DCGAN and Diffusion Implementation

I. Introduction:

For this homework we were required to implement and compare the results by implementing a generative adversarial network based model and a generative diffusion based model on a subset of CelebA Dataset provided to us. Below is a brief description of Generative Adversarial Networks and Generative Diffusion:

1. Generative Adversarial Networks(GAN):
    a. The main objective of a GAN is to be able to learn the probability distribution of the training dataset, and on the basis of the same generate the samples from that distribution with gaussian noise as input.
    b. The GAN has two components, the Generator and the Discriminator. It is the Generator's job to transform a noise-vector into an image that would look like those in the training dataset. And it is the Discriminator's job to review the output from the generator and ensure that the output is from the probability distribution of the training dataset.
    c. Through the training the discriminator is trained to become more and more efficient at telling whether an image belongs to the probability distribution of the training dataset, so that it can tell the fakes(not belonging to the probability distribution) and the the real images(belonging to the probability distribution) apart.
    d. For the loss function, we use the min max function based on the binary cross entropy function, which can be found below, which returns the parameters of the discriminator which maximizes the expectation that the the generated image belongs to the probability distribution of the train data and maximizes the expectation of it not belonging to the probability distribution of the test data and returns the parameters of the generator which minimize the expectation that the image generated by passing the noise through the generator and then passing the output through the discriminator belongs to a probability distribution that is not the probability distribution of the training data :

$$\min_{\theta_g} \max_{\theta_d} \left[E_{x \sim p_{data}} [\log D(x)] + E_{z \sim p_Z} [\log(1 - D(G(z)))] \right]$$

Fig 1

2. Generative Diffusion:
    a. Generative Diffusion on the other hand is based on two parallel markov processes, one of stepwise adding noise to an image so that post a time the image is just noise.
    b. And the second markov process, which models the noise that has been added at each step to generate the noisy output at that step.
    c. While for each step the loss function tries to find out the parameters for the network that maximize the probability that the image being generated from the model will be the actual that we had input into the markov process initially.

II. GAN/ DCGAN Implementation

- For this part the network that I used for the Discriminator and the Generator is identical to the one implemented in the DL Studio Module, with the difference that the data on which it is being trained, is the CelebA dataset.

- For the same, I created a dataset and dataloader for the celebA dataset, which was ingested into the model for training and images from the same were input into the saved generator model to generate 1000 new images based on the CelebA dataset.
- Code for the Dataset and dataloader:

```
# Dataset and Dataloader for CelebA subset dataset
class CustomImageDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.image_filenames = [os.path.join(root_dir, filename) for filename in os.listd
ir(root_dir)]

    def __len__(self):
        return len(self.image_filenames)

    def __getitem__(self, idx):
        # if torch.is_tensor(idx):
        #     idx = idx.tolist()

        img_name = self.image_filenames[idx]
        image = Image.open(img_name)

        if self.transform:
            image = self.transform(image)

        return image

# Example usage:
data_transform = tvt.Compose([
    tvt.Resize((64, 64)),
    tvt.CenterCrop((64,64)),
    tvt.ToTensor(),
    tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

dataset = CustomImageDataset(root_dir='/content/drive/MyDrive/DLStudio/DLStudio-2.4.2/Exa
mplesAdversarialLearning/celeba_dataset_64x64/celeba_dataset_64x64', transform=data_trans
form)
```

- Code for the Discriminator-Generator Network and weight initialization:

```
# This code has been picked up from the DL Studio Library
class DiscriminatorDG1(nn.Module):
            def __init__(self):
                super(DiscriminatorDG1, self).__init__()
                self.conv_in = nn.Conv2d(  3,    64,      kernel_size=4,      stride=2,
padding=1)
                self.conv_in2 = nn.Conv2d( 64,   128,     kernel_size=4,      stride=2,
padding=1)
```

```
                self.conv_in3 = nn.Conv2d( 128,  256,     kernel_size=4,      stride=2,
padding=1)
                self.conv_in4 = nn.Conv2d( 256,  512,     kernel_size=4,      stride=2,
padding=1)
                self.conv_in5 = nn.Conv2d( 512,  1,       kernel_size=4,      stride=1,
padding=0)
                self.bn1  = nn.BatchNorm2d(128)
                self.bn2  = nn.BatchNorm2d(256)
                self.bn3  = nn.BatchNorm2d(512)
                self.sig = nn.Sigmoid()
            def forward(self, x):
                x = torch.nn.functional.leaky_relu(self.conv_in(x), negative_slope=0.2, i
nplace=True)
                x = self.bn1(self.conv_in2(x))
                x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
                x = self.bn2(self.conv_in3(x))
                x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
                x = self.bn3(self.conv_in4(x))
                x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
                x = self.conv_in5(x)
                x = self.sig(x)
                return x

class GeneratorDG1(nn.Module):
            def __init__(self):
                super(GeneratorDG1, self).__init__()
                self.latent_to_image = nn.ConvTranspose2d( 100,   512,  kernel_size=4, st
ride=1, padding=0, bias=False)
                self.upsampler2 = nn.ConvTranspose2d( 512, 256, kernel_size=4, stride=2,
padding=1, bias=False)
                self.upsampler3 = nn.ConvTranspose2d (256, 128, kernel_size=4, stride=2,
padding=1, bias=False)
                self.upsampler4 = nn.ConvTranspose2d (128, 64,  kernel_size=4, stride=2,
padding=1, bias=False)
                self.upsampler5 = nn.ConvTranspose2d(  64,  3,  kernel_size=4, stride=2,
padding=1, bias=False)
                self.bn1 = nn.BatchNorm2d(512)
                self.bn2 = nn.BatchNorm2d(256)
                self.bn3 = nn.BatchNorm2d(128)
                self.bn4 = nn.BatchNorm2d(64)
                self.tanh  = nn.Tanh()
            def forward(self, x):
                x = self.latent_to_image(x)
                x = torch.nn.functional.relu(self.bn1(x))
                x = self.upsampler2(x)
                x = torch.nn.functional.relu(self.bn2(x))
                x = self.upsampler3(x)
                x = torch.nn.functional.relu(self.bn3(x))
                x = self.upsampler4(x)
                x = torch.nn.functional.relu(self.bn4(x))
                x = self.upsampler5(x)
```

```
                x = self.tanh(x)
                return x
```

- Code for running the GAN network:

```
# This code has been picked up from the DL Studio Library with some changes from my end
def run_gan_code(discriminator, generator, results_dir, dataloader):
            dir_name_for_results = results_dir
            if os.path.exists(dir_name_for_results):
                files = glob.glob(dir_name_for_results + "/*")
                for file in files:
                    if os.path.isfile(file):
                        os.remove(file)
                    else:
                        files = glob.glob(file + "/*")
                        list(map(lambda x: os.remove(x), files))
            else:
                os.mkdir(dir_name_for_results)
            #  Set the number of channels for the 1x1 input noise vectors for the Generat
or:
            nz = 100
            netD = discriminator.to("cuda:0" if torch.cuda.is_available() else "cpu")
            netG = generator.to("cuda:0" if torch.cuda.is_available() else "cpu")
            #  Initialize the parameters of the Discriminator and the Generator networks
according to the
            #  definition of the "weights_init()" method:
            netD.apply(weights_init)
            netG.apply(weights_init)
            #  We will use a the same noise batch to periodically check on the progress m
ade for the Generator:
            fixed_noise = torch.randn(32, nz, 1, 1, device="cuda:0" if torch.cuda.is_avai
lable() else "cpu")
            #  Establish convention for real and fake labels during training
            real_label = 1
            fake_label = 0
            #  Adam optimizers for the Discriminator and the Generator:
            optimizerD = optim.Adam(netD.parameters(), lr=1e-4, betas=(0.9, 0.999))
            optimizerG = optim.Adam(netG.parameters(), lr=1e-4, betas=(0.9, 0.999))
            #  Establish the criterion for measuring the loss at the output of the Discri
minator network:
            criterion = nn.BCELoss()
            #  We will use these lists to store the results accumulated during training:
            img_list = []
            G_losses = []
            D_losses = []
            iters = 0
            print("\n\nStarting Training Loop...\n\n")
            start_time = time.perf_counter()
            for epoch in range(30):
                g_losses_per_print_cycle = []
                d_losses_per_print_cycle = []
```

```
                # For each batch in the dataloader
                for i, data in enumerate(dataloader, 0):

                    ##  Maximization Part of the Min-Max Objective of Eq. (3):
                    ##
                    ##  As indicated by Eq. (3) in the DCGAN part of the doc section at t
he beginning of this
                    ##  file, the GAN training boils down to carrying out a min-max optim
ization. Each iterative
                    ##  step of the max part results in updating the Discriminator parame
ters and each iterative
                    ##  step of the min part results in the updating of the Generator par
ameters.  For each
                    ##  batch of the training data, we first do max and then do min.  Sin
ce the max operation
                    ##  affects both terms of the criterion shown in the doc section, it
has two parts: In the
                    ##  first part we apply the Discriminator to the training images usin
g 1.0 as the target;
                    ##  and, in the second part, we supply to the Discriminator the outpu
t of the Generator
                    ##  and use 0 as the target. In what follows, the Discriminator is be
ing applied to
                    ##  the training images:
                    netD.zero_grad()
                    real_images_in_batch = data[0].to("cuda:0" if torch.cuda.is_available
() else "cpu")
                    #  Need to know how many images we pulled in since at the tailend of
the dataset, the
                    #  number of images may not equal the user-specified batch size:
                    b_size = real_images_in_batch.size(0)
                    label = torch.full((b_size,), real_label, dtype=torch.float, device
="cuda:0" if torch.cuda.is_available() else "cpu")
                    output = netD(real_images_in_batch).view(-1)
                    lossD_for_reals = criterion(output, label)
                    lossD_for_reals.backward()
                    ##  That brings us the second part of what it takes to carry out the
max operation on the
                    ##  min-max criterion shown in Eq. (3) in the doc section at the begi
nning of this file.
                    ##  part calls for applying the Discriminator to the images produced
by the Generator from
                    ##  noise:
                    noise = torch.randn(b_size, nz, 1, 1, device="cuda:0" if torch.cuda.i
s_available() else "cpu")
                    fakes = netG(noise)
                    label.fill_(fake_label)
                    ##  The call to fakes.detach() in the next statement returns a copy o
f the 'fakes' tensor
                    ##  that does not exist in the computational graph. That is, the call
shown below first
```

```
                    ##  makes a copy of the 'fakes' tensor and then removes it from the c
omputational graph.
                    ##  The original 'fakes' tensor continues to remain in the computatio
nal graph.  This ploy
                    ##  ensures that a subsequent call to backward() in the 3rd statement
below would only
                    ##  update the netD weights.
                    output = netD(fakes.detach()).view(-1)
                    lossD_for_fakes = criterion(output, label)
                    ##  At this point, we do not care if the following call also calculat
es the gradients
                    ##  wrt the Discriminator weights since we are going to next iteratio
n with 'netD.zero_grad()':
                    lossD_for_fakes.backward()
                    lossD = lossD_for_reals + lossD_for_fakes
                    d_losses_per_print_cycle.append(lossD)
                    ##  Only the Discriminator weights are incremented:
                    optimizerD.step()

                    ##  Minimization Part of the Min-Max Objective of Eq. (3):
                    ##
                    ##  That brings to the min part of the max-min optimization described
in Eq. (3) the doc
                    ##  section at the beginning of this file.  The min part requires tha
t we minimize
                    ##  "1 - D(G(z))" which, since D is constrained to lie in the interva
l (0,1), requires that
                    ##  we maximize D(G(z)).  We accomplish that by applying the Discrimi
nator to the output
                    ##  of the Generator and use 1 as the target for each image:
                    netG.zero_grad()
                    label.fill_(real_label)
                    output = netD(fakes).view(-1)
                    lossG = criterion(output, label)
                    g_losses_per_print_cycle.append(lossG)
                    lossG.backward()
                    ##  Only the Generator parameters are incremented:
                    optimizerG.step()
                    if i % 100 == 99:
                        current_time = time.perf_counter()
                        elapsed_time = current_time - start_time
                        mean_D_loss = torch.mean(torch.FloatTensor(d_losses_per_print_cyc
le))
                        mean_G_loss = torch.mean(torch.FloatTensor(g_losses_per_print_cyc
le))
                        print("[epoch=%d/%d   iter=%4d   elapsed_time=%5d secs]     mean_
D_loss=%7.4f    mean_G_loss=%7.4f" %
                                      ((epoch+1),30,(i+1),elapsed_time,mean_D_loss,mean_G
_loss))
                        d_losses_per_print_cycle = []
                        g_losses_per_print_cycle = []
```

```
                    G_losses.append(lossG.item())
                    D_losses.append(lossD.item())
                    if (iters % 500 == 0) or ((epoch == 30-1) and (i == len(dataloader)-
1)):
                        with torch.no_grad():
                            fake = netG(fixed_noise).detach().cpu()  ## detach() removes
the fake from comp. graph.
                                                                    ## for creating its
CPU compatible version
                        img_list.append(torchvision.utils.make_grid(fake, padding=1, pad_
value=1, normalize=True))
                    iters += 1
            #  At the end of training, make plots from the data in G_losses and D_losses:
            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.savefig(dir_name_for_results + "/gen_and_disc_loss_training.png")
            plt.show()
            #  Make an animated gif from the Generator output images stored in img_list:
            images = []
            for imgobj in img_list:
                img = tvtF.to_pil_image(imgobj)
                images.append(img)
            imageio.mimsave(dir_name_for_results + "/generation_animation.gif", images, f
ps=5)
            #  Make a side-by-side comparison of a batch-size sampling of real images dra
wn from the
            #  training data and what the Generator is capable of producing at the end of
training:
            real_batch = next(iter(dataloader))
            real_batch = real_batch[0]
            plt.figure(figsize=(15,15))
            plt.subplot(1,2,1)
            plt.axis("off")
            plt.title("Real Images")
            plt.imshow(np.transpose(torchvision.utils.make_grid(real_batch.to("cuda:0" if
torch.cuda.is_available() else "cpu"),
                                                    padding=1, pad_value=1, normalize=Tru
e).cpu(),(1,2,0)))
            plt.subplot(1,2,2)
            plt.axis("off")
            plt.title("Fake Images")
            plt.imshow(np.transpose(img_list[-1],(1,2,0)))
            plt.savefig(dir_name_for_results + "/real_vs_fake_images.png")
            plt.show()
```

```
            torch.save(netG.state_dict(), "/content/drive/MyDrive/DLStudio-2.4.2/Examples
AdversarialLearning/results_DG1/save_model_loss_1")
```

- Code for generation of new images:

```
generator.load_state_dict(torch.load("/content/drive/MyDrive/DLStudio-2.4.2/ExamplesAdver
sarialLearning/results_DG1/save_model_loss_1"))
input_noise = torch.randn(1000, 100, 1, 1, device=torch.device("cpu"))
fake_imgs = []
j = 1
with torch.no_grad():
  newfake = generator(input_noise).detach().cpu()
fake_imgs.append(newfake)

for i in range(len(fake_imgs[0])):
  image = fake_imgs[0][i].permute(1, 2, 0).cpu().detach().numpy()
  plt.imshow(image)
  plt.axis('off')  # Hide axis
  plt.savefig(f'/content/drive/MyDrive/DLStudio-2.4.2/ExamplesAdversarialLearning/results
_DG1/gen_fake_images1/image_{i}.png')
```

- FID(Fréchet Inception Distance) calculation:

```
def list_files_in_folder(folder_path):
    files = []
    for file_name in os.listdir(folder_path):
        if os.path.isfile(os.path.join(folder_path, file_name)):
            files.append(os.path.join(folder_path, file_name))
    return files

real_folder_path = '/content/drive/MyDrive/DLStudio-2.4.2/ExamplesAdversarialLearning/cel
eba_dataset_64x64/0'
fake_folder_path_1 = '/content/drive/MyDrive/DLStudio-2.4.2/ExamplesAdversarialLearning/r
esults_DG1/gen_fake_images1'

real_image_paths = list_files_in_folder(real_folder_path)
fake_image_paths_1 = list_files_in_folder(fake_folder_path_1)

#FID for GAN
from pytorch_fid.fid_score import calculate_activation_statistics, calculate_frechet_dist
ance
from pytorch_fid.inception import InceptionV3
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
dims = 2048
block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
model = InceptionV3([block_idx]).to(device)
m1 , s1 = calculate_activation_statistics(real_image_paths , model , device = device )
m2 , s2 = calculate_activation_statistics(fake_image_paths_1 , model , device = device )
```

```
fid_value = calculate_frechet_distance(m1 , s1 , m2 , s2)
print(f'FID: { fid_value: .2f}')
```

```
100%|██████████| 200/200 [00:43<00:00,  4.57it/s]
100%|██████████| 41/41 [00:31<00:00,  1.32it/s]
FID:  216.30
```

Output 1

- Plot of the adversarial losses over training iterations for both the generator and the discriminator:

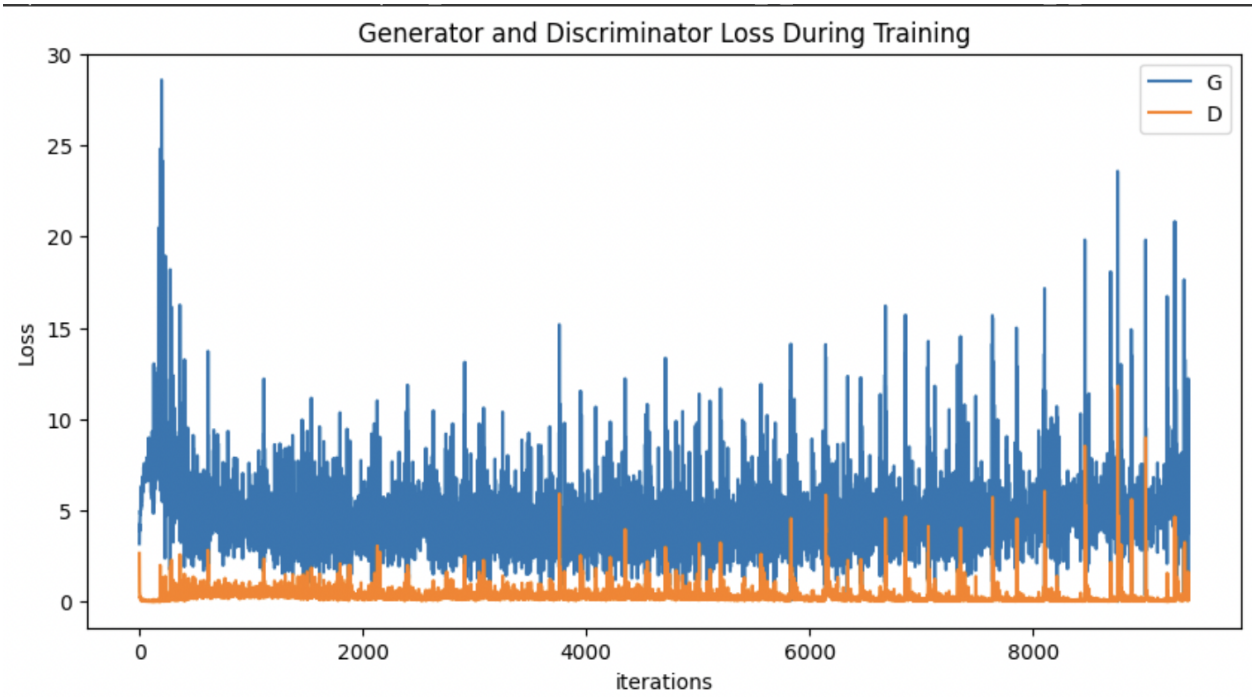

Fig 2

- Sample Output:

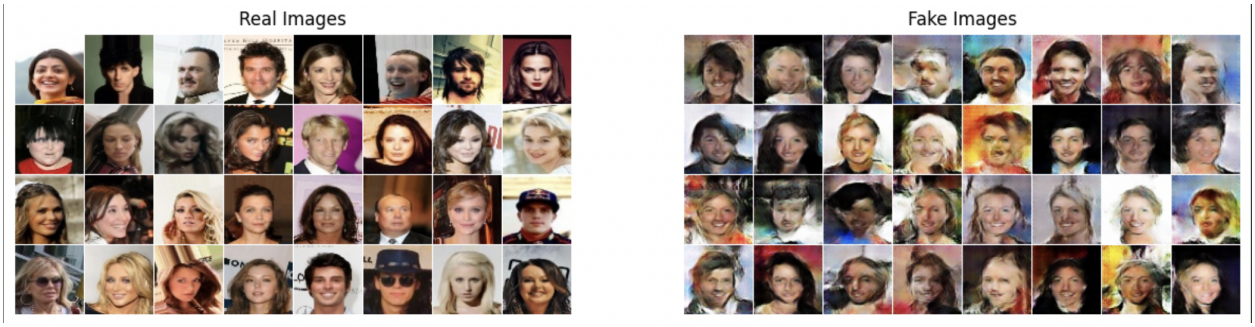

Fig 3

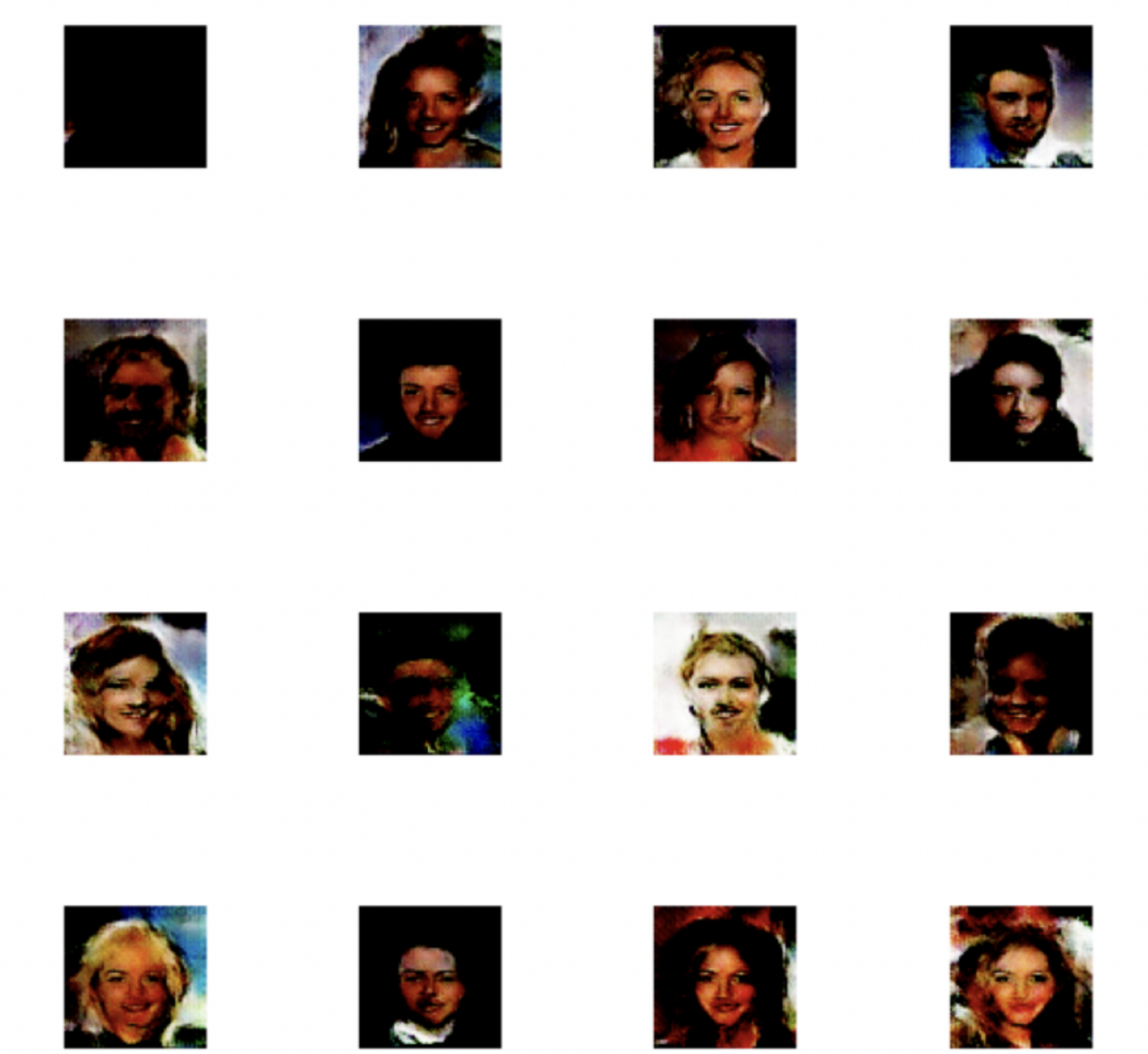

Fig 4

III. Generative Diffusion:

- For this part, as the diffusion model for running parallel markov processes and optimizing the network being used for denoising would be too computationally expensive, we will be using the save network weight provided to us for the homework.
- The saved network weights will be used in the mentioned script to generate the fake images based from the diffusion mode, the script can be found below:

```
# This code has been picked up from the DL Studio Library
# Diffusion Image Generation using the provided model weights
import os
import sys
import numpy as np
```

```
import torch

sys.path.append("/content/drive/MyDrive/DLStudio-2.4.2/DLStudio")
sys.path.append("/content/drive/MyDrive/DLStudio-2.4.2/GenerativeDiffusion")

from GenerativeDiffusion import *

gauss_diffusion   =  GaussianDiffusion(
                        num_diffusion_timesteps = 1000,

                    )


network =  UNetModel(
                       in_channels=3,
                       model_channels   =  128,
                       out_channels     =  3,
                       num_res_blocks   =  2,
                       attention_resolutions =  (4, 8),              ## for 64x64 images
                       channel_mult     =    (1, 2, 3, 4),           ## for 64x64 images
                       num_heads        =    1,
                       attention        =    True            ## <<< Must be the same as f
or RunCodeForDiffusion.py
#                       attention        =    False          ## <<< Must be the same as f
or RunCodeForDiffusion.py

                     )


top_level = GenerativeDiffusion(
                        gen_new_images        =        True,
                        image_size            =        64,
                        num_channels          =        128,
                        ema_rate              =        0.9999,
                        diffusion = gauss_diffusion,
                        network = network,
                        ngpu = 1,
                        path_saved_model = "/content/drive/MyDrive/DLStudio-2.4.2/Example
sDiffusion/saved_model/", ##using the provided diffusion model weights
                        clip_denoised=True,
                        num_samples=1000,
                        batch_size_image_generation=4,
             )

if sys.argv[1] == '--model_path':
    model_path = sys.argv[2]


network.load_state_dict( torch.load("/content/drive/MyDrive/DLStudio-2.4.2/ExamplesDiffus
ion/saved_model/diffusion.pt") )
```

```
network.to(top_level.device)
network.eval()

print("sampling...")
all_images = []

while len(all_images) * top_level.batch_size_image_generation < top_level.num_samples:
    sample = gauss_diffusion.p_sampler_for_image_generation(
        network,
        (top_level.batch_size_image_generation, 3, top_level.image_size, top_level.image_
size),
        device = top_level.device,
        clip_denoised = top_level.clip_denoised,
    )
    sample = ((sample + 1) * 127.5).clamp(0, 255).to(torch.uint8)
    sample = sample.permute(0, 2, 3, 1)
    sample = sample.contiguous()
    gathered_samples = [sample]
    all_images.extend([sample.cpu().numpy() for sample in gathered_samples])
    print(f"created {len(all_images) * top_level.batch_size_image_generation} samples")

arr = np.concatenate(all_images, axis=0)
arr = arr[: top_level.num_samples]

shape_str = "x".join([str(x) for x in arr.shape])
out_path = os.path.join("/content/drive/MyDrive/DLStudio-2.4.2/ExamplesDiffusion", f"samp
les_{shape_str}.npz")

np.savez(out_path, arr)

print("image generation completed")
```

- Code for generation of new images:

```
# This code has been picked up from the DL Studio Library

data = np.load("/content/drive/MyDrive/DLStudio-2.4.2/ExamplesDiffusion/samples_2048x64x6
4x3.npz")

print("\n\n[visualize_sample.py]  the data object: ", data)
## NpzFile 'RESULTS/samples_8x64x64x3.npz' with keys: arr_0
print("\n\n[visualize_sample.py]  type of the data object: ", type(data))
## <class 'numpy.lib.npyio.NpzFile'>
print("\n\n[visualize_sample.py]  shape of the object data['arr_0']: ", data['arr_0'].sha
pe)      ## (8, 64, 64, 3)

for i, img in enumerate(data['arr_0']):
    plt.figure()
    plt.imshow(img)
    plt.axis("off")
```

```
    plt.savefig(f"/content/drive/MyDrive/DLStudio-2.4.2/ExamplesDiffusion/visualize_sampl
es/test_{i}.jpeg")
```

- FID(Fréchet Inception Distance) calculation:

```
def list_files_in_folder(folder_path):
    files = []
    for file_name in os.listdir(folder_path):
        if os.path.isfile(os.path.join(folder_path, file_name)):
            files.append(os.path.join(folder_path, file_name))
    return files

real_folder_path = '/content/drive/MyDrive/DLStudio-2.4.2/ExamplesAdversarialLearning/cel
eba_dataset_64x64/0'
fake_folder_path_1 = '/content/drive/MyDrive/DLStudio-2.4.2/ExamplesAdversarialLearning/r
esults_DG1/gen_fake_images1'

real_image_paths = list_files_in_folder(real_folder_path)
fake_image_paths_diffusion = list_files_in_folder(fake_folder_diffusion)

from pytorch_fid.fid_score import calculate_activation_statistics, calculate_frechet_dist
ance
from pytorch_fid.inception import InceptionV3
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
dims = 2048
block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
model = InceptionV3([block_idx]).to(device)
m1 , s1 = calculate_activation_statistics(real_image_paths , model , device = device )
m2 , s2 = calculate_activation_statistics(fake_image_paths_diffusion , model , device = d
evice )
fid_value = calculate_frechet_distance(m1 , s1 , m2 , s2)
print(f'FID: { fid_value: .2f}')
```

```
100%|██████████| 200/200 [1:07:14<00:00, 20.17s/it]
100%|██████████| 41/41 [00:23<00:00,  1.77it/s]
FID:  204.86
```

Output 2

- Sample Output:

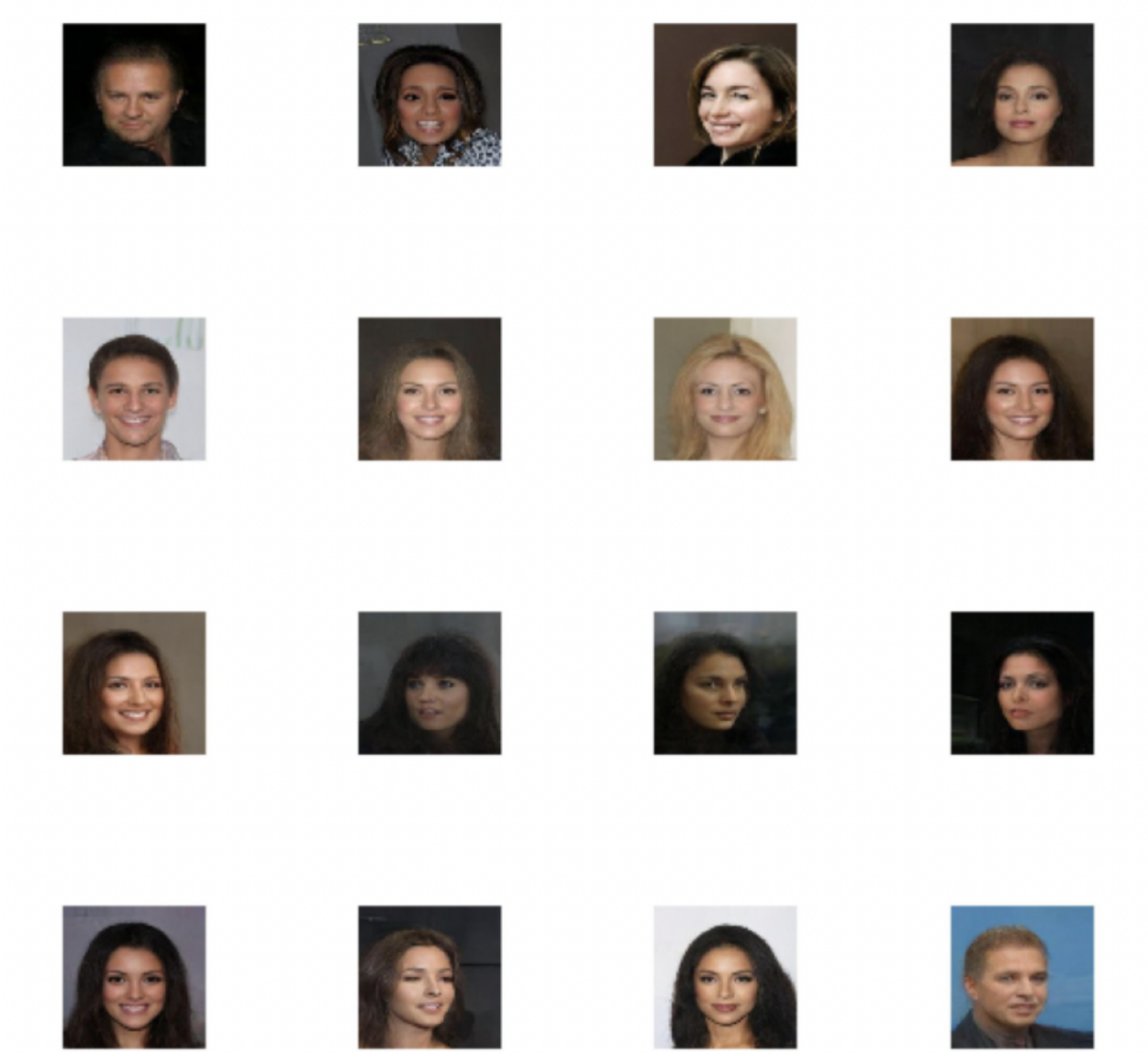

Fig 5

IV. Quantitative and Qualitative Results:

- Qualitative Results:

From the 4 X 4 matrix of images generated for GAN(Fig 4) and Generative Diffusion(Fig 5), we can see that the images generated by both the models are an amalgamation of pixels belonging to the probability distribution of the images in the training dataset that follow a similar kind of local distribution when creating the new sample. From the two figures mentioned above, it can be seen clearly that the outputs generated by the Generative Diffusion model are much better in terms of clarity and differentiation than the outputs generated by the GAN.

- Quantitative Results based on FID(Fréchet Inception Distance) Value:

In the feature space, FID calculates the Fréchet distance between the two distributions. It is a metric for comparing two probability distributions that considers their spreads (covariances) and locations (means) is called the Fréchet distance. The distributions of generated and real images are more comparable in the feature space when the FID score is lower. This means that the generated images more closely mimic the visual traits seen in the original

photographs. Here, from the images called Output 1(GAN) and Output2(Diffusion), we can see that the FID value for the images generated by Diffusion is lower, meaning that the images generated by the Diffusion model are more similar to the probability distribution of the training dataset. This means that the images generated by diffusion capture the visual characteristics present in the real images more accurately.

V. Conclusion or Which Model is Better?:

- From the results, we can see that the images generated by Diffusion are much better than the ones generated by the BCE-GAN. But Generative Diffusion is much more resource and time intensive as compared to the BCE-GAN.
- The reason for this seems to be the iterative process via which the model is optimized for each time step, which allows for the model to learn the features of the training data on a more granular level.
- And as we are explicitly trying to learn the noise and remove it from the data in Generative Diffusion, we are less likely to face the issue of mode collapse that GANs face in case of highly variable data.
- Also, as we are using the likelihood based training function in Generative Diffusion, it allows us to capture the complexity of the data in a much more extensive manner than a binary cross entropy function would in case of BCE-GANs.
- Hence, overall Generative Diffusion would perform better than BCE-GAN. But depending upon the constraints of our use case such as the type of data, computation time, resources etc, it might be suitable to use GANs in place of Generative Diffusion.

Source Code:

- GAN:

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


get_ipython().run_line_magic('cd', '/content/drive/MyDrive/DLStudio/DLStudio-2.4.2/Exampl
esAdversarialLearning')


# In[ ]:


get_ipython().system('unzip celeba_dataset_64x64.zip -d /content/drive/MyDrive/DLStudio/D
LStudio-2.4.2/ExamplesAdversarialLearning/celeba_dataset_64x64/')


# In[ ]:


import os, sys
import torch
import torchvision.transforms as tvt
```

```
from PIL import Image
import numpy as np
import glob
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as tvt
from torchvision import utils
import seaborn as sns
get_ipython().run_line_magic('matplotlib', 'inline')
import matplotlib.pyplot as plt
import random
import time
import skimage.io as io
from pycocotools.coco import COCO
import copy
from scipy.ndimage import zoom
import torch.optim as optim
import re
import math
import random
import copy
import matplotlib.pyplot as plt
import gzip
import pickle
import time
import logging

seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)


sys.path.append( "/content/drive/MyDrive/DLStudio/DLStudio-2.4.2/DLStudio/" )
sys.path.append( "/content/drive/MyDrive/DLStudio/DLStudio-2.4.2/AdversarialLearning" )
##  watch -d -n 0.5 nvidia-smi


# In[ ]:


# Dataset and Dataloader for CelebA subset dataset
class CustomImageDataset(Dataset):
```

```
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.image_filenames = [os.path.join(root_dir, filename) for filename in os.listd
ir(root_dir)]

    def __len__(self):
        return len(self.image_filenames)

    def __getitem__(self, idx):
        # if torch.is_tensor(idx):
        #     idx = idx.tolist()

        img_name = self.image_filenames[idx]
        image = Image.open(img_name)

        if self.transform:
            image = self.transform(image)

        return image

# Example usage:
data_transform = tvt.Compose([
    tvt.Resize((64, 64)),
    tvt.CenterCrop((64,64)),
    tvt.ToTensor(),
    tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

dataset = CustomImageDataset(root_dir='/content/drive/MyDrive/DLStudio/DLStudio-2.4.2/Exa
mplesAdversarialLearning/celeba_dataset_64x64/celeba_dataset_64x64', transform=data_trans
form)


# In[ ]:


dataloader = DataLoader(dataset, batch_size=32, shuffle=True)


# In[ ]:


# This code has been picked up from the DL Studio Library
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)
```

```
# In[ ]:


# This code has been picked up from the DL Studio Library
class DiscriminatorDG1(nn.Module):
            def __init__(self):
                super(DiscriminatorDG1, self).__init__()
                self.conv_in = nn.Conv2d(  3,    64,      kernel_size=4,      stride=2,
padding=1)
                self.conv_in2 = nn.Conv2d( 64,   128,     kernel_size=4,      stride=2,
padding=1)
                self.conv_in3 = nn.Conv2d( 128,  256,     kernel_size=4,      stride=2,
padding=1)
                self.conv_in4 = nn.Conv2d( 256,  512,     kernel_size=4,      stride=2,
padding=1)
                self.conv_in5 = nn.Conv2d( 512,  1,       kernel_size=4,      stride=1,
padding=0)
                self.bn1  = nn.BatchNorm2d(128)
                self.bn2  = nn.BatchNorm2d(256)
                self.bn3  = nn.BatchNorm2d(512)
                self.sig = nn.Sigmoid()
            def forward(self, x):
                x = torch.nn.functional.leaky_relu(self.conv_in(x), negative_slope=0.2, i
nplace=True)
                x = self.bn1(self.conv_in2(x))
                x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
                x = self.bn2(self.conv_in3(x))
                x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
                x = self.bn3(self.conv_in4(x))
                x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
                x = self.conv_in5(x)
                x = self.sig(x)
                return x

class GeneratorDG1(nn.Module):
            def __init__(self):
                super(GeneratorDG1, self).__init__()
                self.latent_to_image = nn.ConvTranspose2d( 100,   512,  kernel_size=4, st
ride=1, padding=0, bias=False)
                self.upsampler2 = nn.ConvTranspose2d( 512, 256, kernel_size=4, stride=2,
padding=1, bias=False)
                self.upsampler3 = nn.ConvTranspose2d (256, 128, kernel_size=4, stride=2,
padding=1, bias=False)
                self.upsampler4 = nn.ConvTranspose2d (128, 64,  kernel_size=4, stride=2,
padding=1, bias=False)
                self.upsampler5 = nn.ConvTranspose2d(  64,  3,  kernel_size=4, stride=2,
padding=1, bias=False)
                self.bn1 = nn.BatchNorm2d(512)
                self.bn2 = nn.BatchNorm2d(256)
```

```
                self.bn3 = nn.BatchNorm2d(128)
                self.bn4 = nn.BatchNorm2d(64)
                self.tanh  = nn.Tanh()
            def forward(self, x):
                x = self.latent_to_image(x)
                x = torch.nn.functional.relu(self.bn1(x))
                x = self.upsampler2(x)
                x = torch.nn.functional.relu(self.bn2(x))
                x = self.upsampler3(x)
                x = torch.nn.functional.relu(self.bn3(x))
                x = self.upsampler4(x)
                x = torch.nn.functional.relu(self.bn4(x))
                x = self.upsampler5(x)
                x = self.tanh(x)
                return x


# In[ ]:


# This code has been picked up from the DL Studio Library with some changes from my end
def run_gan_code(discriminator, generator, results_dir, dataloader):
            dir_name_for_results = results_dir
            if os.path.exists(dir_name_for_results):
                files = glob.glob(dir_name_for_results + "/*")
                for file in files:
                    if os.path.isfile(file):
                        os.remove(file)
                    else:
                        files = glob.glob(file + "/*")
                        list(map(lambda x: os.remove(x), files))
            else:
                os.mkdir(dir_name_for_results)
            #  Set the number of channels for the 1x1 input noise vectors for the Generat
or:
            nz = 100
            netD = discriminator.to("cuda:0" if torch.cuda.is_available() else "cpu")
            netG = generator.to("cuda:0" if torch.cuda.is_available() else "cpu")
            #  Initialize the parameters of the Discriminator and the Generator networks
according to the
            #  definition of the "weights_init()" method:
            netD.apply(weights_init)
            netG.apply(weights_init)
            #  We will use a the same noise batch to periodically check on the progress m
ade for the Generator:
            fixed_noise = torch.randn(32, nz, 1, 1, device="cuda:0" if torch.cuda.is_avai
lable() else "cpu")
            #  Establish convention for real and fake labels during training
            real_label = 1
            fake_label = 0
            #  Adam optimizers for the Discriminator and the Generator:
```

```
            optimizerD = optim.Adam(netD.parameters(), lr=1e-4, betas=(0.9, 0.999))
            optimizerG = optim.Adam(netG.parameters(), lr=1e-4, betas=(0.9, 0.999))
            #  Establish the criterion for measuring the loss at the output of the Discri
minator network:
            criterion = nn.BCELoss()
            #  We will use these lists to store the results accumulated during training:
            img_list = []
            G_losses = []
            D_losses = []
            iters = 0
            print("\n\nStarting Training Loop...\n\n")
            start_time = time.perf_counter()
            for epoch in range(30):
                g_losses_per_print_cycle = []
                d_losses_per_print_cycle = []
                # For each batch in the dataloader
                for i, data in enumerate(dataloader, 0):

                    ##  Maximization Part of the Min-Max Objective of Eq. (3):
                    ##
                    ##  As indicated by Eq. (3) in the DCGAN part of the doc section at t
he beginning of this
                    ##  file, the GAN training boils down to carrying out a min-max optim
ization. Each iterative
                    ##  step of the max part results in updating the Discriminator parame
ters and each iterative
                    ##  step of the min part results in the updating of the Generator par
ameters.  For each
                    ##  batch of the training data, we first do max and then do min.  Sin
ce the max operation
                    ##  affects both terms of the criterion shown in the doc section, it
has two parts: In the
                    ##  first part we apply the Discriminator to the training images usin
g 1.0 as the target;
                    ##  and, in the second part, we supply to the Discriminator the outpu
t of the Generator
                    ##  and use 0 as the target. In what follows, the Discriminator is be
ing applied to
                    ##  the training images:
                    netD.zero_grad()
                    real_images_in_batch = data[0].to("cuda:0" if torch.cuda.is_available
() else "cpu")
                    #  Need to know how many images we pulled in since at the tailend of
the dataset, the
                    #  number of images may not equal the user-specified batch size:
                    b_size = real_images_in_batch.size(0)
                    label = torch.full((b_size,), real_label, dtype=torch.float, device
="cuda:0" if torch.cuda.is_available() else "cpu")
                    output = netD(real_images_in_batch).view(-1)
                    lossD_for_reals = criterion(output, label)
                    lossD_for_reals.backward()
```

```
                    ##  That brings us the second part of what it takes to carry out the
max operation on the
                    ##  min-max criterion shown in Eq. (3) in the doc section at the begi
nning of this file.
                    ##  part calls for applying the Discriminator to the images produced
by the Generator from
                    ##  noise:
                    noise = torch.randn(b_size, nz, 1, 1, device="cuda:0" if torch.cuda.i
s_available() else "cpu")
                    fakes = netG(noise)
                    label.fill_(fake_label)
                    ##  The call to fakes.detach() in the next statement returns a copy o
f the 'fakes' tensor
                    ##  that does not exist in the computational graph. That is, the call
shown below first
                    ##  makes a copy of the 'fakes' tensor and then removes it from the c
omputational graph.
                    ##  The original 'fakes' tensor continues to remain in the computatio
nal graph.  This ploy
                    ##  ensures that a subsequent call to backward() in the 3rd statement
below would only
                    ##  update the netD weights.
                    output = netD(fakes.detach()).view(-1)
                    lossD_for_fakes = criterion(output, label)
                    ##  At this point, we do not care if the following call also calculat
es the gradients
                    ##  wrt the Discriminator weights since we are going to next iteratio
n with 'netD.zero_grad()':
                    lossD_for_fakes.backward()
                    lossD = lossD_for_reals + lossD_for_fakes
                    d_losses_per_print_cycle.append(lossD)
                    ##  Only the Discriminator weights are incremented:
                    optimizerD.step()

                    ##  Minimization Part of the Min-Max Objective of Eq. (3):
                    ##
                    ##  That brings to the min part of the max-min optimization described
in Eq. (3) the doc
                    ##  section at the beginning of this file.  The min part requires tha
t we minimize
                    ##  "1 - D(G(z))" which, since D is constrained to lie in the interva
l (0,1), requires that
                    ##  we maximize D(G(z)).  We accomplish that by applying the Discrimi
nator to the output
                    ##  of the Generator and use 1 as the target for each image:
                    netG.zero_grad()
                    label.fill_(real_label)
                    output = netD(fakes).view(-1)
                    lossG = criterion(output, label)
                    g_losses_per_print_cycle.append(lossG)
                    lossG.backward()
```

```
                    ##  Only the Generator parameters are incremented:
                    optimizerG.step()
                    if i % 100 == 99:
                        current_time = time.perf_counter()
                        elapsed_time = current_time - start_time
                        mean_D_loss = torch.mean(torch.FloatTensor(d_losses_per_print_cyc
le))
                        mean_G_loss = torch.mean(torch.FloatTensor(g_losses_per_print_cyc
le))
                        print("[epoch=%d/%d   iter=%4d   elapsed_time=%5d secs]     mean_
D_loss=%7.4f    mean_G_loss=%7.4f" %
                                      ((epoch+1),30,(i+1),elapsed_time,mean_D_loss,mean_G
_loss))
                        d_losses_per_print_cycle = []
                        g_losses_per_print_cycle = []
                    G_losses.append(lossG.item())
                    D_losses.append(lossD.item())
                    if (iters % 500 == 0) or ((epoch == 30-1) and (i == len(dataloader)-
1)):
                        with torch.no_grad():
                            fake = netG(fixed_noise).detach().cpu()  ## detach() removes
the fake from comp. graph.
                                                                    ## for creating its
CPU compatible version
                        img_list.append(torchvision.utils.make_grid(fake, padding=1, pad_
value=1, normalize=True))
                    iters += 1
            #  At the end of training, make plots from the data in G_losses and D_losses:
            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.savefig(dir_name_for_results + "/gen_and_disc_loss_training.png")
            plt.show()
            #  Make an animated gif from the Generator output images stored in img_list:
            images = []
            for imgobj in img_list:
                img = tvtF.to_pil_image(imgobj)
                images.append(img)
            imageio.mimsave(dir_name_for_results + "/generation_animation.gif", images, f
ps=5)
            #  Make a side-by-side comparison of a batch-size sampling of real images dra
wn from the
            #  training data and what the Generator is capable of producing at the end of
training:
            real_batch = next(iter(dataloader))
            real_batch = real_batch[0]
            plt.figure(figsize=(15,15))
```

```
            plt.subplot(1,2,1)
            plt.axis("off")
            plt.title("Real Images")
            plt.imshow(np.transpose(torchvision.utils.make_grid(real_batch.to("cuda:0" if
torch.cuda.is_available() else "cpu"),
                                                    padding=1, pad_value=1, normalize=Tru
e).cpu(),(1,2,0)))
            plt.subplot(1,2,2)
            plt.axis("off")
            plt.title("Fake Images")
            plt.imshow(np.transpose(img_list[-1],(1,2,0)))
            plt.savefig(dir_name_for_results + "/real_vs_fake_images.png")
            plt.show()
            torch.save(netG.state_dict(), "/content/drive/MyDrive/DLStudio-2.4.2/Examples
AdversarialLearning/results_DG1/save_model_loss_1")


# In[ ]:


discriminator =  DiscriminatorDG1()
generator =  GeneratorDG1()

num_learnable_params_disc = sum(p.numel() for p in discriminator.parameters() if p.requir
es_grad)
print("\n\nThe number of learnable parameters in the Discriminator: %d\n" % num_learnable
_params_disc)
num_learnable_params_gen = sum(p.numel() for p in generator.parameters() if p.requires_gr
ad)
print("\nThe number of learnable parameters in the Generator: %d\n" % num_learnable_param
s_gen)
num_layers_disc = len(list(discriminator.parameters()))
print("\nThe number of layers in the discriminator: %d\n" % num_layers_disc)
num_layers_gen = len(list(generator.parameters()))
print("\nThe number of layers in the generator: %d\n\n" % num_layers_gen)

run_gan_code(discriminator=discriminator, generator=generator, results_dir="results_DG1",
dataloader=dataloader)


# In[ ]:


## Code to generate 1000 images from the saved generator model
generator.load_state_dict(torch.load("/content/drive/MyDrive/DLStudio-2.4.2/ExamplesAdver
sarialLearning/results_DG1/save_model_loss_1"))
input_noise = torch.randn(1000, 100, 1, 1, device=torch.device("cpu"))
fake_imgs = []
j = 1
with torch.no_grad():
  newfake = generator(input_noise).detach().cpu()
```

```
fake_imgs.append(newfake)


# In[ ]:


## Code to save the generated images
for i in range(len(fake_imgs[0])):
  image = fake_imgs[0][i].permute(1, 2, 0).cpu().detach().numpy()
  plt.imshow(image)
  plt.axis('off')  # Hide axis
  plt.savefig(f'/content/drive/MyDrive/DLStudio-2.4.2/ExamplesAdversarialLearning/results
_DG1/gen_fake_images1/image_{i}.png')


# In[ ]:


# getting the list of image paths for FID error calculation
def list_files_in_folder(folder_path):
    files = []
    for file_name in os.listdir(folder_path):
        if os.path.isfile(os.path.join(folder_path, file_name)):
            files.append(os.path.join(folder_path, file_name))
    return files

real_folder_path = '/content/drive/MyDrive/DLStudio-2.4.2/ExamplesAdversarialLearning/cel
eba_dataset_64x64/0'
fake_folder_path_1 = '/content/drive/MyDrive/DLStudio-2.4.2/ExamplesAdversarialLearning/r
esults_DG1/gen_fake_images1'

real_image_paths = list_files_in_folder(real_folder_path)
fake_image_paths_1 = list_files_in_folder(fake_folder_path_1)


# In[ ]:


#FID for GAN
from pytorch_fid.fid_score import calculate_activation_statistics, calculate_frechet_dist
ance
from pytorch_fid.inception import InceptionV3
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
dims = 2048
block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
model = InceptionV3([block_idx]).to(device)
m1 , s1 = calculate_activation_statistics(real_image_paths , model , device = device )
m2 , s2 = calculate_activation_statistics(fake_image_paths_1 , model , device = device )
fid_value = calculate_frechet_distance(m1 , s1 , m2 , s2)
print(f'FID: { fid_value: .2f}')
```

```
# In[ ]:


# Displaying sample of generated images
def display_random_images(folder_path, num_images=16, rows=4, cols=4):
    # List all files in the folder
    files = os.listdir(folder_path)
    # Select num_images random files
    random_files = random.sample(files, num_images)

    # Create a subplot grid with the specified number of rows and columns
    fig, axes = plt.subplots(rows, cols, figsize=(10, 10))
    for i, ax in enumerate(axes.flat):
        # Open and display each image
        img_path = os.path.join(folder_path, random_files[i])
        img = Image.open(img_path)
        ax.imshow(img)
        ax.axis('off')  # Turn off axis labels
    plt.show()

folder_path_gan = "/content/drive/MyDrive/DLStudio-2.4.2/ExamplesAdversarialLearning/resu
lts_DG1/gen_fake_images1"
display_random_images(folder_path_gan)
```

- Generative Diffusion:

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


import os, sys
import torch
import torchvision.transforms as tvt
from PIL import Image
import numpy as np
import glob
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms as tvt
from torchvision import utils
import seaborn as sns
```

```
get_ipython().run_line_magic('matplotlib', 'inline')
import matplotlib.pyplot as plt
import random
import time
import skimage.io as io
from pycocotools.coco import COCO
import copy
from scipy.ndimage import zoom
import torch.optim as optim
import re
import math
import random
import copy
import matplotlib.pyplot as plt
import gzip
import pickle
import time
import logging

seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)


# In[ ]:


# This code has been picked up from the DL Studio Library
# Diffusion Image Generation using the provided model weights
import os
import sys
import numpy as np

import torch

sys.path.append("/content/drive/MyDrive/DLStudio-2.4.2/DLStudio")
sys.path.append("/content/drive/MyDrive/DLStudio-2.4.2/GenerativeDiffusion")

from GenerativeDiffusion import *

gauss_diffusion   =  GaussianDiffusion(
                        num_diffusion_timesteps = 1000,

                    )
```

```
network =  UNetModel(
                       in_channels=3,
                       model_channels   =  128,
                       out_channels     =  3,
                       num_res_blocks   =  2,
                       attention_resolutions =  (4, 8),              ## for 64x64 images
                       channel_mult     =    (1, 2, 3, 4),           ## for 64x64 images
                       num_heads        =    1,
                       attention        =    True            ## <<< Must be the same as f
or RunCodeForDiffusion.py
#                       attention        =    False          ## <<< Must be the same as f
or RunCodeForDiffusion.py

                     )


top_level = GenerativeDiffusion(
                        gen_new_images        =        True,
                        image_size            =        64,
                        num_channels          =        128,
                        ema_rate              =        0.9999,
                        diffusion = gauss_diffusion,
                        network = network,
                        ngpu = 1,
                        path_saved_model = "/content/drive/MyDrive/DLStudio-2.4.2/Example
sDiffusion/saved_model/", ##using the provided diffusion model weights
                        clip_denoised=True,
                        num_samples=1000,
                        batch_size_image_generation=4,
             )

if sys.argv[1] == '--model_path':
    model_path = sys.argv[2]


network.load_state_dict( torch.load("/content/drive/MyDrive/DLStudio-2.4.2/ExamplesDiffus
ion/saved_model/diffusion.pt") )

network.to(top_level.device)
network.eval()

print("sampling...")
all_images = []

while len(all_images) * top_level.batch_size_image_generation < top_level.num_samples:
    sample = gauss_diffusion.p_sampler_for_image_generation(
        network,
        (top_level.batch_size_image_generation, 3, top_level.image_size, top_level.image_
size),
        device = top_level.device,
        clip_denoised = top_level.clip_denoised,
```

```
    )
    sample = ((sample + 1) * 127.5).clamp(0, 255).to(torch.uint8)
    sample = sample.permute(0, 2, 3, 1)
    sample = sample.contiguous()
    gathered_samples = [sample]
    all_images.extend([sample.cpu().numpy() for sample in gathered_samples])
    print(f"created {len(all_images) * top_level.batch_size_image_generation} samples")

arr = np.concatenate(all_images, axis=0)
arr = arr[: top_level.num_samples]

shape_str = "x".join([str(x) for x in arr.shape])
out_path = os.path.join("/content/drive/MyDrive/DLStudio-2.4.2/ExamplesDiffusion", f"samp
les_{shape_str}.npz")

np.savez(out_path, arr)

print("image generation completed")


# In[ ]:


# This code has been picked up from the DL Studio Library

data = np.load("/content/drive/MyDrive/DLStudio-2.4.2/ExamplesDiffusion/samples_2048x64x6
4x3.npz")

print("\n\n[visualize_sample.py]  the data object: ", data)
## NpzFile 'RESULTS/samples_8x64x64x3.npz' with keys: arr_0
print("\n\n[visualize_sample.py]  type of the data object: ", type(data))
## <class 'numpy.lib.npyio.NpzFile'>
print("\n\n[visualize_sample.py]  shape of the object data['arr_0']: ", data['arr_0'].sha
pe)      ## (8, 64, 64, 3)

for i, img in enumerate(data['arr_0']):
    plt.figure()
    plt.imshow(img)
    plt.axis("off")
    plt.savefig(f"/content/drive/MyDrive/DLStudio-2.4.2/ExamplesDiffusion/visualize_sampl
es/test_{i}.jpeg")


# In[ ]:


def list_files_in_folder(folder_path):
    files = []
    for file_name in os.listdir(folder_path):
        if os.path.isfile(os.path.join(folder_path, file_name)):
            files.append(os.path.join(folder_path, file_name))
```

```
    return files

real_folder_path = '/content/drive/MyDrive/DLStudio-2.4.2/ExamplesAdversarialLearning/cel
eba_dataset_64x64/0'
fake_folder_path_1 = '/content/drive/MyDrive/DLStudio-2.4.2/ExamplesAdversarialLearning/r
esults_DG1/gen_fake_images1'

real_image_paths = list_files_in_folder(real_folder_path)
fake_image_paths_diffusion = list_files_in_folder(fake_folder_diffusion)


# In[ ]:


from pytorch_fid.fid_score import calculate_activation_statistics, calculate_frechet_dist
ance
from pytorch_fid.inception import InceptionV3
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
dims = 2048
block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
model = InceptionV3([block_idx]).to(device)
m1 , s1 = calculate_activation_statistics(real_image_paths , model , device = device )
m2 , s2 = calculate_activation_statistics(fake_image_paths_diffusion , model , device = d
evice )
fid_value = calculate_frechet_distance(m1 , s1 , m2 , s2)
print(f'FID: { fid_value: .2f}')


# In[ ]:


# Displaying sample of generated images
def display_random_images(folder_path, num_images=16, rows=4, cols=4):
    files = os.listdir(folder_path)
    random_files = random.sample(files, num_images)

    fig, axes = plt.subplots(rows, cols, figsize=(10, 10))
    for i, ax in enumerate(axes.flat):
        # Open and display each image
        img_path = os.path.join(folder_path, random_files[i])
        img = Image.open(img_path)
        ax.imshow(img)
        ax.axis('off')
    plt.show()

folder_path_diffusion = "/content/drive/MyDrive/DLStudio-2.4.2/ExamplesDiffusion/visualiz
e_samples"
display_random_images(folder_path_diffusion)
```