Homework 9 - Deep Learning (ECE 60146)

Ayan Biswas Pranta
PUID 037714249

April 07, 2025

Introduction

In this homework, two generative models—Generative Adversarial Networks (GANs) and Diffusion Models—are explored and compared. Both models are used to generate images, which are then analyzed both qualitatively and quantitatively. The generated images are evaluated in terms of visual quality, diversity, and fidelity. Following this, the GAN model is fine-tuned using images produced by the Diffusion model.

1. GAN

1.1 Model

Generative Adversarial Network (GAN) generate images from noise with model that gets matured through training. The model consists of two neural networks:

- **Generator**: The network that converts a noise vector into an image.

The generator takes in random noise as input and applies **transpose convolutions** to **upsample** it into a full-sized image. Each Transpose Convolution layer in the generator follows this pattern to progressively increase the feature map size. **Batch Normalization** and **ReLU** activations refine details. The last layer uses **tanh activation** to scale pixel values to [-1,1], suitable for image data. The discriminator takes an image (real or generated) and outputs a probability of it being real.

Input: Random noise vector (latent space)

Dense Layer: Fully connected, projecting to a higher-dimensional space

Reshape Layer: Reshapes to a spatial structure (e.g., 4×4×512)

Transposed Convolutions:

ConvTranspose2D (512 → 256) + BatchNorm + ReLU

ConvTranspose2D (256 → 128) + BatchNorm + ReLU

ConvTranspose2D (128 → 64) + BatchNorm + ReLU

ConvTranspose2D (64 → 3) + Tanh (Output Image)

- **Discriminator**: The network that classifies images as real or fake.

Input: Generated or real image (e.g., 64×64×3)

Convolutions:

Conv2D (3 → 64) + LeakyReLU

Conv2D (64 → 128) + BatchNorm + LeakyReLU

Conv2D (128 → 256) + BatchNorm + LeakyReLU

Conv2D (256 → 512) + BatchNorm + LeakyReLU

Flatten + Dense Layer: Outputs probability (Real/Fake) using a Sigmoid function.

```
# Borrowed from Sushant Gautam (gan-beginner-tutorial-on-celeba-dataset)
# Modified by me (Ayan Biswas Pranta) for this homework

import torch
import torch.nn as nn
from torch.utils.data import DataLoader
from torchvision.utils import make_grid, save_image
from torchvision.datasets import ImageFolder
import torchvision.transforms as T
import matplotlib.pyplot as plt
from IPython.display import Image
import cv2
import os
from tqdm.notebook import tqdm
import torch.nn.functional as F

data_directory = './celeba_dataset_64x64_100k/'

image_size = 64
batch_size = 128
mean_std_for_norm = (0.5, 0.5, 0.5), (0.5, 0.5, 0.5)  # mean, std for normalize
images
```

```
train_dataset = ImageFolder(root=data_directory, transform=T.Compose([
    T.Resize(image_size),
    T.CenterCrop(image_size),
    T.ToTensor(),
    T.Normalize(*mean_std_for_norm)
]))
train_dataloader = DataLoader(train_dataset, batch_size, shuffle=True,
num_workers=0, pin_memory=True)

def denormalization(img_tensors):
    return img_tensors * mean_std_for_norm[1][0] + mean_std_for_norm[0][0]

def display_images(images, nmax=64):
    fig, ax = plt.subplots(figsize=(8,8))
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(make_grid(denormalization(images.detach()[:nmax]), nrow=8).permute(1,
2, 0))

def to_device(data, device):
    """Move tensor(s) to chosen device"""
    if isinstance(data, (list, tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)

class DeviceDataLoader():
    """Wrap a dataloader to move data to a device"""
    def __init__(self, dl, device):
        self.dl = dl
        self.device = device

    def __iter__(self):
        """Yield a batch of data after moving it to device"""
        for b in self.dl:
            yield to_device(b, self.device)

    def __len__(self):
        """Number of batches"""
        return len(self.dl)

device = torch.device('cuda')
train_dataloader = DeviceDataLoader(train_dataloader, device)
```

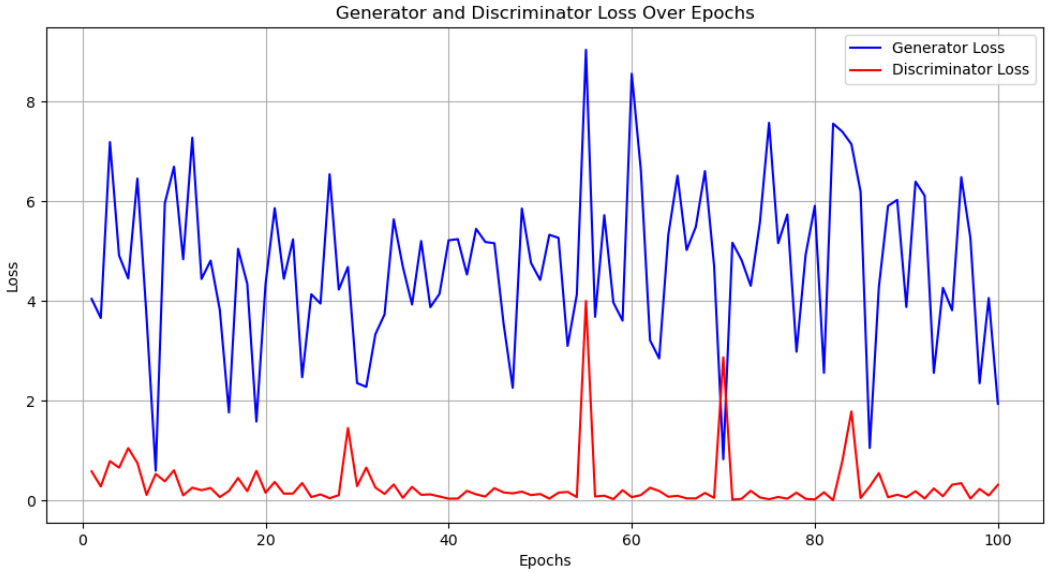
```
discriminator = nn.Sequential(
    # in: 3x 64 x 64
    nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(64),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 64 x 32 x 32

    nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(128),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 128 x 16 x 16

    nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 256 x 8 x 8

    nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(512),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 512 x 4 x 4

    nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0, bias=False),
    # out: 1 x 1 x 1

    nn.Flatten(),
    nn.Sigmoid()
)


discriminator = to_device(discriminator, device)


generator_input_size = 128


generator = nn.Sequential(
    # in: noise_size x 1 x 1
    nn.ConvTranspose2d(generator_input_size, 512, kernel_size=4, stride=1,
padding=0, bias=False),
    nn.BatchNorm2d(512),
    nn.ReLU(True),
```

```
    # out: 512 x 4 x 4

    nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.ReLU(True),
    # out: 256 x 8 x 8

    nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(128),
    nn.ReLU(True),
    # out: 128 x 16 x 16

    nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(64),
    nn.ReLU(True),
    # out: 64 x 32 x 32

    nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1, bias=False),
    nn.Tanh()  # output is between -1 to 1
    # out: 3 x 64 x 64
)

generator = to_device(generator, device)  # move generator to device

def train_discriminator(real_images, opt_d):
    opt_d.zero_grad()
    real_preds = discriminator(real_images)
    real_targets = torch.ones(real_images.size(0), 1, device=device)
    real_loss = F.binary_cross_entropy(real_preds, real_targets)

    starter_noise_for_training = torch.randn(batch_size, generator_input_size, 1, 1,
device=device)
    fake_images = generator(starter_noise_for_training)
    fake_targets = torch.zeros(fake_images.size(0), 1, device=device)
    fake_preds = discriminator(fake_images)
    fake_loss = F.binary_cross_entropy(fake_preds, fake_targets)

    loss = real_loss + fake_loss
    loss.backward()
    opt_d.step()
    return loss.item()
```

```
def train_generator(opt_g):
    opt_g.zero_grad()
    starter_noise = torch.randn(batch_size, generator_input_size, 1, 1,
device=device)
    fake_images = generator(starter_noise)
    preds = discriminator(fake_images)
    targets = torch.ones(batch_size, 1, device=device)
    loss = F.binary_cross_entropy(preds, targets)
    loss.backward()
    opt_g.step()
    return loss.item()

# Directory to store checkpoints and losses plot
sample_dir = 'generated'
os.makedirs(sample_dir, exist_ok=True)

# Removed the per-epoch image saving function call

def main_training(epochs, lr):
    torch.cuda.empty_cache()
    losses_generator = []
    losses_discriminator = []
    optimizer_discriminator = torch.optim.Adam(discriminator.parameters(), lr=lr,
betas=(0.5, 0.999))
    optimizer_generator = torch.optim.Adam(generator.parameters(), lr=lr,
betas=(0.5, 0.999))

    for epoch in range(epochs):
        for real_images, _ in tqdm(train_dataloader, disable=True):
            loss_discriminator = train_discriminator(real_images,
optimizer_discriminator)
            loss_generator = train_generator(optimizer_generator)
        losses_generator.append(loss_generator)
        losses_discriminator.append(loss_discriminator)
        print("Epoch {}/{}, Generator Loss: {:.4f}, Discriminator Loss:
{:.4f}".format(
            epoch+1, epochs, loss_generator, loss_discriminator))

    return losses_generator, losses_discriminator

# Hyperparameters
lr = 0.00025
```

```
epochs = 100

losses_generator, losses_discriminator = main_training(epochs, lr)

# Save the model checkpoints
torch.save(generator.state_dict(), 'Generator_saved_model.pth')
torch.save(discriminator.state_dict(), 'Discriminator_saved_model.pth')

plt.figure(figsize=(12,6))
plt.plot(losses_discriminator, '-')
plt.plot(losses_generator, '-')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Discriminator', 'Generator'])
plt.title('Losses')
plt.savefig('losses_plot.png')
plt.show()

# ------------------------------
# Final image generation: generate 1024 images
# ------------------------------

# Create a new directory for final generated images
final_dir = 'generated'
os.makedirs(final_dir, exist_ok=True)


# Generate 1024 noise samples
final_noise = torch.randn(1024, generator_input_size, 1, 1, device=device)
# Generate 1024 images
final_images = generator(final_noise)
final_images = denormalization(final_images)


# Save each image in a separate file
for i in range(final_images.shape[0]):
    # Each image is of shape [3, 64, 64]
    filename = os.path.join(final_dir, f"generated_image_{i:04d}.png")
    save_image(final_images[i], filename)
```

1.2 Training Curve



The generator training curve isn't converging because discriminator is overpowering generator. If the discriminator is too strong, the generator struggles to improve but may still generate some passable samples. One trick is to train generator more than discriminator, i.e. Instead of the usual 1:1 training ratio, updating the generator 15-20 times before updating the discriminator. This allows the generator to catch up and learn meaningful updates before the discriminator gets too strong.

As GAN generated images are satisfactory in this homework (section 3.1), I didn't change the training ratio.

2. Diffusion

2.1 Generate Images

```
# modified the given code for generation by diffusion
# Original code is given by Kak/ instructors of ECE 60146
# I modified it so that diffusion.pt can be loaded and
# visualized in the same code

import os
import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
```

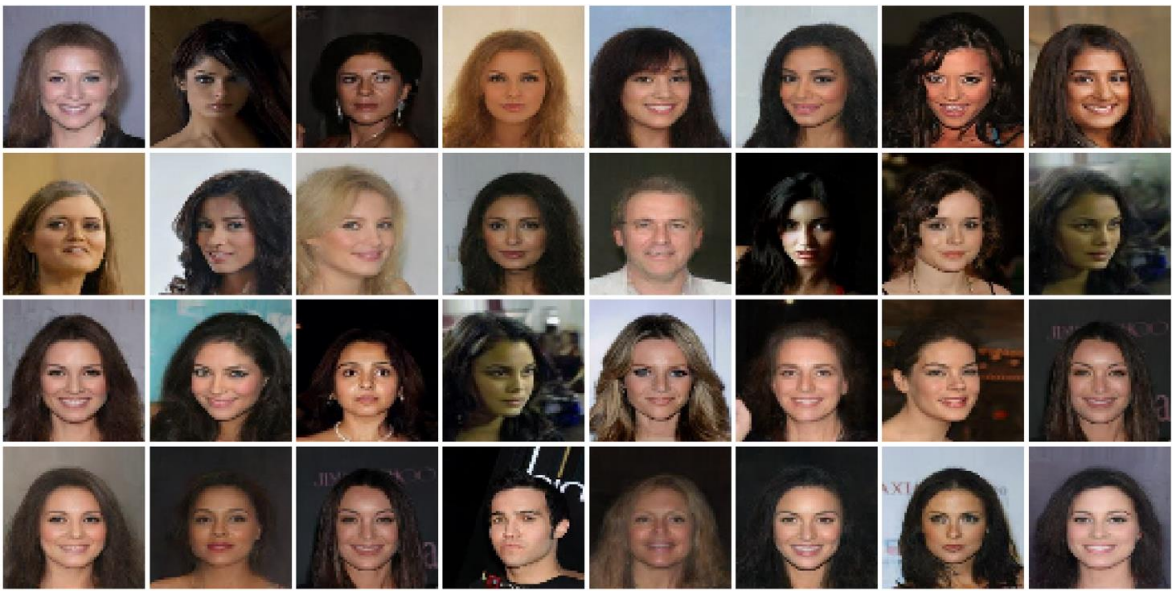
```
import glob
from PIL import Image
from GenerativeDiffusion import *

# Default model path
MODEL_PATH = "diffusion.pt"
RESULTS_DIR = "RESULTS_1024"
VISUALIZATION_DIR = "visualize_samples_1024"
IMAGE_DISPLAY_SIZE = (256, 256)

# Ensure necessary directories exist
os.makedirs(RESULTS_DIR, exist_ok=True)
os.makedirs(VISUALIZATION_DIR, exist_ok=True)

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 for 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=RESULTS_DIR,
    clip_denoised=True,
    num_samples=1024,
    batch_size_image_generation=8,
)
```

```
# Load the model
print(f"Loading model from {MODEL_PATH}...")
network.load_state_dict(torch.load(MODEL_PATH))
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).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(RESULTS_DIR, f"samples_{shape_str}.npz")

np.savez(out_path, arr)
print("image generation completed")

# Visualization
print("\n\nPutting together a collage of the generated images for display\n")
print("\nFor the individual images, check the directory 'visualize_samples'\n\n")

npz_archive = out_path
data = np.load(npz_archive)

for i, arr in enumerate(data['arr_0']):
    img = Image.fromarray(arr)
```

```
    img = img.resize(IMAGE_DISPLAY_SIZE)
    img.save(f"{VISUALIZATION_DIR}/test_{i}.jpg")

if os.path.exists(VISUALIZATION_DIR):
    im_tensor_all = torch.from_numpy(data['arr_0'][:64]).float()
    im_tensor_all = torch.transpose(im_tensor_all, 1, 3)
    im_tensor_all = torch.transpose(im_tensor_all, 2, 3)
    plt.figure(figsize=(25, 15))
    plt.imshow(np.transpose(torchvision.utils.make_grid(im_tensor_all, padding=2,
pad_value=1, normalize=True).cpu(), (1, 2, 0)))
    plt.title("Fake Images")
    plt.savefig(VISUALIZATION_DIR +  "/fake_images.png")
    plt.show()
```

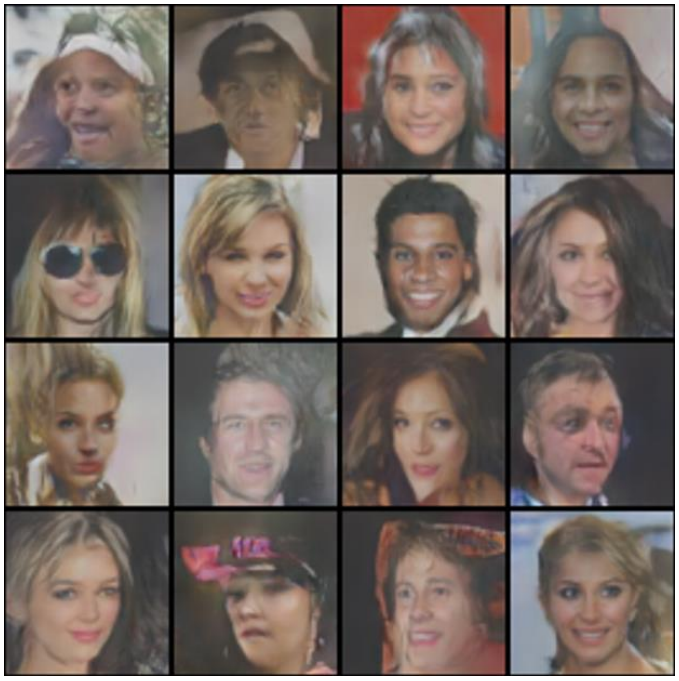
Parameters chosen for generation:

Parameter	Value	Description
model_channels	128	Base channel width of the network
num_res_blocks	2	Number of residual blocks per level
attention_resolutions	(4, 8)	Attention at specific resolution (64×64)
channel_mult	(1, 2, 3, 4)	Channel scaling factors at different resolutions
num_heads	1	Number of attention heads
num_channels	128	Number of feature channels
ema_rate	0.9999	EMA decay rate for model weights
clip_denoised	True	Clip values after denoising

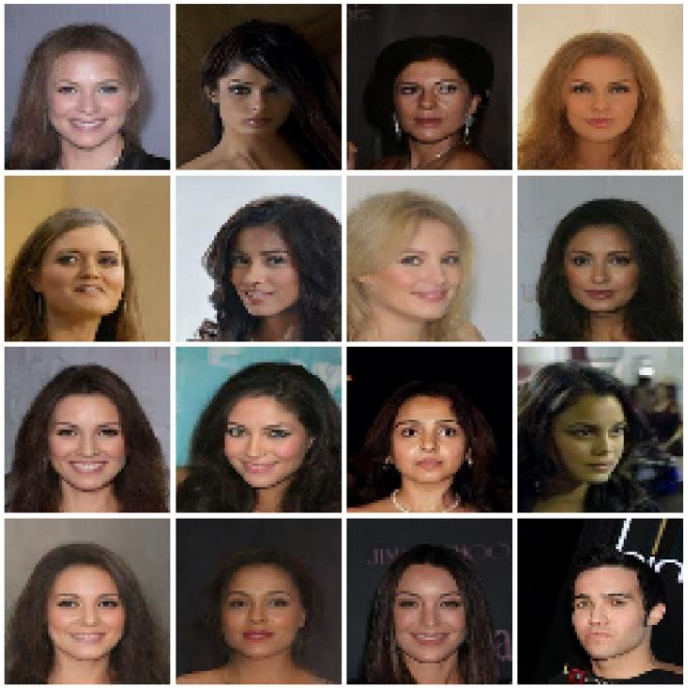


3. FID Score

3.1 Display 4x4 (16 images) images Generated by GAN



3.2 Display 4x4 (16 images) images Generated by Diffusion



The diffusion model images appear sharper, with more refined details. They are generally more realistic, with more natural skin textures, lighting, and symmetry. The GAN-generated faces sometimes have odd textures and distortions. The GAN-generated images have more visible artifacts (blurriness, unnatural blending, or strange textures), while diffusion models produce smoother and more natural results. GANs sometimes suffer from "mode collapse," generating similar-looking faces, whereas diffusion models tend to have a better distribution of facial diversity.

3.3 + 3.4 GAN FID + Diffusion FID

Model	FID Score
Diffusion	65.442
GAN	46.230

GAN demonstrates lower FID than diffusion model which implies GAN captures the distribution better than the latter. However, this doesn't necessarily mean GAN images are better qualitatively from human perception. FID primarily measures statistical similarity between generated and real image distributions, not perceptual quality. Diffusion models, despite having higher FID, often produce images with finer details and fewer artifacts, which can appear more realistic or visually appealing to human observers.

```
import os
import torch
import torchvision.transforms as transforms
from PIL import Image
from torch_fidelity import calculate_metrics

def resize_images(folder_path, size=(64, 64)):
    """Resizes all images in a folder to the specified size."""
    transform = transforms.Compose([
        transforms.Resize(size),
        transforms.ToTensor()
    ])

    for filename in os.listdir(folder_path):
        img_path = os.path.join(folder_path, filename)
        try:
            with Image.open(img_path) as img:
                img = img.convert("RGB")
```

```
                img_resized = transform(img)
                img_resized = transforms.ToPILImage()(img_resized)
                img_resized.save(img_path)
        except Exception as e:
            print(f"Error processing {img_path}: {e}")

def compute_fid(real_images_path, fake_images_path):
    metrics = calculate_metrics(
        input1=real_images_path,
        input2=fake_images_path,
        fid=True,  # Compute FID
        verbose=False
    )
    return metrics['frechet_inception_distance']

if __name__ == "__main__":
    real_images_folder = "./training_images"
    diffusion_images_folder = "./diffusion_images"
    gan_images_folder = "./gan_images"

    # Resize diffusion images before computing FID
    resize_images(diffusion_images_folder, size=(64, 64))

    fid_diffusion = compute_fid(real_images_folder, diffusion_images_folder)
    fid_gan = compute_fid(real_images_folder, gan_images_folder)

    print("+---------------------------+------------------+")
    print("| Model                     | FID Score        |")
    print("+---------------------------+------------------+")
    print(f"| Diffusion                 | {fid_diffusion:.3f}            |")
    print(f"| GAN                       | {fid_gan:.3f}           |")
    print("+---------------------------+------------------+")
```

4. Finetune GAN

```
# Borrowed from Sushant Gautam (gan-beginner-tutorial-on-celeba-dataset)
# Modified by me (Ayan Biswas Pranta) for this homework

import torch
import torch.nn as nn
from torch.utils.data import DataLoader
```

```
from torchvision.utils import make_grid, save_image
from torchvision.datasets import ImageFolder
import torchvision.transforms as T
import matplotlib.pyplot as plt
from IPython.display import Image
import cv2
import os
from tqdm.notebook import tqdm
import torch.nn.functional as F
import random


# Update the training dataset directory for fine-tuning
data_directory = './dataset_for_finetuning/'


image_size = 64
batch_size = 128
mean_std_for_norm = (0.5, 0.5, 0.5), (0.5, 0.5, 0.5)  # mean, std for normalize images


train_dataset = ImageFolder(root=data_directory, transform=T.Compose([
    T.Resize(image_size),
    T.CenterCrop(image_size),
    T.ToTensor(),
    T.Normalize(*mean_std_for_norm)
]))
train_dataloader = DataLoader(train_dataset, batch_size, shuffle=True, num_workers=0,
pin_memory=True)


def denormalization(img_tensors):
    return img_tensors * mean_std_for_norm[1][0] + mean_std_for_norm[0][0]


def display_images(images, nmax=64):
    fig, ax = plt.subplots(figsize=(8,8))
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(make_grid(denormalization(images.detach()[:nmax]), nrow=8).permute(1, 2, 0))


def to_device(data, device):
```

```
    """Move tensor(s) to chosen device"""
    if isinstance(data, (list, tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)


class DeviceDataLoader():
    """Wrap a dataloader to move data to a device"""
    def __init__(self, dl, device):
        self.dl = dl
        self.device = device

    def __iter__(self):
        """Yield a batch of data after moving it to device"""
        for b in self.dl:
            yield to_device(b, self.device)

    def __len__(self):
        """Number of batches"""
        return len(self.dl)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
train_dataloader = DeviceDataLoader(train_dataloader, device)

# Define the discriminator
discriminator = nn.Sequential(
    nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(64),
    nn.LeakyReLU(0.2, inplace=True),

    nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(128),
    nn.LeakyReLU(0.2, inplace=True),

    nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.LeakyReLU(0.2, inplace=True),

    nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(512),
    nn.LeakyReLU(0.2, inplace=True),
```

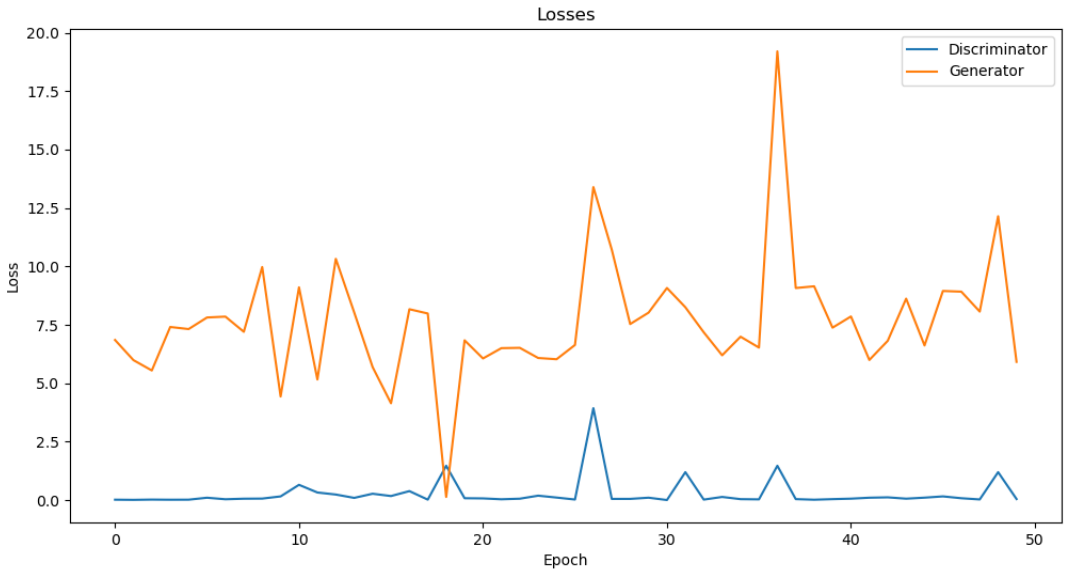
```
    nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0, bias=False),
    nn.Flatten(),
    nn.Sigmoid()
)


discriminator = to_device(discriminator, device)

generator_input_size = 128

# Define the generator
generator = nn.Sequential(
    nn.ConvTranspose2d(generator_input_size, 512, kernel_size=4, stride=1, padding=0,
bias=False),
    nn.BatchNorm2d(512),
    nn.ReLU(True),

    nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.ReLU(True),

    nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(128),
    nn.ReLU(True),

    nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(64),
    nn.ReLU(True),

    nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1, bias=False),
    nn.Tanh()
)
generator = to_device(generator, device)


# Load the pre-trained model checkpoints
discriminator.load_state_dict(torch.load('Discriminator_saved_model.pth'))
generator.load_state_dict(torch.load('Generator_saved_model.pth'))


def train_discriminator(real_images, opt_d):
    opt_d.zero_grad()
```

```
    real_preds = discriminator(real_images)
    real_targets = torch.ones(real_images.size(0), 1, device=device)
    real_loss = F.binary_cross_entropy(real_preds, real_targets)

    starter_noise_for_training = torch.randn(batch_size, generator_input_size, 1, 1,
device=device)
    fake_images = generator(starter_noise_for_training)
    fake_targets = torch.zeros(fake_images.size(0), 1, device=device)
    fake_preds = discriminator(fake_images)
    fake_loss = F.binary_cross_entropy(fake_preds, fake_targets)

    loss = real_loss + fake_loss
    loss.backward()
    opt_d.step()
    return loss.item()


def train_generator(opt_g):
    opt_g.zero_grad()
    starter_noise = torch.randn(batch_size, generator_input_size, 1, 1, device=device)
    fake_images = generator(starter_noise)
    preds = discriminator(fake_images)
    targets = torch.ones(batch_size, 1, device=device)
    loss = F.binary_cross_entropy(preds, targets)
    loss.backward()
    opt_g.step()
    return loss.item()


# Directory to store checkpoints and losses plot
sample_dir = 'generated'
os.makedirs(sample_dir, exist_ok=True)


# Modify the training function to use 50 epochs
def main_training(epochs, lr):
    torch.cuda.empty_cache()
    losses_generator = []
    losses_discriminator = []
    optimizer_discriminator = torch.optim.Adam(discriminator.parameters(), lr=lr,
betas=(0.5, 0.999))
    optimizer_generator = torch.optim.Adam(generator.parameters(), lr=lr, betas=(0.5,
```

```
0.999))

    for epoch in range(epochs):
        for real_images, _ in tqdm(train_dataloader, disable=True):
            loss_discriminator = train_discriminator(real_images, optimizer_discriminator)
            loss_generator = train_generator(optimizer_generator)
        losses_generator.append(loss_generator)
        losses_discriminator.append(loss_discriminator)
        print("Epoch {}/{}, Generator Loss: {:.4f}, Discriminator Loss: {:.4f}".format(
            epoch+1, epochs, loss_generator, loss_discriminator))

    return losses_generator, losses_discriminator

# Hyperparameters: set learning rate and 50 epochs for fine-tuning
lr = 0.00025
epochs = 50

losses_generator, losses_discriminator = main_training(epochs, lr)

# Save the fine-tuned model checkpoints
torch.save(generator.state_dict(), 'generated_saved_model_finetuned.pth')
torch.save(discriminator.state_dict(), 'discriminator_saved_model_finetuned.pth')

plt.figure(figsize=(12,6))
plt.plot(losses_discriminator, '-')
plt.plot(losses_generator, '-')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend(['Discriminator', 'Generator'])
plt.title('Losses')
plt.savefig('losses_plot_finetuning.png')
plt.show()


# ------------------------------
# Final image generation: generate 1024 images
# ------------------------------

final_dir = 'generated_finetuned'
os.makedirs(final_dir, exist_ok=True)

final_noise = torch.randn(1024, generator_input_size, 1, 1, device=device)
```

```
final_images = generator(final_noise)
final_images = denormalization(final_images)

for i in range(final_images.shape[0]):
    filename = os.path.join(final_dir, f"generated_image_{i:04d}.png")
    save_image(final_images[i], filename)


# Select 16 random images from the 1024 generated images
random_images = random.sample(range(1024), 16)
selected_images = torch.stack([final_images[i] for i in random_images])  # Convert list to
tensor
selected_images = selected_images.cpu()  # Move to CPU if necessary


# Create a 4x4 grid of images
plt.figure(figsize=(8, 8))
plt.axis("off")
plt.imshow(make_grid(selected_images, nrow=4).permute(1, 2, 0).cpu().numpy())  # Ensure
numpy conversion
plt.title("Randomly Selected 4x4 Generated Images")
plt.show()
```

4.1 Training curves



After resuming training from where it ended for GAN, it didn't improve anymore. Discriminator is very dominant with very low loss where generator is having a hard time fooling the discriminator. This can happen for two reasons:

1. The number of diffusion generated images (which are used as training data for fine-tuning) is only 1024, where original GAN was trained with 10000 training samples.
2. Diffusion generated images are fake, they aren't as good as celebA dataset.

4.2 Visual observations of Fine-tuned GAN against GAN

From my Qualitative observation, originally trained GAN looks better than fine-tuned GAN. My assumption is, diffusion generated images aren't good as training data because at the end of the day they are fake, they aren't better training sample than the original celebrity images (celebA dataset). Therefore, with further training with diffusion generated images, the result of fine-tuned GAN gets worse.

4.3 Visual observations of Fine-tuned GAN against Diffusion

Diffusion generated images are far better than fine-tuned GAN. The Fine-tuned GAN-generated faces have odd textures and distortions. They have more visible artifacts (blurriness, unnatural blending, or strange textures), sometimes suffer from "mode collapse," generating similar-looking faces. On the other hand, the diffusion model images appear sharper, with more refined details. They are generally more realistic, with more natural skin textures, lighting, and symmetry.

4.4 Display 4x4 (16 images) images generated by finetuned GAN

Model	FID Score
Diffusion	65.442
GAN	46.230
GAN (fine-tuned)	46.230

5. Bonus

5.1 Late

The homework is submitted before the deadline.