Homework 8

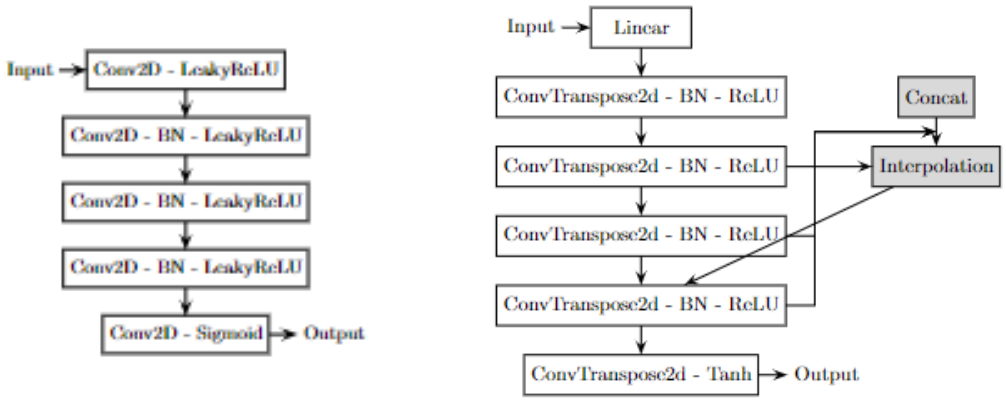
Alexandre Olive Pellicer

Building a GAN

For the discriminator, we have used a similar structure as the one proposed in the DCGAN. Nevertheless, we changed a bit the implementation and some hyperparameters compared from the implementation in the "AdversarialLearning.py" file from DLStudio in order to try to achieve a better performance and faster training. The implementation can be found at the end of the document.

For the discriminator, we have improved the DCGAN implementation in order to achieve an improvement in the quality of the generated images. We still used the Transpose Convolutions as stated in the instructions, but we added skip connections in order to enhance the performance. See Fig 1 for the new implementation of the generator. Also find the code at the end of the document. Some reasons that encouraged us to add the skip connections are the following:

- Preservation of Information. Skip connections allow the network to retain information from earlier layers and propagate it to later layers. This can help in preserving fine-grained details of the input image that may otherwise be lost during the up sampling process.
- Faster Training. Skip connections facilitate the flow of gradients through the network, enabling faster convergence during training.
- Mitigation of Vanishing Gradient Problem
- Improved Performance



(a) (b)

Fig 1: (a) Block diagram of the discriminator (b) Block diagram of the generator

In order to implement the training logic we have followed the implementation from the Slide 64 through 69 of Week 11. We have used the nn.BCELoss for training both the generator and the discriminator. These are the adversarial losses over training iterations for both the generator and the discriminator:

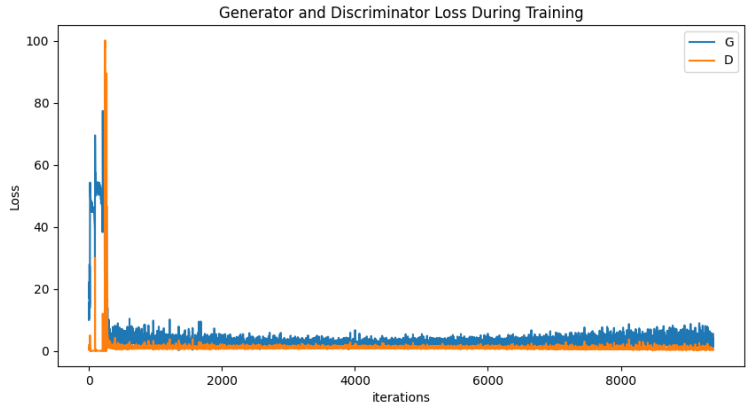


Fig 2: Adversarial losses over training iterations for both the generator and the discriminator

From the plot of the training losses, we can see that both the generator and the discriminator loss get stabilized after the first iterations and that the loss values keep close to a small value as expected. Therefore, we do not get a mode collapse. This is also proven by the quality of the generated samples.

Generating synthetic samples

Synthetic images generated by our implemented GAN using Transpose Convolutions:

Fig 3: Synthetic images generated by our implemented GAN

From the obtained results using the GAN, we can see that our implementation of the GAN can generate images where we can identify faces of people. Something similar in most of the images is that we can distinguish the gender of the person and the parts of the faces. The images do not look realistic at all. A human could easily detect that these images are synthetic. We can see that the textures are not well defined at all. Images that contain objects from the real world are complex images. They contain many different textures and edges. Our implementation of the GAN struggles a bit when it comes to generating these textures. About the differences, in some images, like fourth column fourth row, we can see that the generator is mixing two different patches of faces, to create the final picture of the woman. It looks like the two eyes are different and the color of the face doesn't look uniform.

Synthetic images generated by the Diffusion Model:

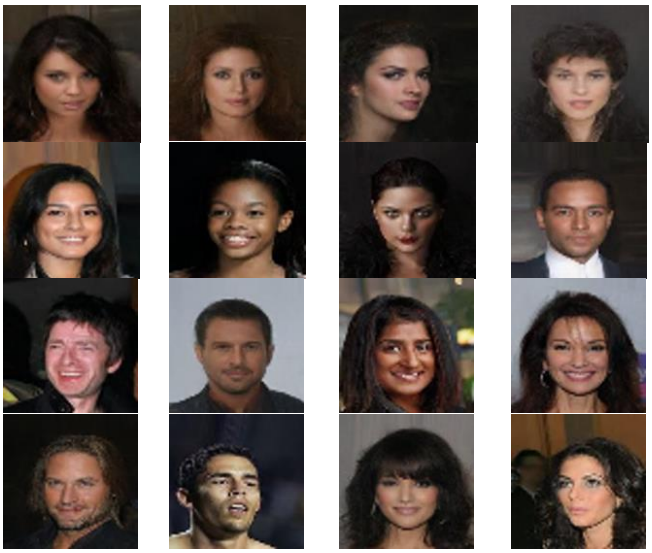


Fig 4: Synthetic images generated by the Diffusion Model

From the obtained results using the Diffusion Model with the pretrained weights provided, we see that in the generated images we can clearly identify the faces of people. We see that the colors of the skin look coherent in the entire face. Something similar in all the generated images is that the backgrounds tend to be uniform. About differences, in some cases, the images look a bit blurry as if they had been filtered with a low pass filter. For example, the first row fourth column. Also, in the last row first and second column it looks like there is some sparky noise (pixels with random value) in the faces of the two men.

Calculating FID

In order to calculate the FID we have used the snippet code provided in the instructions. In the instructions it is mentioned to compute the FID with 1000 real and synthetic images. When doing it we got the following error "ValueError: Imaginary component". As mentioned on "Piazza" looking in the discussion from https://github.com/mseitzer/pytorch-fid/issues/13 we need to use a minimum of 2048 real and synthetic images. Therefore, we compute the FID score using 2048 real and synthetic images. This is the code used:

```
from pytorch_fid.fid_score \
    import calculate_activation_statistics, \
    calculate_frechet_distance
from pytorch_fid.inception import InceptionV3
import os
import random

device = "cuda:1"

# take 2500 random real samples
directory_real =
'/home/aolivepe/ECE60146/HW8/data/celeba_dataset_64x64/0'
real_paths = os.listdir(directory_real)
real_paths = random.sample(real_paths, 2048)
```

```
real_paths = [os.path.join(directory_real, file) for file in real_paths]

# take the 2500 synthetic samples
directory_fake = '/home/aolivepe/ECE60146/HW8/DLStudio-
2.4.3/ExamplesDiffusion/visualize_samples_resized'
fake_paths = os.listdir(directory_fake)
fake_paths = random.sample(fake_paths, 2048)
fake_paths = [os.path.join(directory_fake, file) for file in fake_paths]

dims = 2048

block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
model = InceptionV3([block_idx]).to(device)
m1, s1 = calculate_activation_statistics(
real_paths, model, device=device)
m2, s2 = calculate_activation_statistics(
fake_paths, model, device=device)
fid_value = calculate_frechet_distance(m1, s1, m2, s2)
print(f'FID: {fid_value:.2f}')
```

For computing the FID of the samples generated by the Diffusion Model, we previously resized the samples obtained after running the "VisualizeSamples.py" file from 256 to 64 in order to have the same size as the real data.

FID for the samples generated with the BCE-GAN: **72.03**

FID for the samples generated with the Diffusion Model: **51.19**

Since the FID is a distance, we are interested in having the lowest value possible. Thus, from the quantitative point of view, the best performance is achieved when using the Diffusion Model.

Comparing BCE-GAN and Diffusion Model

Doing a qualitative comparison between the results obtained using the BCE-GAN and the Diffusion Model, we can say that the best results are achieved when using the Diffusion Model. The images generated with the Diffusion Model look more realistic although in some cases a human could be able to identify the fake ones because some of them look blurry. The images generated by the BCE-GAN can easily be labeled as synthetic by a human because of the fact that contours are not well defined and in some cases, we could see that faces look like created from different patches (two eyes don't look equal or different skin color in the same face).

From the qualitative analysis we mentioned that the best performance is achieved when using the Diffusion Model. This qualitative analysis is backed by the quantitative metric, the FID. From the results we see that the lowest FID (i.e. the most realistic samples) is achieved when using the Diffusion Model.

In conclusion, as seen in the qualitative and quantitative analysis, the best performance is achieved when using the Diffusion Model.

Code

mygan.py

```
import random
import numpy
import torch
import os
import sys
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as tvt
import glob
import torch.optim as optim
import
imageio

import matplotlib.pyplot as plt
import torchvision.transforms.functional as tvtF
import numpy as np

# Discriminator network definition
class Discriminator(nn.Module):
    def __init__(self, num_colors=3, depths=128, image_size=64):
        super(Discriminator, self).__init__()

        # Convolutional layers with LeakyReLU activations
        self.conv1 = nn.Sequential(
            nn.Conv2d(num_colors, depths, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(depths, depths * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(depths * 2),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(depths * 2, depths * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(depths * 4),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.conv4 = nn.Sequential(
            nn.Conv2d(depths * 4, depths * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(depths * 8),
            nn.LeakyReLU(0.2, inplace=True)
        )
```

```
        self.conv5 = nn.Sequential(
            nn.Conv2d(depths * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

        self.final_size = image_size // 16

    def forward(self, x):
        # Forward pass through convolutional layers
        conv1_out = self.conv1(x)
        conv2_out = self.conv2(conv1_out)
        conv3_out = self.conv3(conv2_out)
        conv4_out = self.conv4(conv3_out)
        conv5_out = self.conv5(conv4_out)
        return conv5_out.squeeze()

# Generator network definition
class Generator(nn.Module):
    def __init__(self, num_noises=100, num_colors=3, depths=128,
image_size=64):
        super(Generator, self).__init__()

        if image_size % 16 != 0:
            raise Exception("Size of the image must be divisible by 16")
        self.final_size = image_size // 16
        self.depths = depths

        # Linear layer and convolutional layers with ReLU activations
        self.lin = nn.Linear(num_noises, depths * 8 * self.final_size *
self.final_size)
        self.conv1 = nn.Sequential(
            nn.ConvTranspose2d(depths * 8, depths * 4, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(depths * 4),
            nn.ReLU()
        )
        self.conv2 = nn.Sequential(
            nn.ConvTranspose2d(depths * 4, depths * 2, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(depths * 2),
            nn.ReLU()
        )
        self.conv3 = nn.Sequential(
            nn.ConvTranspose2d(depths * 2, depths, 4, 2, 1, bias=False),
            nn.BatchNorm2d(depths),
            nn.ReLU()
        )
        self.conv4 = nn.Sequential(
```

```
            nn.ConvTranspose2d(depths * 2 + depths, depths, 4, 2, 1,
bias=False),  # Skip connection
            nn.BatchNorm2d(depths),
            nn.ReLU()
        )
        self.conv5 = nn.Sequential(
            nn.ConvTranspose2d(depths, num_colors, 4, 2, 1, bias=False),
            nn.Tanh()
        )

    def forward(self, x):
        # Forward pass through linear and convolutional layers
        lin_out = self.lin(x)
        lin_out = lin_out.view(-1, self.depths * 8, self.final_size,
self.final_size)
        conv1_out = self.conv1(lin_out)
        conv2_out = self.conv2(conv1_out)
        conv3_out = self.conv3(conv2_out)
        resized_conv2_out = nn.functional.interpolate(conv2_out,
size=(conv3_out.size(2), conv3_out.size(3)), mode='bilinear',
align_corners=False)
        cat_input = torch.cat((conv3_out, resized_conv2_out), dim=1)  #
Concatenate with conv2_out
        conv4_out = self.conv4(cat_input)
        conv5_out = self.conv5(conv4_out)
        conv5_out = nn.functional.interpolate(conv5_out, size=(64, 64),
mode='bilinear', align_corners=False)
        return conv5_out

## Defining variables ----------------------------------------------------
---
dataroot = "./../../data/celeba_dataset_64x64/"
image_size = [64,64]
batch_size = 32
num_workers = 2
dir_name_for_results = "final_experiment"
device = "cuda:0"

learning_rate = 0.0002
beta1 = 0.5
epochs = 30
## ----------------------------------------------------------------------
---

## Dataset and dataloader ------------------------------------------------
---
dataset = torchvision.datasets.ImageFolder(root=dataroot,
                                           transform =
tvt.Compose([
```

```
                                                tvt.Resize(image_size
),
                                                tvt.CenterCrop(image_
size),
                                                tvt.ToTensor(),

                                                tvt.Normalize((0.5,
0.5, 0.5), (0.5, 0.5, 0.5)),
                                            ]))
train_dataloader = torch.utils.data.DataLoader(dataset,
batch_size=batch_size, shuffle=True, num_workers=num_workers)
## ------------------------------------------------------------------
----

# Function for initializing weights of the network
def weights_init(m):
    """
    Uses the DCGAN initializations for the weights
    """
    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)

# Creating instances of discriminator and generator
discriminator =  Discriminator()
generator =  Generator()

nz = 100
netD = discriminator.to(device)
netG = generator.to(device)

# Applying weight initialization to discriminator and generator
netD.apply(weights_init)
netG.apply(weights_init)

fixed_noise = torch.randn(batch_size, nz, device=device)

real_label = 1
fake_label = 0

optimizerD = optim.Adam(netD.parameters(), lr=learning_rate,
betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=learning_rate,
betas=(beta1, 0.999))

criterion = nn.BCELoss()
```

```
G_losses = []
D_losses = []

# Training logic. Inspired from the implementation available in the
"AdversarialLearning.py" file from DL-Studio
for epoch in range(epochs):
    g_losses_per_print_cycle = [] # List to store generator losses for
each print cycle
    d_losses_per_print_cycle = [] # List to store discriminator losses
for each print cycle
    for i, data in enumerate(train_dataloader, 0):

        netD.zero_grad()

        real_images_in_batch = data[0].to(device) # Get real images
        b_size = real_images_in_batch.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float,
device=device)
        output = netD(real_images_in_batch).view(-1) # Forward pass real
images through discriminator
        lossD_for_reals = criterion(output, label) # Calculate loss for
real images
        lossD_for_reals.backward()

        noise = torch.randn(b_size, nz, device=device)
        fakes = netG(noise) # Generate fake images
        label.fill_(fake_label)
        output = netD(fakes.detach()).view(-1) # Forward pass fake images
through discriminator
        lossD_for_fakes = criterion(output, label) # Calculate loss for
fake images
        lossD_for_fakes.backward()

        lossD = lossD_for_reals + lossD_for_fakes # Total discriminator
loss
        d_losses_per_print_cycle.append(lossD)
        optimizerD.step()

        netG.zero_grad() # Reset gradients of the generator
        label.fill_(real_label)
        output = netD(fakes).view(-1) # Forward pass fake images through
discriminator
        lossG = criterion(output, label) # Calculate generator loss
        g_losses_per_print_cycle.append(lossG)
        lossG.backward()
        optimizerG.step()

        if i % 100 == 99:
```

```
            mean_D_loss =
torch.mean(torch.FloatTensor(d_losses_per_print_cycle)) # Calculate mean
discriminator loss
            mean_G_loss =
torch.mean(torch.FloatTensor(g_losses_per_print_cycle)) # Calculate mean
generator loss
            print("[epoch=%d/%d   iter=%4d]     mean_D_loss=%7.4f    mean
_G_loss=%7.4f" % ((epoch+1),epochs,(i+1),mean_D_loss,mean_G_loss))
            d_losses_per_print_cycle = []
            g_losses_per_print_cycle = []

        G_losses.append(lossG.item()) # Append generator loss to overall
list
        D_losses.append(lossD.item()) # Append discriminator loss to
overall list

# Save generator model
torch.save(netG.state_dict(), './netG.pth')

# Plot generator and discriminator losses during training
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")
```

mygan_inference.py

```
import random
import numpy
import torch
import os
import sys
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as tvt
import glob
import torch.optim as optim
import time
```

```
import
imageio

import matplotlib.pyplot as plt
import torchvision.transforms.functional as tvtF
import numpy as np

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)

# Discriminator network definition
class Discriminator(nn.Module):
    def __init__(self, num_colors=3, depths=128, image_size=64):
        super(Discriminator, self).__init__()

        # Convolutional layers with LeakyReLU activations
        self.conv1 = nn.Sequential(
            nn.Conv2d(num_colors, depths, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(depths, depths * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(depths * 2),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(depths * 2, depths * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(depths * 4),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.conv4 = nn.Sequential(
            nn.Conv2d(depths * 4, depths * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(depths * 8),
            nn.LeakyReLU(0.2, inplace=True)
        )
        self.conv5 = nn.Sequential(
            nn.Conv2d(depths * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

        self.final_size = image_size // 16

    def forward(self, x):
```

```
        # Forward pass through convolutional layers
        conv1_out = self.conv1(x)
        conv2_out = self.conv2(conv1_out)
        conv3_out = self.conv3(conv2_out)
        conv4_out = self.conv4(conv3_out)
        conv5_out = self.conv5(conv4_out)
        return conv5_out.squeeze()

# Generator network definition
class Generator(nn.Module):
    def __init__(self, num_noises=100, num_colors=3, depths=128,
image_size=64):
        super(Generator, self).__init__()

        if image_size % 16 != 0:
            raise Exception("Size of the image must be divisible by 16")
        self.final_size = image_size // 16
        self.depths = depths

        # Linear layer and convolutional layers with ReLU activations
        self.lin = nn.Linear(num_noises, depths * 8 * self.final_size *
self.final_size)
        self.conv1 = nn.Sequential(
            nn.ConvTranspose2d(depths * 8, depths * 4, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(depths * 4),
            nn.ReLU()
        )
        self.conv2 = nn.Sequential(
            nn.ConvTranspose2d(depths * 4, depths * 2, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(depths * 2),
            nn.ReLU()
        )
        self.conv3 = nn.Sequential(
            nn.ConvTranspose2d(depths * 2, depths, 4, 2, 1, bias=False),
            nn.BatchNorm2d(depths),
            nn.ReLU()
        )
        self.conv4 = nn.Sequential(
            nn.ConvTranspose2d(depths * 2 + depths, depths, 4, 2, 1,
bias=False),  # Skip connection
            nn.BatchNorm2d(depths),
            nn.ReLU()
        )
        self.conv5 = nn.Sequential(
            nn.ConvTranspose2d(depths, num_colors, 4, 2, 1, bias=False),
            nn.Tanh()
        )
```

```
    def forward(self, x):
        # Forward pass through linear and convolutional layers
        lin_out = self.lin(x)
        lin_out = lin_out.view(-1, self.depths * 8, self.final_size,
self.final_size)
        conv1_out = self.conv1(lin_out)
        conv2_out = self.conv2(conv1_out)
        conv3_out = self.conv3(conv2_out)
        resized_conv2_out = nn.functional.interpolate(conv2_out,
size=(conv3_out.size(2), conv3_out.size(3)), mode='bilinear',
align_corners=False)
        cat_input = torch.cat((conv3_out, resized_conv2_out), dim=1)  #
Concatenate with conv2_out
        conv4_out = self.conv4(cat_input)
        conv5_out = self.conv5(conv4_out)
        conv5_out = nn.functional.interpolate(conv5_out, size=(64, 64),
mode='bilinear', align_corners=False)
        return conv5_out

## Defining variables ----------------------------------------------------
---
dataroot = "./../../data/celeba_dataset_64x64/"
image_size = [64,64]
batch_size = 32
num_workers = 2
dir_name_for_results = "results_mix"
device = "cuda:1"

learning_rate = 0.0002
beta1 = 0.5
epochs = 30
## ----------------------------------------------------------------------
---

## Dataset and dataloader ------------------------------------------------
---
dataset = torchvision.datasets.ImageFolder(root=dataroot,
                                           transform =
tvt.Compose([
                                                      tvt.Resize(image_size
),
                                                      tvt.CenterCrop(image_
size),
                                                      tvt.ToTensor(),

                                                      tvt.Normalize((0.5,
0.5, 0.5), (0.5, 0.5, 0.5)),
                                           ]))
```

```
train_dataloader = torch.utils.data.DataLoader(dataset,
batch_size=batch_size, shuffle=True, num_workers=num_workers)
## ------------------------------------------------------------------------
----

discriminator =  Discriminator()
generator =  Generator()

nz = 100
netD = discriminator.to(device)
netG = generator.to(device)
netG.load_state_dict(torch.load('./netG.pth'))

# Generate 2048 samples
for i in range(2048):
    fixed_noise = torch.randn(1, nz, device=device)
    fake = netG(fixed_noise).detach().cpu()
    img = tvtF.to_pil_image(torchvision.utils.make_grid(fake, padding=1,
pad_value=1, normalize=True))
    imageio.imwrite(f"results_mygan_Gskip_Dskip_experiments_final/image_{
i}.png", img)
```

fid.py

```
from pytorch_fid.fid_score \
    import calculate_activation_statistics, \
    calculate_frechet_distance
from pytorch_fid.inception import InceptionV3
import os
import random

device = "cuda:1"

# take 2048 random real samples
directory_real =
'/home/aolivepe/ECE60146/HW8/data/celeba_dataset_64x64/0'
real_paths = os.listdir(directory_real)
real_paths = random.sample(real_paths, 2048)
real_paths = [os.path.join(directory_real, file) for file in real_paths]

# take the 2048 synthetic samples
directory_fake = '/home/aolivepe/ECE60146/HW8/DLStudio-
2.4.3/ExamplesAdversarialLearning/results_mygan_Gskip_Dskip_experiments_f
inal'
fake_paths = os.listdir(directory_fake)
fake_paths = random.sample(fake_paths, 2048)
fake_paths = [os.path.join(directory_fake, file) for file in fake_paths]
```

```
dims = 2048

block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
model = InceptionV3([block_idx]).to(device)
m1, s1 = calculate_activation_statistics(
real_paths, model, device=device)
m2, s2 = calculate_activation_statistics(
fake_paths, model, device=device)
fid_value = calculate_frechet_distance(m1, s1, m2, s2)
print(f'FID: {fid_value:.2f}')
```