ECE 57000 Assignment 07 Exercise

Your Name: Mohmmad Alwakeel

For this assignment, you will do an ablation study on the DCGAN model discussed in class and implemented WGAN with weight clipping and (optional) WGAN with gradient penalty.

Exercise 1: Ablation Study on DCGAN

An [ablation study (https://en.wikipedia.org/wiki/Ablation_(artificial_intelligence)](https://en.wikipedia.org/wiki/Ablation_(artificial_intelligence)) measures performance changes after changing certain components in the AI system. The goal is to understand the contribution on each component for the overall system.

Task 1.0 Original DCGAN on MNIST from class note

Here is the copy of the code implementation from [course website (https://www.davidinouye.com/course/ece57000-fall-2021/lectures/dcgan-mnist-edit.pdf)](https://www.davidinouye.com/course/ece57000-fall-2021/lectures/dcgan-mnist-edit.pdf). Please run the code to obtain the result and **use it as a baseline to compare the results** with the following the ablation tasks.

Hyper-parameter and Dataloader setup

In [2]:
```
from __future__ import print_function
#%matplotlib inline
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
torch.cuda.manual_seed(manualSeed)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmarks = False
os.environ['PYTHONHASHSEED'] = str(manualSeed)

# Root directory for dataset
# dataroot = "data/celeba"

# Number of workers for dataloader
workers = 1

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
#image_size = 64
image_size = 32

# Number of channels in the training images. For color images this is 3
#nc = 3
nc = 1

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
#ngf = 64
ngf = 8

# Size of feature maps in discriminator
```

```
#ndf = 64
ndf = 8

# Number of training epochs
num_epochs = 5
num_epochs_wgan = 15
num_iters = 250

# Learning rate for optimizers
lr = 0.0002
lr_rms = 5e-4

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) els
e "cpu")

# Initialize BCELoss function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.0
fake_label = 0.0

# Several useful functions
def initialize_net(net_class, init_method, device, ngpu):

    # Create the generator
    net_inst = net_class(ngpu).to(device)

    # Handle multi-gpu if desired
    if (device.type == 'cuda') and (ngpu > 1):
        net_inst = nn.DataParallel(net_inst, list(range(ngpu)))

    # Apply the weights_init function to randomly initialize all weights
    #  to mean=0, stdev=0.2.
    if init_method is not None:
        net_inst.apply(init_method)

    # Print the model
    print(net_inst)

    return net_inst

def plot_GAN_loss(losses, labels):

    plt.figure(figsize=(10,5))
    plt.title("Losses During Training")
```

```
    for loss, label in zip(losses, labels):
        plt.plot(loss,label=f"{label}")

    plt.xlabel("iterations")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()


def plot_real_fake_images(real_batch, fake_batch):

    # Plot the real images
    plt.figure(figsize=(15,15))
    plt.subplot(1,2,1)
    plt.axis("off")
    plt.title("Real Images")
    plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], pa
dding=5, normalize=True).cpu(),(1,2,0)))

    # Plot the fake images from the last epoch
    plt.subplot(1,2,2)
    plt.axis("off")
    plt.title("Fake Images")
    plt.imshow(np.transpose(fake_batch[-1],(1,2,0)))
    plt.show()


# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

# Download the MNIST dataset
dataset = dset.MNIST(
    'data', train=True, download=True,
   transform=transforms.Compose([
       transforms.Resize(image_size), # Resize from 28 x 28 to 32 x 32 (so pow
er of 2)
       transforms.CenterCrop(image_size),
       transforms.ToTensor(),
       transforms.Normalize((0.5,), (0.5,))
   ]))

# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
```

```
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], paddin
g=2, normalize=True).cpu(),(1,2,0)))
```

```
Random Seed:  999
```

Out[2]: <matplotlib.image.AxesImage at 0x18d47080588>

Architectural design for generator and discriminator

In [4]:
```
# Generator Code
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution, state size. nz x 1 x 1
            nn.ConvTranspose2d( nz, ngf * 4, kernel_size=4, stride=1, padding=
0, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True), # inplace ReLU
            # current state size. (ngf*4) x 4 x 4
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # current state size. (ngf*2) x 8 x 8
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # current state size. ngf x 16 x 16
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            # current state size. nc x 32 x 32
            # Produce number between -1 and 1, as pixel values have been norma
lized to be between -1 and 1
            nn.Tanh()
        )

    def forward(self, input):
        return self.main(input)


class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 32 x 32
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 16 x 16
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 8 x 8
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 4 x 4
            nn.Conv2d(ndf * 4, 1, 4, 1, 0, bias=False),
            # state size. (ndf*4) x 1 x 1
            nn.Sigmoid()  # Produce probability
        )

    def forward(self, input):
        return self.main(input)
```

Loss function and Training function

In [8]:
```
# Initialize networks
netG = initialize_net(Generator, weights_init, device, ngpu)
netD = initialize_net(Discriminator, weights_init, device, ngpu)

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))


# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Add the gradients from the all-real and all-fake batches
        errD = errD_real + errD_fake
        # Update D
```

```
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost
        # Since we just updated D, perform another forward pass of all-fake ba
tch through D
        output = netD(fake).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G
(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

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

        iters += 1
```

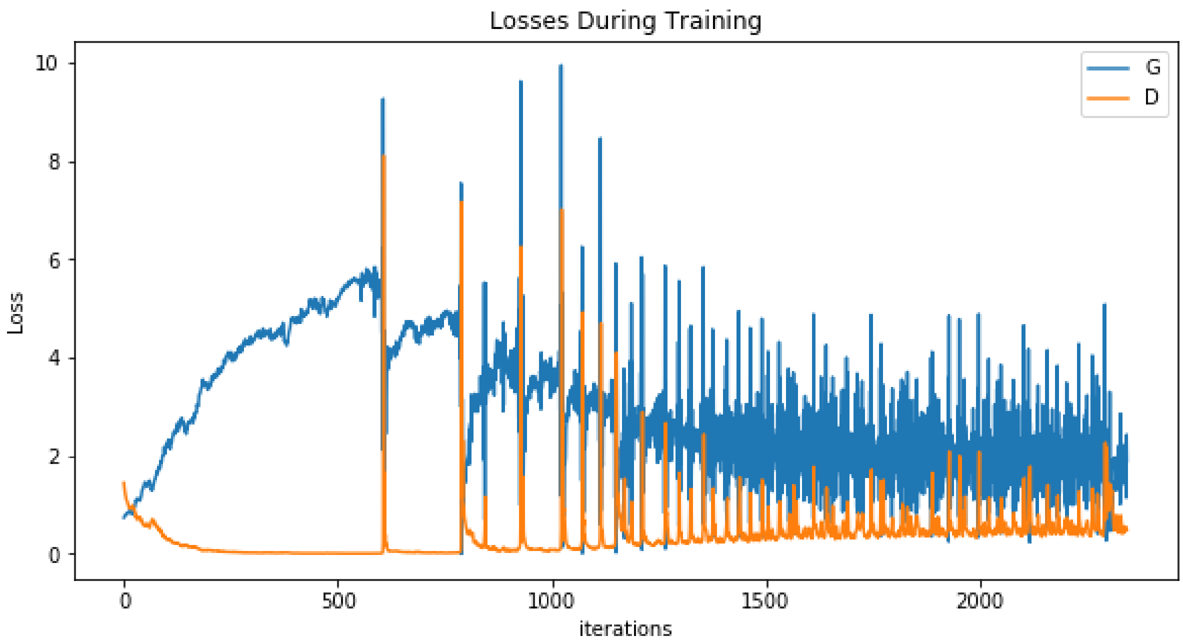
```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 32, kernel_size=(4, 4), stride=(1, 1), bias=Fal
se)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(32, 16, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1), bias=False)
    (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(16, 8, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1), bias=False)
    (7): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(8, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (10): Tanh()
  )
)
Discriminator(
  (main): Sequential(
    (0): Conv2d(1, 8, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias
=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(8, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bia
s=False)
    (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bi
as=False)
    (6): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(32, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (9): Sigmoid()
  )
)
Starting Training Loop...
[0/5][0/469]    Loss_D: 1.4493  Loss_G: 0.7415  D(x): 0.4824    D(G(z)): 0.50
60 / 0.4805
[0/5][50/469]   Loss_D: 0.5649  Loss_G: 1.4601  D(x): 0.8009    D(G(z)): 0.28
27 / 0.2385
[0/5][100/469]  Loss_D: 0.3072  Loss_G: 2.2178  D(x): 0.8648    D(G(z)): 0.14
38 / 0.1146
[0/5][150/469]  Loss_D: 0.1532  Loss_G: 2.6514  D(x): 0.9455    D(G(z)): 0.09
04 / 0.0784
[0/5][200/469]  Loss_D: 0.0809  Loss_G: 3.3859  D(x): 0.9648    D(G(z)): 0.04
32 / 0.0394
[0/5][250/469]  Loss_D: 0.0421  Loss_G: 3.9898  D(x): 0.9821    D(G(z)): 0.02
34 / 0.0215
[0/5][300/469]  Loss_D: 0.0268  Loss_G: 4.3464  D(x): 0.9912    D(G(z)): 0.01
77 / 0.0156
[0/5][350/469]  Loss_D: 0.0232  Loss_G: 4.4663  D(x): 0.9930    D(G(z)): 0.01
```

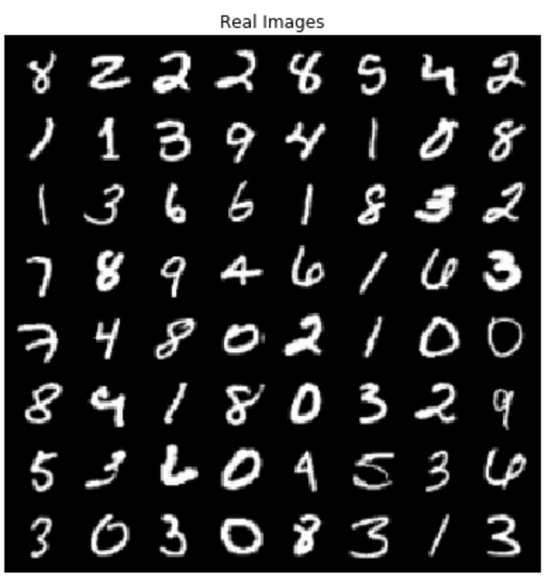
```
60 / 0.0138
[0/5][400/469]  Loss_D: 0.0161  Loss_G: 4.7768  D(x): 0.9955   D(G(z)): 0.01
15 / 0.0091
[0/5][450/469]  Loss_D: 0.0099  Loss_G: 5.1826  D(x): 0.9978   D(G(z)): 0.00
76 / 0.0066
[1/5][0/469]    Loss_D: 0.0127  Loss_G: 5.0169  D(x): 0.9975   D(G(z)): 0.01
01 / 0.0073
[1/5][50/469]   Loss_D: 0.0095  Loss_G: 5.4539  D(x): 0.9966   D(G(z)): 0.00
60 / 0.0050
[1/5][100/469]  Loss_D: 0.0106  Loss_G: 5.5828  D(x): 0.9947   D(G(z)): 0.00
52 / 0.0039
[1/5][150/469]  Loss_D: 0.0707  Loss_G: 4.1313  D(x): 0.9751   D(G(z)): 0.04
38 / 0.0174
[1/5][200/469]  Loss_D: 0.0339  Loss_G: 4.6644  D(x): 0.9839   D(G(z)): 0.01
73 / 0.0111
[1/5][250/469]  Loss_D: 0.0243  Loss_G: 4.7487  D(x): 0.9870   D(G(z)): 0.01
10 / 0.0097
[1/5][300/469]  Loss_D: 0.0263  Loss_G: 4.6570  D(x): 0.9838   D(G(z)): 0.00
99 / 0.0109
[1/5][350/469]  Loss_D: 0.2402  Loss_G: 3.1571  D(x): 0.8952   D(G(z)): 0.11
06 / 0.0513
[1/5][400/469]  Loss_D: 0.1165  Loss_G: 3.5505  D(x): 0.9490   D(G(z)): 0.05
72 / 0.0379
[1/5][450/469]  Loss_D: 0.0839  Loss_G: 3.6389  D(x): 0.9443   D(G(z)): 0.01
47 / 0.0361
[2/5][0/469]    Loss_D: 0.2277  Loss_G: 2.5304  D(x): 0.9503   D(G(z)): 0.15
79 / 0.0913
[2/5][50/469]   Loss_D: 0.0605  Loss_G: 3.8779  D(x): 0.9763   D(G(z)): 0.03
55 / 0.0236
[2/5][100/469]  Loss_D: 0.1419  Loss_G: 3.1828  D(x): 0.9319   D(G(z)): 0.06
70 / 0.0496
[2/5][150/469]  Loss_D: 0.1373  Loss_G: 3.2217  D(x): 0.9511   D(G(z)): 0.08
17 / 0.0465
[2/5][200/469]  Loss_D: 0.1551  Loss_G: 2.8188  D(x): 0.9248   D(G(z)): 0.07
05 / 0.0715
[2/5][250/469]  Loss_D: 0.3293  Loss_G: 3.3079  D(x): 0.7501   D(G(z)): 0.02
67 / 0.0410
[2/5][300/469]  Loss_D: 0.1709  Loss_G: 2.7689  D(x): 0.9298   D(G(z)): 0.09
01 / 0.0727
[2/5][350/469]  Loss_D: 0.2094  Loss_G: 2.5953  D(x): 0.8997   D(G(z)): 0.09
47 / 0.0850
[2/5][400/469]  Loss_D: 0.2756  Loss_G: 2.4953  D(x): 0.8781   D(G(z)): 0.12
89 / 0.0956
[2/5][450/469]  Loss_D: 0.2437  Loss_G: 2.3766  D(x): 0.8823   D(G(z)): 0.10
64 / 0.1074
[3/5][0/469]    Loss_D: 0.5736  Loss_G: 0.8547  D(x): 0.6049   D(G(z)): 0.02
86 / 0.4503
[3/5][50/469]   Loss_D: 0.2849  Loss_G: 2.3658  D(x): 0.9202   D(G(z)): 0.17
65 / 0.1052
[3/5][100/469]  Loss_D: 0.7536  Loss_G: 2.7158  D(x): 0.9708   D(G(z)): 0.49
35 / 0.0768
[3/5][150/469]  Loss_D: 0.5502  Loss_G: 2.6759  D(x): 0.9472   D(G(z)): 0.36
96 / 0.0825
[3/5][200/469]  Loss_D: 0.3190  Loss_G: 2.1664  D(x): 0.8549   D(G(z)): 0.14
02 / 0.1291
[3/5][250/469]  Loss_D: 0.9409  Loss_G: 1.0806  D(x): 0.4383   D(G(z)): 0.03
58 / 0.3602
```

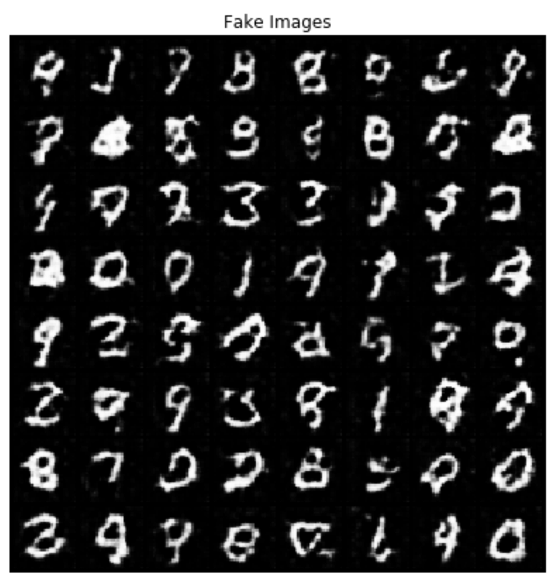
```
[3/5][300/469]  Loss_D: 0.3255  Loss_G: 2.4825  D(x): 0.8707    D(G(z)): 0.16
24 / 0.0953
[3/5][350/469]  Loss_D: 0.4015  Loss_G: 2.0034  D(x): 0.8289    D(G(z)): 0.17
72 / 0.1557
[3/5][400/469]  Loss_D: 0.4051  Loss_G: 1.6725  D(x): 0.7982    D(G(z)): 0.15
09 / 0.2106
[3/5][450/469]  Loss_D: 0.5482  Loss_G: 2.3155  D(x): 0.8970    D(G(z)): 0.33
85 / 0.1133
[4/5][0/469]    Loss_D: 0.3806  Loss_G: 2.4335  D(x): 0.8588    D(G(z)): 0.19
33 / 0.0993
[4/5][50/469]   Loss_D: 0.5730  Loss_G: 3.5892  D(x): 0.9247    D(G(z)): 0.37
05 / 0.0336
[4/5][100/469]  Loss_D: 0.4288  Loss_G: 1.6665  D(x): 0.7756    D(G(z)): 0.14
45 / 0.2139
[4/5][150/469]  Loss_D: 0.4800  Loss_G: 1.1722  D(x): 0.7310    D(G(z)): 0.13
64 / 0.3342
[4/5][200/469]  Loss_D: 0.4490  Loss_G: 3.1399  D(x): 0.9111    D(G(z)): 0.28
49 / 0.0510
[4/5][250/469]  Loss_D: 0.5594  Loss_G: 1.1648  D(x): 0.6734    D(G(z)): 0.10
90 / 0.3419
[4/5][300/469]  Loss_D: 0.4193  Loss_G: 1.6752  D(x): 0.7690    D(G(z)): 0.12
99 / 0.2102
[4/5][350/469]  Loss_D: 0.4460  Loss_G: 2.5648  D(x): 0.8833    D(G(z)): 0.25
95 / 0.0928
[4/5][400/469]  Loss_D: 0.9967  Loss_G: 3.4140  D(x): 0.9776    D(G(z)): 0.60
06 / 0.0376
[4/5][450/469]  Loss_D: 0.5906  Loss_G: 1.9718  D(x): 0.7761    D(G(z)): 0.26
39 / 0.1564
```

Visualization of the results

In [9]:
```
# plot the loss for generator and discriminator
plot_GAN_loss([G_losses, D_losses], ["G", "D"])

# Grab a batch of real images from the dataloader
plot_real_fake_images(next(iter(dataloader)), img_list)
```







Task 1.1 Ablation study on batch normalization

1. Please modify the code provided in the Task 1.0 so that the neural network architure does not contain any batch normalization layer.
   Hint: modify the ***Architectural design for generator and discriminator*** section in Task 1.0
2. Train the model with modified networks and visualize the results.

In [5]:
```
# Generator Code
class Generator_woBN(nn.Module):
    def __init__(self, ngpu):
        super(Generator_woBN, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            ################################ YOUR CODE ######################
#########


            nn.ConvTranspose2d( nz, ngf * 4, kernel_size=4, stride=1, padding=
0, bias=False),
            ## nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True), # inplace ReLU
            # current state size. (ngf*4) x 4 x 4
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.ReLU(True),
            # current state size. (ngf*2) x 8 x 8
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.ReLU(True),
            # current state size. ngf x 16 x 16
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            # current state size. nc x 32 x 32
            # Produce number between -1 and 1, as pixel values have been norma
lized to be between -1 and 1
            nn.Tanh()

            ############################# END YOUR CODE #####################
########
        )

    def forward(self, input):
        return self.main(input)


class Discriminator_woBN(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator_woBN, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            ################################ YOUR CODE ######################
#########


            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 16 x 16
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 8 x 8
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 4 x 4
            nn.Conv2d(ndf * 4, 1, 4, 1, 0, bias=False),
            # state size. (ndf*4) x 1 x 1
            nn.Sigmoid()  # Produce probability
```

```
            ############################# END YOUR CODE ######################
########
        )

    def forward(self, input):
        return self.main(input)

netG_noBN = initialize_net(Generator_woBN, weights_init, device, ngpu)
netD_noBN = initialize_net(Discriminator_woBN, weights_init, device, ngpu)
```

```
Generator_woBN(
  (main): Sequential(
    (0): ConvTranspose2d(100, 32, kernel_size=(4, 4), stride=(1, 1), bias=Fal
se)
    (1): ReLU(inplace=True)
    (2): ConvTranspose2d(32, 16, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1), bias=False)
    (3): ReLU(inplace=True)
    (4): ConvTranspose2d(16, 8, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1), bias=False)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(8, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (7): Tanh()
  )
)
Discriminator_woBN(
  (main): Sequential(
    (0): Conv2d(1, 8, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias
=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(8, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bia
s=False)
    (3): LeakyReLU(negative_slope=0.2, inplace=True)
    (4): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bi
as=False)
    (5): LeakyReLU(negative_slope=0.2, inplace=True)
    (6): Conv2d(32, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (7): Sigmoid()
  )
)
```

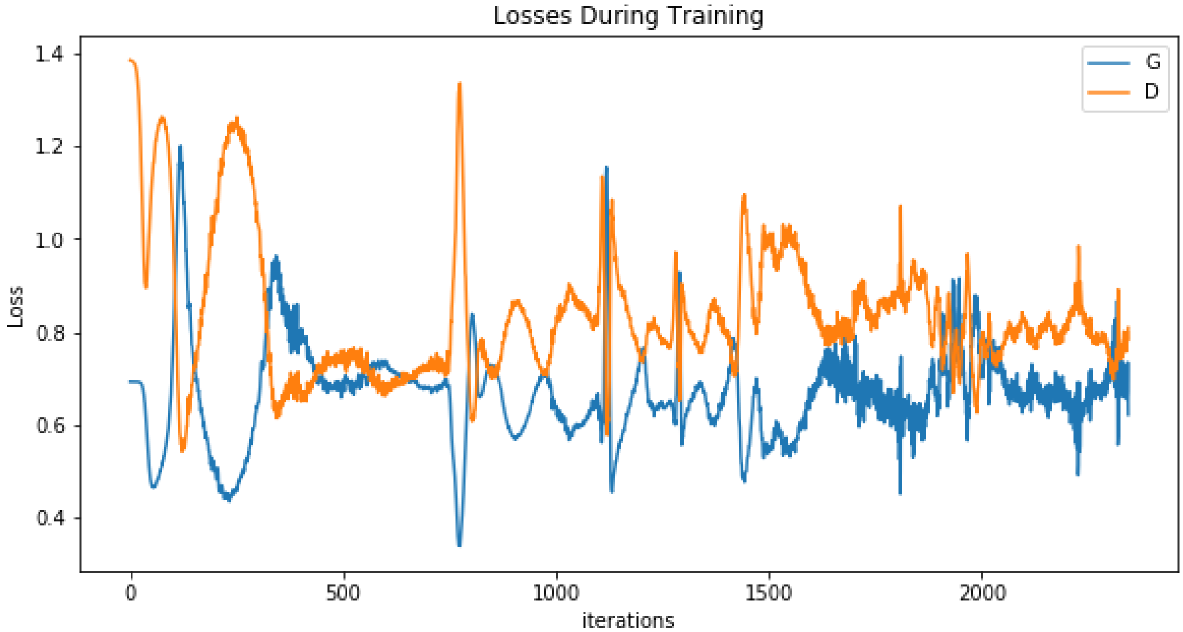
In [22]:
```
# Setup Adam optimizers for both G and D
optimizerD_noBN = optim.Adam(netD_noBN.parameters(), lr=lr, betas=(beta1, 0.99
9))
optimizerG_noBN = optim.Adam(netG_noBN.parameters(), lr=lr, betas=(beta1, 0.99
9))

# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
        netD_noBN.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, device=device)
        # Forward pass real batch through D
        output = netD_noBN(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG_noBN(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD_noBN(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Add the gradients from the all-real and all-fake batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD_noBN.step()

        ############################
```

```
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG_noBN.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost
        # Since we just updated D, perform another forward pass of all-fake ba
tch through D
        output = netD_noBN(fake).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG_noBN.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G
(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

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

        iters += 1
```

```
Starting Training Loop...
[0/5][0/469]    Loss_D: 1.3862  Loss_G: 0.6931  D(x): 0.5001    D(G(z)): 0.50
00 / 0.5000
[0/5][50/469]   Loss_D: 1.1021  Loss_G: 0.4735  D(x): 0.8969    D(G(z)): 0.62
92 / 0.6230
[0/5][100/469]  Loss_D: 1.0617  Loss_G: 0.7312  D(x): 0.6825    D(G(z)): 0.49
29 / 0.4814
[0/5][150/469]  Loss_D: 0.7274  Loss_G: 0.7138  D(x): 0.9526    D(G(z)): 0.49
09 / 0.4903
[0/5][200/469]  Loss_D: 1.0531  Loss_G: 0.4995  D(x): 0.9002    D(G(z)): 0.60
77 / 0.6074
[0/5][250/469]  Loss_D: 1.2455  Loss_G: 0.4615  D(x): 0.7923    D(G(z)): 0.63
47 / 0.6307
[0/5][300/469]  Loss_D: 1.0370  Loss_G: 0.6426  D(x): 0.7651    D(G(z)): 0.53
15 / 0.5261
[0/5][350/469]  Loss_D: 0.6507  Loss_G: 0.8827  D(x): 0.8804    D(G(z)): 0.39
78 / 0.4154
[0/5][400/469]  Loss_D: 0.6759  Loss_G: 0.7737  D(x): 0.9547    D(G(z)): 0.46
46 / 0.4615
[0/5][450/469]  Loss_D: 0.7132  Loss_G: 0.7028  D(x): 0.9813    D(G(z)): 0.49
96 / 0.4953
[1/5][0/469]    Loss_D: 0.7360  Loss_G: 0.6875  D(x): 0.9709    D(G(z)): 0.50
41 / 0.5029
[1/5][50/469]   Loss_D: 0.7618  Loss_G: 0.6754  D(x): 0.9633    D(G(z)): 0.50
85 / 0.5092
[1/5][100/469]  Loss_D: 0.6918  Loss_G: 0.7167  D(x): 0.9872    D(G(z)): 0.49
19 / 0.4884
[1/5][150/469]  Loss_D: 0.6848  Loss_G: 0.7120  D(x): 0.9922    D(G(z)): 0.49
11 / 0.4907
[1/5][200/469]  Loss_D: 0.6991  Loss_G: 0.6916  D(x): 0.9965    D(G(z)): 0.50
12 / 0.5008
[1/5][250/469]  Loss_D: 0.7248  Loss_G: 0.6830  D(x): 0.9852    D(G(z)): 0.50
61 / 0.5051
[1/5][300/469]  Loss_D: 1.2294  Loss_G: 0.3710  D(x): 0.9858    D(G(z)): 0.70
20 / 0.6904
[1/5][350/469]  Loss_D: 0.7619  Loss_G: 0.6635  D(x): 0.9703    D(G(z)): 0.51
81 / 0.5151
[1/5][400/469]  Loss_D: 0.7359  Loss_G: 0.6849  D(x): 0.9716    D(G(z)): 0.50
57 / 0.5042
[1/5][450/469]  Loss_D: 0.8494  Loss_G: 0.5832  D(x): 0.9751    D(G(z)): 0.56
08 / 0.5582
[2/5][0/469]    Loss_D: 0.8059  Loss_G: 0.6273  D(x): 0.9642    D(G(z)): 0.53
62 / 0.5341
[2/5][50/469]   Loss_D: 0.7459  Loss_G: 0.6861  D(x): 0.9661    D(G(z)): 0.50
78 / 0.5045
[2/5][100/469]  Loss_D: 0.8877  Loss_G: 0.5797  D(x): 0.9443    D(G(z)): 0.56
32 / 0.5604
[2/5][150/469]  Loss_D: 0.8267  Loss_G: 0.6318  D(x): 0.9409    D(G(z)): 0.53
30 / 0.5324
[2/5][200/469]  Loss_D: 0.9887  Loss_G: 0.5047  D(x): 0.9676    D(G(z)): 0.61
46 / 0.6039
[2/5][250/469]  Loss_D: 0.7888  Loss_G: 0.6859  D(x): 0.9247    D(G(z)): 0.50
71 / 0.5038
[2/5][300/469]  Loss_D: 0.8040  Loss_G: 0.6379  D(x): 0.9626    D(G(z)): 0.53
45 / 0.5286
[2/5][350/469]  Loss_D: 0.7322  Loss_G: 0.8264  D(x): 0.8649    D(G(z)): 0.43
33 / 0.4390
```

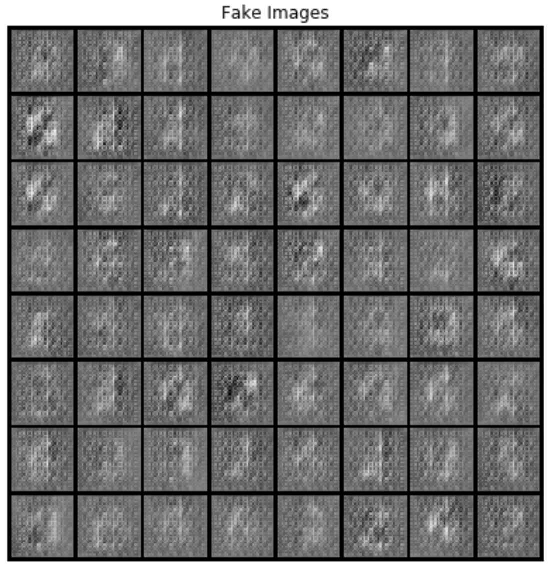
```
[2/5][400/469]  Loss_D: 0.7539  Loss_G: 0.6926  D(x): 0.9609    D(G(z)): 0.50
96 / 0.5006
[2/5][450/469]  Loss_D: 0.8252  Loss_G: 0.6348  D(x): 0.9456    D(G(z)): 0.53
47 / 0.5309
[3/5][0/469]    Loss_D: 0.7772  Loss_G: 0.7198  D(x): 0.9234    D(G(z)): 0.49
89 / 0.4886
[3/5][50/469]   Loss_D: 0.9471  Loss_G: 0.5730  D(x): 0.9151    D(G(z)): 0.57
31 / 0.5647
[3/5][100/469]  Loss_D: 0.9725  Loss_G: 0.5667  D(x): 0.9097    D(G(z)): 0.57
96 / 0.5689
[3/5][150/469]  Loss_D: 0.9877  Loss_G: 0.5592  D(x): 0.8919    D(G(z)): 0.57
87 / 0.5733
[3/5][200/469]  Loss_D: 0.8591  Loss_G: 0.6601  D(x): 0.8772    D(G(z)): 0.51
40 / 0.5179
[3/5][250/469]  Loss_D: 0.7949  Loss_G: 0.7761  D(x): 0.9287    D(G(z)): 0.51
05 / 0.4616
[3/5][300/469]  Loss_D: 0.8385  Loss_G: 0.6871  D(x): 0.9104    D(G(z)): 0.51
64 / 0.5066
[3/5][350/469]  Loss_D: 0.8649  Loss_G: 0.6124  D(x): 0.8698    D(G(z)): 0.50
74 / 0.5442
[3/5][400/469]  Loss_D: 0.9474  Loss_G: 0.6092  D(x): 0.9700    D(G(z)): 0.59
83 / 0.5449
[3/5][450/469]  Loss_D: 0.9090  Loss_G: 0.6099  D(x): 0.8803    D(G(z)): 0.53
93 / 0.5443
[4/5][0/469]    Loss_D: 0.8197  Loss_G: 0.6970  D(x): 0.9251    D(G(z)): 0.52
19 / 0.4990
[4/5][50/469]   Loss_D: 0.8413  Loss_G: 0.7134  D(x): 0.9088    D(G(z)): 0.52
19 / 0.4910
[4/5][100/469]  Loss_D: 0.7529  Loss_G: 0.7709  D(x): 0.9396    D(G(z)): 0.49
70 / 0.4633
[4/5][150/469]  Loss_D: 0.7445  Loss_G: 0.7366  D(x): 0.9086    D(G(z)): 0.47
31 / 0.4791
[4/5][200/469]  Loss_D: 0.7977  Loss_G: 0.6930  D(x): 0.9411    D(G(z)): 0.51
90 / 0.5008
[4/5][250/469]  Loss_D: 0.7729  Loss_G: 0.6904  D(x): 0.9492    D(G(z)): 0.51
17 / 0.5021
[4/5][300/469]  Loss_D: 0.7794  Loss_G: 0.6933  D(x): 0.9647    D(G(z)): 0.52
37 / 0.5002
[4/5][350/469]  Loss_D: 0.9193  Loss_G: 0.4914  D(x): 0.8252    D(G(z)): 0.50
94 / 0.6126
[4/5][400/469]  Loss_D: 0.7829  Loss_G: 0.6836  D(x): 0.9483    D(G(z)): 0.51
36 / 0.5053
[4/5][450/469]  Loss_D: 0.7414  Loss_G: 0.7421  D(x): 0.9389    D(G(z)): 0.48
94 / 0.4766
```

In [23]:
```
# plot the loss for generator and discriminator
plot_GAN_loss([G_losses, D_losses], ["G", "D"])

# Grab a batch of real images from the dataloader
plot_real_fake_images(next(iter(dataloader)), img_list)
```



Real Images

Fake Images



Task 1.2 Ablation study on the trick: "Construct different mini-batches for real and fake"

1. Please modify the code provided in the Task 1.0 so that the discriminator algorithm part computes the forward and backward pass for fake and real images concatenated together (with their corresponding fake and real labels concatenated as well) instead of computing the forward and backward passes for fake and real images separately.
   Hint: modify the ***Loss function and Training function*** section in Task 1.0.
2. Train the model with modified networks and visualize the results.

In [55]:
```
# re-initilizate networks for the generator and discrimintor.
netG = initialize_net(Generator, weights_init, device, ngpu)
netD = initialize_net(Discriminator, weights_init, device, ngpu)

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))

# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################

        ################################ YOUR CODE ###########################
#####

        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)

        noise = torch.randn(b_size, nz, 1, 1, device=device)
        fake = netG(noise)

        data_concat = torch.cat((real_cpu, fake))
        label_concat = torch.cat((torch.full((b_size,), real_label, device=dev
ice),
                                 torch.full((b_size,), fake_label, device=devi
ce)))


        output_concat = netD(data_concat).view(-1)



        errD = criterion(output_concat, label_concat)
        errD.backward()

        # Update D
        optimizerD.step()

        ############################ END YOUR CODE ###########################
###
```

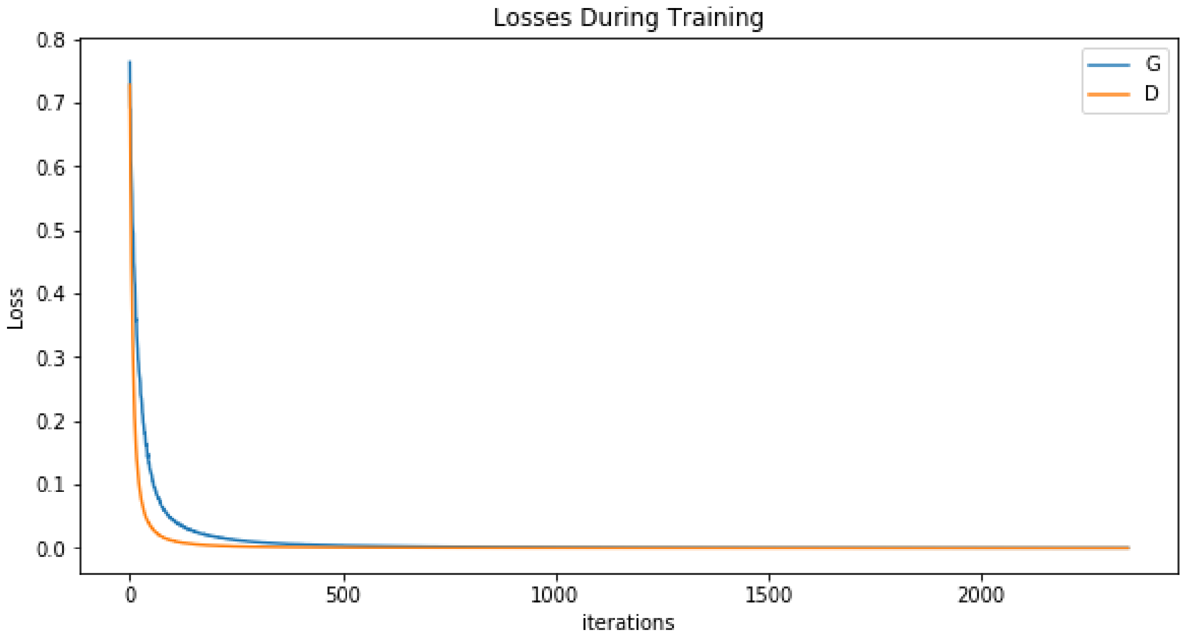
```
        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        fake = netG(noise)

        label = torch.full((b_size,), real_label, device=device)  # fake label
s are real for generator cost
        # Since we just updated D, perform another forward pass of all-fake ba
tch through D
        output = netD(fake).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(G(z)): %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

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

        iters += 1
```

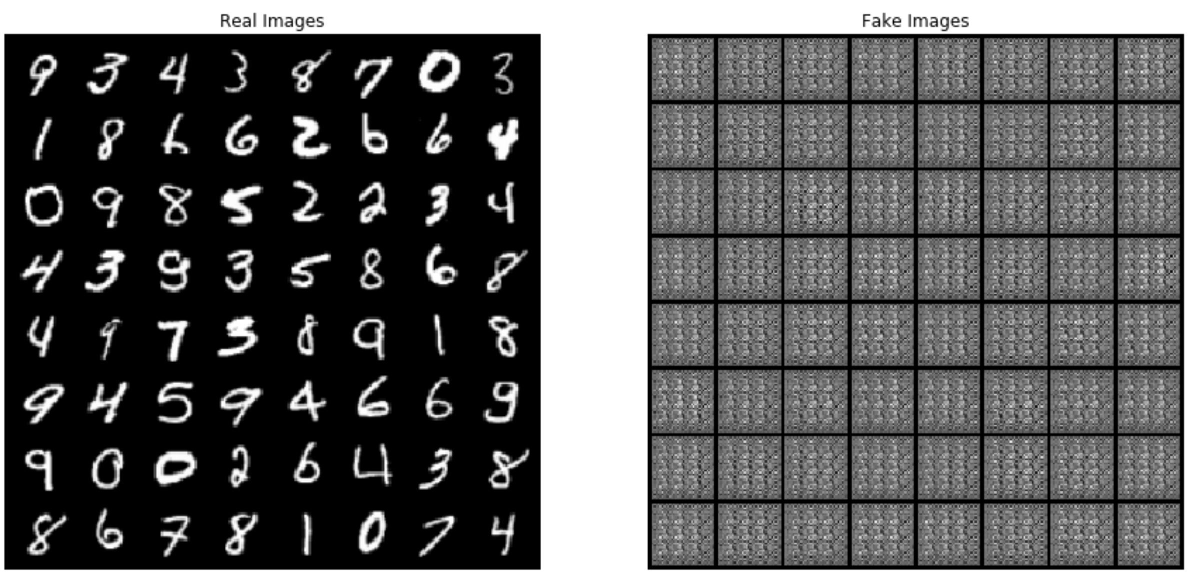
```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 32, kernel_size=(4, 4), stride=(1, 1), bias=Fal
se)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(32, 16, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1), bias=False)
    (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(16, 8, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1), bias=False)
    (7): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(8, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (10): Tanh()
  )
)
Discriminator(
  (main): Sequential(
    (0): Conv2d(1, 8, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias
=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(8, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bia
s=False)
    (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bi
as=False)
    (6): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(32, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (9): Sigmoid()
  )
)
Starting Training Loop...
[0/5][0/469]    Loss_D: 0.7281  Loss_G: 0.7642  D(G(z)): 0.4703
[0/5][50/469]   Loss_D: 0.0334  Loss_G: 0.1177  D(G(z)): 0.8895
[0/5][100/469]  Loss_D: 0.0121  Loss_G: 0.0459  D(G(z)): 0.9552
[0/5][150/469]  Loss_D: 0.0066  Loss_G: 0.0263  D(G(z)): 0.9740
[0/5][200/469]  Loss_D: 0.0043  Loss_G: 0.0182  D(G(z)): 0.9820
[0/5][250/469]  Loss_D: 0.0030  Loss_G: 0.0128  D(G(z)): 0.9873
[0/5][300/469]  Loss_D: 0.0022  Loss_G: 0.0095  D(G(z)): 0.9905
[0/5][350/469]  Loss_D: 0.0018  Loss_G: 0.0075  D(G(z)): 0.9925
[0/5][400/469]  Loss_D: 0.0014  Loss_G: 0.0062  D(G(z)): 0.9938
[0/5][450/469]  Loss_D: 0.0012  Loss_G: 0.0051  D(G(z)): 0.9950
[1/5][0/469]    Loss_D: 0.0010  Loss_G: 0.0048  D(G(z)): 0.9952
[1/5][50/469]   Loss_D: 0.0009  Loss_G: 0.0041  D(G(z)): 0.9959
[1/5][100/469]  Loss_D: 0.0008  Loss_G: 0.0035  D(G(z)): 0.9965
[1/5][150/469]  Loss_D: 0.0006  Loss_G: 0.0031  D(G(z)): 0.9969
[1/5][200/469]  Loss_D: 0.0006  Loss_G: 0.0028  D(G(z)): 0.9972
```

```
[1/5][250/469]  Loss_D: 0.0005  Loss_G: 0.0025  D(G(z)): 0.9975
[1/5][300/469]  Loss_D: 0.0004  Loss_G: 0.0022  D(G(z)): 0.9978
[1/5][350/469]  Loss_D: 0.0004  Loss_G: 0.0020  D(G(z)): 0.9980
[1/5][400/469]  Loss_D: 0.0004  Loss_G: 0.0018  D(G(z)): 0.9982
[1/5][450/469]  Loss_D: 0.0003  Loss_G: 0.0017  D(G(z)): 0.9983
[2/5][0/469]    Loss_D: 0.0003  Loss_G: 0.0017  D(G(z)): 0.9983
[2/5][50/469]   Loss_D: 0.0003  Loss_G: 0.0015  D(G(z)): 0.9985
[2/5][100/469]  Loss_D: 0.0003  Loss_G: 0.0014  D(G(z)): 0.9986
[2/5][150/469]  Loss_D: 0.0002  Loss_G: 0.0013  D(G(z)): 0.9987
[2/5][200/469]  Loss_D: 0.0002  Loss_G: 0.0012  D(G(z)): 0.9988
[2/5][250/469]  Loss_D: 0.0002  Loss_G: 0.0011  D(G(z)): 0.9989
[2/5][300/469]  Loss_D: 0.0002  Loss_G: 0.0011  D(G(z)): 0.9989
[2/5][350/469]  Loss_D: 0.0002  Loss_G: 0.0010  D(G(z)): 0.9990
[2/5][400/469]  Loss_D: 0.0002  Loss_G: 0.0009  D(G(z)): 0.9991
[2/5][450/469]  Loss_D: 0.0002  Loss_G: 0.0009  D(G(z)): 0.9991
[3/5][0/469]    Loss_D: 0.0001  Loss_G: 0.0008  D(G(z)): 0.9992
[3/5][50/469]   Loss_D: 0.0001  Loss_G: 0.0008  D(G(z)): 0.9992
[3/5][100/469]  Loss_D: 0.0001  Loss_G: 0.0007  D(G(z)): 0.9993
[3/5][150/469]  Loss_D: 0.0001  Loss_G: 0.0007  D(G(z)): 0.9993
[3/5][200/469]  Loss_D: 0.0001  Loss_G: 0.0007  D(G(z)): 0.9993
[3/5][250/469]  Loss_D: 0.0001  Loss_G: 0.0006  D(G(z)): 0.9994
[3/5][300/469]  Loss_D: 0.0001  Loss_G: 0.0006  D(G(z)): 0.9994
[3/5][350/469]  Loss_D: 0.0001  Loss_G: 0.0006  D(G(z)): 0.9994
[3/5][400/469]  Loss_D: 0.0001  Loss_G: 0.0005  D(G(z)): 0.9995
[3/5][450/469]  Loss_D: 0.0001  Loss_G: 0.0005  D(G(z)): 0.9995
[4/5][0/469]    Loss_D: 0.0001  Loss_G: 0.0005  D(G(z)): 0.9995
[4/5][50/469]   Loss_D: 0.0001  Loss_G: 0.0005  D(G(z)): 0.9995
[4/5][100/469]  Loss_D: 0.0001  Loss_G: 0.0005  D(G(z)): 0.9995
[4/5][150/469]  Loss_D: 0.0001  Loss_G: 0.0004  D(G(z)): 0.9996
[4/5][200/469]  Loss_D: 0.0001  Loss_G: 0.0004  D(G(z)): 0.9996
[4/5][250/469]  Loss_D: 0.0001  Loss_G: 0.0004  D(G(z)): 0.9996
[4/5][300/469]  Loss_D: 0.0001  Loss_G: 0.0004  D(G(z)): 0.9996
[4/5][350/469]  Loss_D: 0.0001  Loss_G: 0.0004  D(G(z)): 0.9996
[4/5][400/469]  Loss_D: 0.0001  Loss_G: 0.0004  D(G(z)): 0.9996
[4/5][450/469]  Loss_D: 0.0001  Loss_G: 0.0003  D(G(z)): 0.9997
```

In [56]:

```
# plot the loss for generator and discriminator
plot_GAN_loss([G_losses, D_losses], ["G", "D"])

# Grab a batch of real images from the dataloader
plot_real_fake_images(next(iter(dataloader)), img_list)
```





Task 1.3 Ablation study on the generator's loss function

1. Please modify the code provided in the Task 1.0 so that the *Generator* algorithm part minimizes $\log(1 - D(G(z)))$ instead of the modified loss function suggested in the original GAN paper of $-\log(D(G(z)))$.
   A. Modify the ***Loss function and Training function*** section in Task 1.0
   B. (Hint) Try to understand the definition of BCE loss (https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html) first and how the modified loss function was implemented.
2. Train the model with modified networks and visualize the results.

In [12]:
```
# re-initilizate networks for the generator and discrimintor.
netG = initialize_net(Generator, weights_init, device, ngpu)
netD = initialize_net(Discriminator, weights_init, device, ngpu)

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))

# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Add the gradients from the all-real and all-fake batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()
```

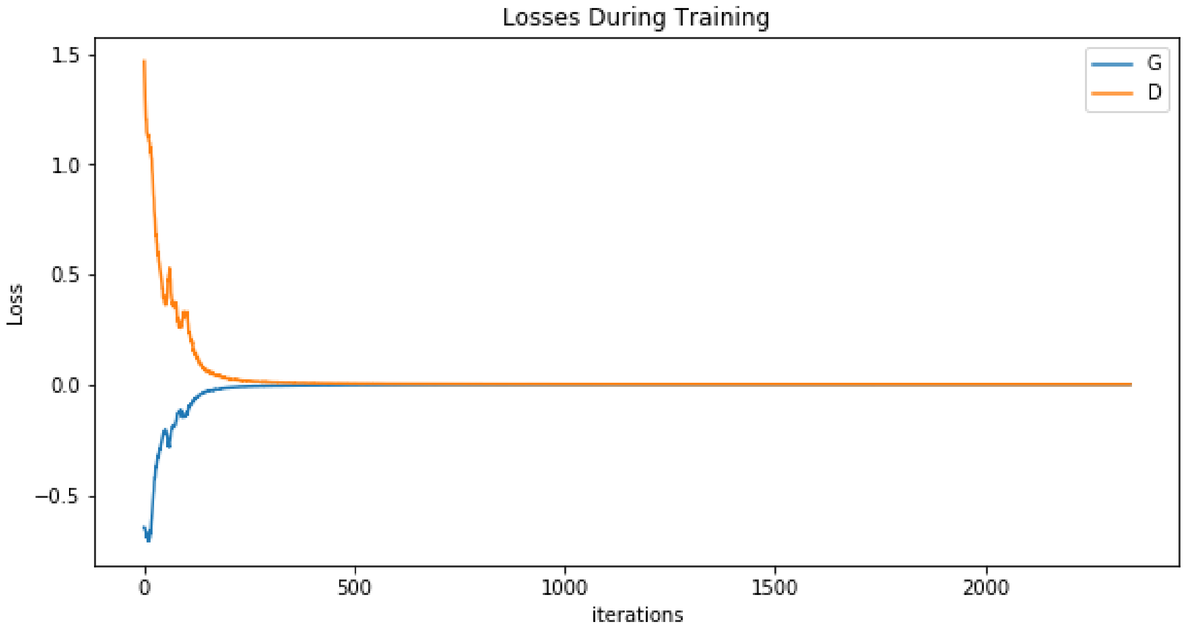
```
        ############################
        # (2) Update G network
        ###########################

        ################################ YOUR CODE ###########################
#####
        ''''
        #label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake).view(-1)
        # Calculate D's loss on the all-fake batch
        #errG = torch.log(1- output)
        #output = torch.log(1-output)
        errG = -criterion(output, label)
        # Calculate the gradients for this batch
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()
        '''
        netG.zero_grad()
        label.fill_(fake_label)  # fake labels to minimize
        # Since we just updated D, perform another forward pass of all-fake ba
tch through D
        output = netD(fake).view(-1)
        # Calculate G's loss based on this output
        errG = -criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()


        ############################ END YOUR CODE ##########################
###

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G
(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

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

        iters += 1
```

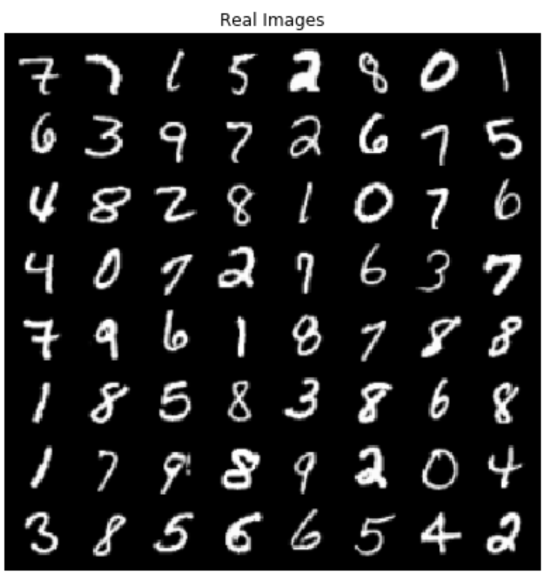
```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 32, kernel_size=(4, 4), stride=(1, 1), bias=Fal
se)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(32, 16, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1), bias=False)
    (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(16, 8, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1), bias=False)
    (7): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(8, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (10): Tanh()
  )
)
Discriminator(
  (main): Sequential(
    (0): Conv2d(1, 8, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias
=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(8, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bia
s=False)
    (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bi
as=False)
    (6): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(32, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (9): Sigmoid()
  )
)
Starting Training Loop...
[0/5][0/469]    Loss_D: 1.4669  Loss_G: -0.6471 D(x): 0.4565    D(G(z)): 0.48
69 / 0.4727
[0/5][50/469]   Loss_D: 0.3643  Loss_G: -0.2051 D(x): 0.8811    D(G(z)): 0.20
80 / 0.1839
[0/5][100/469]  Loss_D: 0.3241  Loss_G: -0.1450 D(x): 0.8609    D(G(z)): 0.14
97 / 0.1331
[0/5][150/469]  Loss_D: 0.0572  Loss_G: -0.0282 D(x): 0.9748    D(G(z)): 0.03
07 / 0.0277
[0/5][200/469]  Loss_D: 0.0267  Loss_G: -0.0140 D(x): 0.9882    D(G(z)): 0.01
47 / 0.0138
[0/5][250/469]  Loss_D: 0.0162  Loss_G: -0.0088 D(x): 0.9929    D(G(z)): 0.00
90 / 0.0087
[0/5][300/469]  Loss_D: 0.0106  Loss_G: -0.0055 D(x): 0.9950    D(G(z)): 0.00
55 / 0.0055
[0/5][350/469]  Loss_D: 0.0086  Loss_G: -0.0047 D(x): 0.9963    D(G(z)): 0.00
```

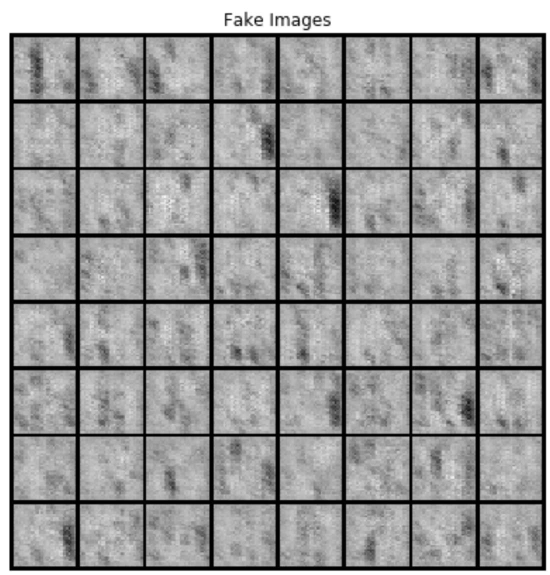
```
48 / 0.0047
[0/5][400/469]  Loss_D: 0.0067  Loss_G: -0.0036 D(x): 0.9969    D(G(z)): 0.00
36 / 0.0036
[0/5][450/469]  Loss_D: 0.0046  Loss_G: -0.0028 D(x): 0.9982    D(G(z)): 0.00
29 / 0.0028
[1/5][0/469]    Loss_D: 0.0045  Loss_G: -0.0027 D(x): 0.9982    D(G(z)): 0.00
27 / 0.0026
[1/5][50/469]   Loss_D: 0.0041  Loss_G: -0.0022 D(x): 0.9981    D(G(z)): 0.00
22 / 0.0022
[1/5][100/469]  Loss_D: 0.0029  Loss_G: -0.0018 D(x): 0.9989    D(G(z)): 0.00
18 / 0.0018
[1/5][150/469]  Loss_D: 0.0026  Loss_G: -0.0016 D(x): 0.9989    D(G(z)): 0.00
16 / 0.0016
[1/5][200/469]  Loss_D: 0.0025  Loss_G: -0.0014 D(x): 0.9990    D(G(z)): 0.00
14 / 0.0014
[1/5][250/469]  Loss_D: 0.0019  Loss_G: -0.0012 D(x): 0.9992    D(G(z)): 0.00
12 / 0.0012
[1/5][300/469]  Loss_D: 0.0017  Loss_G: -0.0011 D(x): 0.9994    D(G(z)): 0.00
11 / 0.0011
[1/5][350/469]  Loss_D: 0.0015  Loss_G: -0.0008 D(x): 0.9994    D(G(z)): 0.00
08 / 0.0008
[1/5][400/469]  Loss_D: 0.0013  Loss_G: -0.0007 D(x): 0.9994    D(G(z)): 0.00
07 / 0.0007
[1/5][450/469]  Loss_D: 0.0013  Loss_G: -0.0008 D(x): 0.9995    D(G(z)): 0.00
08 / 0.0008
[2/5][0/469]    Loss_D: 0.0012  Loss_G: -0.0007 D(x): 0.9996    D(G(z)): 0.00
07 / 0.0007
[2/5][50/469]   Loss_D: 0.0010  Loss_G: -0.0006 D(x): 0.9996    D(G(z)): 0.00
06 / 0.0006
[2/5][100/469]  Loss_D: 0.0010  Loss_G: -0.0006 D(x): 0.9996    D(G(z)): 0.00
06 / 0.0006
[2/5][150/469]  Loss_D: 0.0009  Loss_G: -0.0005 D(x): 0.9996    D(G(z)): 0.00
05 / 0.0005
[2/5][200/469]  Loss_D: 0.0009  Loss_G: -0.0005 D(x): 0.9996    D(G(z)): 0.00
05 / 0.0005
[2/5][250/469]  Loss_D: 0.0008  Loss_G: -0.0004 D(x): 0.9997    D(G(z)): 0.00
04 / 0.0004
[2/5][300/469]  Loss_D: 0.0007  Loss_G: -0.0004 D(x): 0.9997    D(G(z)): 0.00
04 / 0.0004
[2/5][350/469]  Loss_D: 0.0007  Loss_G: -0.0004 D(x): 0.9997    D(G(z)): 0.00
04 / 0.0004
[2/5][400/469]  Loss_D: 0.0007  Loss_G: -0.0004 D(x): 0.9997    D(G(z)): 0.00
04 / 0.0004
[2/5][450/469]  Loss_D: 0.0006  Loss_G: -0.0004 D(x): 0.9998    D(G(z)): 0.00
04 / 0.0004
[3/5][0/469]    Loss_D: 0.0007  Loss_G: -0.0004 D(x): 0.9997    D(G(z)): 0.00
04 / 0.0004
[3/5][50/469]   Loss_D: 0.0005  Loss_G: -0.0003 D(x): 0.9998    D(G(z)): 0.00
03 / 0.0003
[3/5][100/469]  Loss_D: 0.0005  Loss_G: -0.0003 D(x): 0.9997    D(G(z)): 0.00
03 / 0.0003
[3/5][150/469]  Loss_D: 0.0006  Loss_G: -0.0003 D(x): 0.9998    D(G(z)): 0.00
03 / 0.0003
[3/5][200/469]  Loss_D: 0.0005  Loss_G: -0.0003 D(x): 0.9998    D(G(z)): 0.00
03 / 0.0003
[3/5][250/469]  Loss_D: 0.0005  Loss_G: -0.0003 D(x): 0.9998    D(G(z)): 0.00
03 / 0.0003
```

```
[3/5][300/469]  Loss_D: 0.0004  Loss_G: -0.0002 D(x): 0.9998   D(G(z)): 0.00
02 / 0.0002
[3/5][350/469]  Loss_D: 0.0004  Loss_G: -0.0002 D(x): 0.9999   D(G(z)): 0.00
02 / 0.0002
[3/5][400/469]  Loss_D: 0.0004  Loss_G: -0.0002 D(x): 0.9998   D(G(z)): 0.00
02 / 0.0002
[3/5][450/469]  Loss_D: 0.0004  Loss_G: -0.0002 D(x): 0.9998   D(G(z)): 0.00
02 / 0.0002
[4/5][0/469]    Loss_D: 0.0004  Loss_G: -0.0002 D(x): 0.9998   D(G(z)): 0.00
02 / 0.0002
[4/5][50/469]   Loss_D: 0.0003  Loss_G: -0.0002 D(x): 0.9999   D(G(z)): 0.00
02 / 0.0002
[4/5][100/469]  Loss_D: 0.0003  Loss_G: -0.0002 D(x): 0.9999   D(G(z)): 0.00
02 / 0.0002
[4/5][150/469]  Loss_D: 0.0003  Loss_G: -0.0002 D(x): 0.9999   D(G(z)): 0.00
02 / 0.0002
[4/5][200/469]  Loss_D: 0.0003  Loss_G: -0.0002 D(x): 0.9999   D(G(z)): 0.00
02 / 0.0002
[4/5][250/469]  Loss_D: 0.0003  Loss_G: -0.0002 D(x): 0.9999   D(G(z)): 0.00
02 / 0.0002
[4/5][300/469]  Loss_D: 0.0003  Loss_G: -0.0001 D(x): 0.9999   D(G(z)): 0.00
01 / 0.0001
[4/5][350/469]  Loss_D: 0.0002  Loss_G: -0.0001 D(x): 0.9999   D(G(z)): 0.00
01 / 0.0001
[4/5][400/469]  Loss_D: 0.0002  Loss_G: -0.0001 D(x): 0.9999   D(G(z)): 0.00
01 / 0.0001
[4/5][450/469]  Loss_D: 0.0002  Loss_G: -0.0001 D(x): 0.9999   D(G(z)): 0.00
01 / 0.0001
```

In [13]:
```
# plot the loss for generator and discriminator
plot_GAN_loss([G_losses, D_losses], ["G", "D"])

# Grab a batch of real images from the dataloader
plot_real_fake_images(next(iter(dataloader)), img_list)
```







In [ ]:
```

```

Task 1.4 Ablation study on the weight initialization

1. Please use the function `initialize_net` provided in Task 1.0 to initialize the generator and discriminator function without weight initialization (HINT: There is no need to modify the code for `initialize_net` function).
2. Train the model with modified networks and visualize the results.

In [18]:
```
################################## YOUR CODE ##################################
netD_woinit = initialize_net(Discriminator, None, device, ngpu)
netG_woinit = initialize_net(Generator, None, device, ngpu)
############################  END YOUR CODE ################################
```

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(1, 8, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias
=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(8, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bia
s=False)
    (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bi
as=False)
    (6): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(32, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (9): Sigmoid()
  )
)
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 32, kernel_size=(4, 4), stride=(1, 1), bias=Fal
se)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(32, 16, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1), bias=False)
    (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(16, 8, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1), bias=False)
    (7): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(8, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (10): Tanh()
  )
)
```

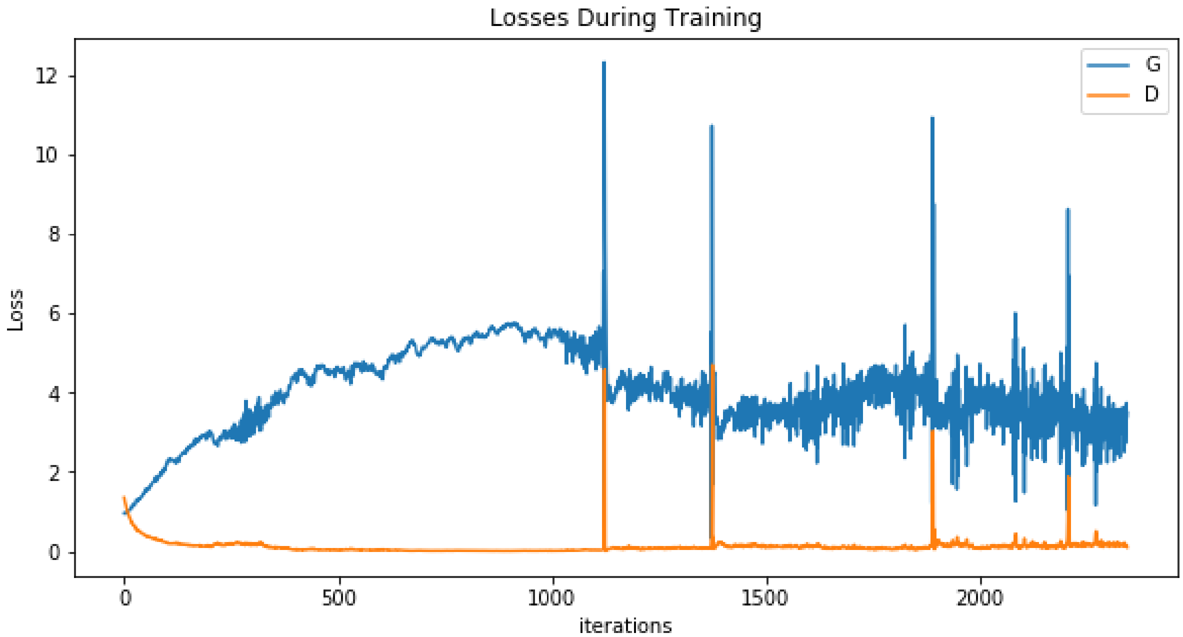
In [19]:
```
# Setup Adam optimizers for both G and D
optimizerD_woinit = optim.Adam(netD_woinit.parameters(), lr=lr, betas=(beta1,
0.999))
optimizerG_woinit = optim.Adam(netG_woinit.parameters(), lr=lr, betas=(beta1,
0.999))

# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
        netD_woinit.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, device=device)
        # Forward pass real batch through D
        output = netD_woinit(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG_woinit(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD_woinit(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Add the gradients from the all-real and all-fake batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD_woinit.step()

        ############################
```

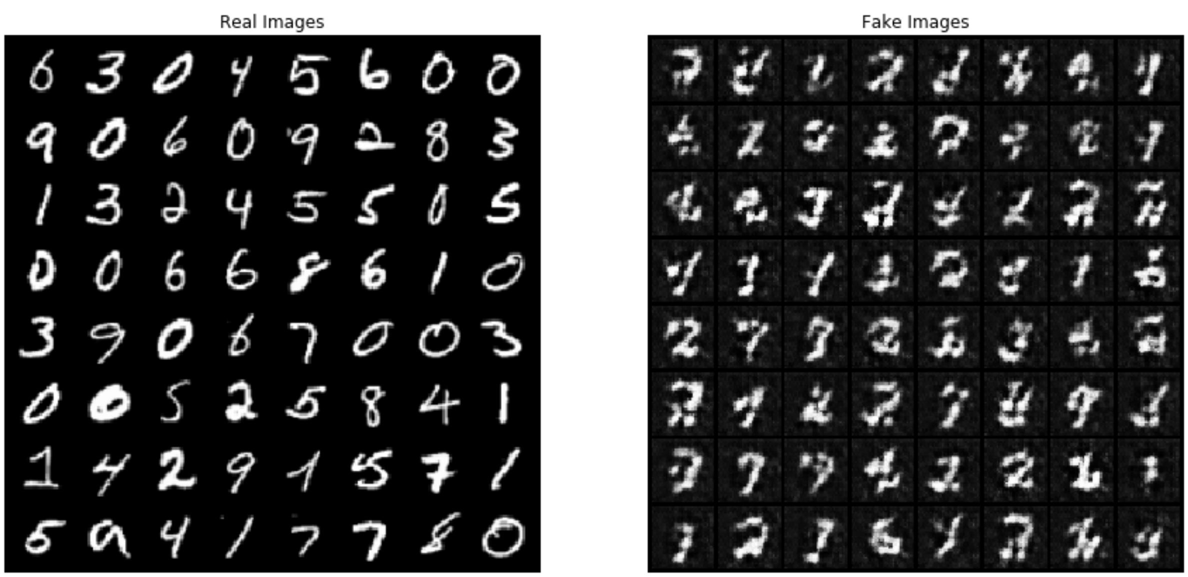
```
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG_woinit.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost
        # Since we just updated D, perform another forward pass of all-fake ba
tch through D
        output = netD_woinit(fake).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG_woinit.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G
(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

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

        iters += 1
```

```
Starting Training Loop...
[0/5][0/469]    Loss_D: 1.3569  Loss_G: 0.9581  D(x): 0.4424    D(G(z)): 0.40
16 / 0.3917
[0/5][50/469]   Loss_D: 0.3826  Loss_G: 1.5581  D(x): 0.8935    D(G(z)): 0.23
40 / 0.2161
[0/5][100/469]  Loss_D: 0.2050  Loss_G: 2.2244  D(x): 0.9373    D(G(z)): 0.12
92 / 0.1130
[0/5][150/469]  Loss_D: 0.1600  Loss_G: 2.5938  D(x): 0.9438    D(G(z)): 0.09
49 / 0.0789
[0/5][200/469]  Loss_D: 0.1197  Loss_G: 2.9223  D(x): 0.9609    D(G(z)): 0.07
43 / 0.0589
[0/5][250/469]  Loss_D: 0.1929  Loss_G: 2.9358  D(x): 0.9060    D(G(z)): 0.07
77 / 0.0605
[0/5][300/469]  Loss_D: 0.1536  Loss_G: 3.3917  D(x): 0.9219    D(G(z)): 0.06
48 / 0.0386
[0/5][350/469]  Loss_D: 0.0882  Loss_G: 3.8163  D(x): 0.9542    D(G(z)): 0.03
71 / 0.0247
[0/5][400/469]  Loss_D: 0.0380  Loss_G: 4.3452  D(x): 0.9835    D(G(z)): 0.02
06 / 0.0145
[0/5][450/469]  Loss_D: 0.0389  Loss_G: 4.6153  D(x): 0.9810    D(G(z)): 0.01
90 / 0.0118
[1/5][0/469]    Loss_D: 0.0354  Loss_G: 4.5837  D(x): 0.9810    D(G(z)): 0.01
55 / 0.0120
[1/5][50/469]   Loss_D: 0.0400  Loss_G: 4.3595  D(x): 0.9883    D(G(z)): 0.02
72 / 0.0151
[1/5][100/469]  Loss_D: 0.0266  Loss_G: 4.5798  D(x): 0.9892    D(G(z)): 0.01
51 / 0.0111
[1/5][150/469]  Loss_D: 0.0378  Loss_G: 4.7847  D(x): 0.9829    D(G(z)): 0.01
41 / 0.0094
[1/5][200/469]  Loss_D: 0.0119  Loss_G: 5.2471  D(x): 0.9954    D(G(z)): 0.00
73 / 0.0057
[1/5][250/469]  Loss_D: 0.0100  Loss_G: 5.3343  D(x): 0.9966    D(G(z)): 0.00
65 / 0.0052
[1/5][300/469]  Loss_D: 0.0128  Loss_G: 5.3213  D(x): 0.9951    D(G(z)): 0.00
78 / 0.0052
[1/5][350/469]  Loss_D: 0.0130  Loss_G: 5.4676  D(x): 0.9946    D(G(z)): 0.00
75 / 0.0047
[1/5][400/469]  Loss_D: 0.0077  Loss_G: 5.5294  D(x): 0.9973    D(G(z)): 0.00
50 / 0.0042
[1/5][450/469]  Loss_D: 0.0098  Loss_G: 5.6918  D(x): 0.9951    D(G(z)): 0.00
48 / 0.0036
[2/5][0/469]    Loss_D: 0.0114  Loss_G: 5.2923  D(x): 0.9968    D(G(z)): 0.00
82 / 0.0055
[2/5][50/469]   Loss_D: 0.0093  Loss_G: 5.4973  D(x): 0.9968    D(G(z)): 0.00
61 / 0.0043
[2/5][100/469]  Loss_D: 0.0143  Loss_G: 5.3100  D(x): 0.9947    D(G(z)): 0.00
89 / 0.0054
[2/5][150/469]  Loss_D: 0.0342  Loss_G: 4.9135  D(x): 0.9807    D(G(z)): 0.01
42 / 0.0082
[2/5][200/469]  Loss_D: 0.0759  Loss_G: 3.8271  D(x): 0.9733    D(G(z)): 0.04
61 / 0.0253
[2/5][250/469]  Loss_D: 0.0886  Loss_G: 3.9323  D(x): 0.9503    D(G(z)): 0.03
05 / 0.0257
[2/5][300/469]  Loss_D: 0.0675  Loss_G: 4.1724  D(x): 0.9698    D(G(z)): 0.03
44 / 0.0209
[2/5][350/469]  Loss_D: 0.1036  Loss_G: 3.6430  D(x): 0.9491    D(G(z)): 0.04
26 / 0.0344
```

```
[2/5][400/469]  Loss_D: 0.0748  Loss_G: 3.8228  D(x): 0.9742   D(G(z)): 0.04
59 / 0.0276
[2/5][450/469]  Loss_D: 0.1798  Loss_G: 3.0636  D(x): 0.9369   D(G(z)): 0.10
32 / 0.0553
[3/5][0/469]    Loss_D: 0.1467  Loss_G: 3.2363  D(x): 0.9337   D(G(z)): 0.06
64 / 0.0508
[3/5][50/469]   Loss_D: 0.1483  Loss_G: 3.8654  D(x): 0.9357   D(G(z)): 0.06
95 / 0.0311
[3/5][100/469]  Loss_D: 0.1401  Loss_G: 3.0381  D(x): 0.9309   D(G(z)): 0.05
80 / 0.0585
[3/5][150/469]  Loss_D: 0.0910  Loss_G: 4.2358  D(x): 0.9710   D(G(z)): 0.05
72 / 0.0198
[3/5][200/469]  Loss_D: 0.1488  Loss_G: 2.8348  D(x): 0.9139   D(G(z)): 0.04
70 / 0.0831
[3/5][250/469]  Loss_D: 0.1099  Loss_G: 2.9160  D(x): 0.9212   D(G(z)): 0.01
77 / 0.0701
[3/5][300/469]  Loss_D: 0.0954  Loss_G: 3.9063  D(x): 0.9425   D(G(z)): 0.02
80 / 0.0259
[3/5][350/469]  Loss_D: 0.0597  Loss_G: 4.2697  D(x): 0.9834   D(G(z)): 0.04
06 / 0.0200
[3/5][400/469]  Loss_D: 0.0492  Loss_G: 4.0862  D(x): 0.9697   D(G(z)): 0.01
72 / 0.0235
[3/5][450/469]  Loss_D: 0.0775  Loss_G: 3.9485  D(x): 0.9674   D(G(z)): 0.03
98 / 0.0286
[4/5][0/469]    Loss_D: 0.0941  Loss_G: 3.6881  D(x): 0.9330   D(G(z)): 0.01
75 / 0.0355
[4/5][50/469]   Loss_D: 0.0999  Loss_G: 3.4199  D(x): 0.9411   D(G(z)): 0.03
19 / 0.0506
[4/5][100/469]  Loss_D: 0.1172  Loss_G: 3.4347  D(x): 0.9255   D(G(z)): 0.03
05 / 0.0482
[4/5][150/469]  Loss_D: 0.0930  Loss_G: 3.8003  D(x): 0.9764   D(G(z)): 0.06
35 / 0.0337
[4/5][200/469]  Loss_D: 0.0839  Loss_G: 3.9679  D(x): 0.9536   D(G(z)): 0.03
29 / 0.0293
[4/5][250/469]  Loss_D: 0.1436  Loss_G: 4.0866  D(x): 0.9600   D(G(z)): 0.09
09 / 0.0257
[4/5][300/469]  Loss_D: 0.0960  Loss_G: 3.1791  D(x): 0.9422   D(G(z)): 0.03
19 / 0.0584
[4/5][350/469]  Loss_D: 0.1126  Loss_G: 3.9329  D(x): 0.9398   D(G(z)): 0.04
46 / 0.0290
[4/5][400/469]  Loss_D: 0.2129  Loss_G: 2.6459  D(x): 0.9199   D(G(z)): 0.11
23 / 0.0933
[4/5][450/469]  Loss_D: 0.1977  Loss_G: 2.3631  D(x): 0.8789   D(G(z)): 0.04
10 / 0.1248
```

In [20]:
```
# plot the loss for generator and discriminator
plot_GAN_loss([G_losses, D_losses], ["G", "D"])

# Grab a batch of real images from the dataloader
plot_real_fake_images(next(iter(dataloader)), img_list)
```





Exercise 2: Implement the WGAN with weight clipping

Wasserstein GAN ([WGAN (https://arxiv.org/abs/1701.07875)](https://arxiv.org/abs/1701.07875)) is an alternative training strategy to traditional GAN. WGAN may provide more stable learning and may avoid problems faced in traditional GAN training like mode collapse.

1. Rewrite the loss functions and training function according to the algorithm introduced in slide 18 in [Lecture note for WGAN (https://www.davidinouye.com/course/ece57000-fall-2021/lectures/wasserstein-gan.pdf)](https://www.davidinouye.com/course/ece57000-fall-2021/lectures/wasserstein-gan.pdf). A few notes/hints:
   A. Keep the same generator as in Exercise 1, Task 1.0, but modify the discriminator so that there is no restriction on the range of the output. (Simply comment out the last `Sigmoid` layer)
   B. Modify the optimizer to be the RMSProp optimizer with a learning rate equal to the value in `lr_rms` (which we set to `5e-4` , which is larger than the rate in the paper but works better for our purposes).
   C. Use [`torch.Tensor.clamp_()` (https://pytorch.org/docs/stable/generated/torch.Tensor.clamp_.html)](https://pytorch.org/docs/stable/generated/torch.Tensor.clamp_.html) function to clip the parameter values. You will need to do this for all parameters of the discriminator. See algorithm for when to do this.
2. Train the model with modified networks and visualize the results.

In [6]:
```
# Generator Code
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution, state size. nz x 1 x 1
            nn.ConvTranspose2d( nz, ngf * 4, kernel_size=4, stride=1, padding=
0, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True), # inplace ReLU
            # current state size. (ngf*4) x 4 x 4
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # current state size. (ngf*2) x 8 x 8
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # current state size. ngf x 16 x 16
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            # current state size. nc x 32 x 32
            # Produce number between -1 and 1, as pixel values have been norma
lized to be between -1 and 1
            nn.Tanh()
        )

    def forward(self, input):
        return self.main(input)

class Discriminator_WGAN(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator_WGAN, self).__init__()
        self.ngpu = ngpu
        ################################ YOUR CODE ###########################
#####
        self.main = nn.Sequential(
            # input is (nc) x 32 x 32
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 16 x 16
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 8 x 8
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 4 x 4
            nn.Conv2d(ndf * 4, 1, 4, 1, 0, bias=False),
            # state size. (ndf*4) x 1 x 1
            #nn.Sigmoid()  # Produce probability
        )
        ########################### END YOUR CODE ############################
####
```

```
    def forward(self, input):
        return self.main(input)

netG = initialize_net(Generator, weights_init, device, ngpu)
netD = initialize_net(Discriminator_WGAN, weights_init, device, ngpu)
```

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 32, kernel_size=(4, 4), stride=(1, 1), bias=Fal
se)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(32, 16, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1), bias=False)
    (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(16, 8, kernel_size=(4, 4), stride=(2, 2), padding=
(1, 1), bias=False)
    (7): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(8, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
    (10): Tanh()
  )
)
Discriminator_WGAN(
  (main): Sequential(
    (0): Conv2d(1, 8, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias
=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(8, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bia
s=False)
    (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bi
as=False)
    (6): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(32, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
  )
)
```

In [10]:
```
############################ YOUR CODE ############################
# Setup RMSprop optimizers for both netG and netD with given learning rate as
 `lr_rms`
optimizerD = optim.RMSprop(netD.parameters(), lr=lr_rms)
optimizerG = optim.RMSprop(netG.parameters(), lr=lr_rms)
######################## # END YOUR CODE ##########################
real_label = 1.
fake_label = 0.
# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
n_critic = 5
c = 0.01
dataloader_iter = iter(dataloader)

print("Starting Training Loop...")
num_iters = 1000

for iters in range(num_iters):

    ##########################################################################
#
    # (1) Train Discriminator more: minimize -(mean(D(real))-mean(D(fake)))
    ##########################################################################
#

    for p in netD.parameters():
        p.requires_grad = True

    for idx_critic in range(n_critic):

        netD.zero_grad()

        try:
            data = next(dataloader_iter)
        except StopIteration:
            dataloader_iter = iter(dataloader)
            data = next(dataloader_iter)

        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        D_real = netD(real_cpu).view(-1)

        noise = torch.randn(b_size, nz, 1, 1, device=device)
        fake = netG(noise)
        D_fake = netD(fake).view(-1)

        ############################ YOUR CODE ############################
        # Define your loss function for variable `D_loss`
        #label = torch.full((b_size,), real_label, dtype=torch.float, device=d
evice)
        #D_real = criterion(D_real, label)
        #D_real.backward()
```

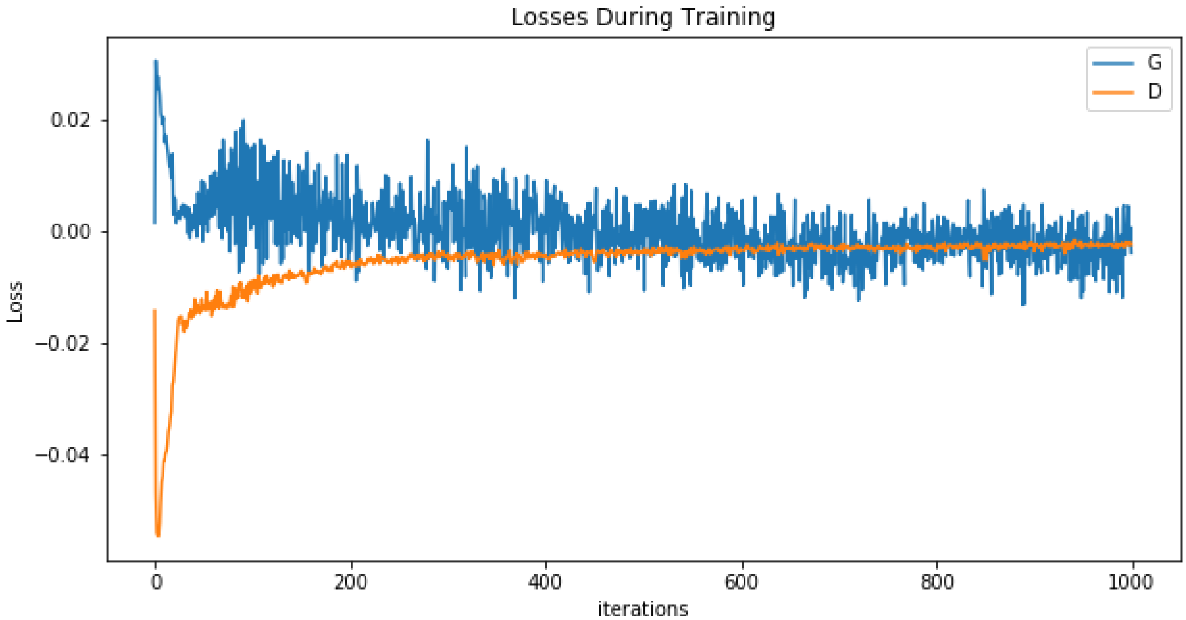
```
        #D_fake = netD(fake.detach()).view(-1)
        #label.fill_(fake_label)
        #D_fake = criterion(D_fake, label)
        #D_fake.backward()

        D_loss = -(D_real.mean() - D_fake.mean())

        # Backpropagate the loss function and upate the optimizer
        D_loss.backward()

        optimizerD.step()
        # Clip the gradient with limit `c` by using `clamp_()` function
        for p in netD.parameters():
            p.data.clamp_(-c, c)

        ######################## # END YOUR CODE ##########################

    ##########################################################################
#
    # (2) Update G network: minimize -mean(D(fake)) (Update only once in 5 epo
chs)
    ##########################################################################
#
    for p in netD.parameters():
        p.requires_grad = False

    netG.zero_grad()

    noise = torch.randn(b_size, nz, 1, 1, device=device)
    fake = netG(noise)
    D_fake = netD(fake).view(-1)

    ################################ YOUR CODE ###############################
#
    # Define your loss function for variable `G_loss`
    G_loss = -(D_fake.mean())

    # Backpropagate the loss function and upate the optimizer
    G_loss.backward()
    optimizerG.step()

    ############################# END YOUR CODE ##############################

    # Output training stats
    if iters % 10 == 0:
        print('[%4d/%4d]   Loss_D: %6.4f    Loss_G: %6.4f'
            % (iters, num_iters, D_loss.item(), G_loss.item()))

    # Save Losses for plotting later
    G_losses.append(G_loss.item())
    D_losses.append(D_loss.item())

    # Check how the generator is doing by saving G's output on fixed_noise
    if (iters % 100 == 0):
        with torch.no_grad():
```

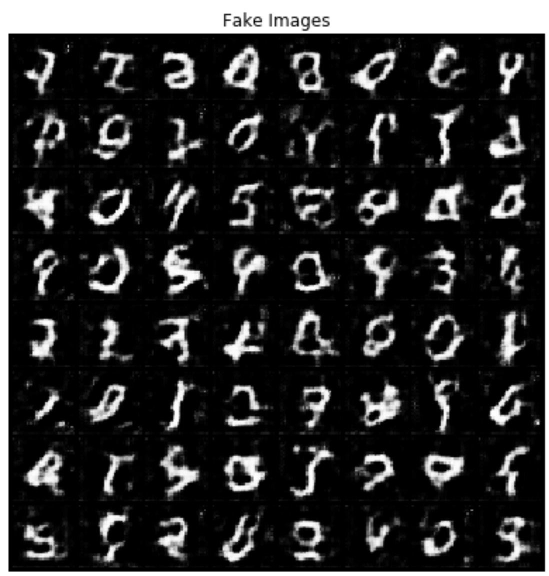
```
            fake = netG(fixed_noise).detach().cpu()
        img_list.append(vutils.make_grid(fake, padding=2, normalize=True))
```

```
Starting Training Loop...
[   0/1000]   Loss_D: -0.0143    Loss_G: 0.0015
[  10/1000]   Loss_D: -0.0412    Loss_G: 0.0160
[  20/1000]   Loss_D: -0.0248    Loss_G: 0.0061
[  30/1000]   Loss_D: -0.0181    Loss_G: 0.0029
[  40/1000]   Loss_D: -0.0151    Loss_G: 0.0005
[  50/1000]   Loss_D: -0.0139    Loss_G: 0.0088
[  60/1000]   Loss_D: -0.0123    Loss_G: 0.0105
[  70/1000]   Loss_D: -0.0134    Loss_G: 0.0164
[  80/1000]   Loss_D: -0.0100    Loss_G: 0.0138
[  90/1000]   Loss_D: -0.0114    Loss_G: -0.0056
[ 100/1000]   Loss_D: -0.0105    Loss_G: 0.0083
[ 110/1000]   Loss_D: -0.0093    Loss_G: 0.0151
[ 120/1000]   Loss_D: -0.0085    Loss_G: 0.0126
[ 130/1000]   Loss_D: -0.0093    Loss_G: 0.0053
[ 140/1000]   Loss_D: -0.0081    Loss_G: -0.0021
[ 150/1000]   Loss_D: -0.0070    Loss_G: 0.0067
[ 160/1000]   Loss_D: -0.0073    Loss_G: 0.0082
[ 170/1000]   Loss_D: -0.0078    Loss_G: 0.0044
[ 180/1000]   Loss_D: -0.0064    Loss_G: 0.0070
[ 190/1000]   Loss_D: -0.0067    Loss_G: -0.0007
[ 200/1000]   Loss_D: -0.0054    Loss_G: -0.0037
[ 210/1000]   Loss_D: -0.0050    Loss_G: 0.0074
[ 220/1000]   Loss_D: -0.0054    Loss_G: -0.0008
[ 230/1000]   Loss_D: -0.0059    Loss_G: -0.0035
[ 240/1000]   Loss_D: -0.0052    Loss_G: -0.0028
[ 250/1000]   Loss_D: -0.0051    Loss_G: -0.0030
[ 260/1000]   Loss_D: -0.0048    Loss_G: 0.0017
[ 270/1000]   Loss_D: -0.0049    Loss_G: -0.0016
[ 280/1000]   Loss_D: -0.0045    Loss_G: -0.0050
[ 290/1000]   Loss_D: -0.0036    Loss_G: 0.0090
[ 300/1000]   Loss_D: -0.0041    Loss_G: -0.0067
[ 310/1000]   Loss_D: -0.0048    Loss_G: 0.0048
[ 320/1000]   Loss_D: -0.0042    Loss_G: 0.0007
[ 330/1000]   Loss_D: -0.0056    Loss_G: 0.0117
[ 340/1000]   Loss_D: -0.0044    Loss_G: 0.0002
[ 350/1000]   Loss_D: -0.0041    Loss_G: 0.0091
[ 360/1000]   Loss_D: -0.0050    Loss_G: 0.0085
[ 370/1000]   Loss_D: -0.0040    Loss_G: 0.0090
[ 380/1000]   Loss_D: -0.0044    Loss_G: -0.0046
[ 390/1000]   Loss_D: -0.0043    Loss_G: 0.0066
[ 400/1000]   Loss_D: -0.0043    Loss_G: -0.0003
[ 410/1000]   Loss_D: -0.0045    Loss_G: -0.0030
[ 420/1000]   Loss_D: -0.0032    Loss_G: 0.0012
[ 430/1000]   Loss_D: -0.0037    Loss_G: -0.0035
[ 440/1000]   Loss_D: -0.0039    Loss_G: -0.0078
[ 450/1000]   Loss_D: -0.0056    Loss_G: 0.0004
[ 460/1000]   Loss_D: -0.0037    Loss_G: -0.0032
[ 470/1000]   Loss_D: -0.0041    Loss_G: -0.0002
[ 480/1000]   Loss_D: -0.0042    Loss_G: 0.0062
[ 490/1000]   Loss_D: -0.0040    Loss_G: -0.0030
[ 500/1000]   Loss_D: -0.0040    Loss_G: 0.0039
[ 510/1000]   Loss_D: -0.0029    Loss_G: 0.0010
[ 520/1000]   Loss_D: -0.0037    Loss_G: -0.0098
[ 530/1000]   Loss_D: -0.0031    Loss_G: 0.0062
[ 540/1000]   Loss_D: -0.0040    Loss_G: 0.0061
[ 550/1000]   Loss_D: -0.0040    Loss_G: -0.0065
```

```
[ 560/1000]   Loss_D: -0.0037    Loss_G: -0.0031
[ 570/1000]   Loss_D: -0.0037    Loss_G: 0.0037
[ 580/1000]   Loss_D: -0.0038    Loss_G: -0.0070
[ 590/1000]   Loss_D: -0.0033    Loss_G: -0.0042
[ 600/1000]   Loss_D: -0.0034    Loss_G: -0.0010
[ 610/1000]   Loss_D: -0.0034    Loss_G: 0.0004
[ 620/1000]   Loss_D: -0.0029    Loss_G: -0.0046
[ 630/1000]   Loss_D: -0.0034    Loss_G: 0.0027
[ 640/1000]   Loss_D: -0.0033    Loss_G: -0.0074
[ 650/1000]   Loss_D: -0.0033    Loss_G: -0.0061
[ 660/1000]   Loss_D: -0.0032    Loss_G: -0.0090
[ 670/1000]   Loss_D: -0.0025    Loss_G: -0.0066
[ 680/1000]   Loss_D: -0.0029    Loss_G: -0.0023
[ 690/1000]   Loss_D: -0.0030    Loss_G: -0.0096
[ 700/1000]   Loss_D: -0.0031    Loss_G: -0.0079
[ 710/1000]   Loss_D: -0.0027    Loss_G: -0.0059
[ 720/1000]   Loss_D: -0.0028    Loss_G: -0.0125
[ 730/1000]   Loss_D: -0.0027    Loss_G: -0.0034
[ 740/1000]   Loss_D: -0.0026    Loss_G: -0.0027
[ 750/1000]   Loss_D: -0.0030    Loss_G: -0.0022
[ 760/1000]   Loss_D: -0.0026    Loss_G: -0.0004
[ 770/1000]   Loss_D: -0.0027    Loss_G: -0.0013
[ 780/1000]   Loss_D: -0.0025    Loss_G: -0.0057
[ 790/1000]   Loss_D: -0.0025    Loss_G: -0.0010
[ 800/1000]   Loss_D: -0.0033    Loss_G: -0.0005
[ 810/1000]   Loss_D: -0.0029    Loss_G: -0.0045
[ 820/1000]   Loss_D: -0.0025    Loss_G: -0.0014
[ 830/1000]   Loss_D: -0.0024    Loss_G: -0.0007
[ 840/1000]   Loss_D: -0.0027    Loss_G: -0.0045
[ 850/1000]   Loss_D: -0.0035    Loss_G: -0.0030
[ 860/1000]   Loss_D: -0.0022    Loss_G: 0.0008
[ 870/1000]   Loss_D: -0.0022    Loss_G: 0.0040
[ 880/1000]   Loss_D: -0.0029    Loss_G: -0.0048
[ 890/1000]   Loss_D: -0.0019    Loss_G: -0.0132
[ 900/1000]   Loss_D: -0.0037    Loss_G: -0.0055
[ 910/1000]   Loss_D: -0.0027    Loss_G: -0.0050
[ 920/1000]   Loss_D: -0.0022    Loss_G: 0.0006
[ 930/1000]   Loss_D: -0.0032    Loss_G: -0.0056
[ 940/1000]   Loss_D: -0.0023    Loss_G: -0.0085
[ 950/1000]   Loss_D: -0.0024    Loss_G: -0.0107
[ 960/1000]   Loss_D: -0.0026    Loss_G: -0.0080
[ 970/1000]   Loss_D: -0.0023    Loss_G: -0.0011
[ 980/1000]   Loss_D: -0.0023    Loss_G: -0.0013
[ 990/1000]   Loss_D: -0.0020    Loss_G: -0.0119
```

In [11]:
```
# plot the loss for generator and discriminator
plot_GAN_loss([G_losses, D_losses], ["G", "D"])

# Grab a batch of real images from the dataloader
plot_real_fake_images(next(iter(dataloader)), img_list)
```





(Optional and ungraded) Exercise 3: Implement the WGAN with Gradient Penalty

1. Use slide 20 in [Lecture note for WGAN (https://www.davidinouye.com/course/ece57000-fall-2021/lectures/wasserstein-gan.pdf)](https://www.davidinouye.com/course/ece57000-fall-2021/lectures/wasserstein-gan.pdf) to implement WGAN-GP algorithm.
   A. Use the same discriminator and generator as in Exercise 2.
   B. Use Adam optimizer for WGAN-GP.
   C. If implemented correctly, we have setup some hyperparameters (different than the original algorithm) that seem to work in this situation.
   D. For calculating the gradient penalty term, you will need to:
      a. Create a batch of interpolated samples.
      b. Pass this interpolated batch through the discriminator.
      c. Compute the gradient of the discriminator with respect to the samples using [`torch.autograd.grad` (https://pytorch.org/docs/stable/generated/torch.autograd.grad.html)](https://pytorch.org/docs/stable/generated/torch.autograd.grad.html). You will need to set:
         i. `outputs`
         ii. `inputs`
         iii. `grad_outputs`
         iv. `create_graph=True` and `retain_graph=True` (because we want to backprop through this gradient calculation for the final objective.)
         v. Hint: Also make sure to understand the return result of this function to extract the gradients as necessary.
      d. Compute the gradient penalty (Hint: For numerical stability, we found that `grad_norm = torch.sqrt((grad**2).sum(1) + 1e-14)` is a simple way to compute the norm.)
      e. Use $\lambda = 10$ for the gradient penalty as in the original paper.
2. Train the model with modified networks and visualize the results.

In [20]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

t = torch.tensor([1,2], device=device)
```

In [ ]:
```
# Setup networks for WGAN-GP
netG = initialize_net(Generator, weights_init, device, ngpu)
netD = initialize_net(Discriminator_WGAN, weights_init, device, ngpu)

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=5e-4, betas=(0.5, 0.9))
optimizerG = optim.Adam(netG.parameters(), lr=5e-4, betas=(0.5, 0.9))

# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
n_critic = 5
dataloader_iter = iter(dataloader)

print("Starting Training Loop...")
num_iters = 1000

for iters in range(num_iters):

    ###########################################################################
#
    # (1) Train Discriminator more: minimize -(mean(D(real))-mean(D(fake)))+GP
    ###########################################################################
#

    for p in netD.parameters():
        p.requires_grad = True

    for idx_critic in range(n_critic):

        netD.zero_grad()

        try:
            data = next(dataloader_iter)
        except StopIteration:
            dataloader_iter = iter(dataloader)

        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        D_real = netD(real_cpu).view(-1)

        noise = torch.randn(b_size, nz, 1, 1, device=device)
        fake = netG(noise)
        D_fake = netD(fake).view(-1)

        ############################ YOUR CODE ############################
        # Compute the gradient penalty term

        # Define your loss function for variable `D_loss`

        # Backpropagate the loss function and upate the optimizer
```

```
        ######################## # END YOUR CODE ##########################

    ###########################################################################
#
    # (2) Update G network: minimize -mean(D(fake)) (Update only once in 5 epo
chs)
    ###########################################################################
#
    for p in netD.parameters():
        p.requires_grad = False

    netG.zero_grad()

    noise = torch.randn(b_size, nz, 1, 1, device=device)
    fake = netG(noise)
    D_fake = netD(fake).view(-1)

    ################################ YOUR CODE ###############################
#
    # Define your loss function for variable `G_loss`

    # Backpropagate the loss function and upate the optimizer

    ############################# END YOUR CODE ##############################

    # Output training stats
    if iters % 10 == 0:
        print('[%4d/%4d]   Loss_D: %6.4f    Loss_G: %6.4f'
            % (iters, num_iters, D_loss.item(), G_loss.item()))

    # Save Losses for plotting later
    G_losses.append(G_loss.item())
    D_losses.append(D_loss.item())

    # Check how the generator is doing by saving G's output on fixed_noise
    if (iters % 100 == 0):
        with torch.no_grad():
            fake = netG(fixed_noise).detach().cpu()
        img_list.append(vutils.make_grid(fake, padding=2, normalize=True))
```

In [ ]:

```
# plot the loss for generator and discriminator
plot_GAN_loss([G_losses, D_losses], ["G", "D"])

# Grab a batch of real images from the dataloader
plot_real_fake_images(next(iter(dataloader)), img_list)
```