P3: GAN (30%)

In this problem, we will train a generative adversarial network (GAN) to generate new celebrities after showing it pictures of many real celebrities.

```
%matplotlib inline

from __future__ import print_function

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)
```

```
Random Seed:  999
<torch._C.Generator at 0x7dd8140fd250>
```

(a) Prepare CelebA Dataset

we will use the Celeb-A Faces dataset which can be downloaded from the link. The dataset will download as a file named `img_align_celeba.zip`. Once downloaded, create a directory named `CelebA` and extract the zip file into that directory. Then, set the `dataroot` input for this notebook to the `CelebA` directory you just created. The resulting directory structure should be:

```
/path/to/CelebA
    -> img_align_celeba
        -> 188242.jpg
        -> 173822.jpg
        -> 284702.jpg
        -> 537394.jpg
           ...
```

This is an important step because we will be using the ImageFolder dataset class, which requires there to be subdirectories in the dataset's root folder.

```
!gdown --fuzzy https://drive.google.com/file/d/0B7EVK8r0v71pZjFTYXZWM3FlRnM/view?usp=share_link&resourcekey=0-dYn9z10tMJOBAkviAcfdyQ img_
```

```
/bin/bash: line 1: img_align_celeba.zip: command not found
Downloading...
From: https://drive.google.com/uc?id=0B7EVK8r0v71pZjFTYXZWM3FlRnM
To: /content/img_align_celeba.zip
100% 1.44G/1.44G [00:20<00:00, 71.0MB/s]
```

```
!mkdir ./CelebA
!unzip /content/img_align_celeba.zip -d ./CelebA
```

```
extracting: ./CelebA/img_align_celeba/202047.jpg
extracting: ./CelebA/img_align_celeba/202048.jpg
extracting: ./CelebA/img_align_celeba/202049.jpg
extracting: ./CelebA/img_align_celeba/202050.jpg
extracting: ./CelebA/img_align_celeba/202051.jpg
extracting: ./CelebA/img_align_celeba/202052.jpg
extracting: ./CelebA/img_align_celeba/202053.jpg
extracting: ./CelebA/img_align_celeba/202054.jpg
extracting: ./CelebA/img_align_celeba/202055.jpg
extracting: ./CelebA/img_align_celeba/202056.jpg
extracting: ./CelebA/img_align_celeba/202057.jpg
extracting: ./CelebA/img_align_celeba/202058.jpg
extracting: ./CelebA/img_align_celeba/202059.jpg
extracting: ./CelebA/img_align_celeba/202060.jpg
extracting: ./CelebA/img_align_celeba/202061.jpg
extracting: ./CelebA/img_align_celeba/202062.jpg
extracting: ./CelebA/img_align_celeba/202063.jpg
extracting: ./CelebA/img_align_celeba/202064.jpg
extracting: ./CelebA/img_align_celeba/202065.jpg
extracting: ./CelebA/img_align_celeba/202066.jpg
extracting: ./CelebA/img_align_celeba/202067.jpg
extracting: ./CelebA/img_align_celeba/202068.jpg
extracting: ./CelebA/img_align_celeba/202069.jpg
extracting: ./CelebA/img_align_celeba/202070.jpg
extracting: ./CelebA/img_align_celeba/202071.jpg
extracting: ./CelebA/img_align_celeba/202072.jpg
extracting: ./CelebA/img_align_celeba/202073.jpg
extracting: ./CelebA/img_align_celeba/202074.jpg
extracting: ./CelebA/img_align_celeba/202075.jpg
extracting: ./CelebA/img_align_celeba/202076.jpg
extracting: ./CelebA/img_align_celeba/202077.jpg
extracting: ./CelebA/img_align_celeba/202078.jpg
extracting: ./CelebA/img_align_celeba/202079.jpg
extracting: ./CelebA/img_align_celeba/202080.jpg
extracting: ./CelebA/img_align_celeba/202081.jpg
extracting: ./CelebA/img_align_celeba/202082.jpg
extracting: ./CelebA/img_align_celeba/202083.jpg
extracting: ./CelebA/img_align_celeba/202084.jpg
extracting: ./CelebA/img_align_celeba/202085.jpg
extracting: ./CelebA/img_align_celeba/202086.jpg
extracting: ./CelebA/img_align_celeba/202087.jpg
extracting: ./CelebA/img_align_celeba/202088.jpg
extracting: ./CelebA/img_align_celeba/202089.jpg
extracting: ./CelebA/img_align_celeba/202090.jpg
extracting: ./CelebA/img_align_celeba/202091.jpg
extracting: ./CelebA/img_align_celeba/202092.jpg
extracting: ./CelebA/img_align_celeba/202093.jpg
extracting: ./CelebA/img_align_celeba/202094.jpg
extracting: ./CelebA/img_align_celeba/202095.jpg
extracting: ./CelebA/img_align_celeba/202096.jpg
extracting: ./CelebA/img_align_celeba/202097.jpg
extracting: ./CelebA/img_align_celeba/202098.jpg
extracting: ./CelebA/img_align_celeba/202099.jpg
extracting: ./CelebA/img_align_celeba/202100.jpg
```

Set parameters for the implemented network.

```
# Root directory for dataset
dataroot = "./CelebA"
# Number of workers for dataloader
workers = 2
# Batch size during training
batch_size = 256
# Spatial size of training images. All images will be resized to this size using a transformer.
image_size = 64
# Number of channels in the training images. For color images this is 3
nc = 3
# Size of z latent vector (i.e. size of generator input)
nz = 100
# Learning rate for optimizers
lr = 0.0005
# Beta1 hyperparam for Adam optimizers
beta1 = 0.5
```

Now, we can create the dataset, create the dataloader, set the device to run on, and finally visualize some of the training data.

```
# We can use an image folder dataset the way we have it setup.
# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# 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)[192:], padding=2, normalize=True).cpu(),(1,2,0)))
```

```
<matplotlib.image.AxesImage at 0x7dd72dc71a20>
```

(b) GAN Implementation

1. Weight Initialization

All model weights in GAN should be randomly initialized from a Normal distribution with `mean = 0, stdev = 0.02`. The `weights_init` function takes an initialized model as input and reinitializes all convolutional, convolutional-transpose, and batch normalization layers to meet this criteria. This function is applied to the models immediately after initialization.

```
# 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)
```

2. Generator

The generator G, is designed to map the latent space vector z to data-space. Since our data are images, converting z to data-space means ultimately creating an RGB image with the same size as the training images (i.e. 3x64x64). In practice, this is accomplished through a series of strided two dimensional convolutional transpose layers, each paired with a 2D batch norm layer and a relu activation. The output of the generator is fed through a tanh function to return it to the input data range of $[-1, 1]$.

```
class Generator(nn.Module):
  def __init__(self):
    super(Generator, self).__init__()
    # TODO: finish implementing the network architecture
    self.seq = nn.Sequential(
      # TODO: the first ConvTranspose2d layer will take the noise as input with size of 100
      # so you will need at least one ConvTranspose2d layer, one BatchNorm2d layer followed by a ReLU activation here
      nn.ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False),
      nn.BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
      nn.ReLU(),
      # ENDS HERE
      nn.ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
      nn.BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
      nn.ReLU(),
      nn.ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
      nn.BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
      nn.ReLU(),
      nn.ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
      nn.BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
      nn.ReLU(),
      nn.ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
      nn.Tanh()
    )

  def forward(self, input):
    x = self.seq(input)
    return x
```

```
# Create the generator
netG = Generator().to(device)

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netG.apply(weights_init)

# Print the model
print(netG)
```

```
Generator(
  (seq): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU()
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

```
Generator(
  (seq): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU()
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

3. Discriminator

The discriminator D is a binary classification network that takes an image as input and outputs a scalar probability that the input image is real (as opposed to fake). Here, D takes a 3x64x64 input image, processes it through a series of Conv2d, BatchNorm2d, and ReLU layers, and outputs the final probability through a Sigmoid activation function. This architecture can be extended with more layers if necessary for the problem, but there is significance to the use of the strided convolution, BatchNorm, and ReLUs.

```
class Discriminator(nn.Module):
  def __init__(self):
    super(Discriminator, self).__init__()
    # TODO: finish implementing the network architecture
    self.seq = nn.Sequential(
        nn.Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
        nn.ReLU(),
        nn.Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
        nn.BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
        nn.ReLU(),
        nn.Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
        nn.BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
        nn.ReLU(),
        nn.Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False),
        nn.BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True),
        nn.ReLU(),
        # TODO: the last Conv2d layer of the doscriminator followed by a sigmoid
        nn.Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), padding=(0, 0), bias=False),
        nn.Sigmoid()
        # ENDS HERE
    )

  def forward(self, input):
    x = self.seq(input)
    return x
```

```
# Create the Discriminator
netD = Discriminator().to(device)

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model
print(netD)
```

```
Discriminator(
  (seq): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): ReLU()
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): ReLU()
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): ReLU()
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU()
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
```

```
      (12): Sigmoid()
    )
  )
```

Loss Functions and Optimizers

With D and G setup, we can specify how they learn through the loss functions and optimizers. We will use the Binary Cross Entropy loss (BCELoss) function which is defined in PyTorch as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -\left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)\right]$$

Notice how this function provides the calculation of both log components in the objective function (i.e. $\log(D(x))$ and $\log(1 - D(G(z)))$). We can specify what part of the BCE equation to use with the y input. This is accomplished in the training loop which is coming up soon, but it is important to understand how we can choose which component we wish to calculate just by changing y (i.e. GT labels).

```
# 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.
fake_label = 0.

# 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))
```

5. Training Loop

Finally, now that we have all of the parts of the GAN framework defined, we can train it. Be mindful that training GANs is somewhat of an art form, as incorrect hyperparameter settings lead to mode collapse with little explanation of what went wrong. Here, we will closely follow Algorithm 1 from Goodfellow's paper, while abiding by some of the best practices shown in ganhacks . Namely, we will construct different mini-batches for real and fake images, and also adjust G's objective function to maximize $\log(D(G(z)))$. Training is split up into two main parts. Part 1 updates the Discriminator and Part 2 updates the Generator.

Part 1: Train the Discriminator

Recall, the goal of training the discriminator is to maximize the probability of correctly classifying a given input as real or fake. In terms of Goodfellow, we wish to "update the discriminator by ascending its stochastic gradient". Practically, we want to maximize $\log(D(x)) + \log(1 - D(G(z)))$. Due to the separate mini-batch suggestion from ganhacks, we will calculate this in two steps. First, we will construct a batch of real samples from the training set, forward pass through D, calculate the loss ($\log(D(x))$), then calculate the gradients in a backward pass. Secondly, we will construct a batch of fake samples with the current generator, forward pass this batch through D, calculate the loss ($\log(1 - D(G(z)))$), and *accumulate* the gradients with a backward pass. Now, with the gradients accumulated from both the all-real and all-fake batches, we call a step of the Discriminator's optimizer.

Part 2: Train the Generator

As stated in the original paper, we want to train the Generator by minimizing $\log(1 - D(G(z)))$ in an effort to generate better fakes. As mentioned, this was shown by Goodfellow to not provide sufficient gradients, especially early in the learning process. As a fix, we instead wish to maximize $\log(D(G(z)))$. In the code we accomplish this by: classifying the Generator output from Part 1 with the Discriminator, computing G's loss *using real labels as GT*, computing G's gradients in a backward pass, and finally updating G's parameters with an optimizer step. It may seem counter-intuitive to use the real labels as GT labels for the loss function, but this allows us to use the $\log(x)$ part of the BCELoss (rather than the $\log(1 - x)$ part) which is exactly what we want.

Finally, we will do some statistic reporting and at the end of each epoch we will push our fixed_noise batch through the generator to visually track the progress of G's training. The training statistics reported are:

- **Loss_D** - discriminator loss calculated as the sum of losses for the all real and all fake batches ($\log(D(x)) + \log(D(G(z)))$).
- **Loss_G** - generator loss calculated as $\log(D(G(z)))$
- **D(x)** - the average output (across the batch) of the discriminator for the all real batch. This should start close to 1 then theoretically converge to 0.5 when G gets better. Think about why this is.
- **D(G(z))** - average discriminator outputs for the all fake batch. The first number is before D is updated and the second number is after D is updated. These numbers should start near 0 and converge to 0.5 as G gets better. Think about why this is.

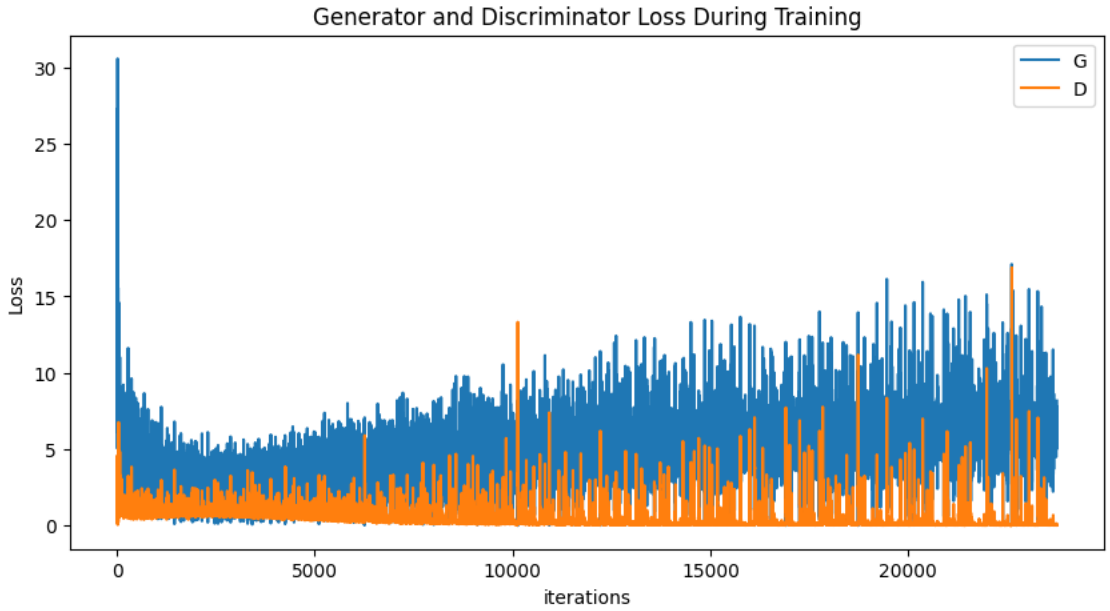
Note: This step might take a while, depending on how many epochs you run and if you removed some data from the dataset.

```
# Training Loop
num_epochs = 30

# 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)))
        ############################################################

        # Initialize the gradient of D
        netD.zero_grad()

        # 1.1 Calculate the loss with all-real batch

        # Format batch
        real_batch = data[0].to(device)
        b_size = real_batch.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
        # Forward pass real batch through D
        output = netD(real_batch).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()

        # 1.2 Calculate the loss 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 and a label array
        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()
        # Calculate the mean of output, D_G_z1, from netD with your fake batch
        D_G_z1 = output.mean().item()

        # 1.3 Combine the loss from all-real batch and all-fake batch
        # Add the gradients from the all-real and all-fake batches
        errD = errD_real + errD_fake
        # Update D with step()
        optimizerD.step()

        ############################################################
        # 2. Update G network: maximize log(D(G(z)))
        ############################################################

        # TODO: Initialize the gradient of G
        netG.zero_grad()
        # TODO: create the label array, remember fake labels are real for generator cost

        b_size = fake.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float, device=device)

        # TODO: Since we just updated D, perform another forward pass of all-fake batch throu$gh D as output (you are not suppose to detac
        output = netD(fake).view(-1)

        # TODO: Calculate G's loss based on this output
        errG = criterion(output, label)
        # TODO: Calculate gradients using backword() for G's loss
        errG.backward()
        # TODO: Calculate the mean of output, D_G_z2, from netD with your fake batch
```

```
        # TODO: Calculate the mean of output D_G_z2 from this batch for printing
        D_G_z2 = output.mean().item()
        # TODO: Update G with step
        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(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1
```

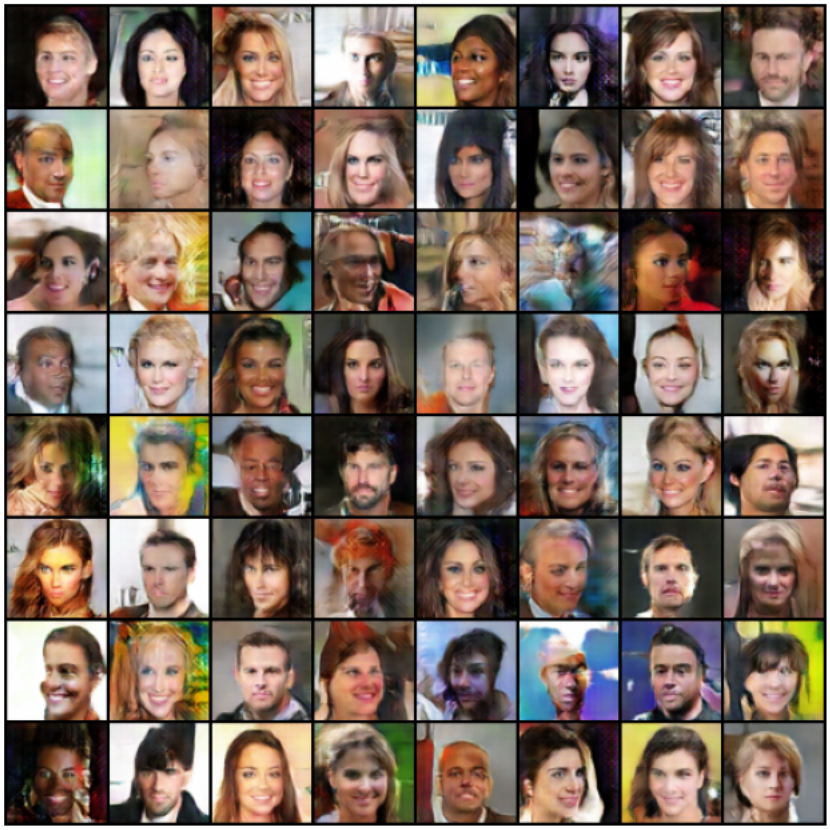
```
[23/30][550/792]        Loss_D: 0.6572  Loss_G: 2.1628  D(x): 0.7085    D(G(z)): 0.1237 / 0.2095
[23/30][600/792]        Loss_D: 0.1144  Loss_G: 5.1625  D(x): 0.9280    D(G(z)): 0.0272 / 0.0153
[23/30][650/792]        Loss_D: 0.0487  Loss_G: 7.0937  D(x): 0.9931    D(G(z)): 0.0356 / 0.0035
[23/30][700/792]        Loss_D: 0.0517  Loss_G: 5.3933  D(x): 0.9803    D(G(z)): 0.0271 / 0.0169
[23/30][750/792]        Loss_D: 0.1566  Loss_G: 3.6292  D(x): 0.8818    D(G(z)): 0.0105 / 0.0869
[24/30][0/792]  Loss_D: 0.1715  Loss_G: 7.8478  D(x): 0.9992    D(G(z)): 0.1277 / 0.0011
[24/30][50/792] Loss_D: 0.2565  Loss_G: 10.5653 D(x): 0.9965    D(G(z)): 0.1685 / 0.0001
[24/30][100/792]        Loss_D: 0.1906  Loss_G: 5.2642  D(x): 0.8647    D(G(z)): 0.0140 / 0.0201
[24/30][150/792]        Loss_D: 0.0520  Loss_G: 6.1463  D(x): 0.9602    D(G(z)): 0.0077 / 0.0090
[24/30][200/792]        Loss_D: 0.0603  Loss_G: 7.4105  D(x): 0.9957    D(G(z)): 0.0451 / 0.0027
[24/30][250/792]        Loss_D: 0.0478  Loss_G: 7.7187  D(x): 0.9716    D(G(z)): 0.0128 / 0.0021
[24/30][300/792]        Loss_D: 0.0760  Loss_G: 5.9679  D(x): 0.9731    D(G(z)): 0.0381 / 0.0096
[24/30][350/792]        Loss_D: 0.0390  Loss_G: 5.6607  D(x): 0.9772    D(G(z)): 0.0136 / 0.0141
[24/30][400/792]        Loss_D: 0.0471  Loss_G: 7.3052  D(x): 0.9879    D(G(z)): 0.0217 / 0.0027
[24/30][450/792]        Loss_D: 0.1189  Loss_G: 7.8220  D(x): 0.9831    D(G(z)): 0.0777 / 0.0013
[24/30][500/792]        Loss_D: 0.0984  Loss_G: 4.7855  D(x): 0.9871    D(G(z)): 0.0656 / 0.0259
[24/30][550/792]        Loss_D: 0.0776  Loss_G: 6.2904  D(x): 0.9850    D(G(z)): 0.0511 / 0.0064
[24/30][600/792]        Loss_D: 0.0662  Loss_G: 5.3100  D(x): 0.9600    D(G(z)): 0.0171 / 0.0212
[24/30][650/792]        Loss_D: 0.0700  Loss_G: 5.7274  D(x): 0.9800    D(G(z)): 0.0180 / 0.0123
[24/30][700/792]        Loss_D: 0.0644  Loss_G: 5.7618  D(x): 0.9978    D(G(z)): 0.0516 / 0.0098
[24/30][750/792]        Loss_D: 0.0333  Loss_G: 5.5257  D(x): 0.9794    D(G(z)): 0.0103 / 0.0147
[25/30][0/792]  Loss_D: 0.2920  Loss_G: 9.9751  D(x): 0.9996    D(G(z)): 0.1975 / 0.0002
[25/30][50/792] Loss_D: 0.0255  Loss_G: 7.8301  D(x): 0.9818    D(G(z)): 0.0062 / 0.0018
[25/30][100/792]        Loss_D: 0.0511  Loss_G: 5.4476  D(x): 0.9714    D(G(z)): 0.0183 / 0.0152
[25/30][150/792]        Loss_D: 0.2562  Loss_G: 6.2525  D(x): 0.9790    D(G(z)): 0.1592 / 0.0061
[25/30][200/792]        Loss_D: 0.0481  Loss_G: 6.0783  D(x): 0.9762    D(G(z)): 0.0194 / 0.0098
[25/30][250/792]        Loss_D: 0.5461  Loss_G: 5.3578  D(x): 0.9328    D(G(z)): 0.2860 / 0.0125
[25/30][300/792]        Loss_D: 0.0339  Loss_G: 7.2994  D(x): 0.9924    D(G(z)): 0.0233 / 0.0034
[25/30][350/792]        Loss_D: 0.6465  Loss_G: 14.6206 D(x): 0.9993    D(G(z)): 0.3711 / 0.0000
[25/30][400/792]        Loss_D: 0.1198  Loss_G: 5.0480  D(x): 0.9842    D(G(z)): 0.0828 / 0.0171
[25/30][450/792]        Loss_D: 0.0429  Loss_G: 5.8961  D(x): 0.9881    D(G(z)): 0.0267 / 0.0110
[25/30][500/792]        Loss_D: 0.0735  Loss_G: 5.8975  D(x): 0.9917    D(G(z)): 0.0576 / 0.0083
[25/30][550/792]        Loss_D: 0.0660  Loss_G: 5.5604  D(x): 0.9585    D(G(z)): 0.0182 / 0.0139
[25/30][600/792]        Loss_D: 0.5295  Loss_G: 7.4288  D(x): 0.9977    D(G(z)): 0.3192 / 0.0016
[25/30][650/792]        Loss_D: 0.1196  Loss_G: 6.7669  D(x): 0.9941    D(G(z)): 0.0786 / 0.0036
[25/30][700/792]        Loss_D: 0.0926  Loss_G: 4.7999  D(x): 0.9559    D(G(z)): 0.0337 / 0.0278
[25/30][750/792]        Loss_D: 0.0481  Loss_G: 5.6837  D(x): 0.9892    D(G(z)): 0.0325 / 0.0128
[26/30][0/792]  Loss_D: 0.0289  Loss_G: 5.8287  D(x): 0.9965    D(G(z)): 0.0241 / 0.0094
[26/30][50/792] Loss_D: 0.0313  Loss_G: 7.3741  D(x): 0.9762    D(G(z)): 0.0056 / 0.0032
[26/30][100/792]        Loss_D: 0.0412  Loss_G: 5.9588  D(x): 0.9771    D(G(z)): 0.0145 / 0.0102
[26/30][150/792]        Loss_D: 0.2954  Loss_G: 2.4196  D(x): 0.7947    D(G(z)): 0.0009 / 0.2300
[26/30][200/792]        Loss_D: 0.0374  Loss_G: 5.8779  D(x): 0.9788    D(G(z)): 0.0139 / 0.0108
[26/30][250/792]        Loss_D: 0.0668  Loss_G: 5.0814  D(x): 0.9584    D(G(z)): 0.0160 / 0.0234
[26/30][300/792]        Loss_D: 0.0239  Loss_G: 6.1323  D(x): 0.9846    D(G(z)): 0.0073 / 0.0082
[26/30][350/792]        Loss_D: 0.0388  Loss_G: 6.0153  D(x): 0.9913    D(G(z)): 0.0279 / 0.0075
[26/30][400/792]        Loss_D: 0.3139  Loss_G: 3.6840  D(x): 0.9221    D(G(z)): 0.1485 / 0.0709
[26/30][450/792]        Loss_D: 0.1083  Loss_G: 5.8080  D(x): 0.9776    D(G(z)): 0.0694 / 0.0083
[26/30][500/792]        Loss_D: 0.0298  Loss_G: 6.8103  D(x): 0.9926    D(G(z)): 0.0204 / 0.0041
[26/30][550/792]        Loss_D: 0.0507  Loss_G: 5.8690  D(x): 0.9919    D(G(z)): 0.0376 / 0.0089
[26/30][600/792]        Loss_D: 0.0423  Loss_G: 6.2575  D(x): 0.9716    D(G(z)): 0.0109 / 0.0096
[26/30][650/792]        Loss_D: 0.0148  Loss_G: 7.1708  D(x): 0.9930    D(G(z)): 0.0067 / 0.0049
[26/30][700/792]        Loss_D: 0.0484  Loss_G: 6.3147  D(x): 0.9971    D(G(z)): 0.0406 / 0.0079
[26/30][750/792]        Loss_D: 0.0553  Loss_G: 7.2559  D(x): 0.9545    D(G(z)): 0.0049 / 0.0039
[27/30][0/792]  Loss_D: 0.4406  Loss_G: 9.6543  D(x): 0.9924    D(G(z)): 0.2608 / 0.0002
[27/30][50/792] Loss_D: 0.0737  Loss_G: 6.7291  D(x): 0.9981    D(G(z)): 0.0604 / 0.0042
[27/30][100/792]        Loss_D: 0.1161  Loss_G: 7.3114  D(x): 0.9826    D(G(z)): 0.0698 / 0.0026
[27/30][150/792]        Loss_D: 0.0695  Loss_G: 6.7972  D(x): 0.9976    D(G(z)): 0.0530 / 0.0040
[27/30][200/792]        Loss_D: 0.2250  Loss_G: 5.6607  D(x): 0.9677    D(G(z)): 0.1399 / 0.0080
[27/30][250/792]        Loss_D: 0.0449  Loss_G: 6.9443  D(x): 0.9892    D(G(z)): 0.0290 / 0.0043
```

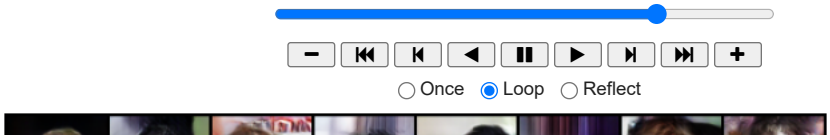
```
1
2 plt.figure(figsize=(10,5))
3 plt.title("Generator and Discriminator Loss During Training")
4 plt.plot(G_losses,label="G")
5 plt.plot(D_losses,label="D")
6 plt.xlabel("iterations")
7 plt.ylabel("Loss")
8 plt.legend()
9 plt.show()
```




```
1 fig = plt.figure(figsize=(8,8))
2 plt.axis("off")
3 ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
4 ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=True)
5
6 HTML(ani.to_jshtml())
```

```
WARNING:matplotlib.animation:Animation size has reached 21210989 bytes, exceeding the limit of 20971520.0. If you're sure you want a la
```






```
# Grab a batch of real images from the dataloader
real_batch = next(iter(dataloader))

# 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], padding=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(img_list[-1],(1,2,0)))
plt.show()
```

1