Lab 10: Implementing GANs for Face Generation

Objectives

This lab aims to provide hands-on experience with Generative Adversarial Networks (GANs) for image generation, focusing on face generation using the CelebA dataset. Students will learn the fundamental architecture and working principles of GANs while implementing and training a DCGAN model using PyTorch. The lab covers exploration of pre-trained GAN models from Hugging Face and experimentation with style transfer techniques, providing practical experience in handling image data and training deep learning models.

Dataset Overview

The CelebFaces Attributes Dataset (CelebA) is a large-scale face attributes dataset with more than 200,000 celebrity images, each with 40 attribute annotations.

Dataset link: https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

Data Dictionary:

- Image Resolution: 178x218 pixels
- Format: JPEG images
- Size: ~1.3 GB
- Attributes: 40 binary attributes including gender, age, hair color, etc.
- Number of Identities: 10,177 unique identities

We use some of the images from this dataset.

Tasks

1. Setup and Data Preparation

- Set up the development environment
- Download and preprocess the CelebA dataset
- Create data loaders and visualization utilities

2. DCGAN Implementation

- Implement the Generator architecture
- Implement the Discriminator architecture
- Define loss functions and optimizers

3. Model Training and Evaluation

- Train the DCGAN model
- Monitor training progress
- Generate and evaluate sample images

4. Advanced Techniques

- Load and use pre-trained models from Hugging Face
- Implement style transfer
- Experiment with different loss functions

Import required libraries

In [1]:
```
#%matplotlib inline
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
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
```

```python
import matplotlib.animation as animation
from IPython.display import HTML
```

In [2]:
```python
# 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.use_deterministic_algorithms(True) # Needed for reproducible results
```

```
Random Seed:  999
```

Setting up some parameters

In [6]:
```python
# Root directory for dataset
dataroot = "Imageceleb\\"

# Number of workers for dataloader
workers = 2

# 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

# 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

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 50

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparameter for Adam optimizers
beta1 = 0.5

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

Loading dataset

In [7]:
```python
# 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() and ngpu > 0) else "cpu")
```

Sample Data

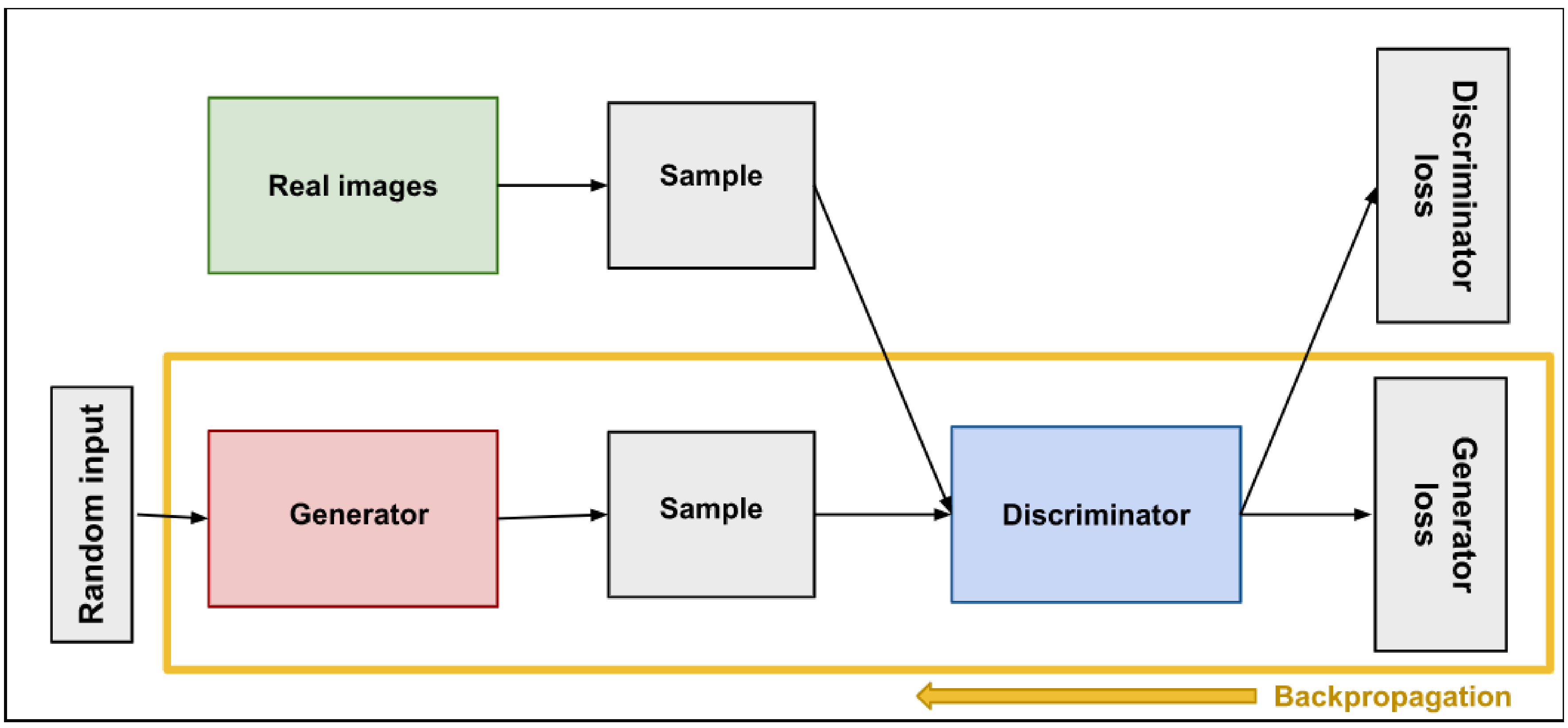
In [8]:
```python
# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padding=2, normalize=True).cpu(),(1,2,0)))
plt.show()
```


```
In [9]: # 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)
```

DCGAN

Normal GANs working

- Generative Adversarial Networks (GANs) are one of the most interesting ideas in computer science today. Two models are trained simultaneously by an adversarial process.
- A generator ("the artist") learns to create images that look real, while a discriminator ("the art critic") learns to tell real images apart from fakes.



Defining Generator

```
In [10]: # 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
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. ``(ngf*8) x 4 x 4``
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. ``(ngf*4) x 8 x 8``
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. ``(ngf*2) x 16 x 16``
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. ``(ngf) x 32 x 32``
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. ``(nc) x 64 x 64``
        )

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

Generator architecture

```
In [11]: # Create the generator
netG = Generator(ngpu).to(device)

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

# Apply the ``weights_init`` function to randomly initialize all weights
#  to ``mean=0``, ``stdev=0.02``.
netG.apply(weights_init)

# Print the model
print(netG)
```

```
Generator(
  (main): 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(inplace=True)
    (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(inplace=True)
    (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(inplace=True)
    (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(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

Defining Discriminator

```
In [12]: class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is ``(nc) x 64 x 64``
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf) x 32 x 32``
            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 16 x 16``
            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 8 x 8``
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*8) x 4 x 4``
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

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

Discriminator architecure

In [13]:
```
# Create the Discriminator
netD = Discriminator(ngpu).to(device)

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

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

# Print the model
print(netD)
```

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (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): LeakyReLU(negative_slope=0.2, inplace=True)
    (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): LeakyReLU(negative_slope=0.2, inplace=True)
    (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): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

Defining loss function and optimizer for Generator and Discriminator

In [14]:
```
# Initialize the ``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))
```

Training of GAN

In [15]:
```
# 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()
```

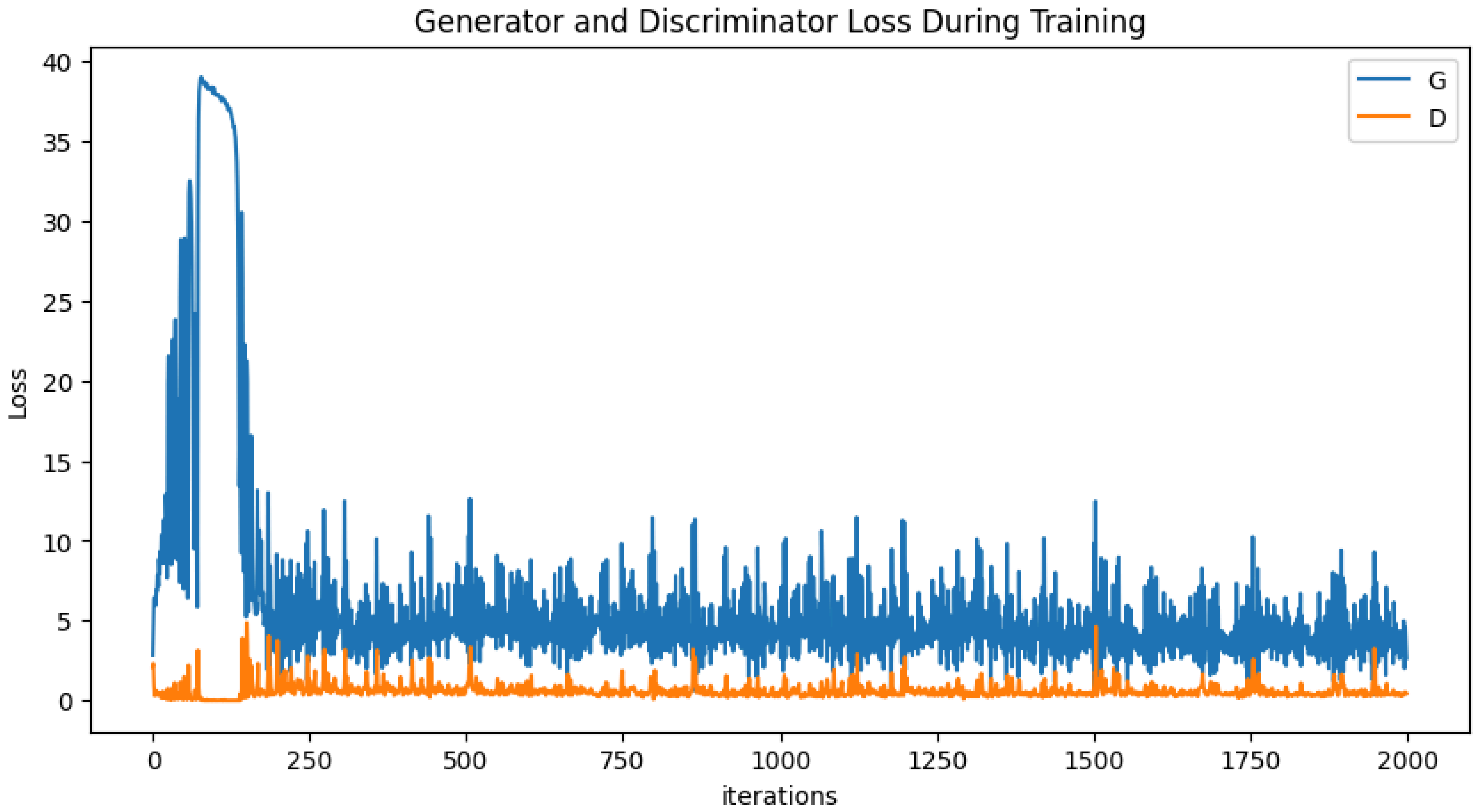
```python
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on 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, accumulated (summed) with previous gradients
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Compute error of D as sum over the fake and the real 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 batch 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(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
```

```
Starting Training Loop...
[0/50][0/40]	Loss_D: 2.0964	Loss_G: 2.8190	D(x): 0.3107	D(G(z)): 0.4352 / 0.0888
[1/50][0/40]	Loss_D: 0.4371	Loss_G: 18.1937	D(x): 0.9660	D(G(z)): 0.2885 / 0.0000
[2/50][0/40]	Loss_D: 0.0185	Loss_G: 38.6524	D(x): 0.9874	D(G(z)): 0.0000 / 0.0000
[3/50][0/40]	Loss_D: 0.0011	Loss_G: 36.9945	D(x): 0.9990	D(G(z)): 0.0000 / 0.0000
[4/50][0/40]	Loss_D: 0.4010	Loss_G: 7.0573	D(x): 0.9655	D(G(z)): 0.1683 / 0.0237
[5/50][0/40]	Loss_D: 1.6104	Loss_G: 7.5502	D(x): 0.9608	D(G(z)): 0.7127 / 0.0012
[6/50][0/40]	Loss_D: 0.3385	Loss_G: 4.1953	D(x): 0.9113	D(G(z)): 0.1821 / 0.0242
[7/50][0/40]	Loss_D: 1.4737	Loss_G: 5.7406	D(x): 0.3943	D(G(z)): 0.0126 / 0.0070
[8/50][0/40]	Loss_D: 0.6993	Loss_G: 3.4771	D(x): 0.7185	D(G(z)): 0.1603 / 0.0561
[9/50][0/40]	Loss_D: 0.7596	Loss_G: 6.1959	D(x): 0.9629	D(G(z)): 0.4448 / 0.0055
[10/50][0/40]	Loss_D: 0.3580	Loss_G: 5.1637	D(x): 0.9353	D(G(z)): 0.2232 / 0.0120
[11/50][0/40]	Loss_D: 1.3705	Loss_G: 11.5522	D(x): 0.9365	D(G(z)): 0.6570 / 0.0001
[12/50][0/40]	Loss_D: 0.7124	Loss_G: 5.1830	D(x): 0.8014	D(G(z)): 0.2892 / 0.0105
[13/50][0/40]	Loss_D: 0.4858	Loss_G: 3.9275	D(x): 0.8526	D(G(z)): 0.2190 / 0.0453
[14/50][0/40]	Loss_D: 0.3525	Loss_G: 4.4513	D(x): 0.9061	D(G(z)): 0.1783 / 0.0283
[15/50][0/40]	Loss_D: 0.2690	Loss_G: 3.9190	D(x): 0.8814	D(G(z)): 0.0961 / 0.0455
[16/50][0/40]	Loss_D: 0.5852	Loss_G: 2.3406	D(x): 0.6577	D(G(z)): 0.0609 / 0.1337
[17/50][0/40]	Loss_D: 0.6469	Loss_G: 6.2424	D(x): 0.9415	D(G(z)): 0.3961 / 0.0046
[18/50][0/40]	Loss_D: 0.4451	Loss_G: 6.1508	D(x): 0.8966	D(G(z)): 0.2305 / 0.0040
[19/50][0/40]	Loss_D: 0.4572	Loss_G: 3.3272	D(x): 0.7621	D(G(z)): 0.0948 / 0.0514
[20/50][0/40]	Loss_D: 1.0514	Loss_G: 9.3445	D(x): 0.9757	D(G(z)): 0.5540 / 0.0004
[21/50][0/40]	Loss_D: 0.6812	Loss_G: 5.3271	D(x): 0.8127	D(G(z)): 0.2813 / 0.0099
[22/50][0/40]	Loss_D: 0.3352	Loss_G: 2.8853	D(x): 0.8692	D(G(z)): 0.1372 / 0.0825
[23/50][0/40]	Loss_D: 0.4089	Loss_G: 3.7670	D(x): 0.7787	D(G(z)): 0.0823 / 0.0495
[24/50][0/40]	Loss_D: 0.4214	Loss_G: 3.0698	D(x): 0.7342	D(G(z)): 0.0351 / 0.0701
[25/50][0/40]	Loss_D: 0.4856	Loss_G: 6.2923	D(x): 0.9341	D(G(z)): 0.2928 / 0.0037
[26/50][0/40]	Loss_D: 0.3950	Loss_G: 7.1944	D(x): 0.9250	D(G(z)): 0.2330 / 0.0019
[27/50][0/40]	Loss_D: 0.5417	Loss_G: 3.4728	D(x): 0.7846	D(G(z)): 0.1482 / 0.0539
[28/50][0/40]	Loss_D: 0.7410	Loss_G: 4.9786	D(x): 0.7753	D(G(z)): 0.2890 / 0.0155
[29/50][0/40]	Loss_D: 0.7997	Loss_G: 5.9233	D(x): 0.9240	D(G(z)): 0.4482 / 0.0051
[30/50][0/40]	Loss_D: 0.5921	Loss_G: 1.9315	D(x): 0.7896	D(G(z)): 0.1773 / 0.2596
[31/50][0/40]	Loss_D: 0.6022	Loss_G: 5.0137	D(x): 0.8257	D(G(z)): 0.2826 / 0.0108
[32/50][0/40]	Loss_D: 0.4545	Loss_G: 2.2398	D(x): 0.7767	D(G(z)): 0.1048 / 0.1799
[33/50][0/40]	Loss_D: 0.6877	Loss_G: 9.4209	D(x): 0.9420	D(G(z)): 0.3753 / 0.0002
[34/50][0/40]	Loss_D: 0.4705	Loss_G: 5.9436	D(x): 0.9026	D(G(z)): 0.2711 / 0.0047
[35/50][0/40]	Loss_D: 0.3816	Loss_G: 2.4975	D(x): 0.7659	D(G(z)): 0.0713 / 0.1196
[36/50][0/40]	Loss_D: 1.0017	Loss_G: 3.8585	D(x): 0.4491	D(G(z)): 0.0102 / 0.0483
[37/50][0/40]	Loss_D: 0.4082	Loss_G: 4.4942	D(x): 0.9145	D(G(z)): 0.2382 / 0.0225
[38/50][0/40]	Loss_D: 0.9808	Loss_G: 4.2245	D(x): 0.4932	D(G(z)): 0.0093 / 0.0481
[39/50][0/40]	Loss_D: 0.6199	Loss_G: 2.3100	D(x): 0.6587	D(G(z)): 0.0645 / 0.1525
[40/50][0/40]	Loss_D: 0.9328	Loss_G: 7.7168	D(x): 0.9721	D(G(z)): 0.5230 / 0.0013
[41/50][0/40]	Loss_D: 0.4027	Loss_G: 2.6354	D(x): 0.7870	D(G(z)): 0.1182 / 0.1092
[42/50][0/40]	Loss_D: 0.3663	Loss_G: 4.0655	D(x): 0.8537	D(G(z)): 0.1476 / 0.0288
[43/50][0/40]	Loss_D: 0.4890	Loss_G: 4.2587	D(x): 0.8314	D(G(z)): 0.2167 / 0.0235
[44/50][0/40]	Loss_D: 0.6485	Loss_G: 5.4888	D(x): 0.7841	D(G(z)): 0.2331 / 0.0092
[45/50][0/40]	Loss_D: 0.4358	Loss_G: 3.4069	D(x): 0.7760	D(G(z)): 0.1205 / 0.0488
[46/50][0/40]	Loss_D: 0.3786	Loss_G: 3.7599	D(x): 0.8541	D(G(z)): 0.1603 / 0.0393
[47/50][0/40]	Loss_D: 0.4728	Loss_G: 5.7944	D(x): 0.9723	D(G(z)): 0.3097 / 0.0074
[48/50][0/40]	Loss_D: 0.2580	Loss_G: 3.9789	D(x): 0.9011	D(G(z)): 0.1221 / 0.0323
[49/50][0/40]	Loss_D: 0.5458	Loss_G: 4.9289	D(x): 0.8728	D(G(z)): 0.2913 / 0.0116
```

Graph of Generator and Discriminator Loss During Training

In [16]:

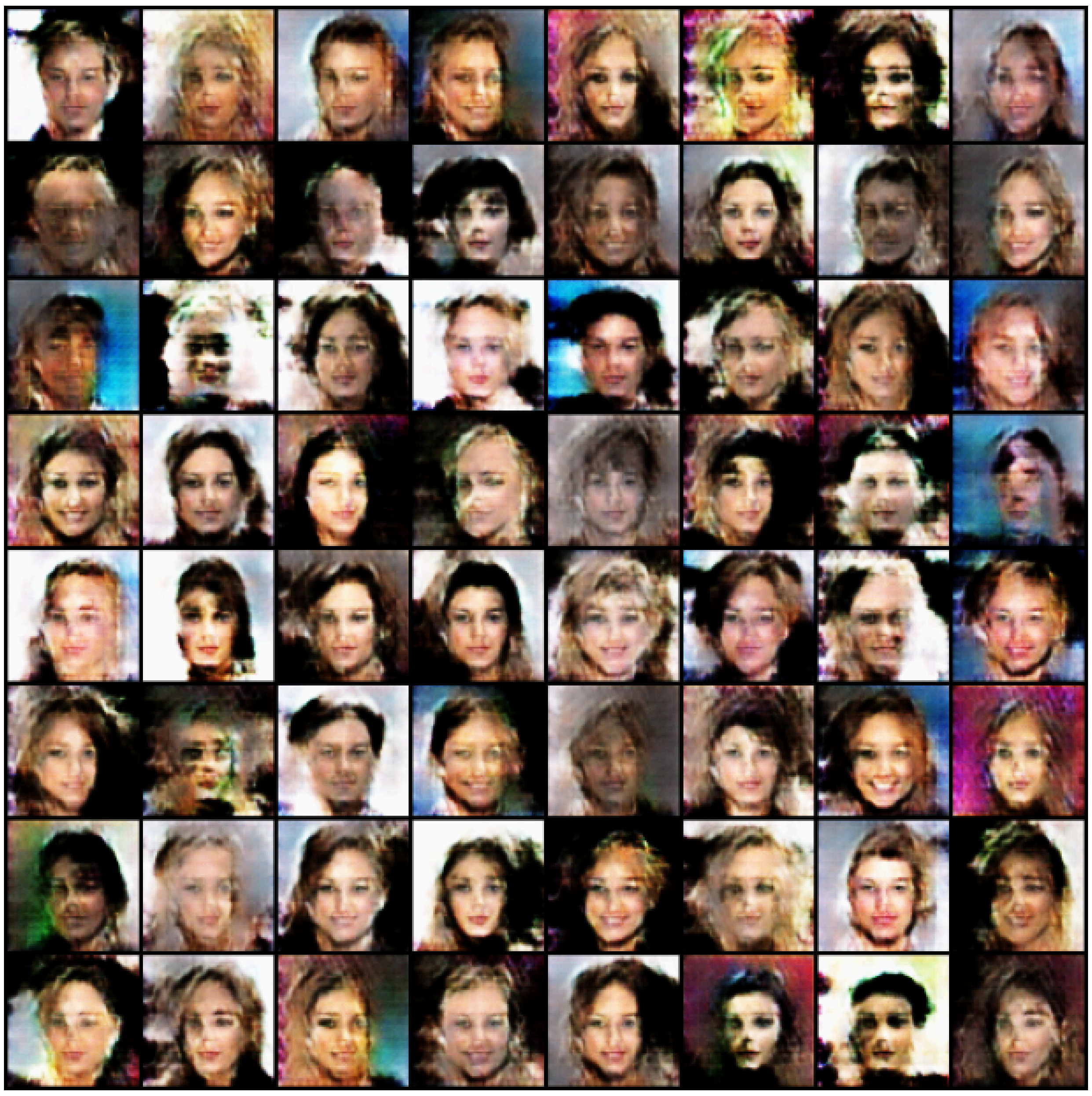
```python
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



In [17]:
```
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=True)

HTML(ani.to_jshtml())
```

Out[17]: No description has been provided for this image

Once Loop Reflect

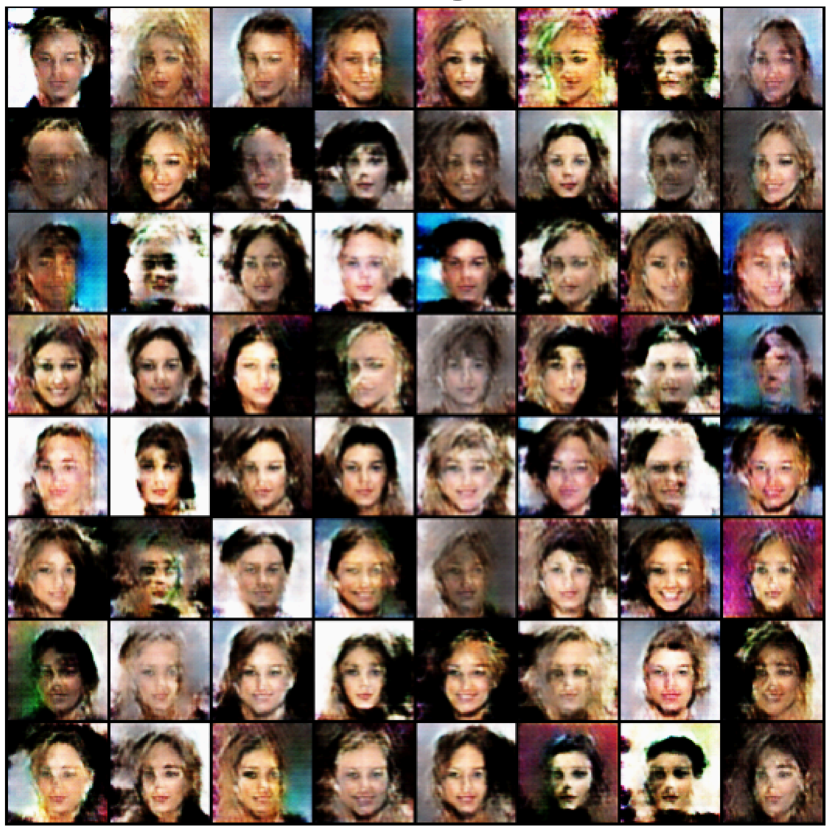


In [18]:
```
# Grab a batch of real images from the dataloader
real_batch = next(iter(dataloader))

# 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()
```

Real Images

Fake Images



We can train this model for more epoches to generate better results.

In [ ]:

```
```