ECE60146_HW8

April 7, 2026

Student Name: Mingqi Yang

1. Task 1: DCGAN Warm-Up on PurdueShapes5GAN

1.1 Running DCGAN (DG1)

In this task, we are asked to run **dcgan DG1.py** in DLStudio package on PurdueShapes5GAN dataset. The script trains an adversarial learning model with 30 epochs. After running the scripts, we should show the following results:

1. The loss curves of discriminator and generator
2. Comparison of real PurdueShapes5GAN images and images generated by the Generator
3. Generator output at different epochs

The script I used is shown below. It is basically the same as the **dcgan_DG1.py** in DLStudio package.

[ ]:
```
#!/usr/bin/env python

##  dcgan_DG1.py

"""
IMPORTANT NOTE:

    You will need to install the PurdueShapes5GAN dataset before you can␣
 ↪execute this script.

    Download the dataset archive

            datasets_for_AdversarialLearning.tar.gz

    through the link "Download the image dataset for AdversarialLearning"␣
 ↪provided at the top
    of the HTML version of the main module doc page and store it in the
    'ExamplesAdversarialLearning' directory of the distribution.  Subsequently,␣
 ↪execute the
    following command in that directory:

            tar zxvf datasets_for_AdversarialLearning.tar.gz
```

```
    This command will create a 'dataGAN' subdirectory and deposit the following␣
 ↪dataset archive
    in that subdirectory:

            PurdueShapes5GAN-20000.tar.gz

    Now execute the following in the "dataGAN" directory:

            tar zxvf PurdueShapes5GAN-20000.tar.gz

    With that, you should be able to execute the adversarial learning based␣
 ↪scripts in the
    'ExamplesAdversarialLearning' directory.

"""



"""
ABOUT THIS SCRIPT:

This script is based on the Discriminator-Generator pair DG1 in the␣
 ↪AdversarialLearning
class of the DLStudio module.

The DG1 pair is an implementation of DCGAN as presented in the paper␣
 ↪"Unsupervised
Representation Learning with Deep Convolutional Generative Adversarial␣
 ↪Networks" by
Radford et al.  DCGAN is short for "Deep Convolutional Generative Adversarial␣
 ↪Network".

The DCGAN neural networks are based on a "4-2-1" scheme for the different␣
 ↪layers in the
two networks.  For both the Generator and the Discriminator, the "4-2-1"␣
 ↪formula stands
for a kernel size of 4x4, a stride of 2x2, and a padding of 1x1.  In the␣
 ↪Discriminator
where the goal is to construct an abstraction pyramid that culminates in a␣
 ↪binary decision
about accepting an image as real or fake, 4x4 convolutional kernels are used in␣
 ↪each layer
of the Discriminator.  In the Generator, on the other hand, a kernel size of␣
 ↪4x4 refers to
```

```
the TransposeConvolutions in the different layers for upsampling needed for␣
 ↪generating an
image from a random noise vector.
"""

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 = "./dataGAN/PurdueShapes5GAN/multiobj/",
#                   dataroot = "/mmt/cloudNAS3/Avi/ImageDatasets/
 ↪PurdueShapes5GAN/multiobj/",
#                    dataroot = "/home/kak/ImageDatasets/PurdueShapes5GAN/
 ↪multiobj/",
                  image_size = [64,64],
                  path_saved_model = "./saved_model",
                  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")
```

The figures below show the deliverables:

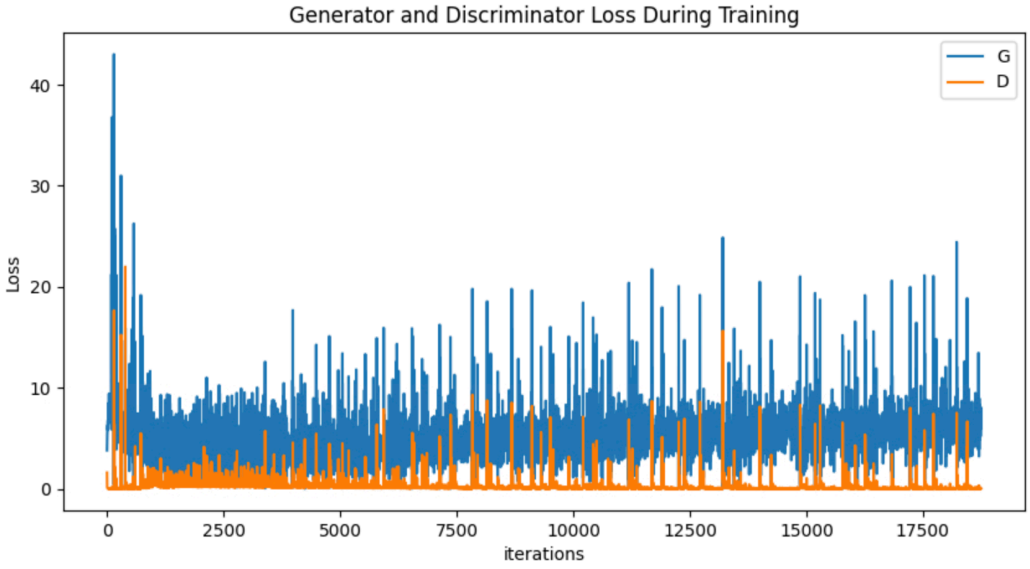


Figure 1: Discriminator and Generator loss curves for DCGAN

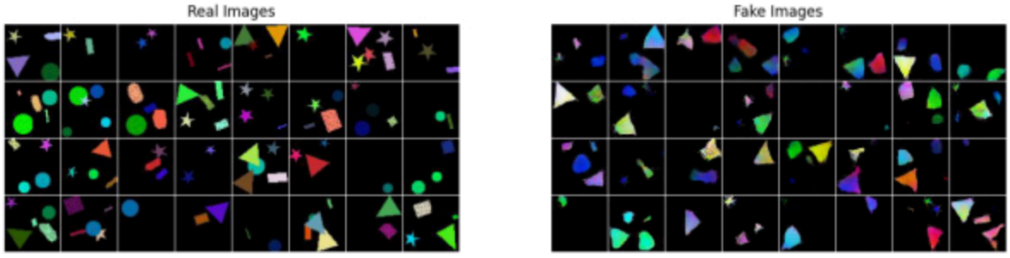
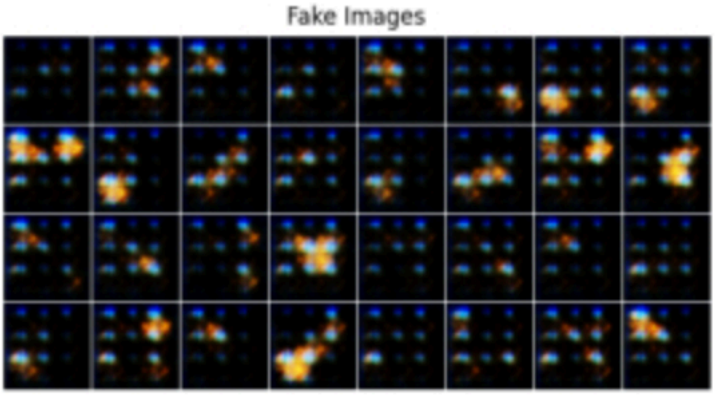
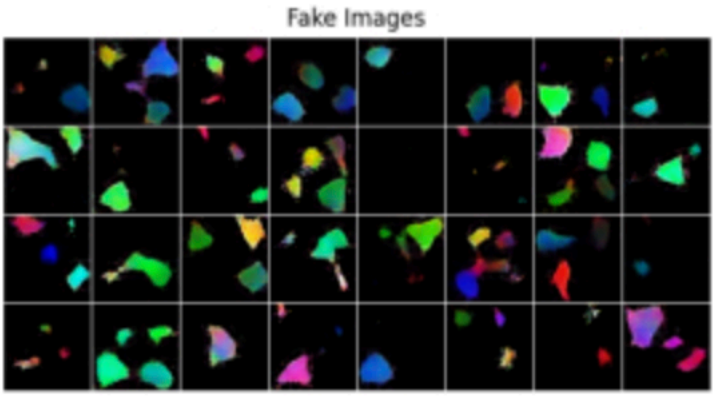


Figure 2: Comparison of real PurdueShapes5GAN images and images generated by the Generator


Fake Images


Fake Images

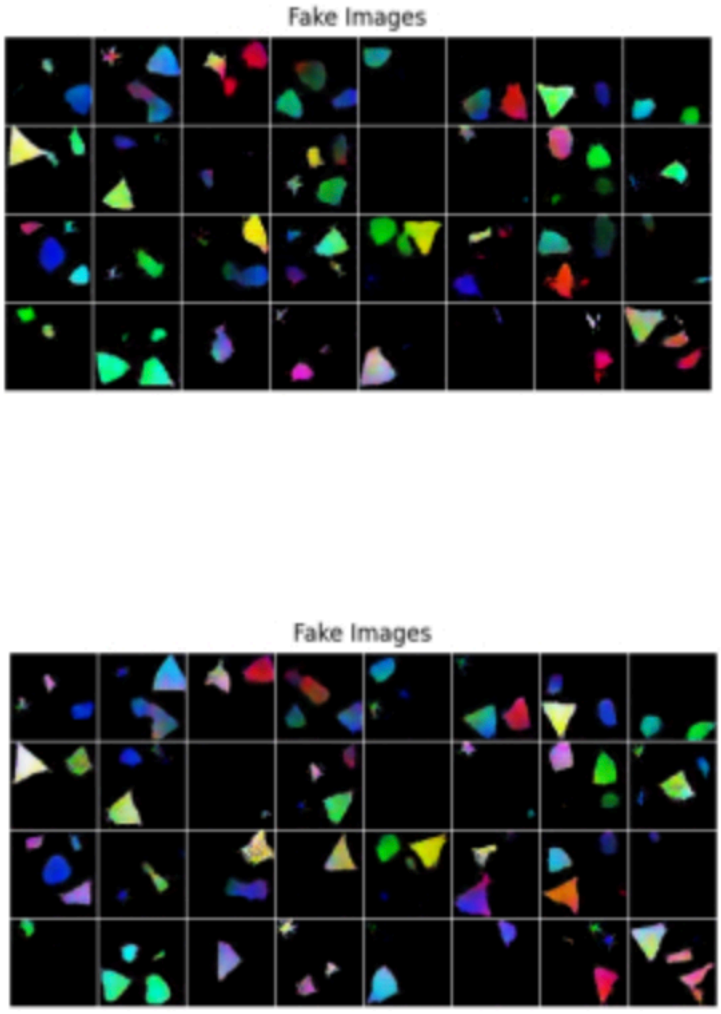


Figure 3: Generator output at epoch 1, 10, 20, 30 (from top to bottom)

3.1.2 Understanding the Architecture and Training

1. Generator Architecture

So, the generator starts with starts with a noisy image with size 1 * 1 * 100 and get a 64 * 64 * 3 image. With the formula $H_{out} = (H_{in} - 1) \times s + k - 2p$, the spatial dimensions changes as follows:

Input: 1 * 1

After 1st ConvTranspose2d layer: 4 * 4

After 2nd ConvTranspose2d layer: 8 * 8

After 3rd ConvTranspose2d layer: 16 * 16

After 4th ConvTranspose2d layer: 32 * 32

After last ConvTranspose2d layer: 64 * 64

The activation function used at the output is Tanh activation function. Tanh function maps the output values to the range [-1, 1], and this is important because the output now have the same intensity scale of the training images.

2. The minimax game

In GAN training, the discriminator and generator play a minimax game. Specifically, the discriminator tries to correctly distinguish real images from fake ones. However, the generator tries to cheat the discriminator. During the discriminator training, there are two phases:

1) When the input is a real image, the target is 1

2) When the input is a generator output, the target is 0

During the generator training, the discriminator is fixed, and the denerator tries to make its generated images look like real images. Therefore, the generator uses target of 1 for its outputs. The reason is that generator wants the discriminator to think about its outputs are real images.

3. The fakes.detach() call

The function fakes.detach() is used to stop gradients from flowing back into the generator when training the discriminator. Specifically, when we pass fakes.detach() into the Discriminator, it assumes the generated images as constant and does not change the generator's parameters. This is important because during discriminator training, we only want to update the discriminator, not the generator. Without fakes.detach(), the gradients from the discriminator loss would also go into the generator. This will update noth discriminator and generator, which is not what we want.

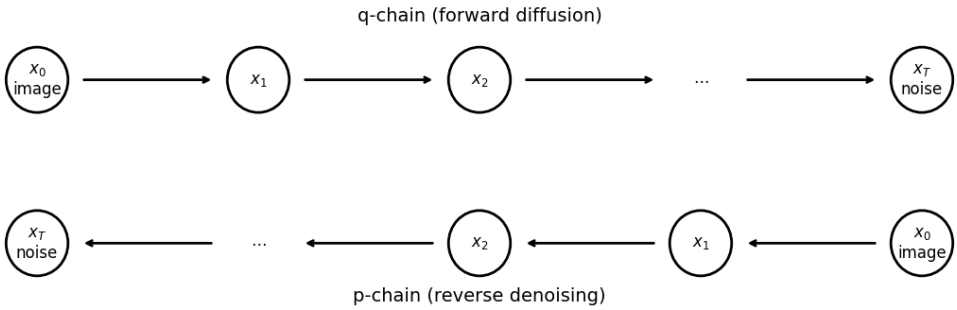
3.2 Task 2: Denoising Diffusion Warm-Up on PurdueShapes5GAN

3.2.1 Understanding the Forward and Reverse Processes

1. The two Markov chains

In diffusion models, there are two Markov chains: the q-chain and the p-chain. For the forward process q-chain, it takes a real image as input and gradually adds Gaussian noise over many steps. As the timestep t increases, the image becomes more and more noisy and finally become pure noise. So, the input of q-chain is clean image and the output of q-chain is noise. For the reverse process p-chain, it starts from pure noise and gradually removes noise step by step to recover a clean image. This process is learned by a denoiser, which predicts how to denoise at each timestep. So, the input of p-chain is noise and the output of p-chain is clean image.

The digram below shows both chains:



2. Forward transition probabilities

The forward transition probability describes how we go from x_{t-1} to x_t by adding noise at each step. It is defined as:

$$q(x_t \mid x_{t-1}) = \mathcal{N}(x_t;\ \sqrt{1-\beta_t}\, x_{t-1},\ \beta_t I)$$

This means that at each timestep t, we take the previous image x_{t-1}, scale it down by $\sqrt{1-\beta_t}$, and then add Gaussian noise with variance β_t. The parameter β_t is called the noise schedule. Specifically, it controls how much noise is added at each step. If β_t is small, only a small amount of noise is added; If β_t is large, more noise is added.

3. The "jump to any timestep" formula

The "jump to any timestep" formula is defined as:

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1-\bar{\alpha}_t}\, \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

This formula allows us to directly generate a noisy version x_t from the original image x_0 in one step without all intermediate steps from 1 to t. This is computationally useful because it avoids repeatedly applying the forward process step-by-step and reduce the computational time and cost.

In this formula, $\alpha_t = 1 - \beta_t$, where β_t is the noise schedule. $\bar{\alpha}_t$ is the product of all α's up to timestep t, which represents how much original signal remains after t steps.

4. What the neural network learns

At each training step, we randomly choose a timestep t and create a noisy image x_t from the original image x_0. The neural network is trained to predict the noise ϵ that was added to the image at that timestep. In this case, the input of the network is the noisy image x_t and timestep t and the output is the noise ϵ. The loss fucntion is then compare the true noise ϵ and the predicted noise $\hat{\epsilon}$.

3.2.2 Noise Schedule Computation and Forward Diffusion Visualization

In this task, we are asked to implement the $\alpha - \beta$ caluclations based on slides 169-171 of lecture notes. Based on the code template in the PDF, we set timesteps T = 100, $\beta_{start} = 0.0001$ and $\beta_{end} = 0.02$. Then, we need to compute the following identities:

1. sqrt_alphas_cumprod $\sqrt{\bar{\alpha}_t}$
2. sqrt_one_minus_alphas_cumprod $\sqrt{1-\bar{\alpha}_t}$
3. posterior_variance $\sigma_t^2 = \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}$
4. posterior_mean_coef1 coef1 $= \frac{\beta_t\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_t}$
5. posterior_mean_coef2 coef2 $= \frac{(1-\bar{\alpha}_{t-1})\sqrt{\alpha_t}}{1-\bar{\alpha}_t}$

The code implementation of compute_noise_scehdule is shown below. It is based on code template in the PDF with the remaining quantity computation.

[ ]:
```
def compute_noise_schedule(
    num_timesteps=1000,
    beta_start=0.0001,
    beta_end=0.02):
    """ Compute all alpha-beta quantities
    needed for diffusion.
    Returns:
        dict with keys:
            'betas', 'alphas',
            'alphas_cumprod',
            'sqrt_alphas_cumprod',
            'sqrt_one_minus_alphas_cumprod',
            'posterior_variance',
            'posterior_mean_coef1',
            'posterior_mean_coef2'
    """
    betas = np.linspace(beta_start, beta_end, num_timesteps)
    alphas = 1.0 - betas
    alphas_cumprod = np.cumprod(alphas, axis=0)

    sqrt_alphas_cumprod = np.sqrt(alphas_cumprod)
    sqrt_one_minus_alphas_cumprod = np.sqrt(1.0 - alphas_cumprod)

    alphas_cumprod_prev = np.empty_like(alphas_cumprod)
    alphas_cumprod_prev[0] = 1.0
    alphas_cumprod_prev[1:] = alphas_cumprod[:-1]

    posterior_variance = (
        betas * (1.0 - alphas_cumprod_prev) / (1.0 - alphas_cumprod)
    )

    posterior_mean_coef1 = (
        betas * np.sqrt(alphas_cumprod_prev) / (1.0 - alphas_cumprod)
    )

    posterior_mean_coef2 = (
```

```
        (1.0 - alphas_cumprod_prev) * np.sqrt(alphas) / (1.0 - alphas_cumprod)
    )

    return {
        'betas': betas,
        'alphas': alphas,
        'alphas_cumprod': alphas_cumprod,
        'sqrt_alphas_cumprod': sqrt_alphas_cumprod,
        'sqrt_one_minus_alphas_cumprod': sqrt_one_minus_alphas_cumprod,
        'posterior_variance': posterior_variance,
        'posterior_mean_coef1': posterior_mean_coef1,
        'posterior_mean_coef2': posterior_mean_coef2,
    }
```

The firgures below show the deliverables:

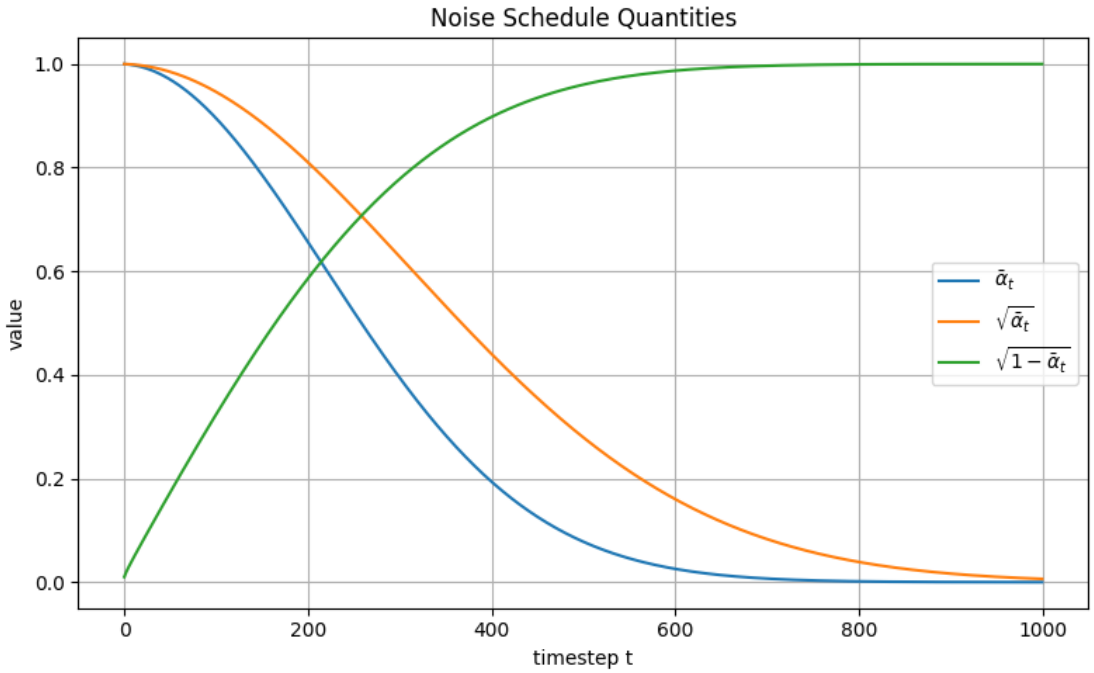


Figure 4: Plot of $\bar{\alpha}_t$, $\sqrt{\bar{\alpha}_t}$, $\sqrt{1-\bar{\alpha}_t}$ as a function of t

From the plot, the noise component $\sqrt{1-\bar{\alpha}_t}$ increases over timesteps while the signal component $\sqrt{\bar{\alpha}_t}$ decreases over timesteps. The point where noise starts to dominate is when the noise curve becomes larger than the signal curves. From the graph, this happens at around t = 250, where two curves intersects. After this point, the noise component is larger than the signal component, meaning the image is more dominated by noise than by the signal.

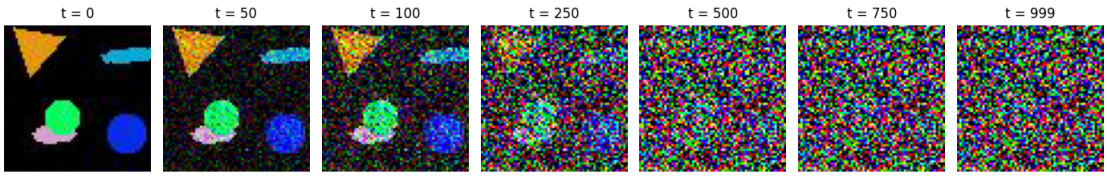


Figure 5: forward diffusion visualization showing the same image at 7 timesteps

From the images, the shapes in the original image are clearly visible at early timesteps when t = 0, 50, 100. At around t = 250, the shapes start to become very blurry and difficult to distinguish. At around t = 500, the image is almost completely dominated by noise, and the original image is no longer recognizable. After that, the images keeps as pure random noise.

3.2.3 Running DLStudio's GenerativeDiffusion

In this task, we are asked to run **RunCodeForDiffusion.py** in DLStudio package on PurdueShape5GAN dataset. This code trains DLStudio's GenerativeDiffsuion model. For this training, we are asked to run at least 40000 iterations and showing the following results:

1. Training loss curves
2. Generate image grids at 2 different training checkpints (For this, I select 20000 and 40000 checkpoints)
3. Compare real PurdueShape5GAN images with diffusion generated images

The script I use is shown below. This is basically the same as the **RunCodeForDiffusion.py** in DLStudio package.

[ ]:
```
##  RunCodeForDiffusion.py

"""
IMPORTANT NOTE:

   You will need to install the PurdueShapes5GAN dataset before you can␣
↪execute this script.

   Download the dataset archive

            datasets_for_AdversarialLearning.tar.gz

   through the link "Download the image dataset for AdversarialLearning and␣
↪Diffusion" provided
   at the top of main doc page for DLStudio and store it in the

            ExamplesDiffusion

   directory of the distribution.  Subsequently, execute the following command␣
↪in that directory:
```

```
            tar zxvf datasets_for_AdversarialLearning.tar.gz

    This command will create a 'dataGAN' subdirectory and deposit the following␣
 ↪dataset archive
    in that subdirectory:

            PurdueShapes5GAN-20000.tar.gz

    Now execute the following in the "dataGAN" directory:

            tar zxvf PurdueShapes5GAN-20000.tar.gz

    With that, you should be able to execute the diffusion related scripts in␣
 ↪the 'ExamplesDiffusion'
    directory.
"""

##  About this script:

##  This is the script you must execute for training a diffusion based model␣
 ↪for your
##  training data.

##  See the README in the ExamplesDiffusion directory for how to use this␣
 ↪script for
##  training a diffusion model.

##  watch -d -n 0.5 nvidia-smi



from GenerativeDiffusion import *

gauss_diffusion   =  GaussianDiffusion( num_diffusion_timesteps = 1000 )

network =  UNetModel(
                      in_channels=3,
                      model_channels    =  128,
                      out_channels      =  3,
                      num_res_blocks    =  2,
                      attention_resolutions =  (4, 8),
                      channel_mult      =    (1, 2, 3, 4),
                      num_heads         =    1,
                      attention         =    True             ## <<<< Make sure␣
 ↪that GenerateNewImageSamples.py has the same
```

```
#                       attention        =    False           ## <<<< Make sure␣
 ↪that GenerateNewImageSamples.py has the same
                     )


top_level = GenerativeDiffusion (
                       data_dir = "./dataGAN/PurdueShapes5GAN/multiobj/0/",
#                       data_dir="/home/kak/ImageDatasets/PurdueShapes5GAN/
 ↪multiobj/0/",
#                       data_dir = "/mnt/cloudNAS3/Avi/ImageDatasets/
 ↪PurdueShapes5GAN/multiobj/0/",

                       image_size            =        64,
                       num_channels          =        128,

                       lr=1e-4,

#                       batch_size=8,                     ## for laptop based␣
 ↪debugging
                       batch_size=32,                    ## on RVL Cloud

#                       log_interval=10,                  ## for laptop based␣
 ↪debugging
#                       save_interval=100,                ## for laptop based␣
 ↪debugging
                       log_interval=100,
                       save_interval=10000,

                       ema_rate  = 0.9999,
                       diffusion = gauss_diffusion,
                       network = network,
                       ngpu = 1,
                       path_saved_model = "RESULTS"
             )


top_level.run_code_for_training()
```

The figures below show the deliverables:

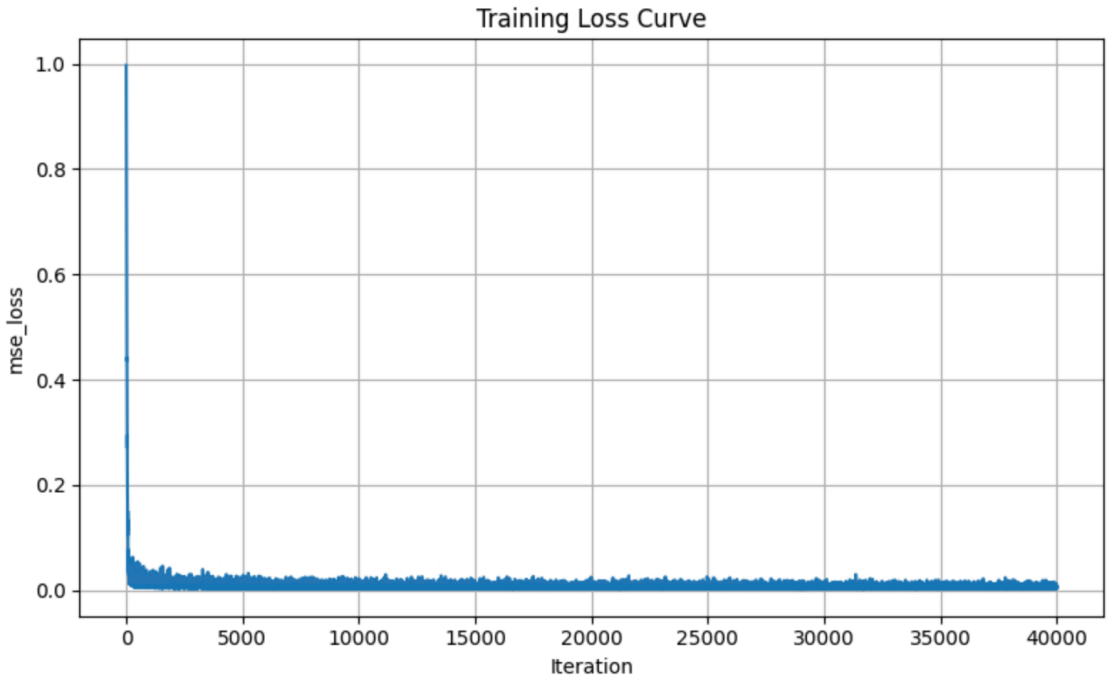
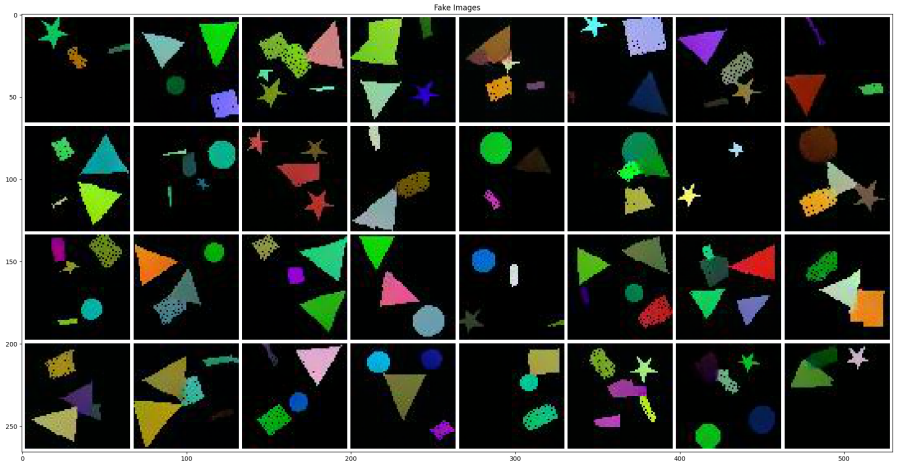


Figure 6: DLStudio's GenerativeDiffusion training loss curve



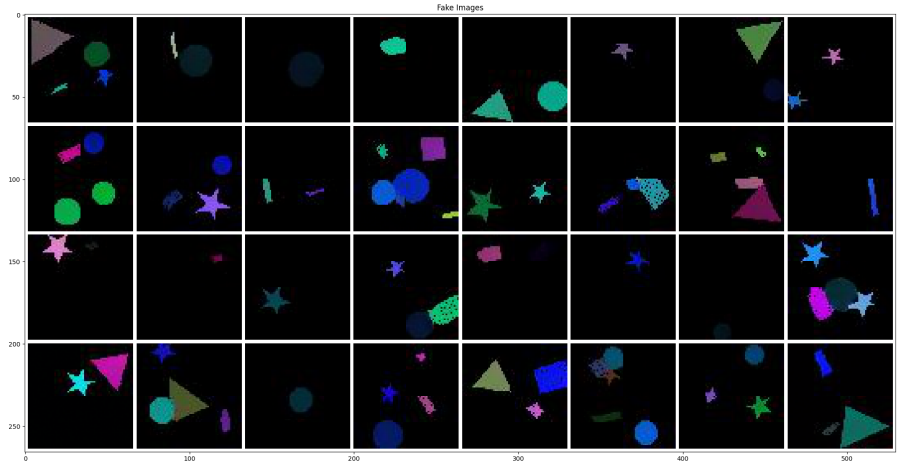
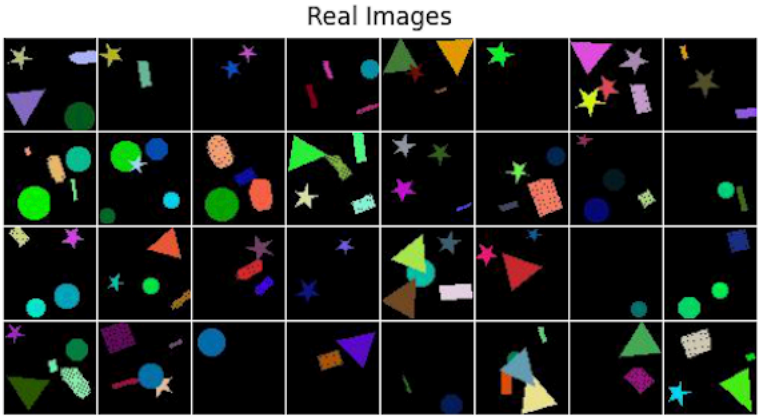


Figure 6: Generated image grids from different training stages (top: 20000 iterations, bottom: 40000 iterations)



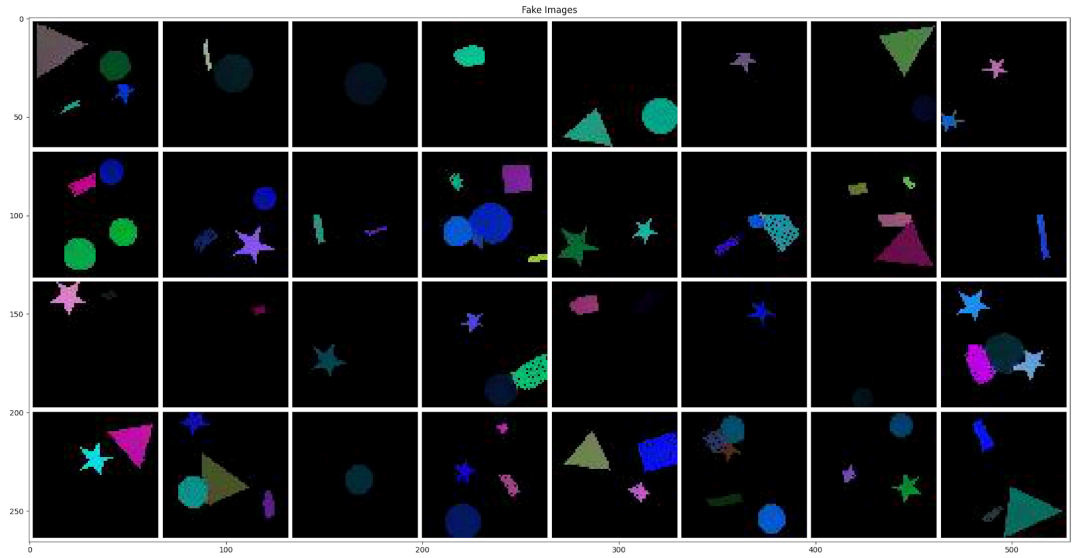


Figure 7: Side-by-side comparison: real PurdueShapes5GAN images vs. diffusion generated images

The diffusion-generated images perform comparatively better than DCGAN-generated images. Specifically, the diffusion-generated images show clearer shapes, better structure, and more consistent object boundaries compared to the DCGAN-generated images. In the diffusion results, most shapes are well-formed and recognizable, with smoother edges and more stable colors. In contrast, the DCGAN images appear noisier and less stable, with blurrier edges and dstorted structures.

3.3 Task 3: CelebA – Train GAN, Run Diffusion, and Compare with FID

3.3.1 Design and Train Your Own DCGAN

In this task, we are asked to design and train our own DCGAN based DLStudio's DiscriminatorDG1 and GeneratorDG1 model. The training and validation dataset is CelabA-64 dataset. Based on the requirements, my designed Genertor takes a 100-dimensional random noise vector and turns it into a $3 \times 64 \times 64$ RGB face image using nn.ConvTranspose2d. My deisgned Discriminator takes a $3 \times 64 \times 64$ image as input and output a single probability uses strided convolution layers followed by a final Sigmoid layer. I also used BatchNorm2d in both networks. For activation functions, I used LeakyReLU in the Discriminator, ReLU in the Generator, and Tanh at the Generator output. For adversarial training, I used nn.BCELoss.

For training setting, I used the suggested configuration from the homework. Specifically, I use Adam optimizer with lr=2e-4 and betas=(0.5, 0.999), batch size = 64, 50 training epochs and image normalization to [-1, 1].

After training, I also get the following deliverables:

1. Total number of learnable parameters in Discriminator and Generator
2. A 4 $\times$4 grid of generated face images at the end of training
3. A sequence of 4 snapshots at different epochs (1, 10, 25, 50)

The script for defining the networks, training loop and image generation is shown below:

[ ]:
```
import os
import math
import random
from pathlib import Path

import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
from PIL import Image
from torch.utils.data import DataLoader
from torchvision import datasets, transforms, utils


class Generator(nn.Module):
    def __init__(self, nz=100, ngf=64, nc=3):
        super().__init__()
        self.main = nn.Sequential(
            nn.ConvTranspose2d(nz, ngf * 8, kernel_size=4, stride=1, padding=0,␣
 ↪bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 8, ngf * 4, kernel_size=4, stride=2,␣
 ↪padding=1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 4, ngf * 2, kernel_size=4, stride=2,␣
 ↪padding=1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 2, ngf, kernel_size=4, stride=2,␣
 ↪padding=1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf, nc, kernel_size=4, stride=2, padding=1,␣
 ↪bias=False),
            nn.Tanh()
        )

    def forward(self, x):
        return self.main(x)
```

```
class Discriminator(nn.Module):
    def __init__(self, ndf=64, nc=3):
        super().__init__()
        self.main = nn.Sequential(
            nn.Conv2d(nc, ndf, kernel_size=4, stride=2, padding=1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf, ndf * 2, kernel_size=4, stride=2, padding=1,␣
 ↪bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 2, ndf * 4, kernel_size=4, stride=2, padding=1,␣
 ↪bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 4, ndf * 8, kernel_size=4, stride=2, padding=1,␣
 ↪bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(ndf * 8, 1, kernel_size=4, stride=1, padding=0,␣
 ↪bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.main(x)


def weights_init(m):
    classname = m.__class__.__name__
    if "Conv" in classname:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif "BatchNorm" in classname:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0.0)


def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)


def denormalize(x):
    return (x + 1.0) / 2.0
```

```python
def save_image_grid(images, filepath, nrow=4, title=None):
    images = denormalize(images.detach().cpu()).clamp(0, 1)
    grid = utils.make_grid(images, nrow=nrow, padding=2)
    plt.figure(figsize=(8, 8))
    plt.axis("off")
    if title is not None:
        plt.title(title)
    plt.imshow(grid.permute(1, 2, 0).numpy())
    plt.tight_layout()
    plt.savefig(filepath, bbox_inches="tight")
    plt.close()


def save_loss_curve(d_losses, g_losses, filepath):
    plt.figure(figsize=(8, 5))
    plt.plot(d_losses, label="Discriminator Loss")
    plt.plot(g_losses, label="Generator Loss")
    plt.xlabel("Iteration")
    plt.ylabel("Loss")
    plt.title("DCGAN Training Loss Curves")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(filepath, bbox_inches="tight")
    plt.close()


def set_seed(seed):
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)


if __name__ == "__main__":
    data_dir = "./Supplementary/celeba_dataset_64x64/"   # change this to your␣
 ↪dataset folder
    output_dir = Path("gan_outputs")
    output_dir.mkdir(parents=True, exist_ok=True)

    seed = 42
    image_size = 64
    nc = 3
    nz = 100
    ngf = 64
    ndf = 64
    batch_size = 64
```

```
    num_epochs = 50
    lr = 2e-4
    beta1 = 0.5
    beta2 = 0.999
    num_workers = 2
    snapshot_epochs = [1, 10, 25, 50]

    set_seed(seed)

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    print(f"Using device: {device}")

    # Dataset and DataLoader
    transform = transforms.Compose([
        transforms.Resize((image_size, image_size)),
        transforms.CenterCrop(image_size),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ])

    dataset = datasets.ImageFolder(
        root=str(Path(data_dir).parent),
        transform=transform
    ) if Path(data_dir).is_dir() and any(Path(data_dir).iterdir()) is False␣
↪else None

    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        pin_memory=torch.cuda.is_available(),
        drop_last=False,
    )

    # Define models
    netG = Generator(nz=nz, ngf=ngf, nc=nc).to(device)
    netD = Discriminator(ndf=ndf, nc=nc).to(device)

    netG.apply(weights_init)
    netD.apply(weights_init)

    print("\nGenerator:\n", netG)
    print("\nDiscriminator:\n", netD)

    print(f"\nGenerator learnable parameters: {count_parameters(netG):,}")
    print(f"Discriminator learnable parameters: {count_parameters(netD):,}")
```

```
    # Define Loss and Optimizers
    criterion = nn.BCELoss()

    optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, beta2))
    optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, beta2))

    fixed_noise = torch.randn(16, nz, 1, 1, device=device)

    real_label = 1.0
    fake_label = 0.0

    G_losses = []
    D_losses = []
    iteration = 0

    # Training Loop
    print("\nStarting Training...\n")
    for epoch in range(1, num_epochs + 1):
        for i, data in enumerate(dataloader):
            netD.zero_grad()

            real_images = data[0].to(device)
            b_size = real_images.size(0)

            label = torch.full((b_size,), real_label, dtype=torch.float,␣
↪device=device)

            output_real = netD(real_images).view(-1)
            errD_real = criterion(output_real, label)
            errD_real.backward()

            noise = torch.randn(b_size, nz, 1, 1, device=device)
            fake_images = netG(noise)
            label.fill_(fake_label)

            output_fake = netD(fake_images.detach()).view(-1)
            errD_fake = criterion(output_fake, label)
            errD_fake.backward()

            errD = errD_real + errD_fake
            optimizerD.step()


            netG.zero_grad()
            label.fill_(real_label)
```

```
            output_fake_for_G = netD(fake_images).view(-1)
            errG = criterion(output_fake_for_G, label)
            errG.backward()
            optimizerG.step()

            D_losses.append(errD.item())
            G_losses.append(errG.item())

            if i % 100 == 0:
                print(
                    f"Epoch [{epoch}/{num_epochs}] "
                    f"Batch [{i}/{len(dataloader)}] "
                    f"Loss_D: {errD.item():.4f} "
                    f"Loss_G: {errG.item():.4f} "
                    f"D(x): {output_real.mean().item():.4f} "
                    f"D(G(z)): {output_fake.mean().item():.4f} /␣
↪{output_fake_for_G.mean().item():.4f}"
                )

            iteration += 1

        # Save snapshot grids at selected epochs
        if epoch in snapshot_epochs:
            with torch.no_grad():
                fake_snapshot = netG(fixed_noise).detach()
            save_image_grid(
                fake_snapshot,
                output_dir / f"generator_snapshot_epoch_{epoch}.png",
                nrow=4,
                title=f"Generator Snapshot - Epoch {epoch}"
            )

    # Final Deliverables
    torch.save(netG.state_dict(), output_dir / "generator_final.pth")
    torch.save(netD.state_dict(), output_dir / "discriminator_final.pth")

    # Save training loss curves
    save_loss_curve(D_losses, G_losses, output_dir / "loss_curves.png")

    # Save final 4x4 generated image grid
    with torch.no_grad():
        final_fake = netG(fixed_noise).detach()
    save_image_grid(
        final_fake,
        output_dir / "generated_faces_final_4x4.png",
        nrow=4,
        title="Generated Faces at End of Training"
```

```
    )

    # mode collapse inspection
    with torch.no_grad():
        inspect_noise = torch.randn(64, nz, 1, 1, device=device)
        inspect_fake = netG(inspect_noise).detach()
    save_image_grid(
        inspect_fake,
        output_dir / "generated_faces_64_for_mode_collapse_check.png",
        nrow=8,
        title="64 Generated Faces for Mode Collapse Inspection"
    )

    print("\nTraining complete.")
    print(f"Outputs saved to: {output_dir.resolve()}")
```

The delieverables are shown below:

Total number of learnable parameters in each network:

1. Discriminator: 2765568
2. Generator: 3576704

Generator dimension trace:

The Generator starts with a noise vector of size (100, 1, 1). Using the ConvTranspose2d formula $H_{out} = (H_{in} - 1) \cdot s + k - 2p$, the spetial diemsnions change as follows:

1. Layer 1: (100, 1, 1) $\rightarrow$ (512, 4, 4)
2. Layer 2: (512, 4, 4) $\rightarrow$ (256, 8, 8)
3. Layer 3: (256, 8, 8) $\rightarrow$ (128, 16, 16)
4. Layer 4: (128, 16, 16) $\rightarrow$ (64, 32, 32)
5. Layer 5: (64, 32, 32) $\rightarrow$ (3, 64, 64)

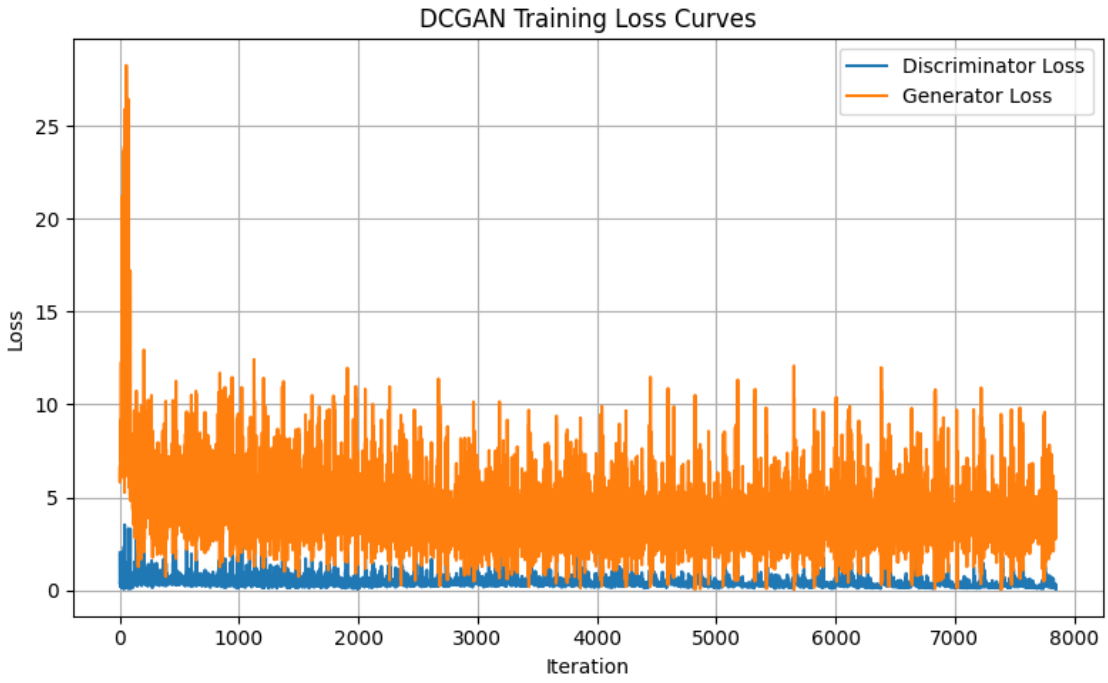


Figure 8: Discriminator and Generator loss curves over training iterations

Figure 9: A 4 $\times$4 grid of generated face images at the end of training

Generator Snapshot - Epoch 1

Generator Snapshot - Epoch 10

Generator Snapshot - Epoch 25

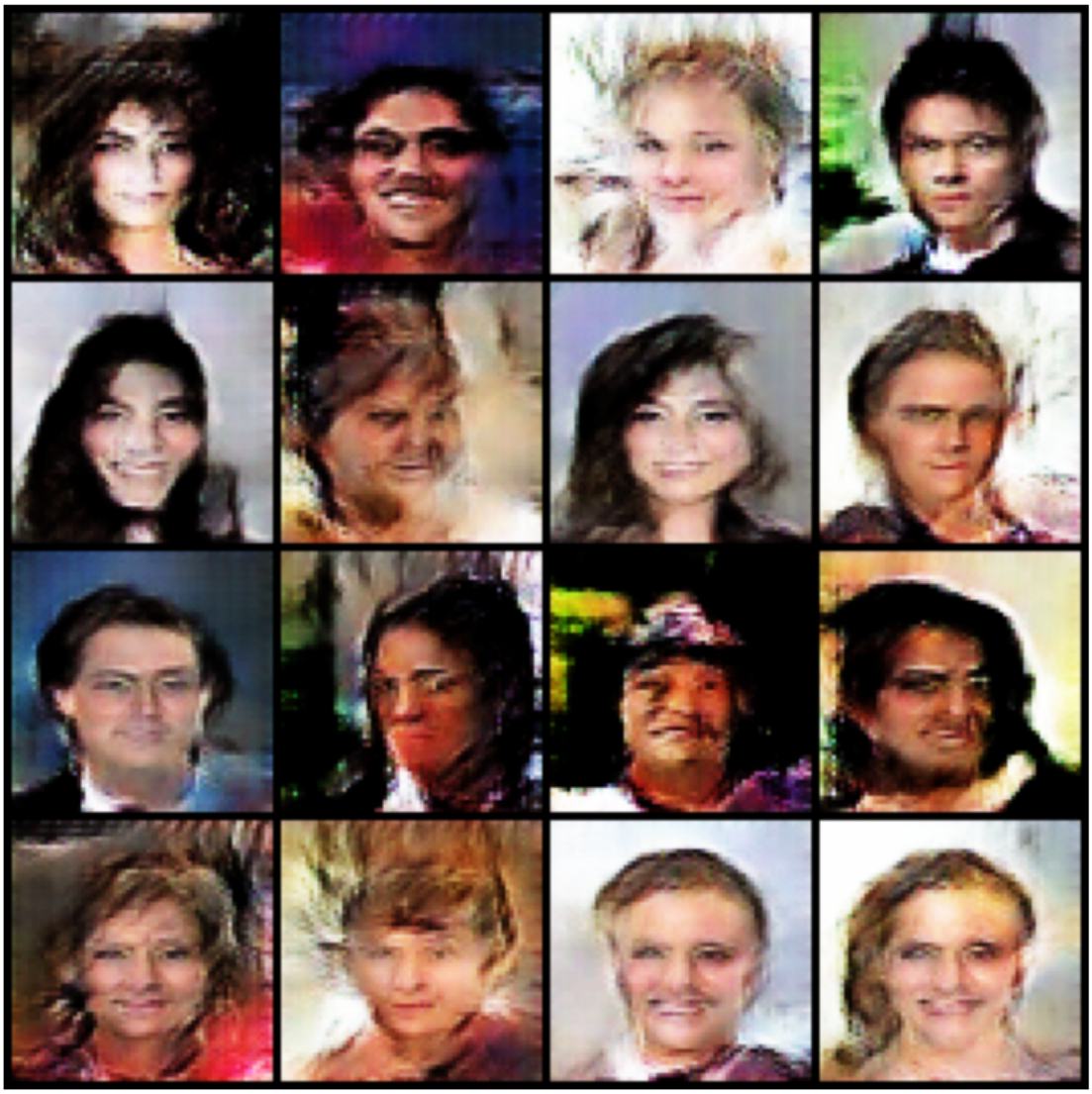


Figure 10: A sequence of 4 snapshots at epoch 1, 10, 25, 50 (from top to bottom)

Analysis:

1. Training dynamics

From the loss curves, the training process shows typical GAN behavior with oscillations rather than smooth convergence, as what we have seen in DLStudio's DCGAN. The Discriminator loss stays relatively low most of the time, while the Generator loss fluctuates significantly and sometimes goes to high values. This indicates that the Generator and Discriminator are continuously competing

2. Visual quality

The generated images improve significantly over training as number of epochs increase. At epoch 1, the images are very blurry and mostly noise, with no clear facial structure. At epoch 10, basic facial structures start to appear, but they are still distorted and unclear. At epoch 25, the faces become more recognizable, with visible eyes, noses and smoother textures. By epoch 50, most generated

images look like realistic human faces. They have clearer facial features and more natural colors. Overall, the final quality is good and most faces are clearly recognizable.

3. Mode collapase check

By inspecting many generated images, the model shows good diversity. The generated faces vary in gender, hair color, facial structure, and face expression, which indicates that the Generator is not producing the same image repeatedly. Overall, the Generator produces a wide range of different faces, which suggests that the model has learned a meaningful distribution across the dataset rather than collapsing to a few modes.

3.3.2 Generate Faces with Pretrained Diffusion

In this task, we are asked to use pretrained diffusion weight **diffusion.pt** to generate 1024 fake face images using **GenerateNewImageSamples.py** and **VisualizeSamples.py** in DLStudio package. We asked to show the following deleiverables:

1. A 4 * 4 grid of diffusion-generated face images

The deliverables are shown below:

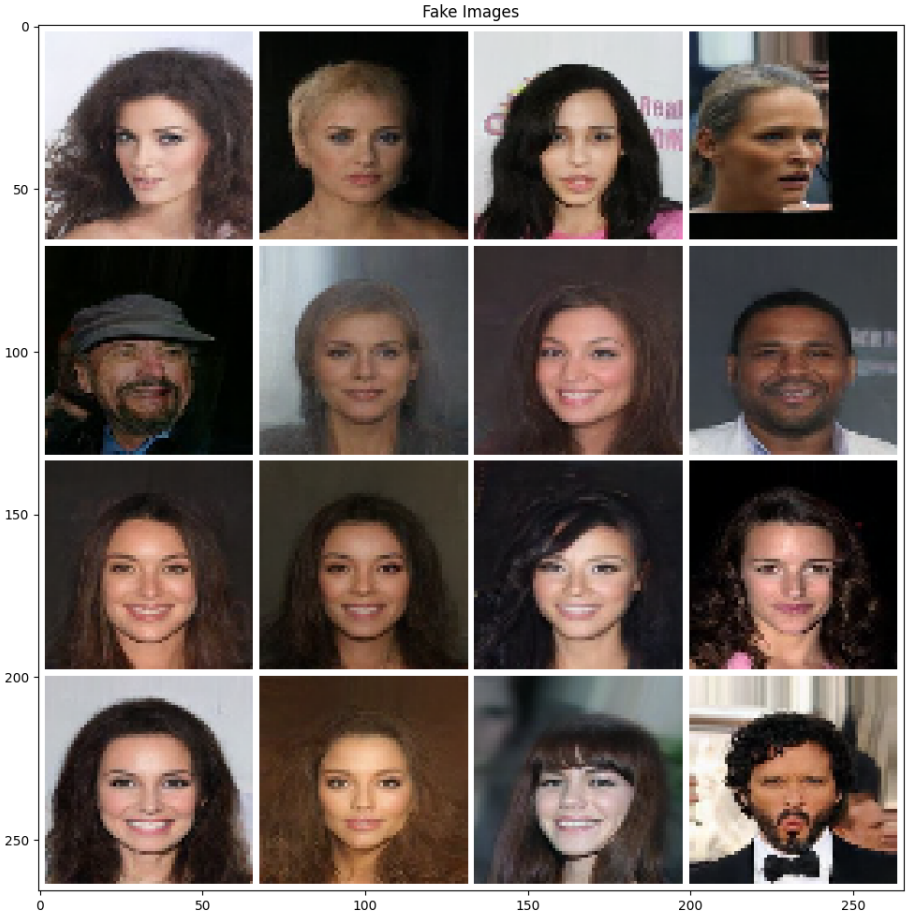


Figure 11: A 4 ×4 grid of diffusion-generated face images

The diffusion-generated images have noticeably higher visual quality than the GAN-generated images. Specifcially, the faces from the diffusion model are sharper, more realistic, and have clearer facial features. In contrast, the GAN-generated images appear blurrier and contain more artifacts, with some faces looking distorted or less natural. The diffusion results also show more consistent lighting details in the backgroud, while the GAN outputs sometimes look noisy. Overall, the diffusion model produces more realistic and visually better faces compared to the GAN.

3.3.3 FID Evaluation: GAN vs. Diffusion

In this task, we are asked to implement Frechet Inception Distance (FID) evaluating metric. This metric is used to measuring the quality and diversity of generated images. If we get low FID score, it means that our generated images have better quality and diversity.

In order to compare our trained GAN and pretrained diffusion model using FID, I generate 1024

images from both model and pytorch-fid package to compute FID. The dataset I use is CelabA-64 dataset.

The code implementation of FID computataion is shown below:

[ ]:
```
import os
from pathlib import Path

import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from PIL import Image
from pytorch_fid.fid_score import \
    calculate_activation_statistics, \
    calculate_frechet_distance
from pytorch_fid.inception import InceptionV3
from torchvision import transforms


class Generator(nn.Module):
    def __init__(self, nz=100, ngf=64, nc=3):
        super().__init__()
        self.main = nn.Sequential(
            nn.ConvTranspose2d(nz, ngf * 8, kernel_size=4, stride=1, padding=0,␣
 ↪bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 8, ngf * 4, kernel_size=4, stride=2,␣
 ↪padding=1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 4, ngf * 2, kernel_size=4, stride=2,␣
 ↪padding=1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 2, ngf, kernel_size=4, stride=2,␣
 ↪padding=1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf, nc, kernel_size=4, stride=2, padding=1,␣
 ↪bias=False),
            nn.Tanh()
        )
```

```
    def forward(self, x):
        return self.main(x)


def save_tensor_as_image(img_tensor, save_path):
    img_tensor = (img_tensor + 1.0) / 2.0
    img_tensor = img_tensor.clamp(0, 1)
    img = transforms.ToPILImage()(img_tensor.cpu())
    img.save(save_path)


def make_4x4_grid(image_paths, save_path, title):
    fig, axes = plt.subplots(4, 4, figsize=(8, 8))
    for ax, img_path in zip(axes.flat, image_paths[:16]):
        img = Image.open(img_path).convert("RGB")
        ax.imshow(img)
        ax.axis("off")
    plt.suptitle(title)
    plt.tight_layout()
    plt.savefig(save_path, bbox_inches="tight")
    plt.close()


def list_image_paths(folder):
    exts = {".jpg", ".jpeg", ".png", ".bmp", ".webp"}
    return sorted(
        [str(p) for p in Path(folder).iterdir() if p.suffix.lower() in exts]
    )


if __name__ == "__main__":
    real_dir = "./Supplementary/celeba_dataset_64x64"
    diffusion_dir = "/home/yang1581/Data/ECE60146/DLStudio-2.5.6/
 ↪ExamplesDiffusion/pretrained_diffusion_v3/"
    gan_model_path = "gan_outputs/generator_final.pth"
    gan_generated_dir = "fid_eval/gan_generated_1024"
    diffusion_grid_path = "fid_eval/diffusion_4x4_grid.png"
    gan_grid_path = "fid_eval/gan_4x4_grid.png"
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    os.makedirs("fid_eval", exist_ok=True)
    os.makedirs(gan_generated_dir, exist_ok=True)

    # Generate 1024 GAN images
    nz = 100
    netG = Generator(nz=nz, ngf=64, nc=3).to(device)
    netG.load_state_dict(torch.load(gan_model_path, map_location=device))
```

```
netG.eval()

batch_size = 64
total_images = 1024
saved_count = 0

with torch.no_grad():
    while saved_count < total_images:
        current_batch_size = min(batch_size, total_images - saved_count)
        noise = torch.randn(current_batch_size, nz, 1, 1, device=device)
        fake_images = netG(noise)

        for i in range(current_batch_size):
            save_path = os.path.join(
                gan_generated_dir,
                f"gan_{saved_count:04d}.png"
            )
            save_tensor_as_image(fake_images[i], save_path)
            saved_count += 1

real_paths = list_image_paths(real_dir)[:1024]
gan_paths = list_image_paths(gan_generated_dir)[:1024]
diffusion_paths = list_image_paths(diffusion_dir)[:1024]

# Compute FID
dims = 2048
block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
model = InceptionV3([block_idx]).to(device)

m1, s1 = calculate_activation_statistics(
    real_paths, model, device=device
)
m2, s2 = calculate_activation_statistics(
    gan_paths, model, device=device
)
fid_gan = calculate_frechet_distance(
    m1, s1, m2, s2
)

m3, s3 = calculate_activation_statistics(
    diffusion_paths, model, device=device
)
fid_diffusion = calculate_frechet_distance(
    m1, s1, m3, s3
)

# Save 4x4 image grids
```

```
    make_4x4_grid(
        gan_paths,
        gan_grid_path,
        "GAN Generated Faces"
    )

    make_4x4_grid(
        diffusion_paths,
        diffusion_grid_path,
        "Diffusion Generated Faces"
    )

    # Print FID
    print("\nFID Comparison")
    print(fid_gan)
    print(fid_diffusion)
```

The delieverables are shown below:

Model	FID (↓ better)
Your DCGAN	89.6761
Diffusion (pretrained)	74.4646

Figure 12: FID comparison table


GAN Generated Faces

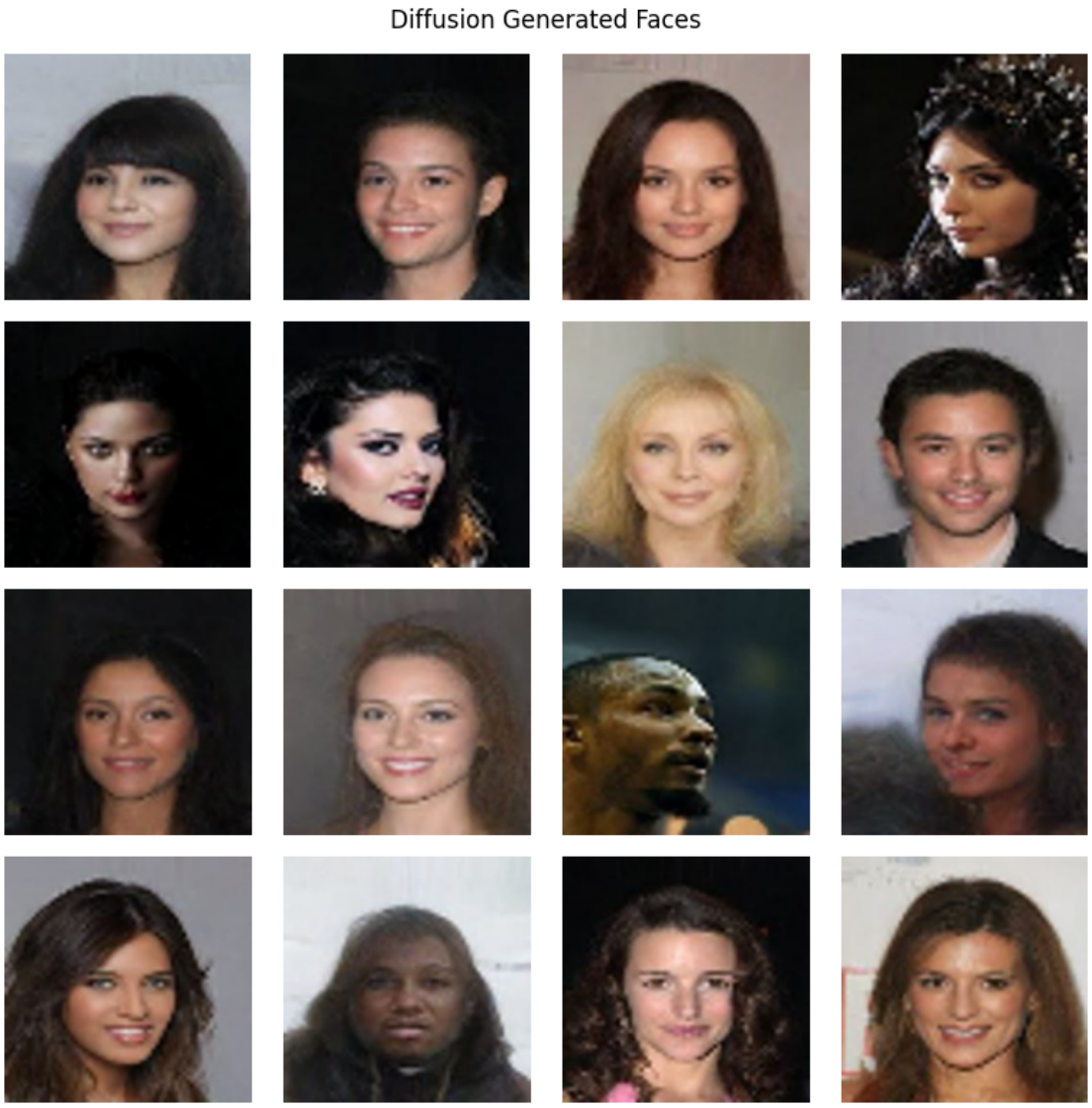


Figure 13: 4 × 4 image grids of GAN-generated and diffusion-generated faces

Visually inspect all 1024 GAN images for mode collapse:

After visually inspecting all 1024 GAN-generated images, there is no clear sign of severe mode collapse. Specifcially, the faces are not identical and still show some variation. However, there are some similarities in blurriness. Specifcially, some facial features appear distorted or repeated in the same way. This suggests that while the GAN is generating different faces, it may be partially collapsing to a few common noisy patterns.

Discussion:

Based on the FID comparison table, the diffusion model achieves a lower FID (74.46) compared to the our GAN (89.68), which means the diffusion model generates images that are closer to the real data distribution. Based on image grids, the diffusion-generated faces also appear more diverse, with noticeable differences in gender, hairstyle, and facial expressions. On the other hand, the GAN-generated images show some diversity but tend to have similar facial textures and often look

blurry or distorted.

Besides, the diffusion model also captures finer details much better. However, the GAN images often have artifacts and lack sharpness. Such difference comes from different models. Specifically, GANs use adversarial training, which can lead to instability and mode collapse because of competetition between generator and discriminator. In diffusion models, it learns to gradually denoise images, which makes training more stable.

3.4 Task 4: Stable Diffusion – Text-to-Face and Inpainting

3.4.2 Text-to-Face Generation

In this task, we are asked to use public Stable Diffusion model to generate face images. Specifically, we are using HuggingFace's diffusers library. We are required to use the following 6 different prompts to generate the face images:

1. "portrait photo of a smiling young woman with brown hair"
2. "portrait photo of an old man with a beard and glasses"
3. "portrait photo of a woman with blonde hair and sunglasses"
4. "portrait photo of a young man, serious expression, black hair"
5. "watercolor painting of a woman's face, artistic style"
6. "pencil sketch portrait of an elderly person"

For each prompt, we are asked to use 2 random seeds, meaning that we need to generate 2 different images for each prompt. For deleiverables, we need to show all the generated images in a 4 * 3 grid.

The script I use to generate images is shown below. The script is mainly based on the code template in the PDF, but I modify it so that it can directly output a 4 * 3 image grid.

[ ]:
```
from diffusers import StableDiffusionPipeline
import torch
import os
import math
from PIL import Image
import matplotlib.pyplot as plt


model_id = "runwayml/stable-diffusion-v1-5"
pipe = StableDiffusionPipeline.from_pretrained(
    model_id, torch_dtype=torch.float16
)
pipe = pipe.to("cuda")
# pipe.enable_attention_slicing()  # if low VRAM


prompts = [
    "portrait photo of a smiling young woman with brown hair",
```

```
    "portrait photo of an old man with a beard and glasses",
    "portrait photo of a woman with blonde hair and sunglasses",
    "portrait photo of a young man, serious expression, black hair",
    "watercolor painting of a woman's face, artistic style",
    "pencil sketch portrait of an elderly person",
]

seeds = [42, 123]

output_dir = "stable_diffusion_outputs"
os.makedirs(output_dir, exist_ok=True)

generated_images = []
titles = []

for prompt_idx, prompt in enumerate(prompts):
    for seed in seeds:
        gen = torch.Generator(
            device="cuda").manual_seed(seed)
        image = pipe(
            prompt,
            num_inference_steps=50,
            guidance_scale=7.5,
            generator=gen
        ).images[0]

        image_filename = f"sd_prompt_{prompt_idx+1}_seed_{seed}.png"
        image.save(os.path.join(output_dir, image_filename))

        generated_images.append(image)
        titles.append(f"P{prompt_idx+1}, seed={seed}")

fig, axes = plt.subplots(4, 3, figsize=(12, 16))

for ax, image, title in zip(axes.flat, generated_images, titles):
    ax.imshow(image)
    ax.set_title(title, fontsize=10)
    ax.axis("off")

plt.tight_layout()
plt.savefig(os.path.join(output_dir, "sd_4x3_grid.png"))
plt.show()
```

The deliverables are shown in the figure below:

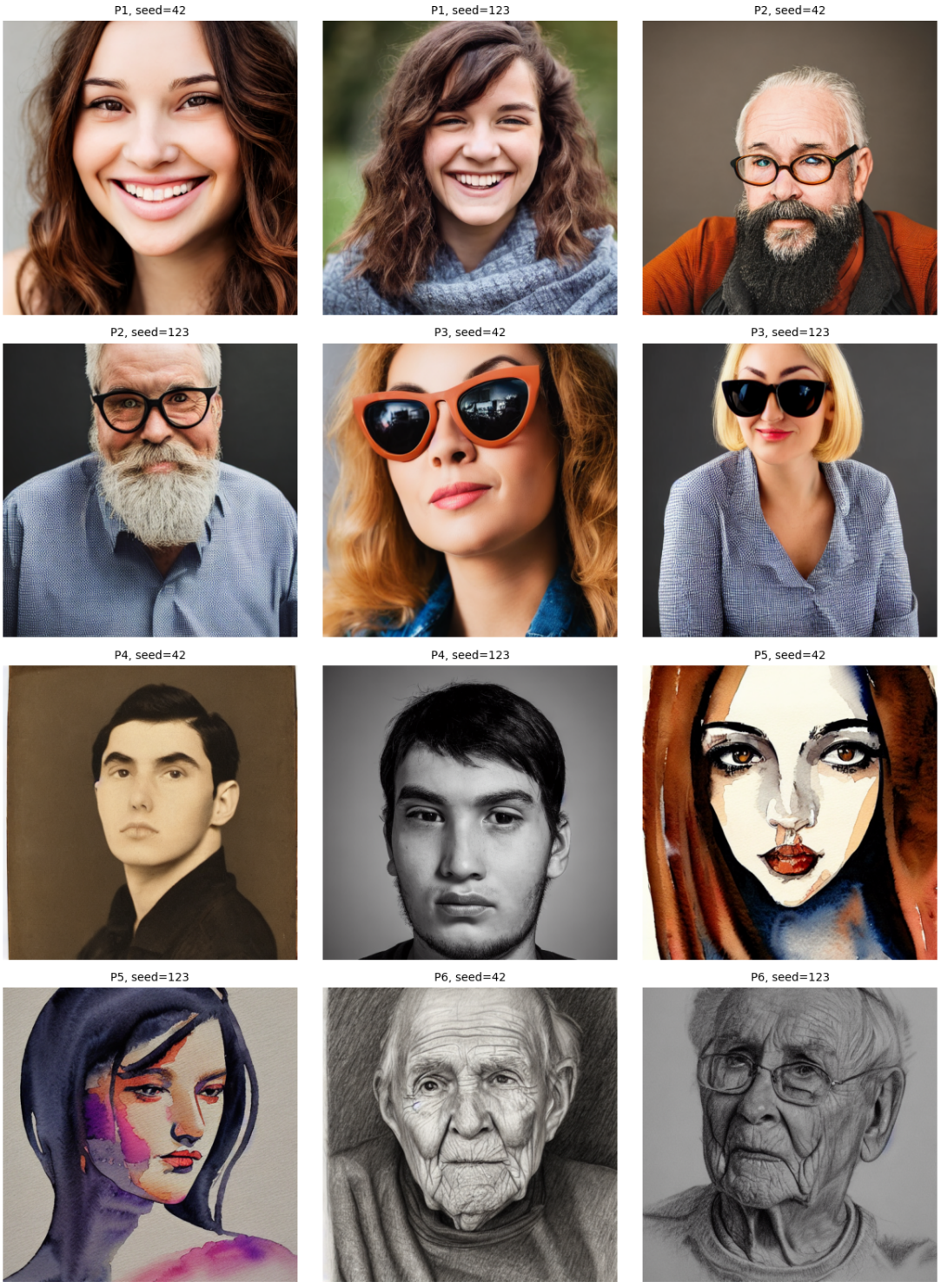


Figure 14: 4 × 3 grid showing all 12 generated images

Based on the images grid, Stable Diffusion captures the requested attributes very well. For the first 4 photorealistic prompts, the generated faces clearly match the descriptions, such as age differences, hair color and accessories. The images look realistic with clear facial features and natural skin

texture. For the last 2 artistic prompts, the model also performs well because the generated images clearly show the artistc style of watercolor and pencil sketches. Compared to the 64×64 GAN and diffusion outputs from Task 4, these 512×512 Stable Diffusion images are much higher quality. Since Stable Diffusion images have higher resolution, they have sharper facial details and fewer artifacts. In contrast, the lower-resolution GAN/diffusion images often look blurry or distorted.

3.4.3 Investigating Guidance Scale and Inference Steps

In this task, we are asked to understand how guidance scale and number of inference steps affect the quality of images generated by Stable Diffusion model. For guidance scale parameter, we need to test with the following different values:

guidance scale $\{1.0, 3.0, 5.0, 7.5, 10.0, 15.0, 20.0\}$

and visualize the generated images for each guidance scale value.

For number of inference steps parameter, we need to test withe the following different values:

num inference steps $\{5, 10, 20, 30, 50, 75, 100\}$

and visualize the generated images for each number of inference steps.

The deliverables are shown i the figures below:

Figure 15: A single row of 7 images for the guidance scale sweep

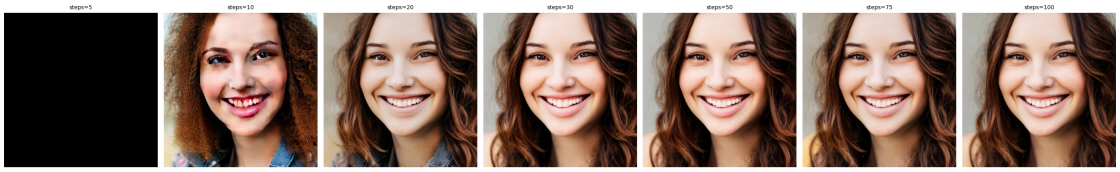


Figure 16: A single row of 7 images for the inference steps sweep

Report:

1. Guidance scale

At guidance scale = 1.0, the image looks natural but slightly less aligned with the prompt. The face is still realistic, but details like expression and facial sharpness are weaker. The reason is that the model is almost behaving like unconditional generation, which means that the model does not strctly follow the prompt. At guidance scale = 20.0, the image becomes overly sharp and somewhat unnatural. The face looks overly perfect, with exaggerated smile and overly smooth skin. This is a sign of losing realism.

For me, the image looks best roughly in the range of 5.0 to 10.0. Within this range, the image is both realistic and clearly follows the prompt, without over smoothing or under smoothing. From the formula, $\epsilon = \epsilon_u + s \cdot (\epsilon_c - \epsilon_u)$. So, when s is very large, the difference $(\epsilon_c - \epsilon_u)$ is overly amplified. in this case, the model is forced to strongly follows the prompt. This causes exaggerated features and over-saturate the image.

2. Inference steps

With only 5 inference steps, the output is very is completely blank and does not form a recognizable face. This is because the model has too few steps to remove noise and recover facial structure. As the number of inference steps increases, the image becomes clearer and more realistic. The quality continues to improve up to around 50 steps. After 50 steps, the improvement becomes very small. So, the image quality saturates around 50–75 steps, and using more steps makes no difference in image quality.

Such behavior matches the reverse Markov chain process. Specifically, with fewer steps, the model must take larger jumps when denoising, which makes it harder to recover details and accurate structures. On the other hand, with more steps, the model takes smaller steps, allowing it to refine fine details.

3.4.4 Text-Guided Inpainting on CelebA Faces

In this task, we are asked to take real CelebA face images, mask out a specific region, and then use Stable Diffusion's inpainting model to regenerate that region based on a text prompt. Based on the code template in the PDF, I implemented the inpainting pipeline using the pretrained Stable Diffusion inpainting model. We need to perform multiple experiments about different impainting masks:

1. Hair color change
2. Add/remove accessories
3. Expression change
4. Creative – your choice (my choice is "thick beard and mustache")

For deliverables, we need to show a row of 4 images for each experiment: original face → mask overlay → inpainted variant(s)

The script I use for impainting experiments is shown below. The script is based on the code template in the PDF, but I modify it so that we can run different inpainting experiments in the same script.

[ ]:
```
from diffusers import StableDiffusionInpaintPipeline
from PIL import Image, ImageDraw
import torch
import os
import matplotlib.pyplot as plt


inpaint_pipe = StableDiffusionInpaintPipeline.from_pretrained(
    "runwayml/stable-diffusion-inpainting",
```

```
    torch_dtype=torch.float16
).to("cuda")


def make_mask(mask_type):
    mask = Image.new("RGB", (512, 512), "black")
    draw = ImageDraw.Draw(mask)

    if mask_type == "hair":
        draw.rectangle([0, 0, 512, 205], fill="white")

    elif mask_type == "eyes":
        draw.rectangle([110, 150, 400, 260], fill="white")

    elif mask_type == "mouth":
        draw.rectangle([120, 300, 400, 460], fill="white")

    elif mask_type == "beard":
        draw.rectangle([130, 250, 390, 500], fill="white")

    return mask


def make_mask_overlay(image, mask):
    image = image.convert("RGBA")
    overlay = Image.new("RGBA", image.size, (255, 0, 0, 0))

    mask_gray = mask.convert("L")
    overlay_draw = ImageDraw.Draw(overlay)

    for y in range(mask.size[1]):
        for x in range(mask.size[0]):
            if mask_gray.getpixel((x, y)) > 0:
                overlay_draw.point((x, y), fill=(255, 0, 0, 80))

    return Image.alpha_composite(image, overlay).convert("RGB")


def run_inpaint_experiment(
    celeb_path,
    mask_type,
    prompts,
    output_prefix,
    output_dir,
    num_inference_steps=50,
    guidance_scale=7.5
):
```

```
celeb = Image.open(celeb_path).convert("RGB")
celeb = celeb.resize((512, 512), Image.LANCZOS)

mask = make_mask(mask_type)
overlay = make_mask_overlay(celeb, mask)

result_images = []

for i, prompt in enumerate(prompts):
    result = inpaint_pipe(
        prompt=prompt,
        image=celeb,
        mask_image=mask,
        num_inference_steps=num_inference_steps,
        guidance_scale=guidance_scale
    ).images[0]

    result.save(
        os.path.join(output_dir, f"{output_prefix}_result_{i+1}.png")
    )
    result_images.append(result)

celeb.save(os.path.join(output_dir, f"{output_prefix}_original.png"))
mask.save(os.path.join(output_dir, f"{output_prefix}_mask.png"))
overlay.save(os.path.join(output_dir, f"{output_prefix}_overlay.png"))

total_cols = 2 + len(result_images)
fig, axes = plt.subplots(1, total_cols, figsize=(4 * total_cols, 4))

axes[0].imshow(celeb)
axes[0].set_title("Original")
axes[0].axis("off")

axes[1].imshow(overlay)
axes[1].set_title("Mask Overlay")
axes[1].axis("off")

for idx, result_img in enumerate(result_images):
    axes[idx + 2].imshow(result_img)
    axes[idx + 2].set_title(f"Result {idx+1}")
    axes[idx + 2].axis("off")

plt.tight_layout()
plt.savefig(os.path.join(output_dir, f"{output_prefix}_row.png"))
plt.close()
```

```python
if __name__ == "__main__":
    output_dir = "inpainting_outputs"
    os.makedirs(output_dir, exist_ok=True)

    celeb_image_1 = "./Supplementary/celeba_dataset_64x64/000041.jpg"
    celeb_image_2 = "./Supplementary/celeba_dataset_64x64/000095.jpg"
    celeb_image_3 = "./Supplementary/celeba_dataset_64x64/000096.jpg"
    celeb_image_4 = "./Supplementary/celeba_dataset_64x64/000134.jpg"

    # Experiment 1: Hair color change
    run_inpaint_experiment(
        celeb_path=celeb_image_1,
        mask_type="hair",
        prompts=[
            "blonde hair, portrait photo",
            "bright blue hair, portrait photo"
        ],
        output_prefix="experiment_1_hair_color_change",
        output_dir=output_dir
    )

    # Experiment 2: Add/remove accessories
    run_inpaint_experiment(
        celeb_path=celeb_image_2,
        mask_type="eyes",
        prompts=[
            "wearing large stylish sunglasses, portrait",
            "natural eyes without glasses, portrait photo"
        ],
        output_prefix="experiment_2_accessories",
        output_dir=output_dir
    )

    # Experiment 3: Expression change
    run_inpaint_experiment(
        celeb_path=celeb_image_3,
        mask_type="mouth",
        prompts=[
            "wide smile showing teeth, portrait photo",
            "serious expression, closed mouth, portrait"
        ],
        output_prefix="experiment_3_expression_change",
        output_dir=output_dir
    )

    # Experiment 4: Creative - your choice
    run_inpaint_experiment(
```

```
        celeb_path=celeb_image_4,
        mask_type="beard",
        prompts=[
            "thick beard and mustache, portrait photo"
        ],
        output_prefix="experiment_4_creative",
        output_dir=output_dir
    )

    print(f"All inpainting results saved to: {output_dir}")
```

The deliverables are shown in the figures below:

Figure 17: Hair color change inpating outpout

Figure 17: Add/remove accessories inpating outpout

Figure 18: Expression change inpating outpout

Figure 19: Creative-your choice inpating outpout

Report:

Based on the images above, the inpainted results are generally convincing, especially when modifying large and well-defined regions like hair and expressions. In the hair color experiment, both blonde and blue hair look natural and match the head shape. This shows good blending with the original face. Similarly, expression changing works well, for the expressions align properly with the face and appear realistic.

However, some experiments struggle, especially when modifying smaller or more complex regions like adding the glasss. In the adding sunglasses experiment, the generated sunglasses add to the part that not perfectly align with the eyes. The creative example of adding beard also looks unnatural and less realistic. For me, I think the text prompt clearly controls what is generated, but the the quality depends on how well the model understands the facial structure of that region. If the model cannot fully intepret the facial region it needs to inpaint, the quality bcome poor and less realistic.

Report:

In diffusion inpainting, the unmasked part of the image is kept fixed at every step. In this case, only the masked region starts from noise and is gradually denoised based on the text prompt. At each timestep, the updated masked region is combined back with the unchanged part, which makes the final image looks consistent. This is very different from a GAN, for GAN generates the whole image from noise and cannot keep part of an existing image unchanged while modifying a specific region.