ECE60146: Homework 8

Manish Kumar Krishne Gowda, 0033682812

(Spring 2024)

1 Introduction

The objective of this homework is to compare the performance of two different approaches for generative modeling of image datasets: Generative Adversarial Networks (GANs) and diffusion-based models. In this assignment, we are provided with a subset of the well-known CelebA dataset, consisting of facial images. Our task involves implementing a GAN from scratch to generate fake images using this dataset. Additionally, we utilized a pre-trained diffusion-based mode to generate fake images.

Diffusion-based modeling is computationally intensive, hence we will focus on generating fake images using the pre-trained model provided to us. The evaluation of the generated images will be done quantitatively using the Fréchet Inception Distance (FID), which measures the similarity between the distributions of real and generated images based on features extracted from an Inception network.

2 Methodology

Like the task instructions suggested, I began by learning about Transpose Convolution, Generative Adversarial Networks (GANs), and diffusion-based models.

Transpose convolution concept helps us to make images bigger while keeping them clear and detailed. It's like zooming in on a picture without making it blurry, i.e. it allows us to upsample feature maps in neural networks. This operation plays a significant role in the architecture of GANs, particularly in the generator network, where it aids in generating high-resolution images from low-dimensional latent vectors.

Generative Adversarial Networks, or GANs for short. are like a game between two players: a creator (i.e. the generator in deep learning parlance) and a critic (i.e. the creator in deep learning parlance). The creator tries to make fake images that look real, while the critic tries to spot which images are fake. They both get better over time, which helps the creator make more realistic images. The two neural networks, namely the generator and discriminator are trained in a competitive fashion. Through adversarial training, GANs have demonstrated remarkable capabilities in generating realistic synthetic data, including images, audio, and text.

Lastly, I looked into diffusion-based models. These models work differently than GANs but also help in making new images. They do this by spreading a kind of "noise" in a picture until it starts to look like a real image.

2.1 Programming Tasks — GAN

I downloaded a subset of the CelebA dataset provided through BrightSpace for this homework. To ensure consistency with the *set_dataloader* method in the *AdversarialLearning* module, which requires images to be organized within their respective class folders, I moved all the images into a folder named "0". Although the task did not require classification based on class, I organized all images within a single class folder to align with the coding structure.

Like discussed in Section I, the training process for Discriminator and Generator networks in the context of GANs involves maximizing and minimizing specific expectations, respectively.

The Discriminator aims to correctly classify input images as real or fake, while the Generator aims to produce images that deceive the Discriminator. This optimization process is formulated as a min-max game, where the networks compete against each other.

To translate this min-max optimization into a training protocol, we update the parameters of the Discriminator and Generator networks separately for each training batch. For the Discriminator, we use a binary cross-entropy loss function to maximize its ability to classify real and fake images correctly. For the Generator, we minimize the loss associated with the Discriminator's output on generated images to improve its ability to generate realistic images.

Deep Convolutional Generative Adversarial Network (DCGAN) implementations, such as the DiscriminatorDG1 and GeneratorDG1 in the AdversarialLearning module of the DLStudio, follow this training protocol. DCGANs are based on fully convolutional networks and use specific layer configurations, such as "4-2-1" parameters, for both Discriminator and Generator networks. The Discriminator network uses strided convolutions with 4x4 kernels and 2x2 strides, while the Generator transforms random noise vectors into realistic-looking images.

Following is the Discriminator-Generator code used for the programming task. The code is leveraged from the DLStudio module [1]. The Discriminator consists of several convolutional layers (nn.Conv2d) followed by batch normalization layers (nn.BatchNorm2d) and leaky ReLU activation functions. The final(Output) layer produces a scalar value for each input image by passing through a sigmoid activation function (nn.Sigmoid). Similar to the Discriminator, the Generator also follows a 4-2-1 topology. The Generator starts with a latent vector and uses transpose convolutional layers (nn.ConvTranspose2d) to upsample the vector and generate an image. The final layer produces an output image with pixel values in the range [-1, 1] using the hyperbolic tangent activation function (nn.Tanh). The Discriminator outputs a probability indicating the likelihood of an input image being real (using BCE Loss), while the Generator produces synthetic images from random noise vectors.

```
        class DiscriminatorDG1(nn.Module):
            def __init__(self):
                super(AdversarialLearning.DataModeling.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, inplace=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(AdversarialLearning.DataModeling.GeneratorDG1, self).
                                                        __init__()
                self.latent_to_image = nn.ConvTranspose2d( 100,   512,
                                                        kernel_size=4, stride=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
```

The Discriminator contains in total 16 layers and about 2.7M (2766529) parameters, while the generator contains 13 layers and about 3.5M (3576704) parameters

```
from DLStudio import *
from AdversarialLearning import *
dls = DLStudio(
                  dataroot = "/content/drive/MyDrive/hw8_data/celeba_dataset_64x64
                                                         ",
                  image_size = [64,64],
                  path_saved_model = "/content/drive/MyDrive/hw8_data/saved_models
                                                         ",
                  learning_rate = 1e-4,       ## <==  try smaller value if mode
                                                         collapse
#                  learning_rate = 5e-3,       ## <==  try smaller value if mode
                                    collapse
                  epochs = 30,
```

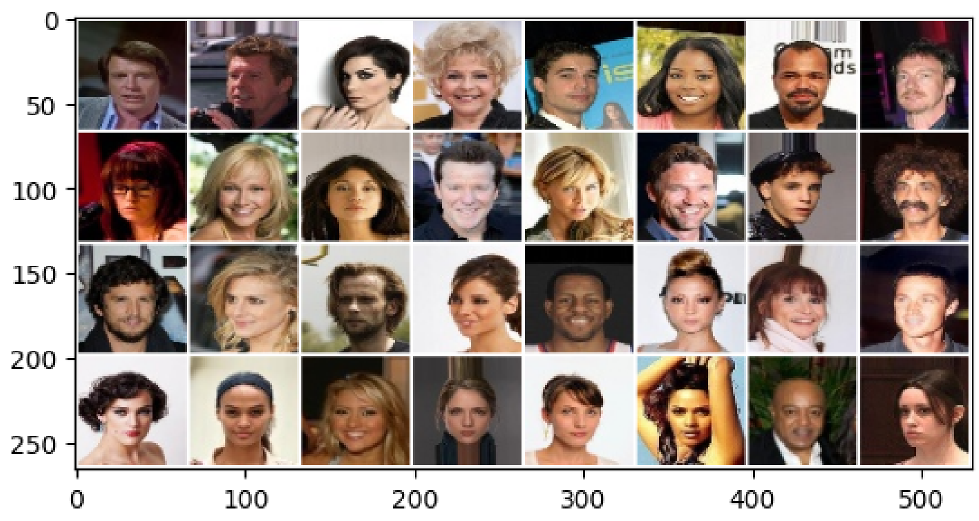

Figure 1: A batch of images from the training dataset

```
                  batch_size = 32,
                  use_gpu = True,
              )

adversarial = AdversarialLearning(
                  dlstudio = dls,
                  ngpu = 1,
                  latent_vector_size = 100,
                  beta1 = 0.5,                          ## for the Adam optimizer
              )

dcgan =   AdversarialLearning.DataModeling( dlstudio = dls, adversarial =
                                           adversarial )


discriminator =  dcgan.DiscriminatorDG1()
generator =  dcgan.GeneratorDG1()
dcgan.set_dataloader()
dcgan.show_sample_images_from_dataset(dls)
dcgan.run_gan_code(dls, adversarial, discriminator=discriminator, generator=
                                           generator, results_dir="results_DG1")
```

DLStudio is instantiated for the purpose of training the GAN. During training, the GAN undergoes optimization over 30 epochs, with each epoch comprising multiple iterations of passing batches of 32 images through the network. The learning process involves updating the parameters of both networks iteratively using the Adam optimizer, with a learning rate of 1e-4. The latent vectors fed into the Generator have a size of 100, and the beta1 parameter of the Adam optimizer is set to 0.5. The training loop involves alternating between updating the parameters of the Discriminator and the Generator networks.

A batch of 32 images is shown in Figure 1. The training loss is shown in Figure 2. Throughout training, two key metrics are monitored: the mean Discriminator loss (mean_D_loss) and the mean Generator loss (mean_G_loss). These metrics provide insights into how well the GAN is learning to generate realistic images.

In the initial epochs, both the Discriminator and Generator losses fluctuate, indicating the

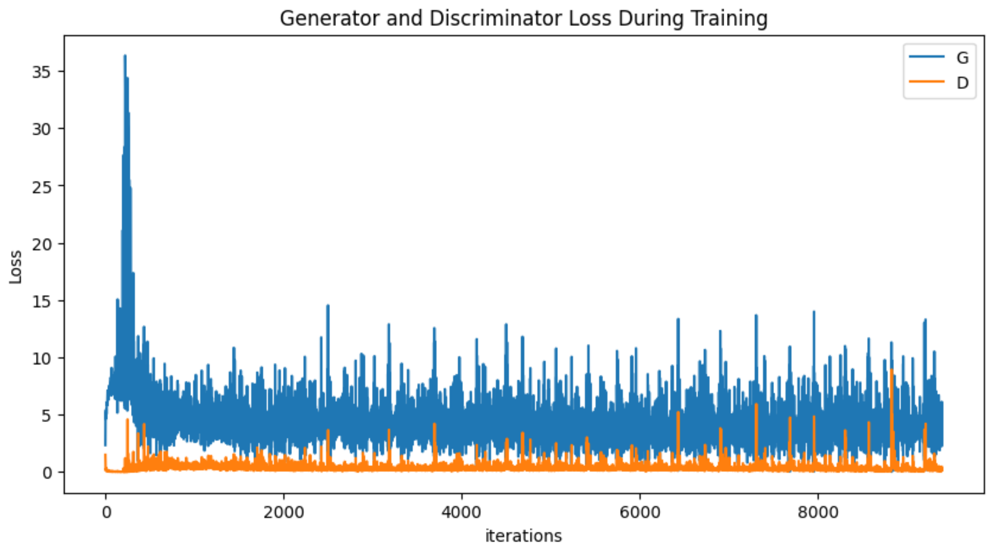

Figure 2: Training Loss GAN

network is still learning. However, as training progresses, both losses stabilize, with the Discriminator loss generally increasing and the Generator loss generally decreasing. This trend suggests that the Generator is gradually improving its ability to generate realistic images, while the Discriminator becomes more adept at distinguishing between real and fake images. This is a typical pattern of GAN training.

Further, a function *generate_fake_images* generates fake images using a given generator model and saves them to the specified directory.Inside the function, a loop iterates 2048 times to generate the same number of fake images. For each iteration, a random noise vector of shape (1, 100, 1, 1) is sampled from a normal distribution. This noise vector is then passed to the generator to produce a fake image. After generating a fake image, it is detached from the computational graph (using .detach()) and moved to the CPU (using .cpu()) to be saved as an image file.

```
import os
import torch
import torchvision

def generate_fake_images(generator, output_dir, num_images):
    os.makedirs(output_dir, exist_ok=True)  # Ensure the output directory exists

    with torch.no_grad():
        for i in range(num_images):
            try:
                # Generate a random noise vector
                noise = torch.randn(1, 100, 1, 1, device=device)

                # Generate a fake image from the noise vector
                fake_image = generator(noise).detach().cpu()

                # Save the fake image
                fake_image_path = os.path.join(output_dir, f"fake_image_{i}.png")
                torchvision.utils.save_image(fake_image, fake_image_path)

                # Print progress
                if (i + 1) % 100 == 0:
                    print(f"Generated {i + 1} fake images.")

            except Exception as e:
                print(f"Error generating image {i + 1}: {str(e)}")

    print(f"All {num_images} fake images saved at: {output_dir}")

# Load the state dictionary of the generator
generator_state_dict = torch.load("/content/drive/MyDrive/hw8_data/saved_models/
                                        results_DG1generator.pt")
generator.load_state_dict(generator_state_dict)

# Define parameters
output_directory = "results_DG1/fake_images"
num_fake_images = 2048

# Generate and save fake images
generate_fake_images(generator, output_directory, num_fake_images)
```

2.1.1 Qualitative Evaluation of GAN

A 4x4 sample of generated fake and trained real Images is shown in Figure 3 and 4

Upon visual inspection, several differences and similarities between the original and fake images are evident.

Similarities

1. **General Appearance :** Both the original and fake images share some similarities in terms of overall composition and structure. The fake images capture the general appearance of human faces, including facial proportions and basic features.

2. **Color Palette :** The color palette used in the fake images is similar to that of the original images, with variations in skin tones, hair colors, and background hues. This helps in creating a semblance of realism in the generated images.

3. **Hair Region :** Although not as detailed as in the original images, the hair region in the fake images shows some resemblance to real hair, with discernible strands and shading.

Differences

1. **Realism :** The original images appear more realistic compared to the fake ones. Real images capture finer details and nuances of human faces, such as skin texture, facial features, and expressions, which are lacking in the generated images.

2. **Detailing :** The original images exhibit sharper edges and clearer features, whereas the fake images have blurry edges and lack finer details. This is particularly noticeable in and around the eyes.

3. **Quality:** While the fake images resemble human faces to some extent, they have a painterly quality to them. The lack of clarity and detail gives them a more artistic or stylized appearance rather than a realistic one.

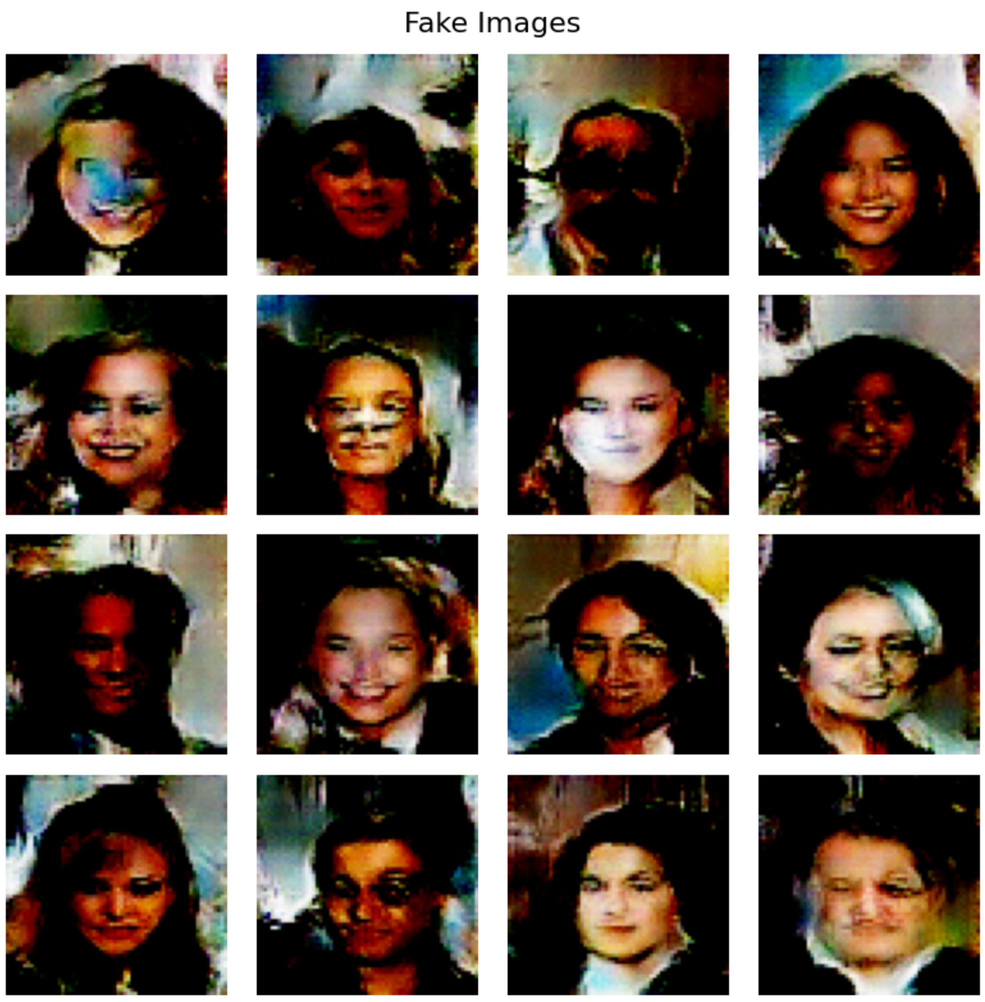

Figure 3: 4x4 Sample of Generated Fake Images

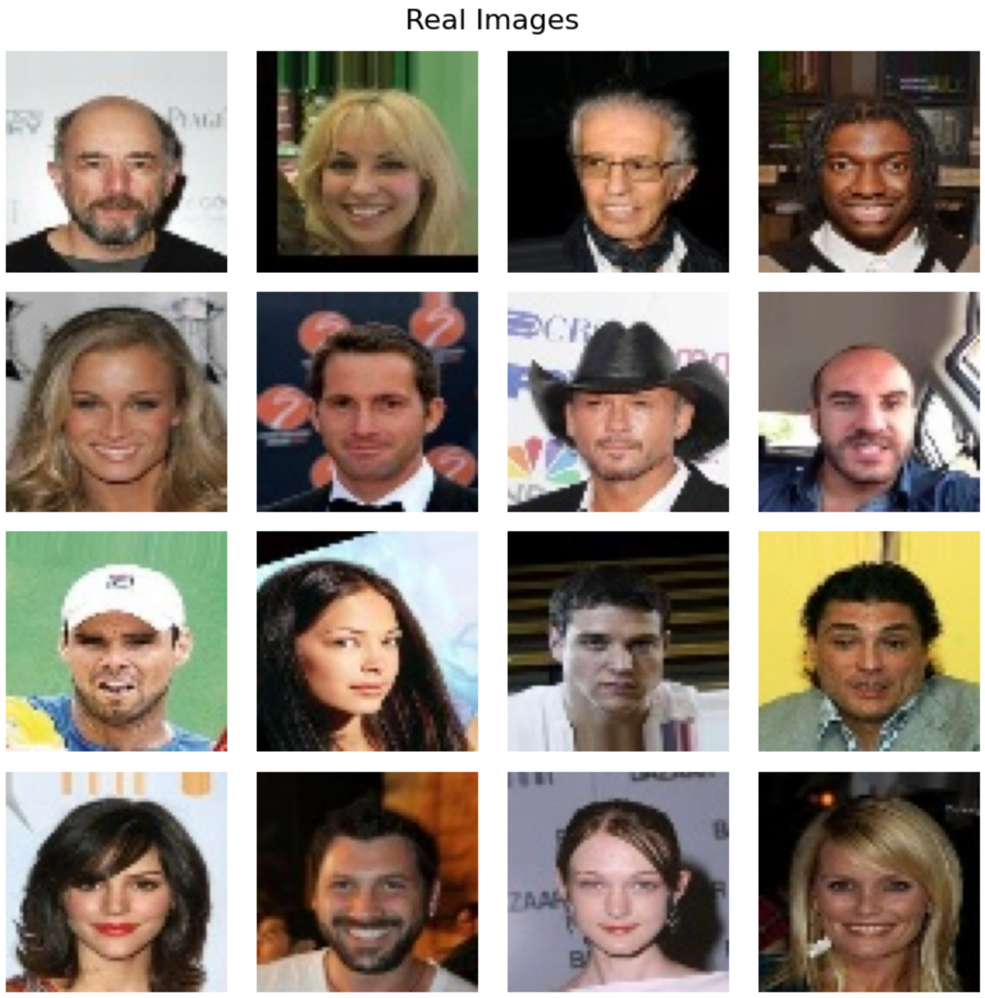

Figure 4: 4x4 Sample of Trained Real Images

In summary, the fake images generated by the BCE-GAN show promise but also highlight areas for improvement. While they do resemble human faces to some extent, the lack of fine detail and clarity diminishes their realism. The blurry edges, particularly around facial features like eyes, contribute to a less convincing appearance. However, the results are encouraging as the generated images exhibit recognizable facial features and capture some aspects of human likeness. With further training and optimization, addressing issues like blurry edges and lack of detail, the BCE-GAN has the potential to generate more realistic and convincing images in the future.

2.1.2 Quantitative Evaluation of GAN

In the quantitative evaluation of our BCE-GAN model's performance, as indicated in the task description, we employed the Fréchet Inception Distance (FID) metric, a widely accepted measure for assessing the quality of generated images. The FID score quantifies the similarity between the distributions of feature vectors extracted from real and generated images. To calculate the FID score, we utilized a pre-trained InceptionV3 network to encode both real and fake images into feature vectors. These feature vectors were then used to model the distribution of feature representations for each set of images as multivariate Gaussian distributions. The FID score is computed as the Fréchet distance between these two multivariate Gaussian distributions.

For our experiment, we computed the FID score using a set of real images obtained from the CelebA dataset and a set of fake images generated by our BCE-GAN model. After extracting activation statistics, namely mean and covariance, from both sets of images, we computed the FID score using the calculated statistics. The resulting FID score we obtained was 105.16.

Interpreting the FID score, a higher value suggests a greater discrepancy between the distributions of real and fake images, indicating poorer image quality and less realism in the generated images. Conversely, a lower FID score signifies a closer match between the distributions of real and fake images, indicating higher image quality and greater realism in the generated images. Therefore, our obtained FID score of 105.16 indicates a significant gap between the distributions of real and fake images, suggesting room for improvement in our BCE-GAN model's image generation capabilities.

2.2 Programming Tasks — Diffusion

The module in DLStudio dedicated to Denoising Diffusion tackles an alternative approach to generative data modeling. This method involves employing two Markov processes: one gradually transforms a training image into pure noise, while the other starts with Gaussian isotropic noise and incrementally denoises it until it resembles images from the training data.

The two Markov chains are denoted as the p-chain and the q-chain. The p-chain progressively denoises isotropic Gaussian noise, while the q-chain progressively applies diffusion to an image until it becomes noise. It's crucial to understand that temporal progression in the chains operates differently: while the q-chain progresses from timestep 0 to T, the p-chain progresses from timestep T to 0.

In practice, isotropic Gaussian noise is commonly used due to its property that multiple consecutive transitions in a Markov chain can be combined into a single calculation. Each training cycle starts by acquiring an image from the dataset, randomly choosing a timestep in the forward q-chain, and estimating the q-chain transition probability based on Bayes' Rule. The amount of denoising in the reverse p-chain is determined by a neural network, which estimates the "exact" opposite of the diffusion extent in the q-chain.

To train the neural network, a timestep is randomly chosen, and a cumulative q-chain transition equivalent to t consecutive transitions is applied to the input image. The posterior probabilities obtained from Bayes' Rule serve as targets for training the neural network to estimate the reverse p-chain transition probability. Various loss functions can be used for training, such as KL-Divergence or MSE error.

The primary objective of training the neural network is to predict the denoising transition probabilities accurately. Training typically occurs in an infinite loop, with checkpoints saved periodically. To witness the image generation capability of a checkpoint, isotropic Gaussian noise is used as input and subjected to all T timestep p-chain transitions to generate a recognizable image.

In our diffusion task, we utilized the diffusion.pt file provided to us, with DLStudio version 2.4.3

```
try:
    # Load the pre-trained model state dictionary
    network.load_state_dict(torch.load(model_path))

    # Move the model to the appropriate device
    network.to(top_level.device)

    # Set the model to evaluation mode
    network.eval()

    print("Starting Sampling...")

    # Initialize a list to store all sampled images
    all_images = []

    # Loop until the desired number of samples is reached
    while len(all_images) * top_level.batch_size_image_generation < top_level.
                                             num_samples:
        # Generate samples using the diffusion process
        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,
        )

        # Scale the sample to [0, 255] and convert to uint8
        sample = ((sample + 1) * 127.5).clamp(0, 255).to(torch.uint8)

        # Permute the dimensions to match the expected image format
        sample = sample.permute(0, 2, 3, 1).contiguous()

        # Store the samples in a list
        all_images.extend([sample.cpu().numpy() for sample in [sample]])

        # Print progress
        print("Generated {} out of {} samples...".format(len(all_images) *
                                                      top_level.
                                                      batch_size_image_generation,
                                                      top_level.num_samples))

    # Concatenate all sampled images and ensure the total number of samples
                                             matches the desired number
    arr = np.concatenate(all_images, axis=0)[:top_level.num_samples]

    # Convert the shape to a string for file naming
    shape_str = "x".join([str(x) for x in arr.shape])
    # Define the output path for the sampled images
    out_path = os.path.join(top_level.path_saved_model , f"samples_{shape_str}.npz"
                                         )

    # Save the sampled images
    np.savez(out_path , arr)

    print("Image generation completed successfully.")

except Exception as e:
    print(f"An error occurred during image generation: {e}")
```

The code defines a GaussianDiffusion object and a UNetModel object for diffusion and network modeling, respectively. These objects are then used to create a GenerativeDiffusion object.With 1000 diffusion timesteps (T) and an image size of 64x64, the script loads the weights of the pre-trained model and sets parameters for image generation. Sampling of new images is conducted using the diffusion process and the trained network, with the generated images processed and saved accordingly. Initially 1048 samples are generated, each batch consisting of 8 images. But FID calculation ran into the issue of "large imaginary component" which necissated atleast 2048 images, So, later 2048 samples were generated. Finally, the script completes the image generation process and saves the resulting samples in the specified directory.

2.2.1 Qualitative and Quantitative Evaluation of Diffusion

A 4x4 images randomly sampled from the 2048 gererated images from the diffusion model is shown in Figure 5 Upon qualitative analysis, it's evident that the images generated by the diffusion model surpass those produced by the GAN in terms of realism and natural appearance. Unlike the GAN-generated images, which sometimes resemble paintings with blurry edges and lack of detail, the diffusion model outputs exhibit a higher degree of clarity and authenticity. The diffusion model effectively captures intricate features such as skin tone and facial expressions, resulting in more lifelike renderings. While occasional instances of blurriness are present, the overall quality of the diffusion model images is commendable. The significantly lower FID score of 49.07 further corroborates the superiority of the diffusion model over the GAN approach. To further enhance the diffusion model's performance, fine-tuning parameters such as the number of diffusion timesteps and network architecture could be explored. Additionally, incorporating advanced techniques for noise modeling and denoising may contribute to refining the generated images, ultimately achieving even greater realism and clarity.

The diffusion model could be better than traditional methods like GANs for making images. It works by gradually cleaning up random noise until it looks like a real picture. This method helps keep more details and make images look more natural. Also, it's good at understanding how images change over time, which means it can make clearer pictures with sharper edges. Plus, tests show that diffusion models often make better images compared to GANs. So, overall, diffusion models seem promising for making high-quality, realistic pictures

3 Conclusion

In conclusion, both GANs and diffusion models offer unique approaches to generating realistic images. While GANs excel in capturing high-level image features, diffusion models stand out in preserving finer details and producing more natural-looking images. However, both methods have

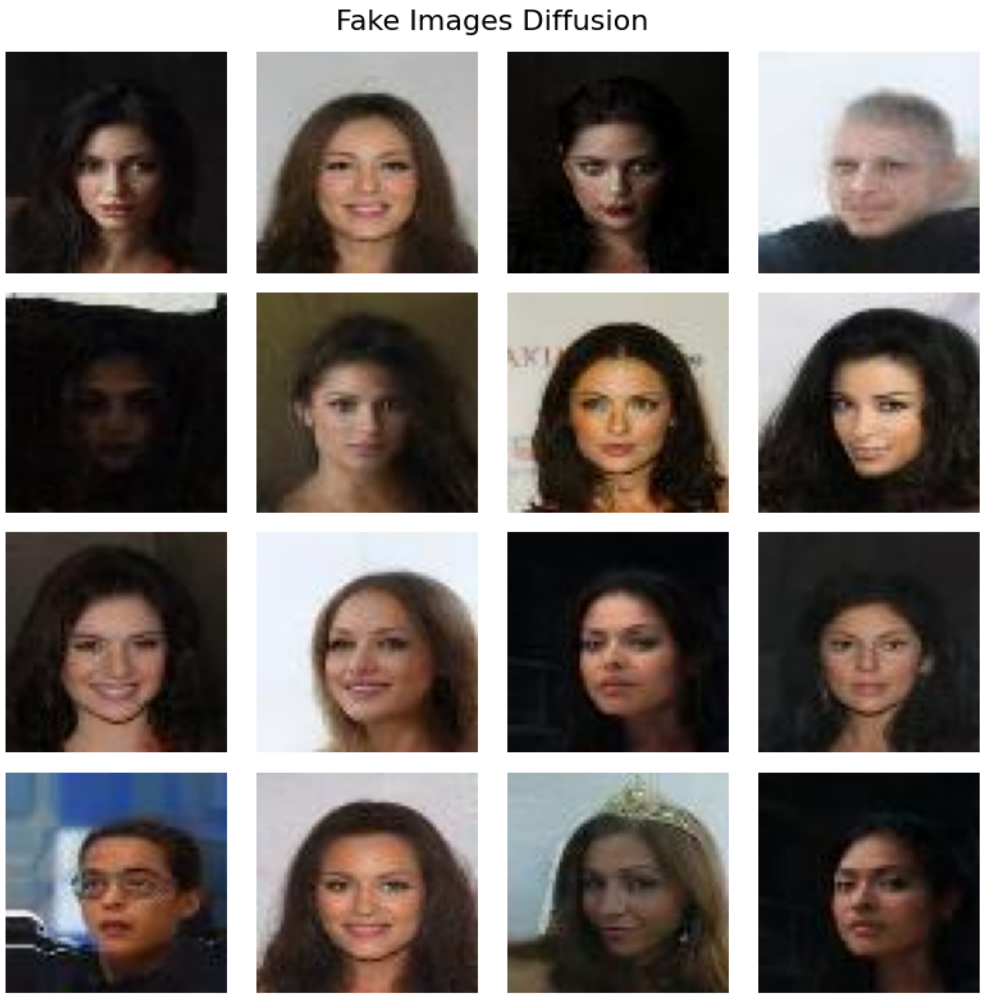

Figure 5: 4x4 Sample of Generated Fake Images from Diffusion model

their strengths and limitations. GANs may suffer from mode collapse and produce artifacts, while diffusion models require significant computational resources. Therefore, the choice between GANs and diffusion models depends on specific requirements and preferences. Nonetheless, advancements in both techniques continue to push the boundaries of generative image modeling, offering promising avenues for future research and applications

References

[1] URL `https://engineering.purdue.edu/kak/distDLS/`