```
### Assignment 4 Generative Adversarial Nets (Unconditional, 10 pts)


In this exercise, we will implement a Generative Adversarial Net (GAN), specifically, a Wasserstein GAN and train it on the M

**Submit**
1. (<font color='red'>Doc A</font>) Include the two figures at the end in the pdf generated by the latex file with Exercise 2
2. (<font color='red'>Doc B</font>) The completed *.ipynb file with all the command outputs (can be created by saving the fil
```

Setup

1. In Colab, open tab Runtime > Change runtime type, choose *python3* and *T4 GPU*.
2. Run the following command to set up the environment. (Takes ~ 1.5 min)

```
! pip install --quiet "ipython[notebook]==7.34.0, <8.17.0" "setuptools>=68.0.0, <68.3.0"  "torch==1.13.0" "matplotlib"  "torc
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 807.9/807.9 kB 5.2 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 890.1/890.1 MB 1.2 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 49.0 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 849.3/849.3 kB 45.7 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 557.1/557.1 MB 1.9 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 317.1/317.1 MB 4.5 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.0/21.0 MB 47.8 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.0/7.0 MB 71.8 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.0/7.0 MB 78.3 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.9/6.9 MB 72.8 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.9/6.9 MB 68.8 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.9/6.9 MB 72.5 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.8/6.8 MB 67.2 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.8/6.8 MB 73.3 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.9/6.9 MB 15.1 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.0/6.0 MB 89.2 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.0/6.0 MB 83.6 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.2/24.2 MB 58.5 MB/s eta 0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.3/24.3 MB 58.5 MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is
torchaudio 2.3.1+cu121 requires torch==2.3.1, but you have torch 1.13.0 which is incompatible.
torchtext 0.18.0 requires torch>=2.3.0, but you have torch 1.13.0 which is incompatible.
```

Let's start with importing our standard set of libraries.

```
import torch
from torch import nn, optim, autograd
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import torchvision.utils as vutils
from dataclasses import dataclass
import time
import sys
%matplotlib inline
torch.set_num_threads(1)
torch.manual_seed(1)


device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")

if device == torch.device("cuda:0"):
  print('Everything looks good; continue')
else:
  # It is OK if you cannot connect to a GPU. In this case, training the model for
  # 2 epoch is sufficient to get full mark. (NOTE THAT 2 epoch takes approximately 1.5 hours to train for CPU)
  print('GPU is not detected. Make sure you have chosen the right runtime type')
```

```
Everything looks good; continue
```

Dataloaders and hyperparameters (0 pt)

```python
@dataclass
class Hyperparameter:
    batchsize: int         = 64
    num_epochs: int        = 5
    latent_size: int       = 32
    n_critic: int          = 5
    critic_size: int       = 1024
    generator_size: int    = 1024
    critic_hidden_size: int = 1024
    gp_lambda: float       = 10.

hp = Hyperparameter()

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

dataset  = torchvision.datasets.MNIST("mnist", download=True, transform=transform)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=hp.batchsize, num_workers=1, shuffle=True, drop_last=True, pin_m
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyt
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyt
100%    9912422/9912422 [00:00<00:00, 14971358.60it/s]
Extracting mnist/MNIST/raw/train-images-idx3-ubyte.gz to mnist/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyt
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyt
100%    28881/28881 [00:00<00:00, 422290.41it/s]
Extracting mnist/MNIST/raw/train-labels-idx1-ubyte.gz to mnist/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte
100%    1648877/1648877 [00:00<00:00, 3850616.34it/s]
Extracting mnist/MNIST/raw/t10k-images-idx3-ubyte.gz to mnist/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
HTTP Error 403: Forbidden

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte
100%    4542/4542 [00:00<00:00, 143919.87it/s]
Extracting mnist/MNIST/raw/t10k-labels-idx1-ubyte.gz to mnist/MNIST/raw
```

Building Models (2 pts)

After examining the preprocessing steps, we can now start building the models, including the generator for generating new images from random noise, and a critic of the realness of the image.

In this assignment we adopt the implementation of [DCGAN](), which is a direct extension of [GAN](), with convolutional and convolutional-transpose layers in the critic and genrator, respectively. Specifically, we will use the [ConvTranspose2d]() layers to upscale the noise.

Moreover, we apply an improved version of [Wasserstein-GAN]() with a [Gradient Penalty]() (you may read Algorithm 1 to fully understand the code we are implementing).

```
# Define the generator

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

        #VVVVVVVVVVVV TO BE COMPLETE (START) VVVVVVVVVVVV
        # Add latent embedding layer to adjust the dimension of the input (1 pt)

        # Hint: you should use the hyperparameters defined above

        self.latent_embedding = nn.Sequential(nn.Linear(hp.latent_size , hp.generator_size ))

        # ^^^^^^^^^^^^ TO BE COMPLETE (END) ^^^^^^^^^^^^


        # Transposed CNN layers to transfer noise to image

        self.tcnn = nn.Sequential(
        # input is Z, going into a convolution
        nn.ConvTranspose2d(hp.generator_size, hp.generator_size, kernel_size=4, stride=1, padding= 0),
        nn.BatchNorm2d(hp.generator_size),
        nn.ReLU(inplace=True),
        # upscaling
        nn.ConvTranspose2d(hp.generator_size, hp.generator_size // 2, 3, 2, 1),
        nn.BatchNorm2d(hp.generator_size // 2),
        nn.ReLU(inplace=True),
        # upscaling
        nn.ConvTranspose2d(hp.generator_size // 2, hp.generator_size // 4, 4, 2, 1),
        nn.BatchNorm2d(hp.generator_size // 4),
        nn.ReLU(inplace=True),
        nn.ConvTranspose2d(hp.generator_size // 4, 1, 4, 2, 1),
        nn.Tanh()
        )


    def forward(self, latent):
        vec_latent = self.latent_embedding(latent).reshape(-1, hp.generator_size, 1, 1)
        return self.tcnn(vec_latent)


# Define the critic

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

        # CNN layers that perform downscaling
        self.cnn_net = nn.Sequential(
        nn.Conv2d(1, hp.critic_size // 4, 3, 2),
        nn.InstanceNorm2d(hp.critic_size // 4, affine=True),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Conv2d(hp.critic_size // 4, hp.critic_size // 2, 3, 2),
        nn.InstanceNorm2d(hp.critic_size // 2, affine=True),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Conv2d(hp.critic_size // 2, hp.critic_size, 3, 2),
        nn.InstanceNorm2d(hp.critic_size, affine=True),
        nn.LeakyReLU(0.2, inplace=True),
        nn.Flatten(),
        )

        # Linear layers that produce the output from the features
        self.critic_net = nn.Sequential(
        nn.Linear(hp.critic_size * 4, hp.critic_hidden_size),
        nn.LeakyReLU(0.2, inplace=True),

        #VVVVVVVVVVVV TO BE COMPLETE (START) VVVVVVVVVVVV
        # Add the last layer to reflect the output (1 pt)

        nn.Linear(hp.critic_hidden_size  ,1)

        # Hint: Given an image, the output of the critic is a value (or a scalar)

        # ^^^^^^^^^^^^ TO BE COMPLETE (END) ^^^^^^^^^^^^
        )

    def forward(self, image):
```

```
        cnn_features = self.cnn_net(image)
        return self.critic_net(cnn_features)

```

Before Training

Next we define the two models and the optimizers. We use the [AdamW](https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html) algorithm.

```
critic, generator = Critic().to(device), Generator().to(device)

critic_optimizer = optim.AdamW(critic.parameters(), lr=1e-4,betas=(0., 0.9))
generator_optimizer = optim.AdamW(generator.parameters(), lr=1e-4,betas=(0., 0.9))
```

Training pipeline (6 points)

Finally, we perform training on the two networks. The training consists of two steps: (1) Updating discriminators for n_critic steps (such that we have an optimal critic): here we use an aggregation of three loss functions, (a) The real loss (the output scalar of the critic for real images); (b) The fake loss (same value for fake images); (c) The [gradient penalty](https://arxiv.org/abs/1704.00028). (2) Updating generators by only considering the fake loss (to fool the critic).

```
img_list, generator_losses, critic_losses = [], [], []
iters = 0
fixed_noise = torch.randn((64, hp.latent_size), device=device)
grad_tensor = torch.ones((hp.batchsize, 1), device=device)
start_time = time.time()

# ref : https://www.youtube.com/watch?v=ILpC3b-819Q
def loss_fn(y_pred, y_true):
    return -torch.mean(y_pred * y_true)

for epoch in range(hp.num_epochs):
    for batch_idx, data in enumerate(dataloader, 0):
        real_images = data[0].to(device)

        real_labels = torch.ones(hp.batchsize, device=device) # real label = 1

        # Update Critic
        critic_optimizer.zero_grad()

        # (a) Real loss
        critic_output_real = critic(real_images)
        critic_loss_real  = loss_fn(critic_output_real, real_labels)  #changed critic_output_real.mean() #

        # (b) Fake loss

        #VVVVVVVVVVV TO BE COMPLETE (START) VVVVVVVVVVV
        # Implement the fake loss

        # (1) Generating a noise tensor (of dimension (batch_size, latent_size)), you are required to
        # use the hyperparameters in the hp class (0.5 pt)

        noise = torch.randn((hp.batchsize , hp.latent_size) , device = device)

        # (2) Generate fake images using the generator (hint: you are not supposed to perform gradient
        # update on the generator) (1.5 pts)


        fake_image = generator.forward(noise)
        fake_labels = -real_labels # fake label = -1
        flipped_fake_labels = real_labels # here, fake label = 1

        # (3) Calculate the fake loss using the output of the generator (1 pt)
        critic_output_fake = critic(fake_image.detach())
        critic_loss_fake = loss_fn(critic_output_fake, fake_labels) #critic_output_fake.mean()  #

        # ^^^^^^^^^^^^ TO BE COMPLETE (END) ^^^^^^^^^^^^

        #  (c) Gradient penalty
        alpha = torch.rand((hp.batchsize, 1, 1, 1), device=device)
        interpolates = (alpha * real_images + ((1. - alpha) * fake_image)).requires_grad_(True)
```

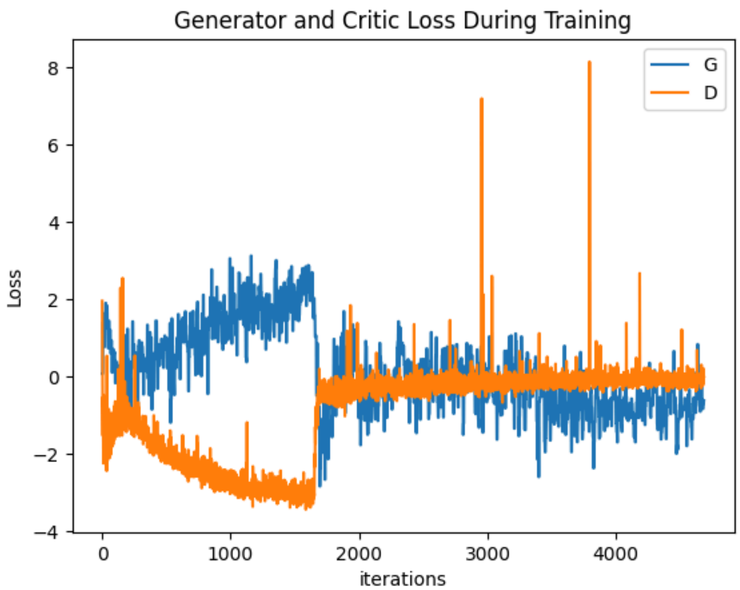
```
        d_interpolates = critic(interpolates)
        gradients = autograd.grad(outputs=d_interpolates, inputs = interpolates, grad_outputs=grad_tensor, create_graph=True,
        gradient_penalty = hp.gp_lambda * ((gradients.view(hp.batchsize, -1).norm(dim=1) - 1.) ** 2).mean()

        #VVVVVVVVVVV TO BE COMPLETE (START) VVVVVVVVVVV
        # Implement the aggregated loss using the above three components, be careful with the signs (1 pt)

        critic_loss = 0.5*(critic_loss_real + critic_loss_fake ) + gradient_penalty

        # ^^^^^^^^^^^^ TO BE COMPLETE (END) ^^^^^^^^^^^

        critic_loss.backward()
        critic_optimizer.step()

        if batch_idx % hp.n_critic == 0:
            # Update Generator
            generator_optimizer.zero_grad()


            #VVVVVVVVVVV TO BE COMPLETE (START) VVVVVVVVVVV
            # Implement the generator loss (2 pts)

            noise = torch.randn((hp.batchsize , hp.latent_size) , device = device)
            fake_image =  generator.forward(noise)
            critic_output_fake = critic(fake_image)
            generator_loss = loss_fn(critic_output_fake, flipped_fake_labels)  #critic_output_fake.mean() #

            # ^^^^^^^^^^^^ TO BE COMPLETE (END) ^^^^^^^^^^^^

            generator_loss.backward()
            generator_optimizer.step()

        # Output training stats
        if batch_idx % 100 == 0:
            elapsed_time = time.time() - start_time
            print(f"[{epoch:>2}/{hp.num_epochs}][{iters:>7}][{elapsed_time:8.2f}s]\t"
                  f"d_loss/g_loss: {critic_loss.item():4.2}/{generator_loss.item():4.2}\t")

        # Save Losses for plotting later
        generator_losses.append(generator_loss.item())
        critic_losses.append(critic_loss.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 500 == 0) or ((epoch == hp.num_epochs - 1) and (batch_idx == len(dataloader) - 1)):
            with torch.no_grad(): fake_images = generator(fixed_noise).cpu()
            img_list.append(vutils.make_grid(fake_images, padding=2, normalize=True))

        iters += 1
```

```
[ 0/5][      0][    6.25s]      d_loss/g_loss:  2.0/0.08
[ 0/5][    100][   24.37s]      d_loss/g_loss: -1.3/-0.16
[ 0/5][    200][   42.92s]      d_loss/g_loss: -1.2/0.44
[ 0/5][    300][   62.12s]      d_loss/g_loss: -1.4/0.71
[ 0/5][    400][   81.73s]      d_loss/g_loss: -1.9/-0.35
[ 0/5][    500][  101.16s]      d_loss/g_loss: -2.1/ 1.0
[ 0/5][    600][  120.31s]      d_loss/g_loss: -2.1/0.47
[ 0/5][    700][  139.48s]      d_loss/g_loss: -2.5/ 1.5
[ 0/5][    800][  158.67s]      d_loss/g_loss: -2.1/ 1.5
[ 0/5][    900][  178.03s]      d_loss/g_loss: -2.7/ 0.9
[ 1/5][    937][  185.35s]      d_loss/g_loss: -2.6/ 0.7
[ 1/5][   1037][  204.60s]      d_loss/g_loss: -2.9/ 1.1
[ 1/5][   1137][  223.80s]      d_loss/g_loss: -3.0/ 1.4
[ 1/5][   1237][  242.99s]      d_loss/g_loss: -3.0/ 2.2
[ 1/5][   1337][  262.22s]      d_loss/g_loss: -3.1/ 1.0
[ 1/5][   1437][  281.38s]      d_loss/g_loss: -2.9/ 2.3
[ 1/5][   1537][  300.73s]      d_loss/g_loss: -3.1/ 1.9
[ 1/5][   1637][  319.96s]      d_loss/g_loss: -3.0/ 1.9
[ 1/5][   1737][  339.24s]      d_loss/g_loss: -0.38/-1.1
[ 1/5][   1837][  358.49s]      d_loss/g_loss: -0.19/-0.33
[ 2/5][   1874][  365.78s]      d_loss/g_loss: -0.43/0.21
[ 2/5][   1974][  385.03s]      d_loss/g_loss: -0.56/0.21
[ 2/5][   2074][  404.31s]      d_loss/g_loss: -0.35/-0.18
[ 2/5][   2174][  423.50s]      d_loss/g_loss: -0.27/-0.44
[ 2/5][   2274][  442.71s]      d_loss/g_loss: -0.48/-0.14
[ 2/5][   2374][  461.92s]      d_loss/g_loss: -0.0017/0.085
[ 2/5][   2474][  481.16s]      d_loss/g_loss: -0.17/-1.1
[ 2/5][   2574][  500.45s]      d_loss/g_loss: -0.27/-0.64
[ 2/5][   2674][  519.71s]      d_loss/g_loss: -0.26/0.17
[ 2/5][   2774][  538.92s]      d_loss/g_loss: 0.028/ 0.4
[ 3/5][   2811][  546.22s]      d_loss/g_loss: -0.033/-1.2
```

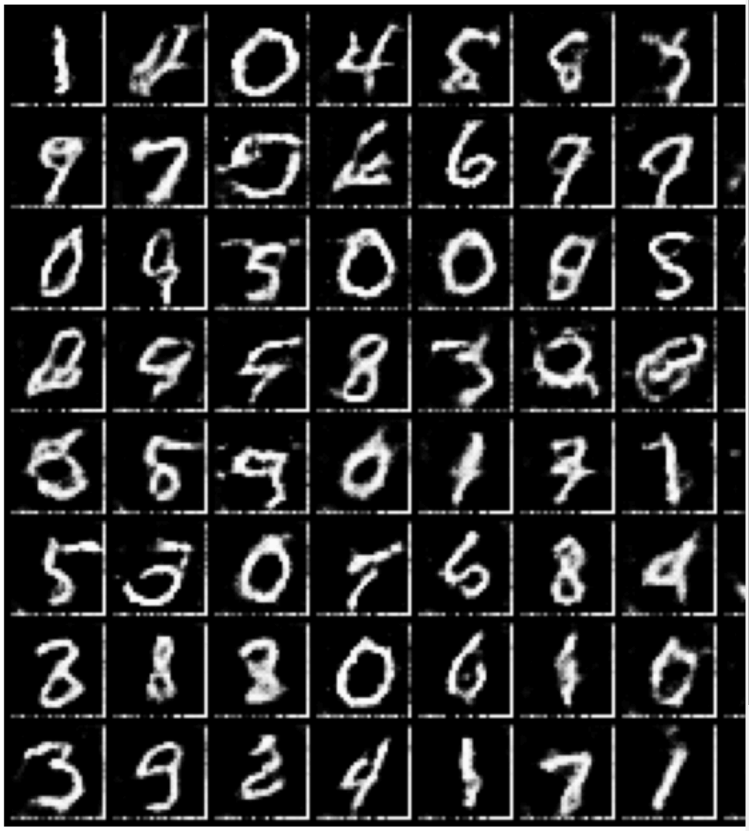
```
[ 3/5][   2911][  565.48s]      d_loss/g_loss: -0.33/-0.23
[ 3/5][   3011][  584.79s]      d_loss/g_loss: -0.026/ 1.0
[ 3/5][   3111][  604.07s]      d_loss/g_loss: -0.42/-0.98
[ 3/5][   3211][  623.32s]      d_loss/g_loss: 0.14/-0.1
[ 3/5][   3311][  642.55s]      d_loss/g_loss: -0.3/ 0.6
[ 3/5][   3411][  661.78s]      d_loss/g_loss: -0.093/0.21
[ 3/5][   3511][  681.13s]      d_loss/g_loss: -0.1/-1.1
[ 3/5][   3611][  700.36s]      d_loss/g_loss: -0.21/-1.2
[ 3/5][   3711][  719.63s]      d_loss/g_loss: -0.11/0.16
[ 4/5][   3748][  726.92s]      d_loss/g_loss: -0.12/-0.4
[ 4/5][   3848][  746.22s]      d_loss/g_loss: -0.045/-0.39
[ 4/5][   3948][  765.48s]      d_loss/g_loss: -0.13/-1.0
[ 4/5][   4048][  784.76s]      d_loss/g_loss: -0.23/-0.88
[ 4/5][   4148][  804.01s]      d_loss/g_loss: -0.2/-1.7
[ 4/5][   4248][  823.22s]      d_loss/g_loss: -0.21/-0.6
[ 4/5][   4348][  842.48s]      d_loss/g_loss: -0.12/-1.6
[ 4/5][   4448][  861.69s]      d_loss/g_loss: -0.1/0.66
[ 4/5][   4548][  880.99s]      d_loss/g_loss: -0.0077/-1.8
[ 4/5][   4648][  900.24s]      d_loss/g_loss: -0.11/-0.75
```

Visualization (2 pts)

```
# Visualize the loss
# include the figure in the latex file (1 pt)
plt.title("Generator and Critic Loss During Training")
plt.plot(generator_losses,label="G")
plt.plot(critic_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```




```
# Visualize the generation (you may scroll to see the animation of training)
# include the final figure in the latex file (1 pt)
import matplotlib.animation as animation
from IPython.display import HTML
#%%capture
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(i.permute(1,2,0), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=True)

HTML(ani.to_jshtml())
```



Once Loop Reflect