BME646 and ECE60146
Homework 9
Spring 2025
Malleswari Kachireddy

Introduction

Generative Models Overview

Generative models are a class of ML methods designed to learn the underlying probability distribution of the dataset, so that new, realistic data samples can be generated. These models not only learn the underlying distribution but also have a framework to generate new synthetic data.
Two predominant generative models are

- Generative Adversarial Networks (GANs) and
- Diffusion Models.

Generative Adversarial Networks (GANs) Network

GAN models consists of two neural networks—the Generator and the Discriminator. The Generator transforms random noise into synthetic data samples, while the Discriminator evaluates these samples against real data. The Generator aims to fool the Discriminator, and the Discriminator strives to accurately distinguish real from fake. In this way, both networks iteratively improve. This training relies on transpose convolution and loss functions like binary cross-entropy to produce high resolution images. Transpose convolution takes a compact, low-resolution feature map and learns how to expand it into a high-resolution image, filling in details as needed.

Diffusion Models:

Diffusion models generate data by reversing a process that gradually adds noise to the training images. Initially, an image is progressively corrupted until it turns into pure noise. The model is then trained to denoise the image step by step, effectively learning how to reconstruct the original data from a noisy input. This process leverages Markov chains and noise schedules, providing a more explicit modeling of the data generation process compared to GANs.

1. Generative Adversarial Networks
   1.1. Model

Our model network uses the DiscriminatorDG1,2 classes and GeneratorDG1,2 classes.

DiscriminatorDG1 class implements the discriminator of a DCGAN using a so-called "4-2-1" topology. This means that the network is designed with layers that progressively reduce the spatial dimensions of the input image through a series of strided convolutions while increasing feature

depth, using Batch Normalization and Leaky ReLU activations to ensure stable and effective learning.

The GeneratorDG1 class is responsible for creating realistic images starting from a random noise. It uses a series of transpose convolution (or deconvolution) layers to upsample a low-dimensional noise vector into a full-sized 64×64 image , applying Batch Normalization and ReLU activations, with a final Tanh activation to produce output images in the desired pixel range.

These GeneratorDG1 and DiscriminatorDG1 architecture forms the basis of a DCGAN, where the generator and discriminator are trained in an adversarial manner to create realistic images.

GeneratorDG2 is identical to GeneratorDG1 whereas DiscriminatorDG2 has one additional convolutional layer.

```
# This code is copied from DLStudio Adversarial Learning by Prof. Avinash Kak.
# I trained two different DC-GAN models (DCGAN1 and DCGAN2)

from DLStudio import DLStudio

import sys,os,os.path
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as tvt
import torchvision.transforms.functional as tvtF
import torch.optim as optim
import numpy as np
import math
import random
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time
import glob
import imageio

#_____________________________  AdversarialLearning Class Definition  _____________________________
```

```
class AdversarialLearning(object):

    def __init__(self, *args, **kwargs ):
        if args:
            raise ValueError(
                  '''AdversarialLearning constructor can only be called with keyword arguments for the following
                     keywords: epochs, learning_rate, batch_size, momentum, image_size, dataroot, path_saved_model,
                     use_gpu, latent_vector_size, ngpu, dlstudio, device, LAMBDA, clipping_threshold, and beta1''')
        allowed_keys = 'dataroot','image_size','path_saved_model','momentum','learning_rate','epochs','batch_size', \
                       'classes','use_gpu','latent_vector_size','ngpu','dlstudio', 'beta1', 'LAMBDA', 'clipping_threshold'
        keywords_used = kwargs.keys()
        for keyword in keywords_used:
            if keyword not in allowed_keys:
                raise SyntaxError(keyword + ":  Wrong keyword used --- check spelling")
        learning_rate = epochs = batch_size = convo_layers_config = momentum = None
        image_size = fc_layers_config = dataroot =  path_saved_model = classes = use_gpu = None
        latent_vector_size = ngpu = beta1 = LAMBDA = clipping_threshold = None
        if 'latent_vector_size' in kwargs           :   latent_vector_size = kwargs.pop('latent_vector_size')
        if 'ngpu' in kwargs                         :   ngpu  = kwargs.pop('ngpu')
        if 'dlstudio' in kwargs                     :   dlstudio  = kwargs.pop('dlstudio')
        if 'beta1' in kwargs                        :   beta1  = kwargs.pop('beta1')
        if 'LAMBDA' in kwargs                       :   LAMBDA  = kwargs.pop('LAMBDA')
        if 'clipping_threshold' in kwargs           :   clipping_threshold = kwargs.pop('clipping_threshold')
        if latent_vector_size:
            self.latent_vector_size = latent_vector_size
        if ngpu:
            self.ngpu = ngpu
        if dlstudio:
            self.dlstudio = dlstudio
        if beta1:
            self.beta1 = beta1
        if LAMBDA:
            self.LAMBDA = LAMBDA
        if clipping_threshold:
            self.clipping_threshold = clipping_threshold
```

```
    def show_sample_images_from_dataset(self, dlstudio):
        data = next(iter(self.train_dataloader))
        real_batch = data[0]
        self.dlstudio.display_tensor_as_image(torchvision.utils.make_grid(real_batch, padding=2, pad_value=1,
normalize=True))

    def set_dataloader(self):
        dataset = torchvision.datasets.ImageFolder(root=self.dlstudio.dataroot,
                    transform = tvt.Compose([
                                    tvt.Resize(self.dlstudio.image_size),
                                    tvt.CenterCrop(self.dlstudio.image_size),
                                    tvt.ToTensor(),
                                    tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                    ]))
        self.train_dataloader = torch.utils.data.DataLoader(dataset, batch_size=self.dlstudio.batch_size,
                                                            shuffle=True, num_workers=2)

    def weights_init(self,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)

    ######################################   Discriminator-Generator DG1
#######################################
    class DiscriminatorDG1(nn.Module):
        """
        This is an implementation of the DCGAN Discriminator. I refer to the DCGAN network topology as
        the 4-2-1 network.  Each layer of the Discriminator network carries out a strided
        convolution with a 4x4 kernel, a 2x2 stride and a 1x1 padding for all but the final
        layer. The output of the final convolutional layer is pushed through a sigmoid to yield
        a scalar value as the final output for each image in a batch.
```

```
    Class Path:  AdversarialLearning  ->  DiscriminatorDG1
    """
    def __init__(self):
        super(AdversarialLearning.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):
    """
    This is an implementation of the DCGAN Generator. As was the case with the Discriminator network,
    you again see the 4-2-1 topology here.  A Generator's job is to transform a random noise
    vector into an image that is supposed to look like it came from the training dataset. (We refer
    to the images constructed from noise vectors in this manner as fakes.)  As you will see later
    in the "run_gan_code()" method, the starting noise vector is a 1x1 image with 100 channels.  In
    order to output 64x64 output images, the network shown below use the Transpose Convolution
    operator nn.ConvTranspose2d with a stride of 2.  If (H_in, W_in) are the height and the width
    of the image at the input to a nn.ConvTranspose2d layer and (H_out, W_out) the same at the
    output, the size pairs are related by
```

```
            H_out   =   (H_in - 1) * s   +   k   -   2 * p
            W_out   =   (W_in - 1) * s   +   k   -   2 * p

    were s is the stride and k the size of the kernel.  (I am assuming square strides, kernels, and
    padding). Therefore, each nn.ConvTranspose2d layer shown below doubles the size of the input.

    Class Path:  AdversarialLearning  ->   GeneratorDG1
    """
    def __init__(self):
        super(AdversarialLearning.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
  ########################################  DG1 Definition ENDS
##########################################

  ####################################  Discriminator-Generator DG2
####################################
```

```
class DiscriminatorDG2(nn.Module):
    """
    This is essentially the same network as the DCGAN for DG1, except for the extra layer
    "self.extra" shown below.  We also declare a batchnorm for this extra layer in the form
    of "self.bnX".  In the implementation of "forward()", we invoke the extra layer at the
    beginning of the network.

    Class Path:  AdversarialLearning  ->   DiscriminatorDG2
    """
    def __init__(self, skip_connections=True, depth=16):
        super(AdversarialLearning.DiscriminatorDG2, self).__init__()
        self.conv_in = nn.Conv2d(  3,    64,     kernel_size=4,     stride=2,    padding=1)
        self.extra =   nn.Conv2d(  64,   64,     kernel_size=4,     stride=1,    padding=2)
        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.bnX  = nn.BatchNorm2d(64)
        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.bnX(self.extra(x))
        x = torch.nn.functional.leaky_relu(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 GeneratorDG2(nn.Module):
```

```
        """
        The Generator for DG2 is exactly the same as for the DG1.  So please the comment block for that
        Generator.

        Class Path:  AdversarialLearning  ->   GeneratorDG2
        """
        def __init__(self):
            super(AdversarialLearning.GeneratorDG2, 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
    ##########################################   DG2 Definition ENDS
##########################################



##########################################################################################################
################
    ##  The training routines follow, first for a GAN constructed using either the DG1 and or the DG2
```

```
    ##  Discriminator-Generator Networks, and then for a WGAN constructed using either the CG1 or the CG2
    ##  Critic-Generator Networks.

##########################################################################################################
###############

    def run_gan_code(self, dlstudio, discriminator, generator, results_dir):
        """
        This function is meant for training a Discriminator-Generator based Adversarial Network.
        The implementation shown uses several programming constructs from the "official" DCGAN
        implementations at the PyTorch website and at GitHub.

        Regarding how to set the parameters of this method, see the following script

                    dcgan_DG1.py

        in the "ExamplesAdversarialLearning" directory of the distribution.
        """
        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 Generator:
        nz = 100
        netD = discriminator.to(self.dlstudio.device)
        netG = generator.to(self.dlstudio.device)
        #  Initialize the parameters of the Discriminator and the Generator networks according to the
        #  definition of the "weights_init()" method:
        netD.apply(self.weights_init)
        netG.apply(self.weights_init)
```

```
        #  We will use a the same noise batch to periodically check on the progress made for the Generator:
        fixed_noise = torch.randn(self.dlstudio.batch_size, nz, 1, 1, device=self.dlstudio.device)
        #  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=dlstudio.learning_rate, betas=(self.beta1, 0.999))
        optimizerG = optim.Adam(netG.parameters(), lr=dlstudio.learning_rate, betas=(self.beta1, 0.999))
        #  Establish the criterion for measuring the loss at the output of the Discriminator 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",f"{dlstudio.epochs} and {len(self.train_dataloader)}")
        start_time = time.perf_counter()
        for epoch in range(dlstudio.epochs):
            g_losses_per_print_cycle = []
            d_losses_per_print_cycle = []
            # For each batch in the dataloader
            for i, data in enumerate(self.train_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 the beginning of this
                ##  file, the GAN training boils down to carrying out a min-max optimization. Each iterative
                ##  step of the max part results in updating the Discriminator parameters and each iterative
                ##  step of the min part results in the updating of the Generator parameters.  For each
                ##  batch of the training data, we first do max and then do min.  Since 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 using 1.0 as the target;
                ##  and, in the second part, we supply to the Discriminator the output of the Generator
                ##  and use 0 as the target. In what follows, the Discriminator is being applied to
                ##  the training images:
                netD.zero_grad()
                real_images_in_batch = data[0].to(self.dlstudio.device)
```

```
            #  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=self.dlstudio.device)
            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 beginning 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=self.dlstudio.device)
            fakes = netG(noise)
            label.fill_(fake_label)
            ##  The call to fakes.detach() in the next statement returns a copy of 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 computational graph.
            ##  The original 'fakes' tensor continues to remain in the computational 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)
            lossD_for_fakes.backward()
            ##  The following is just for displaying the losses:
            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 us 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 that we minimize
            ##  "1 - D(G(z))" which, since D is constrained to lie in the interval (0,1), requires that
            ##  we maximize D(G(z)).  We accomplish that by applying the Discriminator to the output
            ##  of the Generator and use 1 as the target for each image:
            netG.zero_grad()
```

```
            label.fill_(real_label)
            ##  The following forward prop will compute the partials wrt the discriminator params also, but
            ##  they will never get used for updating param vals for two reasons: (1) We call "step()" on
            ##  just optimizerG as shown later below; and (2) We call "netD.zero_grad()" at the beginning of
            ##  each training cycle.
            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_cycle))
                mean_G_loss = torch.mean(torch.FloatTensor(g_losses_per_print_cycle))
                print("[epoch=%d/%d   iter=%4d   elapsed_time=%5d secs]     mean_D_loss=%7.4f
mean_G_loss=%7.4f" %
                             ((epoch+1),dlstudio.epochs,(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 == dlstudio.epochs-1) and (i == len(self.train_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, fps=5)
        #  Make a side-by-side comparison of a batch-size sampling of real images drawn from the
        #  training data and what the Generator is capable of producing at the end of training:
        real_batch = next(iter(self.train_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(self.dlstudio.device),
                                        padding=1, pad_value=1, normalize=True).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()


import random
import numpy
import torch
import os, sys


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)



##  watch -d -n 0.5 nvidia-smi

from DLStudio import *
from AdversarialLearning import *

import sys

dls = DLStudio(
                dataroot = "/content/Supplementary/celeba_dataset_64x64",
#                 dataroot = "/mnt/cloudNAS3/Avi/ImageDatasets/PurdueShapes5GAN/multiobj/",
#                 dataroot = "/home/kak/ImageDatasets/PurdueShapes5GAN/multiobj/",
                image_size = [64,64],
                path_saved_model = "/content/DLStudio-
2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_DG1.pt",
                learning_rate = 1e-4,       ## <==  try smaller value if mode collapse
#                 learning_rate = 5e-3,      ## <==  try smaller value if mode collapse
                epochs = 30,
                batch_size = 32,
                use_gpu = True,
            )

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


discriminator =  dcgan.DiscriminatorDG1()
generator =  dcgan.GeneratorDG1()
```

```
num_learnable_params_disc = sum(p.numel() for p in discriminator.parameters() if p.requires_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_grad)
print("\nThe number of learnable parameters in the Generator: %d\n" % num_learnable_params_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)

dcgan.set_dataloader()

print("\n\nHere is one batch of images from the training dataset:")
dcgan.show_sample_images_from_dataset(dls)

dcgan.run_gan_code(dls, discriminator=discriminator, generator=generator, results_dir="results_DG1")

torch.save(generator.state_dict(), "/content/DLStudio-
2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_G1.pt")
torch.save(discriminator.state_dict(), "/content/DLStudio-
2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_D1.pt")



dls = DLStudio(
                dataroot = "/content/Supplementary/celeba_dataset_64x64",
#                 dataroot = "/mnt/cloudNAS3/Avi/ImageDatasets/PurdueShapes5GAN/multiobj/",
#                 dataroot = "/home/kak/ImageDatasets/PurdueShapes5GAN/multiobj/",

                image_size = [64,64],
                path_saved_model = "/content/DLStudio-
2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_DG2.pt",
                learning_rate = 2e-4,     ## <== try smaller value if mode collapse
                epochs = 30,
                batch_size = 32,
                use_gpu = True,
            )
```

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

discriminator =  dcgan.DiscriminatorDG2()
generator =  dcgan.GeneratorDG2()

num_learnable_params_disc = sum(p.numel() for p in discriminator.parameters() if p.requires_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_grad)
print("\nThe number of learnable parameters in the Generator: %d\n" % num_learnable_params_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)

dcgan.set_dataloader()

dcgan.show_sample_images_from_dataset(dls)

dcgan.run_gan_code(dls, discriminator=discriminator, generator=generator, results_dir="results_DG2")


torch.save(generator.state_dict(), "/content/DLStudio-
2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_G2.pt")
torch.save(discriminator.state_dict(), "/content/DLStudio-
2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_D2.pt")
```

1.2. Training curves: As is expected the generator loss is very high at the beginning and the discriminator loss increases as the generator becomes better at generating fake images that look realistic.

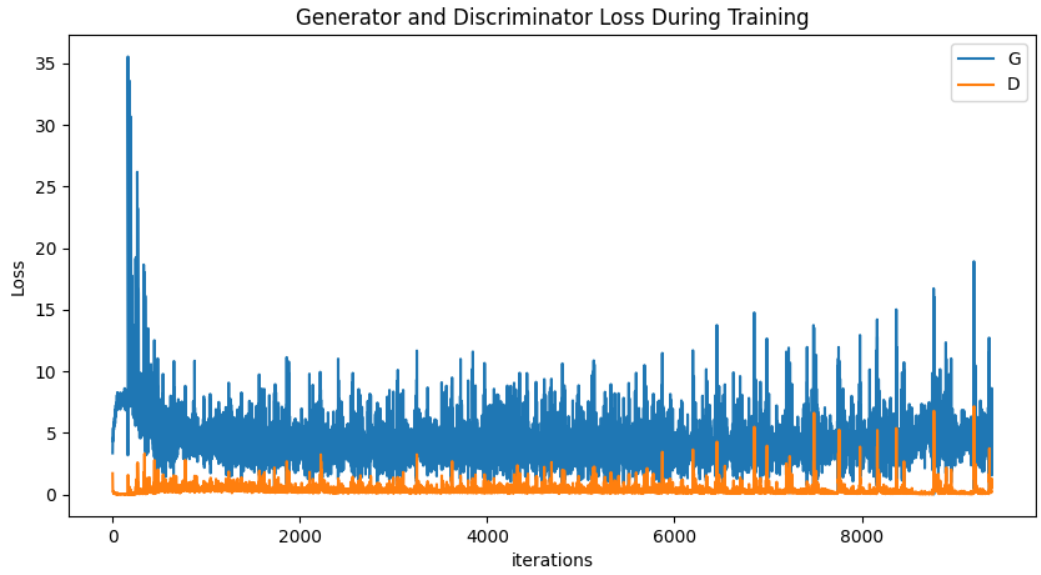

Figure 1: Training loss curve for DC GAN 1

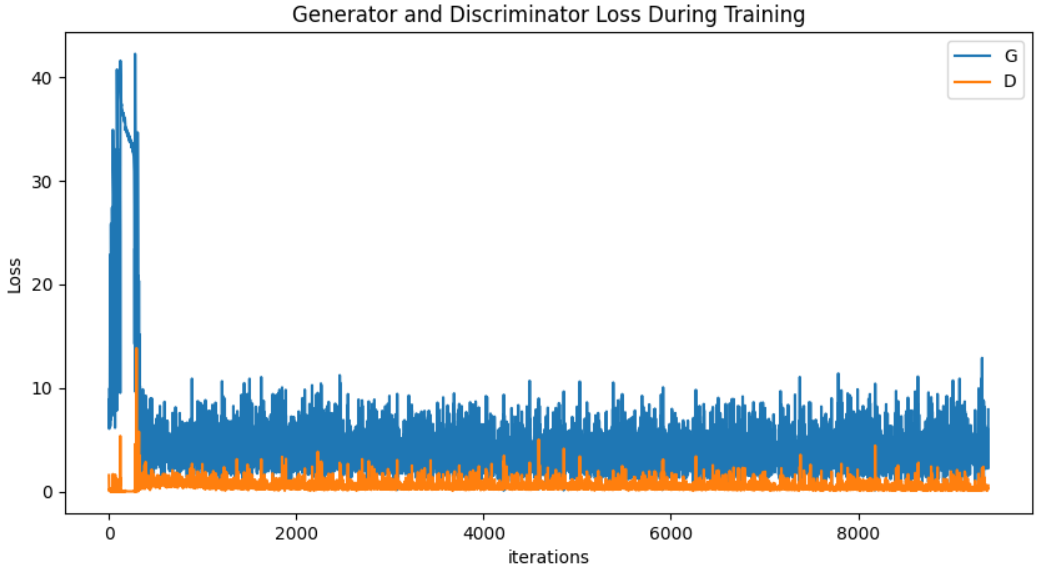

Figure 2: Training loss curve for DC GAN 2

2. Diffusion

   2.1.Generated images:

   Using the code and model weights provided by Prof. Kak, I generated and visualized 1024 images by calling GenerateNewImageSamples.py and VisualizeSamples.py.

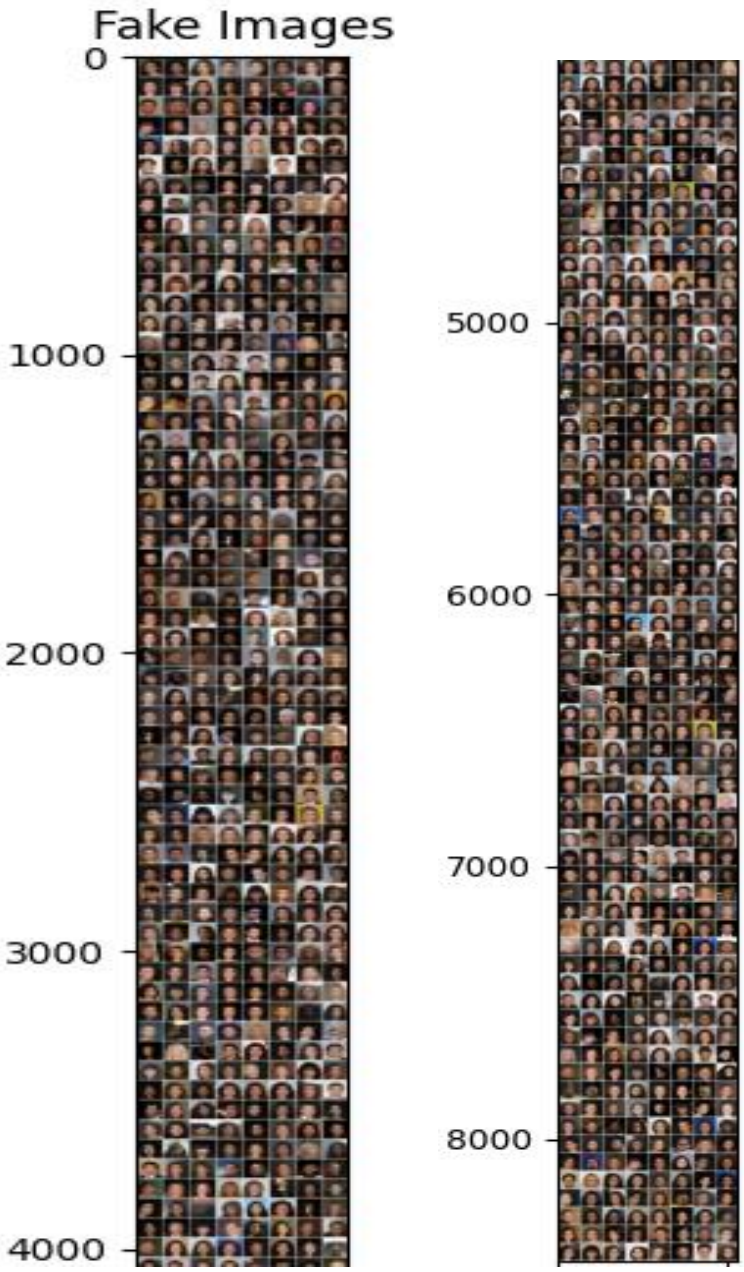

Figure 3: 1024 Fake Images generated by Diffusion model

Figure 4: Sample images generated by diffusion model

The outputs generated by diffusion model are very realistic. However, using an A100 GPU, it took more than 24 minutes to generate 1024 images using the code provided. However, generating 1024 images using the DC-GAN networks took a few seconds. So, the high quality of images comes at the cost of speed.

3. FID Score

Code to generate images from DC GAN generators:

```
def generate_DCGAN_images(output_dir, path_saved_model, dataroot, total_images, image_size, learning_rate,
epochs, batch_size, use_gpu, latent_vector_size, beta1):
  if not os.path.exists(output_dir):
      os.makedirs(output_dir)

  device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

  dlstudio =
DLStudio(dataroot=dataroot,image_size=image_size,path_saved_model=path_saved_model,learning_rate=learning_r
ate,epochs=epochs,batch_size=batch_size,use_gpu=use_gpu)
  adv = AdversarialLearning(dlstudio=dlstudio,ngpu = 1,latent_vector_size=latent_vector_size,beta1=beta1)

  netG = AdversarialLearning.GeneratorDG1()
  netG.load_state_dict(torch.load(adv.dlstudio.path_saved_model, map_location=device))
  netG.to(device)
  netG.eval()

  num_batches = total_images // adv.dlstudio.batch_size
  for batch_idx in range(num_batches):
      noise = torch.randn(adv.dlstudio.batch_size, adv.latent_vector_size, 1, 1, device=device)
      with torch.no_grad():
          fake_images = netG(noise)
      fake_images = (fake_images + 1) / 2.0
      for i in range(adv.dlstudio.batch_size):
          img_idx = batch_idx * adv.dlstudio.batch_size + i
          save_image(fake_images[i], f"{output_dir}/fake_image_{img_idx:04d}.png")
      print(f"Generated images {batch_idx * adv.dlstudio.batch_size} to {(batch_idx + 1) * adv.dlstudio.batch_size - 1}")

  print(f"Successfully generated {total_images} fake images in the '{output_dir}' directory")

  sample_fake_images = []
  for i in range(16):
      img_path = f"{output_dir}/fake_image_{i:04d}.png"
```

```
    img = Image.open(img_path)
    img_tensor = torchvision.transforms.ToTensor()(img)
    sample_fake_images.append(img_tensor)

grid = torchvision.utils.make_grid(sample_fake_images, nrow=4, padding=2, normalize=False)  # Changed nrow to 4
save_image(grid, f"{output_dir}/grid_4x4_sample.png")
print(f"Created a 4x4 grid visualization at '{output_dir}/grid_4x4_sample.png'")

from DLStudio import *
from AdversarialLearning import *

dataroot="/content/Supplementary/celeba_dataset_64x64"
total_images = 1024
image_size=[64,64]
learning_rate = 1e-4
epochs = 1
batch_size = 32
use_gpu = True
latent_vector_size=100
beta1=0.5

output_dir = "/content/DLStudio-2.5.2/ExamplesAdversarialLearning/results_DG1/images"
path_saved_model="/content/DLStudio-2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_G1.pt"
generate_DCGAN_images(output_dir, path_saved_model, dataroot, total_images, image_size, learning_rate, epochs,
batch_size, use_gpu, latent_vector_size, beta1)
shutil.move('/content/DLStudio-2.5.2/ExamplesAdversarialLearning/results_DG1/images/grid_4x4_sample.png',
        '/content/DLStudio-2.5.2/ExamplesAdversarialLearning/results_DG1/grid_4x4_sample.png')

output_dir = "/content/DLStudio-2.5.2/ExamplesAdversarialLearning/results_DG2/images"
path_saved_model="/content/DLStudio-2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_G2.pt"
generate_DCGAN_images(output_dir, path_saved_model, dataroot, total_images, image_size, learning_rate, epochs,
batch_size, use_gpu, latent_vector_size, beta1)
shutil.move('/content/DLStudio-2.5.2/ExamplesAdversarialLearning/results_DG2/images/grid_4x4_sample.png',
        '/content/DLStudio-2.5.2/ExamplesAdversarialLearning/results_DG2/grid_4x4_sample.png')
```

3.1.4x4 images generated by GAN

Figure 5: Sample images generated by DC-GAN1

Figure 6: Sample images generated by DC-GAN2

3.2.4x4 images generated by Diffusion

```
sample_fake_images = []
for i in range(16):
    img_path = f"/content/DLStudio-2.5.2/ExamplesDiffusion/visualize_samples/test_{i}.jpg"
    img = Image.open(img_path)
    img_tensor = torchvision.transforms.ToTensor()(img)
    sample_fake_images.append(img_tensor)

grid = torchvision.utils.make_grid(sample_fake_images, nrow=4, padding=2, normalize=False)  # Changed nrow to 4
save_image(grid, f"/content/DLStudio-2.5.2/ExamplesDiffusion/RESULTS/grid_4x4_sample.png")
print(f"Created a 4x4 grid visualization at '/content/DLStudio-
2.5.2/ExamplesDiffusion/RESULTS/grid_4x4_sample.png'")
```

Figure 7: Sample images generated by Diffusion Model

3.3.GAN FID

3.4.Diffusion FID

Model	FID Score
DC-GAN1	111.88
DC-GAN2	106.88
Diffusion	76.66

Table 1: FID Scores for various generative models

Code to compute FID score: I selected a random selection of 1024 images from the real paths to make sure the number of images is the same in both real and fake folders.

```
def calculate_fid_score(real_images_dir, fake_images_dir, match_count=True):
    real_paths = [os.path.join(real_images_dir, filename) for filename in os.listdir(real_images_dir)]
    fake_paths = [os.path.join(fake_images_dir, filename) for filename in os.listdir(fake_images_dir)]
    if match_count and len(real_paths) > len(fake_paths):
        real_paths = random.sample(real_paths, len(fake_paths))
    dims = 2048
    block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    model = InceptionV3([block_idx]).to(device)
    m1,s1 = calculate_activation_statistics(real_paths,model,batch_size=32,dims=dims,device=device) # Added
batch_size and dims parameters
    m2,s2 = calculate_activation_statistics(fake_paths,model,batch_size=32,dims=dims,device=device) # Added
batch_size and dims parameters
    fid_value = calculate_frechet_distance(m1,s1,m2,s2)
    return fid_value

real_images_dir = "/content/Supplementary/celeba_dataset_64x64/0"
fake_images_dir = "/content/DLStudio-2.5.2/ExamplesAdversarialLearning/results_DG1/images"
fid_value = calculate_fid_score(real_images_dir,fake_images_dir)
print(f"FID value: {fid_value:.2f}")

real_images_dir = "/content/Supplementary/celeba_dataset_64x64/0"
fake_images_dir = "/content/DLStudio-2.5.2/ExamplesAdversarialLearning/results_DG2/images"
fid_value = calculate_fid_score(real_images_dir,fake_images_dir)
```

```
print(f"FID value: {fid_value:.2f}")

real_images_dir = "/content/Supplementary/celeba_dataset_64x64/0"
fake_images_dir = "/content/DLStudio-2.5.2/ExamplesDiffusion/visualize_samples"
fid_value = calculate_fid_score(real_images_dir,fake_images_dir)
print(f"FID value: {fid_value:.2f}")
```

4. Finetuning

I commented out these two lines in run_gan_code:

```
        # netD.apply(self.weights_init)
        # netG.apply(self.weights_init)
```

Training code inspired by dcgan_DG1.py:

```
import random
import numpy
import torch
import os, sys

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)

from DLStudio import *
import sys

dls = DLStudio(
                dataroot = "/content/Supplementary/New_training_folder",
                image_size = [64,64],
                path_saved_model = "/content/DLStudio-
2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_DG1.pt",
                learning_rate = 1e-5,
```

```
                epochs = 50,
                batch_size = 8,
                use_gpu = True,
            )

dcgan = AdversarialLearning(dlstudio = dls,ngpu = 1,latent_vector_size = 100,beta1 = 0.5)

discriminator =  dcgan.DiscriminatorDG1()
generator =  dcgan.GeneratorDG1()

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

generator.load_state_dict(torch.load("/content/DLStudio-
2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_G1.pt", map_location=device))
generator.to(device)

discriminator.load_state_dict(torch.load("/content/DLStudio-
2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_D1.pt", map_location=device))
discriminator.to(device)

num_learnable_params_disc = sum(p.numel() for p in discriminator.parameters() if p.requires_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_grad)
print("\nThe number of learnable parameters in the Generator: %d\n" % num_learnable_params_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)

dcgan.set_dataloader()

print("\n\nHere is one batch of images from the fine-tuning dataset:")
dcgan.show_sample_images_from_dataset(dls)

dcgan.run_gan_code(dls, discriminator=discriminator, generator=generator, results_dir="results_finetuned_DG1")
```

```
torch.save(generator.state_dict(), "/content/DLStudio-
2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_finetuned_G1.pt")
torch.save(discriminator.state_dict(), "/content/DLStudio-
2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_finetuned_D1.pt")
```

4.1. Training loss curves

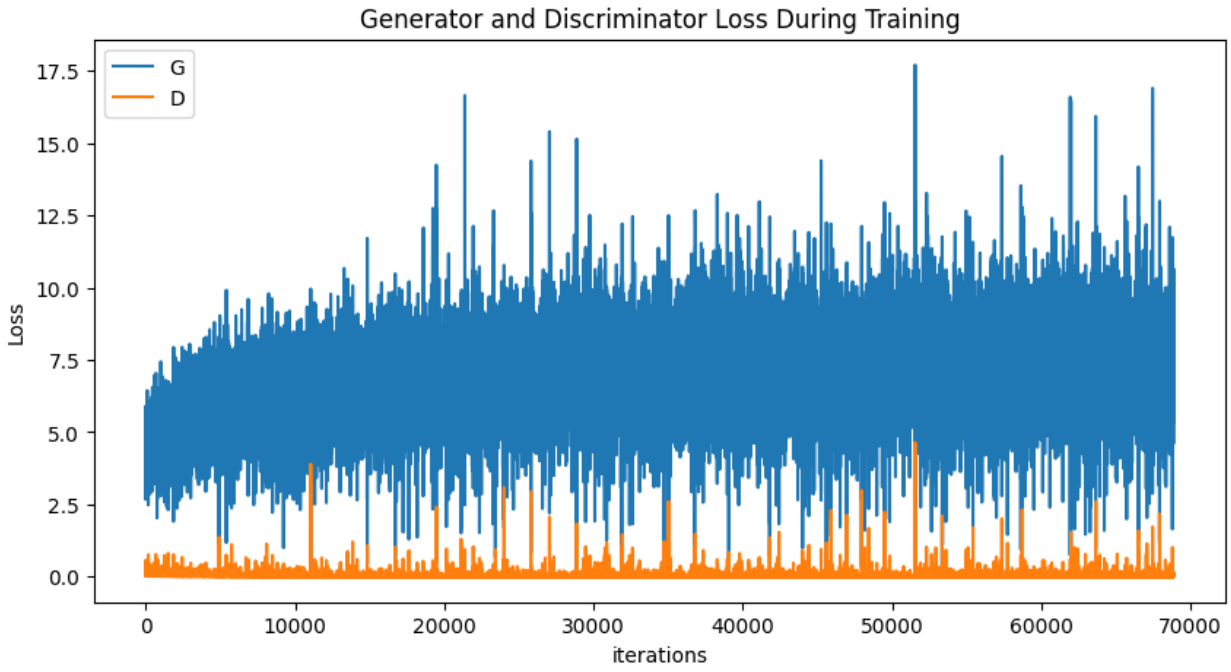

Figure 8: Training loss curve while finetuning DG-GAN1

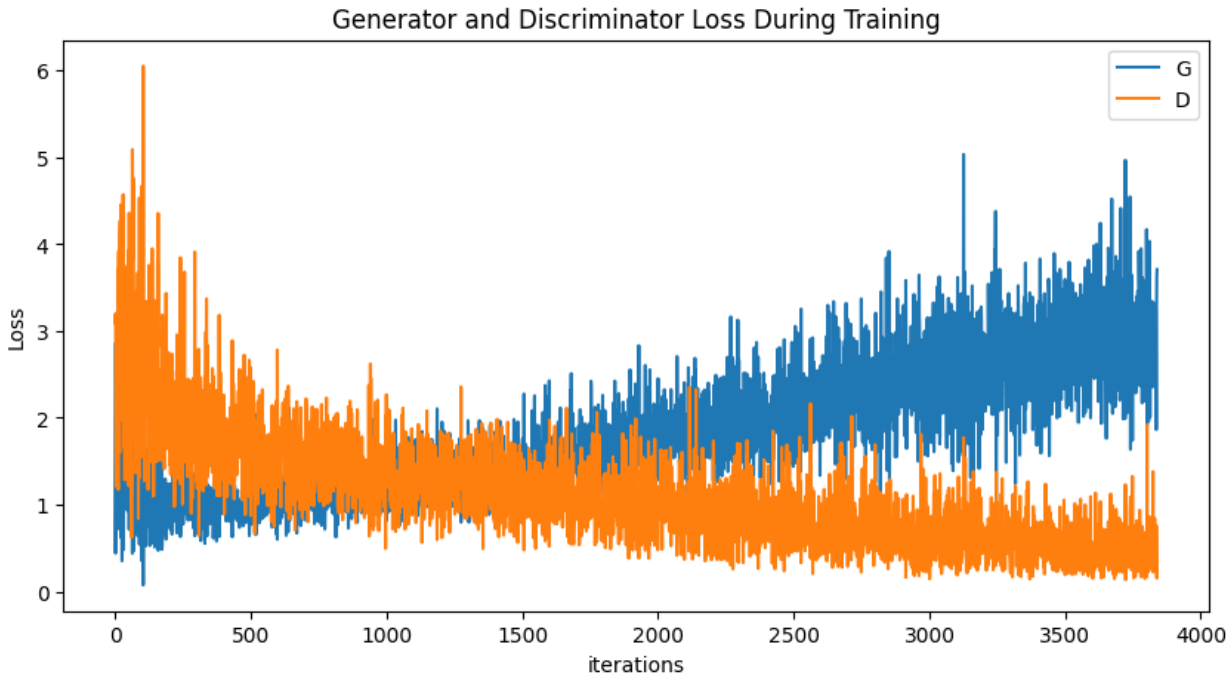

Figure 9: Training loss curve while finetuning DG-GAN2

4.2. Visual observations of finetuned GAN vs. GAN

Before Fine-tuning DC-GAN1	After Fine-tuning DC-GAN1

The images generated before fine-tuning had higher resolution, sharper details and faces were in similar proportions to real people. The lighting in most of the images were consistent with real scenes. The backgrounds are colored in somewhat strange but refined colors.	Images generated after fine-tuning have caricature like appearance, blurrier details. The lighting and background appear unnatural.

4.3. Visual observations of finetuned GAN vs diffusion model

	
Diffusion Model	**After Fine-tuning DC-GAN1**
Majority of the images are professional headshot like portraits with mostly young people's images generated predominantly. We can still notice weird artifacts in teeth and eyes in some of the images.	Images generated after fine-tuning have caricature like appearance, blurrier details. The lighting and background appear unnatural.

4.4.Finetuned GAN sample images

```
dataroot="/content/Supplementary/celeba_dataset_64x64"
```

```
total_images = 1024
image_size=[64,64]
learning_rate = 1e-4
epochs = 1
batch_size = 32
use_gpu = True
latent_vector_size=100
beta1=0.5

output_dir = "/content/DLStudio-2.5.2/ExamplesAdversarialLearning/finetuned_DG1/images"
path_saved_model="/content/DLStudio-2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_finetuned_G1.pt"
generate_DCGAN_images(output_dir, path_saved_model, dataroot, total_images, image_size, learning_rate, epochs,
batch_size, use_gpu, latent_vector_size, beta1)
shutil.move('/content/DLStudio-2.5.2/ExamplesAdversarialLearning/finetuned_DG1/images/grid_4x4_sample.png',
            '/content/DLStudio-2.5.2/ExamplesAdversarialLearning/results_finetuned_DG1/grid_4x4_sample.png')


output_dir = "/content/DLStudio-2.5.2/ExamplesAdversarialLearning/results_finetuned_DG2/images"
path_saved_model="/content/DLStudio-2.5.2/ExamplesAdversarialLearning/saved_model/dcgan_finetuned_G2.pt"
generate_DCGAN_images(output_dir, path_saved_model, dataroot, total_images, image_size, learning_rate, epochs,
batch_size, use_gpu, latent_vector_size, beta1)
shutil.move('/content/DLStudio-
2.5.2/ExamplesAdversarialLearning/results_finetuned_DG2/images/grid_4x4_sample.png',
            '/content/DLStudio-2.5.2/ExamplesAdversarialLearning/results_finetuned_DG2/grid_4x4_sample.png')
```

Figure 10: Sample images generated by finetuned DC-GAN 1

Figure 11: Sample images generated by finetuned DC-GAN2

4.5.FID Score

```
real_images_dir = "/content/Supplementary/celeba_dataset_64x64/0"
fake_images_dir = "/content/DLStudio-2.5.2/ExamplesAdversarialLearning/finetuned_DG1/images"
fid_value = calculate_fid_score(real_images_dir,fake_images_dir)
print(f"FID value: {fid_value:.2f}")

real_images_dir = "/content/Supplementary/celeba_dataset_64x64/0"
fake_images_dir = "/content/DLStudio-2.5.2/ExamplesAdversarialLearning/results_finetuned_DG2/images"
fid_value = calculate_fid_score(real_images_dir,fake_images_dir)
print(f"FID value: {fid_value:.2f}")
```

Model	FID Score (Before Finetuning)	FID Score (After Finetuning)
DC-GAN1	111.88	121.64
DC-GAN2	106.88	117.44
Diffusion	76.66	N/A

Table 2: FID Scores for various generative models before and after finetuning