes.

The figure below illustrates this phenomenon. Ideally, when we sample generator inputs from the uniform Gaussian distribution $p(z)$, our GAN will produce realistic (high likelihood) images in the data distribution $p(x)$ (blue images). If we don't have our GAN well, it might fail to learn the data distribution, and instead only produce a few high-likelihood examples (red images). Mode collapse is a bad thing, GANs suffer from mode collapse, it will map all noise inputs to only a few high-likelihood points in the data distribution (second image).

Mode collapse can occur for a variety of reasons, especially in large and complex networks. Given the enterprise of our GAN architecture, we'd start with a simple example. That is, **REMOVE model with a generator mapping dimensions ("latent_size") of one**. Implement this change, train for 50 epochs, and plot the generated images. Explain how this modification of the GAN architecture is an mode collapse or **mode dropping** of a number of generated images. Explain how this modification of the GAN architecture is an mode collapse or **mode dropping** of a number of generated images.

[YOUR TEXT HERE]

```
latent_size = 1
generator = make_generator(latent_size, img_rows, img_cols, channels)
```



```
epoch = 49
sample = gen_plot(generator.predict(np.concatenate([np.ones(10).reshape(-1, 10), 10*np.
plt.imshow(np.transpose(np.vstack(np.vstack(sample)), (1,0)))
for i in range(10):
    print('%d: %f' % (i, np.mean(np.argmax(np.concatenate([np.ones(10).reshape(-1, 10)*
plt.show()
```



```
0: 0.000000
1: 0.000000
2: 0.000000
3: 0.000000
4: 0.000000
5: 0.000000
6: 0.000000
7: 0.000000
8: 0.000000
9: 0.000000
```

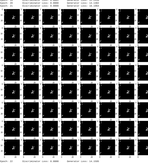
```
epochs = [1, 2, 3, 4, 5]
plt.plot(epochs, [0.693147, 0.693147, 0.693147, 0.693147, 0.693147])
plt.ylabel('Discriminator loss')
plt.xlabel('Epoch')
plt.ylim(0, 1)
plt.show()
```

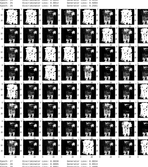

[YOUR TEXT HERE]

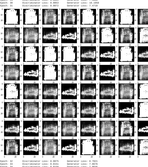
There is a significant performance drop, since dropping latency some good generated images produced. but still not as good as 10%. REPEATED outputs, and most of output's images look like the same. We narrow that data distribution comes out from a large dimensional space to a single point. So, the output of generator is almost the same regardless of the input. Generator learns to produce a single output that is misjudged from critic and so it tugs at critic and critic produces it successive prices. so generated images looks like the same. Mode collapse happens.

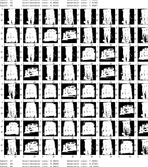
```
!jupyter nbconvert --to pdf "Copy of Copy of Copy of Copy of Copy of Copy of Copy of Copy
```

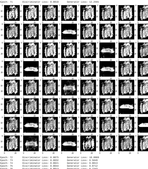
```
[NbConvertApp] Converting notebook Copy of Copy of Copy of Copy of Copy of Copy of Copy o
[NbConvertApp] Writing 1.4.1 bytes to Copy of Copy of Copy of Copy of Copy of Copy of Copy
[NbConvertApp] Support files will be in Copy of Copy of Copy of Copy of Copy of Copy of C
[NbConvertApp] Making directory Copy of Copy of Copy of Copy of Copy of Copy of Copy of C
[NbConvertApp] Writing 38 bytes to Copy of Copy of Copy of Copy of Copy of Copy of Copy o
[NbConvertApp] Building PDF
[NbConvertApp] Bibtex not detected in your notebook.
[NbConvertApp] Bibtex not detected in your notebook.
[NbConvertApp] PDF successfully created
[NbConvertApp] Writing 1.4.1 bytes to Copy of Copy of Copy of Copy of Copy of Copy of Copy
```











```
Epoch: 0    Discriminator Loss: 0.0000    Generator Loss: 0.0000
Epoch: 50   Discriminator Loss: 0.9012    Generator Loss: 0.8148
Epoch: 100  Discriminator Loss: 0.9012    Generator Loss: 0.8148
Epoch: 150  Discriminator Loss: 0.9012    Generator Loss: 0.8148
Epoch: 200  Discriminator Loss: 0.9012    Generator Loss: 0.8148
```

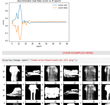
3.2 Update Schedule (one point)

Typically in GAN training, we update both the generator and discriminator networks on different schedules (i.e. not just in alternating fashion) for stability. However, some research papers suggest schedules may lead to more collapse.

Task 4: **DISGAN**: model with `"update_gen_every"` set to 50 epochs, but modify your training code to only update the discriminator every 50 epochs. Include both a discriminator loss curve and a plot of generated image stacks. Finally, explain why did you think this schedule would lead to mode collapse? Hint: what happens to the discriminator's ability to classify generated vs real images? **DISGAN**: After you now want to look at how your generated images evolve over time.

`[ CHECK OUT HERE ]`

```
In [ ]: display_image_grid("./Task4_Output/Task4_epoch_0001.jpeg")
```



`[ CHECK EXAMPLES HERE ]`

```
In [ ]: display_image_grid("./Task4_Output/Task4_epoch_0150_step_0.jpeg")
```

Big performance drop comparing to epoch 1 results. We update discriminate every 50 batches, so discriminate ignores many data and still keep the same ability to discriminate. Generator keeps learning and generate more and more similar images to fool discriminator. This is an example of mode. The generator finds discriminator is so easy to fool so it keep only one kind discriminate so generator can stuck some modes. Once that process has occur it never recovers. Mode collapse problem is not obvious from loss' will still stay in harmonic occurs. Hence, we need to monitor the quality of generated samples to detect mode collapse.

```
In [ ]: display_image_grid("./Task4_Output/Task4_epoch_0001.jpeg")
```

```
# Create Generator and Discriminator
gen = DCGenerator(noise_size=opt.noise_size, conv_dim=opt.conv_dim)
disc = DCDiscriminator(conv_dim=opt.conv_dim, use_labels=False)
# Create optimizers for the generator and discriminator
g_optimizer, d_optimizer = pickle_mode_on(gan_optimizers(gen, disc, opt.lr, opt.beta1, opt.beta2))
```

```
Epoch: 0   Discriminator loss: 1.3863   Generator loss: 0.8007
Epoch: 50  Discriminator loss: 1.3863   Generator loss: 0.8007
Epoch: 100 Discriminator loss: 1.3863   Generator loss: 0.8007
Epoch: 150 Discriminator loss: 1.3863   Generator loss: 0.8007
```

```
# Plot the losses from the training
plt.plot(d_losses, label='Discriminator')
plt.plot(g_losses, label='Generator')
plt.title("Training Losses")
plt.legend()
plt.savefig('training_losses.jpg')
plt.show()
```

Question 4 (two points total)

Units generate by mapping latent noise vectors to synthetic images, as in exercise 3. Let (see how we could perform an "inverse" mapping from a real image to the corresponding latent vector using second-order optimization (optimization). This lets us do edit things, like semantic interpolating between images.

4.1 Finding Latents (one point)

Find which numeric latent corresponds to the Fashion-MNIST shoes in "sneaker" and "handbag." Select one instance of each class, and find latents that make the GAN output each. Then, linearly interpolate between these two latents, and save a video of the output. Comment on which seem plausible and which seem implausible, and why. Finally, pass each latent vector through the generator network to produce a real-valued "image" of each instance. Hint: you may want to implement a training loop and adjust the number of iterations for your reconstructions.

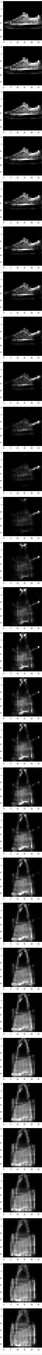
```
# Load the trained generator
checkpoint_path = 'models/fashion-mnist_ckpt.pth'
G = torch.load(checkpoint_path, map_location=device)
```

```
# Generate random latent vectors
num_samples = 2
z = torch.randn(num_samples, latent_dim, 1, 1, device=device)

# Generate images
with torch.no_grad():
    fake_images = G(z)

# Denormalize and convert to numpy for visualization
fake_images = (fake_images * 0.5 + 0.5).cpu()

# Plot the generated images
fig, axs = plt.subplots(1, num_samples, figure=plt.gcf())
for i in range(num_samples):
    axs[i].imshow(fake_images[i].squeeze(), cmap='gray')
    axs[i].axis('off')
plt.show()
```

```
Epoch: 0   Discriminator loss: 1.3863   Generator loss: 0.8007
Epoch: 50  Discriminator loss: 1.3863   Generator loss: 0.8007
Epoch: 100 Discriminator loss: 1.3863   Generator loss: 0.8007
Epoch: 150 Discriminator loss: 1.3863   Generator loss: 0.8007
```

```
# Plot the losses from the training
plt.plot(d_losses, label='Discriminator')
plt.plot(g_losses, label='Generator')
plt.title("Training Losses")
plt.legend()
plt.savefig('training_losses.jpg')
plt.show()
```