BME646 and ECE60146: Homework 8

Spring 2026
Due Date: Tuesday, April 7, 2026, 11:59 pm ET
Progress Report Due Date: Tuesday, March 31, 2026, 11:59 pm ET
TA: Somosmita Mitra (mitra26@purdue.edu)

Turn in typed solutions via Gradescope. Post questions to Piazza. Additional instructions can be found at the end. **Late submissions will be accepted with penalty: -10 points per late day, up to 5 days.**

Early proof-of-progress deadline: You must submit a brief progress report by **11:59 pm ET, March 31, 2026** showing that you have: (1) successfully run DLStudio's DCGAN on PurdueShapes5GAN with initial loss curves, (2) loaded the CelebA-64 dataset and displayed sample images, and (3) successfully downloaded and run Stable Diffusion inference on at least one prompt.

Your final homework submission will NOT be graded if you do not submit the progress report by the March 31 deadline. No exceptions will be made.

1 Introduction

In the previous homeworks, you built networks for classification of increasing sophistication (HW4–5), multi-instance object detection and semantic segmentation (HW6), and metric learning with CLIP (HW7). In all of those cases, the network learned to produce outputs conditioned on ground-truth labels: class labels, bounding boxes, segmentation masks, or embedding distances.

In this homework, you will shift from *discriminative* models (which learn decision boundaries between classes) to *generative* models (which learn the underlying probability distribution of the data and can synthesize new samples from it). You will work with **two datasets**. Tasks 1–2 use DLStudio's PurdueShapes5GAN dataset for warm-up experiments with existing DLStudio code. Starting from Task 3 onward, you switch to a **subset of the CelebA face dataset**, where you will train your own GAN on real celebrity faces, run diffusion-based generation, and ultimately use Stable Diffusion for text-guided face generation and **inpainting**, where you mask out parts of

real CelebA faces and use text prompts to regenerate them (e.g., changing hair color, adding sunglasses, or modifying facial expressions).

The homework is organized into four tasks spanning three generative approaches:

1. **Adversarial Learning (GANs):** You will run DLStudio's DCGAN on PurdueShapes5GAN to understand adversarial training, then design and train your own DCGAN on CelebA faces.

2. **Denoising Diffusion:** You will study the theory of forward and reverse Markov chains, implement the noise schedule, run DLStudio's `GenerativeDiffusion` module on PurdueShapes5GAN, and generate CelebA faces from pretrained diffusion weights.

3. **Stable Diffusion:** You will download pretrained Stable Diffusion weights and generate face images from text prompts, explore how inference parameters affect output, and perform text-guided inpainting on real CelebA faces.

As discussed in the lecture on generative data modeling [2], these approaches introduce several new concepts:

- **Minimax optimization:** Unlike standard supervised learning with a single loss, adversarial learning [4] involves two networks with opposing objectives. The Discriminator maximizes its ability to distinguish real from fake, while the Generator minimizes the Discriminator's success.

- **Transpose convolutions for generation:** The Generator in a DCGAN uses `nn.ConvTranspose2d` to progressively upsample a noise vector into a full-resolution image, the same operation you encountered in the decoder of your segmentation network in HW6.

- **Forward and reverse Markov chains:** Diffusion models [5] define a forward process that gradually destroys an image with noise and a reverse process that learns to undo each noising step. After training, the reverse process can generate images from pure noise.

- **Text-conditioned generation and inpainting with Stable Diffusion:** As mentioned on Slides 132–133 of the lecture [2], Stable Diffusion [6] combines the CLIP text encoder [7] (which you used in HW7) with latent diffusion to generate images from text prompts. Stable Diffusion can also do *inpainting*: filling in masked parts of an image using a text prompt. You will try both in this homework.

- **Fréchet Inception Distance (FID):** A quantitative metric for comparing the quality and diversity of generated images against real images, based on Inception network features.

Here is what you will do:

- Run DLStudio's DCGAN (DG1) on PurdueShapes5GAN and analyze the training dynamics, architecture, and minimax game of adversarial learning
- Study the theory of denoising diffusion, implement the noise schedule, visualize the forward process, and run DLStudio's `GenerativeDiffusion` module on PurdueShapes5GAN
- Design and train your own DCGAN on CelebA-64 faces, generate faces from pretrained diffusion weights, and quantitatively compare both using FID
- Download pretrained Stable Diffusion weights and generate face images from text prompts, exploring how guidance scale and inference steps affect quality
- Perform text-guided inpainting on real CelebA faces, masking hair, eyes, or mouth and regenerating them with text control

1.1 Quick Reference: Lecture Slide Numbers

Use this table to look up the relevant lecture material for each topic in this homework.

HW8 Topic	**Lecture**	**Slides**
Part 1: Adversarial Learning [2]		
PurdueShapes5GAN dataset	GenerativeDataModeling	53–59
DCGAN: Discriminator and Generator	GenerativeDataModeling	60–80
DCGAN mode collapse (DG2)	GenerativeDataModeling	81–85
Part 2: Diffusion [2]		
Stable Diffusion overview	GenerativeDataModeling	131–133
Forward and reverse Markov chains	GenerativeDataModeling	134–141
Loss function for denoising diffusion	GenerativeDataModeling	142–157
Gaussian noise for diffusion	GenerativeDataModeling	158–172
GenerativeDiffusion module in DLStudio	GenerativeDataModeling	174–186
Background		
Transpose convolutions (review from HW6)	SemanticSeg [3]	29–62

Which slides to read for each task:

Task	Key slides
Task 1: DCGAN warm-up	GenerativeDataModeling 53–85
Task 2: Diffusion warm-up	GenerativeDataModeling 131–186
Task 3: CelebA GAN + Diffusion + FID	GenerativeDataModeling 60–85, 174–186
Task 4: Stable Diffusion + Inpainting	GenerativeDataModeling 131–133

2 Setting Up Your Environment

2.1 Prerequisites from Previous Homeworks

This homework builds on your previous setup. Ensure you have:

- DLStudio version 2.5.6 installed, including the `AdversarialLearning` and `GenerativeDiffusion` modules
- The PurdueShapes5GAN dataset (see Section 2.2)
- Python 3.8+ with PyTorch, NumPy, Matplotlib, and SciPy installed

2.2 Downloading the PurdueShapes5GAN Dataset (Tasks 1–3)

Download the dataset archive `datasets_for_AdversarialNetworks.tar.gz` through the link "Download the image dataset for Adversarial Learning" provided at the top of the HTML version of the DLStudio page [1]. Store it in the `ExamplesAdversarialLearning` directory and execute:

```
tar zxvf datasets_for_AdversarialNetworks.tar.gz
```

This creates a `dataGAN` subdirectory containing `PurdueShapes5GAN-20000.tar.gz`. Execute in the `dataGAN` directory:

```
tar zxvf PurdueShapes5GAN-20000.tar.gz
```

The PurdueShapes5GAN dataset consists of 20,000 images of size 64×64. Each image contains a random number of up to five shapes (rectangle, triangle, disk, oval, star), each randomly located, oriented, and colored. See Slides 53–59 of the lecture [2] for details.

2.3 The CelebA-64 Dataset (Tasks 3–4)

Download the supplementary material from **Brightspace** under HW8. This includes:

- A subset of 10,000 CelebA face images, all resized to 64×64
- Pretrained diffusion model weights for CelebA-64

2.4 Setting Up DLStudio's Diffusion Code

The diffusion code is in DLStudio's `GenerativeDiffusion` module. The `ExamplesDiffusion` directory contains three scripts:

1. `RunCodeForDiffusion.py` – Trains the denoising neural network
2. `GenerateNewImageSamples.py` – Generates new images from a trained checkpoint
3. `VisualizeSamples.py` – Extracts and displays individual generated images

Read the README in the `ExamplesDiffusion` directory before running any scripts.

2.5 Setting Up Stable Diffusion (for Task 4)

Task 4 requires downloading two pretrained Stable Diffusion models. **Start these downloads early** as the weights total approximately 9 GB.

```
pip install diffusers transformers \
    accelerate scipy safetensors pytorch-fid
```

The first time you run each pipeline in your code, it will automatically download the model weights from HuggingFace. You can also pre-download them:

```
from diffusers import StableDiffusionPipeline
from diffusers import \
    StableDiffusionInpaintPipeline
import torch

# Text-to-image model (~5 GB)
model_id = "runwayml/stable-diffusion-v1-5"
pipe = StableDiffusionPipeline.from_pretrained(
    model_id, torch_dtype=torch.float16)

```

```
# Inpainting model (~4 GB)
inpaint_id = \
    "runwayml/stable-diffusion-inpainting"
inpaint_pipe = \
    StableDiffusionInpaintPipeline \
    .from_pretrained(
    inpaint_id, torch_dtype=torch.float16)
```

GPU strongly recommended for Tasks 3 and 4. Stable Diffusion inference on CPU is extremely slow (10–20 minutes per image). On a GPU with $\geq$8 GB VRAM, inference takes 5–15 seconds per image. Use Google Colab's free GPU if you do not have a local GPU.

3 Programming Tasks (100 Points Total)

3.1 Task 1: DCGAN Warm-Up on PurdueShapes5GAN (10 points)

Objective: Gain familiarity with adversarial learning by running DLStudio's DCGAN code on the PurdueShapes5GAN dataset and studying its architecture and training loop, before designing your own GAN for CelebA in Task 3.

3.1.1 Running DCGAN (DG1) (5 points)

Run the DLStudio adversarial learning example script `dcgan_DG1.py` in the `ExamplesAdversarialLearning` directory. Train for 30 epochs on PurdueShapes5GAN. Refer to Slides 60–80 of the lecture [2] for the architecture (`DiscriminatorDG1` on Slide 71, `GeneratorDG1` on Slide 72) and the training loop (Slides 75–77).

Deliverables:

- Discriminator and Generator loss curves over 30 epochs (overlaid on one plot with legend)
- A side-by-side figure showing: (left) a batch of real PurdueShapes5GAN images, and (right) the images produced by the Generator from noise vectors at the end of training (similar to Slide 79)
- A sequence of at least 4 snapshots of Generator output at different epochs (e.g., epoch 1, 10, 20, 30) using the **same** noise vectors, to show how generated images evolve during training

3.1.2 Understanding the Architecture and Training (5 points)

Study the Discriminator (`DiscriminatorDG1`, Slide 71) and Generator (`GeneratorDG1`, Slide 72) code, as well as the training loop on Slides 75–77. **Report** (answer each in 1–2 paragraphs):

- **Generator architecture:** The Generator starts with a 1×1 noise image with 100 channels and must produce a $64 \times 64 \times 3$ output. Using the formula $H_{\text{out}} = (H_{\text{in}} - 1) \times s + k - 2p$, trace the spatial dimensions through each `ConvTranspose2d` layer. What activation function is used at the output and why?
- **The minimax game:** Referring to Eq. (35) on Slide 64, explain the two phases of Discriminator training (what targets are used when the Discriminator sees real images? What targets when it sees Generator output?) and the Generator training phase (why do we use a target of 1?).
- **The `fakes.detach()` call:** In Line (B) of the training loop (Slide 76), why is `fakes.detach()` used when training the Discriminator with Generator output? What would happen if we omitted `detach()`?

Grading:

- Successful DCGAN execution with all visual deliverables (5 points)
- Clear explanations of architecture and training dynamics (5 points)

3.2 Task 2: Denoising Diffusion Warm-Up on PurdueShapes5GAN (20 points)

Objective: Understand the theory of denoising diffusion probabilistic models, implement the noise schedule, visualize the forward diffusion process, and run DLStudio's `GenerativeDiffusion` module on PurdueShapes5GAN.

3.2.1 Understanding the Forward and Reverse Processes (7 points)

Study Slides 134–167 of the lecture [2]. **Report** (answer each clearly):

- **The two Markov chains:** Describe the q-chain (forward diffusion) and the p-chain (reverse denoising). What is the input and output of each? What role does the timestep parameter T play? Draw or describe a diagram showing both chains (similar to Slide 139).

- **Forward transition probabilities:** Explain Eq. (70) on Slide 160:

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

  What is β_t (the noise schedule)? What happens to the image as t increases toward T?

- **The "jump to any timestep" formula:** Explain Eqs. (71)–(72) on Slide 162:

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

  Why is this formula computationally useful? (Hint: it avoids stepping through all intermediate timesteps.) What do α_t and $\bar{\alpha}_t$ represent?

- **What the neural network learns:** In each training iteration, a single random timestep t is chosen. What is the neural network's prediction target at that timestep? How does the loss function drive the network to learn effective denoising? (Slides 164–167.)

3.2.2 Noise Schedule Computation and Forward Diffusion Visualization (7 points)

Implement the α-β calculations from Slides 169–171 for a noise schedule with $T = 1000$ timesteps, $\beta_{\text{start}} = 0.0001$, and $\beta_{\text{end}} = 0.02$.

```
import numpy as np

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)
```

```
    # YOUR CODE: compute remaining quantities
    # following the formulas on Slides 169-171
```

Using your noise schedule and Eq. (72), visualize the forward diffusion process on a single PurdueShapes5GAN image at timesteps $t = 0, 50, 100, 250, 500, 750, 999$.

Deliverables:

- Your complete `compute_noise_schedule` implementation
- Plots of $\bar{\alpha}_t$, $\sqrt{\bar{\alpha}_t}$, and $\sqrt{1-\bar{\alpha}_t}$ as functions of t (three curves on one figure)
- A paragraph interpreting the plots: at approximately what timestep does the noise component dominate the signal?
- The forward diffusion visualization showing the same image at 7 timesteps
- A brief paragraph describing what you observe: at what point does the image become unrecognizable?

3.2.3 Running DLStudio's GenerativeDiffusion (6 points)

Run `RunCodeForDiffusion.py` on PurdueShapes5GAN. Train for at least 40,000 iterations. Then use `GenerateNewImageSamples.py` and `VisualizeSamples.py` to generate new images from one or more checkpoints.

GPU strongly recommended. If you only have CPU, reduce iterations (minimum 20,000) or use Google Colab. Document your compute setup.

Deliverables:

- Training loss curve over training iterations
- Generated image grids from at least 2 checkpoints at different training stages (e.g., at 20,000 and 40,000 iterations)
- Side-by-side comparison: real PurdueShapes5GAN images vs. diffusion-generated images from your best checkpoint
- A paragraph comparing diffusion-generated image quality to DCGAN-generated images from Task 1

Grading:

- Clear explanations of forward/reverse processes and learning target (7 points)

- Noise schedule implementation with plots, forward visualization, and observations (7 points)
- Successful diffusion training with generated samples and comparison to DCGAN (6 points)

3.3 Task 3: CelebA – Train GAN, Run Diffusion, and Compare with FID (40 points)

Objective: Design and train your own DCGAN on CelebA-64 celebrity face images, generate faces from pretrained diffusion weights, and quantitatively compare both generative models using Fréchet Inception Distance (FID).

3.3.1 Design and Train Your Own DCGAN (20 points)

Drawing inspiration from DLStudio's DG1 (Task 1), design your own Generator and Discriminator networks for CelebA faces. As in previous homeworks, you have **total freedom** in network design, but:

- Your **Generator** must transform a random noise vector (e.g., 100-dimensional) into a $3 \times 64 \times 64$ RGB face image using transpose convolutions (`nn.ConvTranspose2d`)
- Your **Discriminator** must take a $3 \times 64 \times 64$ image and output a single scalar probability (real vs. fake) using strided convolutions and a final `Sigmoid`
- Use `nn.BatchNorm2d` in both networks (except the Discriminator's first layer and the Generator's last layer, following DCGAN conventions)
- Use `LeakyReLU` in the Discriminator and `ReLU` in the Generator (with `Tanh` at the Generator's output)
- Use `nn.BCELoss` for training

Training configuration (suggested):

- Optimizer: `Adam(lr=2e-4, betas=(0.5, 0.999))` for both networks
- Batch size: 32 or 64
- Epochs: 50–100

- Noise vector dimension: 100
- Normalize images to $[-1, 1]$ (matching the `Tanh` output)

Deliverables:

- Your complete Generator and Discriminator class definitions
- Total number of learnable parameters in each network
- Trace the spatial dimensions through each Generator layer using the `ConvTranspose2d` formula
- Your complete training loop code
- Discriminator and Generator loss curves over training iterations (overlaid on one plot with legend)
- A 4×4 grid of generated face images at the end of training
- A sequence of at least 4 snapshots at different epochs (e.g., epoch 1, 10, 25, 50) using the **same** noise vectors, showing how generated faces evolve
- Analysis (1–2 paragraphs each): training dynamics (convergence, oscillation?), visual quality (recognizable features?), mode collapse check (inspect 64+ images: do different noise vectors produce diverse faces in gender, hair color, expression?)

3.3.2 Generate Faces with Pretrained Diffusion (5 points)

Using the pretrained CelebA-64 diffusion weights provided on Brightspace:

1. Update the results directory path in `GenerateNewImageSamples.py` to point to the downloaded weights
2. Run the script to generate **1024 fake face images**
3. Update paths in `VisualizeSamples.py` and visualize the results

Deliverables:

- A 4×4 grid of diffusion-generated face images
- A brief paragraph comparing visual quality to your GAN output

3.3.3 FID Evaluation: GAN vs. Diffusion (15 points)

Fréchet Inception Distance (FID) [8] is a widely used metric for measuring both the quality and diversity of generated images. It encodes real and generated images into feature vectors using a pretrained Inception network, fits a multivariate Gaussian to each set, and computes the Fréchet distance between the two Gaussians. **Lower FID = better.**

1. Generate **1024 images** from your trained GAN using randomly sampled noise vectors. Save them as individual image files.
2. Use the **1024 images** you generated from the diffusion model above.
3. Use the `pytorch-fid` package [9] to compute FID for both sets against the real CelebA-64 training images.

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

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)
# Repeat for diffusion
```

Deliverables:

- Your FID computation code
- 4×4 image grids of GAN-generated and diffusion-generated faces (16 randomly chosen from 1024 each)
- Visually inspect all 1024 GAN images for **mode collapse**: does the GAN produce the same face pattern repeatedly across batches?
- FID comparison table:

Model	FID (↓ better)
Your DCGAN	?
Diffusion (pretrained)	?

- Discussion (1–2 paragraphs): Which model achieves lower FID? Which produces more diverse faces? Which captures finer details (eyes, hair texture, skin)? Relate to the differences between adversarial training and diffusion-based denoising.

Grading:

- GAN design, training, with loss plots and image grids (20 points)
- Successful diffusion generation with visual comparison (5 points)
- Correct FID computation, mode collapse analysis, and comparative discussion (15 points)

3.4 Task 4: Stable Diffusion – Text-to-Face and Inpainting (30 points)

Objective: Download pretrained Stable Diffusion, generate face images from text prompts, explore how inference parameters affect output, and perform text-guided inpainting on real CelebA faces (masking and regenerating hair, eyes, or expressions with text control).

As discussed on Slides 132–133 of the lecture [2], Stable Diffusion uses the CLIP text encoder (which you worked with in HW7) to condition a latent diffusion model. In this task you will *use* the full pretrained Stable Diffusion pipeline for both generation and inpainting.

3.4.1 Background: How Stable Diffusion Works

The lecture mentions Stable Diffusion on Slides 132–133 [2]. Before you begin the experiments, read the following background carefully so you understand what the system is actually doing internally. The link on Slide 133 (https://www.assemblyai.com/blog/how-to-run-stable-diffusion-locally-to-generate-images) is also a useful reference.

The three components of Stable Diffusion. Unlike the pixel-space diffusion model you ran in Task 2 (which adds and removes noise directly on 64×64 images), Stable Diffusion operates in a compressed **latent space** and has three main components:

1. **VAE Encoder/Decoder:** A pretrained Variational Autoencoder compresses a $512 \times 512 \times 3$ image into a $64 \times 64 \times 4$ latent representation (the encoder), and reconstructs images from latent representations (the decoder). All diffusion happens in this compact latent space, which is why Stable Diffusion can handle high-resolution images without the huge compute cost of pixel-space diffusion.

2. **UNet with CLIP Conditioning:** This is the denoising network, which is similar in spirit to the UNet you saw in DLStudio's `GenerativeDiffusion`, but with one important difference: at each denoising timestep, the UNet receives a **CLIP text embedding** that tells it what kind of image to produce. The CLIP embedding is injected into the UNet through **cross-attention layers**. At each spatial position in the UNet's feature maps, cross-attention allows the network to "look at" the text embedding and decide how that position should be denoised. This is how text ends up controlling what shows up in the generated image.

3. **CLIP Text Encoder:** This is the same CLIP model you used in HW7 for metric learning. It takes a text prompt (e.g., "a smiling woman with blonde hair") and produces an embedding vector (or a sequence of token embeddings) that encodes what the text means. This embedding is what gets fed into the UNet's cross-attention layers.

The generation pipeline. When you call `pipe(prompt, ...)` in the `diffusers` library, the following happens:

1. The text prompt is tokenized and passed through CLIP's text encoder to produce a text embedding

2. Random Gaussian noise is generated in the latent space ($64 \times 64 \times 4$)

3. The UNet iteratively denoises the latent, conditioned on the text embedding, for N steps (the `num_inference_steps` parameter)

4. The final denoised latent is passed through the VAE decoder to produce a $512 \times 512 \times 3$ pixel image

Classifier-Free Guidance (the `guidance_scale` parameter). At each denoising step, the UNet actually runs *twice*: once with the text embedding (conditioned prediction ϵ_c) and once without it (unconditional prediction ϵ_u). The final noise prediction is:

$$\epsilon = \epsilon_u + s \cdot (\epsilon_c - \epsilon_u)$$

where s is the guidance scale. When $s = 1$, you get the raw conditioned output. As s increases, the model pushes harder toward the text-conditioned direction, producing images that match the prompt more closely, but at the cost of diversity and, at very high values, visual artifacts (over-saturation, unnatural colors). Typical values of s are between 5 and 15, with 7.5 being the default.

How inpainting works with diffusion. In standard diffusion, the entire image starts as noise and is progressively denoised. In inpainting:

- The **unmasked region** (the part of the image you want to keep) is preserved at every timestep
- The **masked region** (the part you want to change) starts from noise and is denoised as usual, but conditioned on the text prompt
- At each timestep, the denoised masked region is blended back with the preserved unmasked region

The result is that the masked region is regenerated to match the text prompt while blending in with the surrounding context. This is very different from how a GAN works: a GAN generates an entire image from noise, but inpainting only fills in a specific region while keeping the rest of the image intact.

3.4.2 Text-to-Face Generation (10 points)

Download pretrained Stable Diffusion weights and generate face images using HuggingFace's `diffusers` library. If you have not already installed the required packages, see Section 2.5.

```
from diffusers import StableDiffusionPipeline
import torch

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

prompt = "portrait photo of a smiling woman " \
         "with blonde hair, photorealistic"
gen = torch.Generator(
    device="cuda").manual_seed(42)
image = pipe(
    prompt,
```

```
    num_inference_steps=50,
    guidance_scale=7.5,
    generator=gen
).images[0]
image.save("sd_face_01.png")
```

Generate images for the following **6 prompts**, using **2 different random seeds** each (12 images total):

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"`

Deliverables:

- Your complete Stable Diffusion inference script
- A 4×3 grid showing all 12 generated images (6 prompts $\times$ 2 seeds), labeled with the prompt and seed
- A paragraph discussing: How well does Stable Diffusion capture the requested attributes (age, hair color, expression, style)? How do the photorealistic prompts (1–4) compare to the artistic prompts (5–6)? How do these 512×512 SD outputs compare to your 64×64 GAN/diffusion faces from Task 4?

3.4.3 Investigating Guidance Scale and Inference Steps (10 points)

The two most important inference-time parameters in Stable Diffusion are the **guidance scale** (also called classifier-free guidance scale) and the **number of inference steps**.

Experiment A – Guidance Scale. Choose one prompt from above. Generate images with the following guidance scale values, keeping the number of inference steps fixed at 50 and using the same random seed for all:

$$\texttt{guidance\_scale} \in \{1.0,\ 3.0,\ 5.0,\ 7.5,\ 10.0,\ 15.0,\ 20.0\}$$

Experiment B – Number of Inference Steps. Using the same prompt and seed, fix `guidance_scale=7.5` and generate images with:

$$\texttt{num\_inference\_steps} \in \{5,\ 10,\ 20,\ 30,\ 50,\ 75,\ 100\}$$

Deliverables:

- A single row of 7 images for the guidance scale sweep (labeled with each value)
- A single row of 7 images for the inference steps sweep (labeled with each value)
- **Report** (answer each in 1–2 paragraphs):
  - **Guidance scale:** What happens at 1.0 (nearly unconditional)? At 20.0? At what range does the image look best? Referring to the classifier-free guidance formula $\epsilon = \epsilon_u + s \cdot (\epsilon_c - \epsilon_u)$ from the Background section, explain why very high s values over-saturate the image.
  - **Inference steps:** What does the output look like with only 5 steps? At what number of steps does quality saturate? Relate this to the reverse Markov chain from Task 3: fewer steps mean larger jumps in the denoising process, leaving less room for the model to refine details at each transition.

3.4.4 Text-Guided Inpainting on CelebA Faces (10 points)

In this subtask you will take **real CelebA images**, mask out a region (hair, eyes, mouth, or accessories), and use Stable Diffusion's inpainting pipeline to regenerate that region guided by a text prompt. This is how modern AI photo-editing tools work.

```
from diffusers import \
    StableDiffusionInpaintPipeline
from PIL import Image, ImageDraw
import torch

inpaint_pipe = \
    StableDiffusionInpaintPipeline \
    .from_pretrained(
    "runwayml/stable-diffusion-inpainting",
    torch_dtype=torch.float16
).to("cuda")

```

```
# Load a CelebA face, resize to 512x512
celeb = Image.open(
    "celeba64/img_00042.jpg"
    ).convert("RGB")
celeb = celeb.resize((512, 512),
    Image.LANCZOS)

# Mask: white = inpaint, black = keep
mask = Image.new("RGB", (512, 512), "black")
draw = ImageDraw.Draw(mask)
draw.rectangle([0, 0, 512, 205],
    fill="white")  # mask hair region

result = inpaint_pipe(
    prompt="bright red spiky hair, "
           "portrait photo",
    image=celeb,
    mask_image=mask,
    num_inference_steps=50,
    guidance_scale=7.5
).images[0]
result.save("inpainted_red_hair.png")
```

Perform the following **4 inpainting experiments** on different CelebA images (or the same image with different masks and prompts):

1. **Hair color change:** Mask the hair region (top portion of the face). Inpaint with two different prompts: (a) `"blonde hair, portrait photo"` and (b) `"bright blue hair, portrait photo"`. Show both results side-by-side with the original.

2. **Add/remove accessories:** Mask the eye region. Inpaint with: (a) `"wearing large stylish sunglasses, portrait"` and (b) `"natural eyes without glasses, portrait photo"`.

3. **Expression change:** Mask the lower face (mouth and chin area). Inpaint with: (a) `"wide smile showing teeth, portrait photo"` and (b) `"serious expression, closed mouth, portrait"`.

4. **Creative – your choice:** Choose your own mask region and prompt combination. Be creative. Examples: "thick beard and mustache", "colorful face paint", "cyberpunk visor over eyes", "wearing a crown", etc.

Deliverables:

- Your complete inpainting script

- For each of the 4 experiments, display a row of 3–4 images: original face → mask overlay → inpainted variant(s)
- **Report** (1–2 paragraphs): How convincing are the inpainted results? Does the inpainted region blend well with the unmasked portions of the face? Which experiments produce the most convincing results and which struggle? How does the text prompt control the content of the inpainted region?
- **Report** (1 paragraph): Referring to the "How inpainting works with diffusion" explanation in the Background section above, explain how the masked and unmasked regions are treated differently at each denoising timestep. Why is this very different from how a GAN works? (Hint: a GAN generates an entire image from noise and has no way to keep parts of an existing image while regenerating other parts.)

Grading:

- Text-to-face generation with image grid and discussion (10 points)
- Guidance scale and inference steps experiments with explanations (10 points)
- Text-guided inpainting with 4 experiments and analysis (10 points)

4 Submission Instructions

4.1 What to Submit in the Report

Read this section carefully. Incomplete submissions will lose points. Every item listed below must be present in your submission.

1. **Task 1: DCGAN Warm-Up (10 points)**
   (a) **Code:** Script used to run DG1
   (b) **Figure:** D and G loss curves (overlaid, with legend)
   (c) **Figure:** Side-by-side real vs. generated images at end of training
   (d) **Figure:** Generator output evolution over 4+ epochs (same noise vectors)
   (e) **Text:** Generator dimension trace through each `ConvTranspose2d`
   (f) **Text:** Minimax training explanation (targets for each phase)

(g) **Text:** `fakes.detach()` explanation

2. **Task 2: Diffusion Warm-Up (20 points)**

   (a) **Text:** Two Markov chains description with diagram

   (b) **Text:** Forward transition probability explanation

   (c) **Text:** "Jump to any timestep" formula explanation

   (d) **Text:** Neural network learning target explanation

   (e) **Code:** Complete `compute_noise_schedule` implementation

   (f) **Figure:** $\bar{\alpha}_t$, $\sqrt{\bar{\alpha}_t}$, $\sqrt{1-\bar{\alpha}_t}$ plots with interpretation paragraph

   (g) **Figure:** Forward diffusion visualization (7 timesteps) with observations paragraph

   (h) **Figure:** Diffusion training loss curve

   (i) **Figure:** Generated image grids from 2+ checkpoints

   (j) **Figure:** Real vs. diffusion-generated side-by-side comparison

   (k) **Text:** Comparison of diffusion vs. DCGAN quality

3. **Task 3: CelebA GAN + Diffusion + FID (40 points)**

   (a) **Code:** Generator and Discriminator class definitions

   (b) **Text:** Parameter counts and Generator dimension trace

   (c) **Code:** Adversarial training loop

   (d) **Figure:** D and G loss curves (overlaid, with legend)

   (e) **Figure:** 4×4 grid of generated faces at end of training

   (f) **Figure:** Generator evolution over 4+ epochs (same noise vectors)

   (g) **Text:** Training dynamics, visual quality, and mode collapse analysis (1–2 paragraphs each)

   (h) **Figure:** 4×4 diffusion-generated face grid

   (i) **Text:** Diffusion vs. GAN visual quality comparison

   (j) **Code:** FID computation script

   (k) **Table:** FID values for GAN and diffusion

   (l) **Text:** FID comparative discussion (1–2 paragraphs)

4. **Task 4: Stable Diffusion + Inpainting (30 points)**

   (a) **Code:** Stable Diffusion inference script

(b) **Figure:** Grid of 12 generated images (6 prompts × 2 seeds), labeled

(c) **Text:** Attribute capture and comparison to Task 3 discussion

(d) **Figure:** Guidance scale sweep (7 images, one row, labeled)

(e) **Figure:** Inference steps sweep (7 images, one row, labeled)

(f) **Text:** Guidance scale explanation (low vs. high, over-saturation, formula)

(g) **Text:** Inference steps explanation (relation to reverse Markov chain)

(h) **Code:** Inpainting script

(i) **Figure:** 4 inpainting experiments (original → mask → result variant(s), one row each)

(j) **Text:** Inpainting quality discussion (blending, best/worst experiments)

(k) **Text:** How inpainting relates to diffusion; how it differs from GANs

4.2 Code Files to Submit

1. `noise_schedule.py`: `compute_noise_schedule` implementation and forward diffusion visualization
2. `gan.py`: Generator and Discriminator class definitions, training loop, and image generation
3. `fid_eval.py`: FID computation code
4. `run_stable_diffusion.py`: SD inference, guidance scale, and inference steps experiments
5. `inpainting.py`: Inpainting experiments
6. `requirements.txt`: Dependencies (must include `diffusers`, `transformers`, `pytorch-fid`)
7. `README.md`: Instructions to run your code, including compute setup and model download instructions

4.3 Autograder Interface Requirements

The autograder will import your `noise_schedule.py` file and call the following function. If the file name or function signature does not match exactly, the autograder will fail and you will lose points.

4.3.1 Required File Names

Submit exactly this file name (case-sensitive):

- `noise_schedule.py`

4.3.2 `noise_schedule.py`

Must contain:

- `compute_noise_schedule(num_timesteps=1000, beta_start=0.0001, beta_end=0.02)`: Returns a dictionary with keys `'betas'`, `'alphas'`, `'alphas_cumprod'`, `'sqrt_alphas_cumprod'`, `'sqrt_one_minus_alphas_cumprod'`, `'posterior_variance'`, `'posterior_mean_coef1'`, `'posterior_mean_coef2'`. All values are NumPy arrays of shape `(num_timesteps,)`.

The autograder will run:

```
from noise_schedule import \
    compute_noise_schedule
import numpy as np

# Test with small T
sched = compute_noise_schedule(
    num_timesteps=10,
    beta_start=0.0001,
    beta_end=0.02)
assert sched['betas'].shape == (10,)
assert sched['alphas'].shape == (10,)
assert sched['alphas_cumprod'].shape == (10,)
assert sched['sqrt_alphas_cumprod'].shape \
    == (10,)
assert sched[
    'sqrt_one_minus_alphas_cumprod'].shape \
    == (10,)
assert sched['posterior_variance'].shape \
    == (10,)
assert sched['posterior_mean_coef1'].shape \
    == (10,)
assert sched['posterior_mean_coef2'].shape \
    == (10,)
```

```
24
25 # alphas_cumprod[0] should be close to 1
26 assert abs(sched['alphas_cumprod'][0]
27          - (1 - 0.0001)) < 1e-6
28 # alphas_cumprod should be strictly decreasing
29 assert all(np.diff(sched['alphas_cumprod'])
30            < 0)
31 # sqrt values should be consistent
32 assert np.allclose(
33     sched['sqrt_alphas_cumprod']**2,
34     sched['alphas_cumprod'])
```

4.3.3 Autograder Summary

File	Required Names	Autograder Tests
`noise_schedule.py`	`compute_noise_schedule`	Shape checks, monotonicity, known initial values

Important Notes:

- Do **not** rename the file or function.
- Do **not** put executable code at the module level. Guard scripts with `if __name__ == '__main__':` blocks.
- Default argument values must match those shown above.
- **Cite DLStudio source code** wherever you borrow or adapt code.

4.4 Code Quality Requirements

- Well-commented code explaining key steps
- Meaningful variable and function names
- Docstrings for all classes and functions
- Code should run without modification (given correct paths and model downloads)
- Follow PEP 8 style guidelines

4.5 Report Formatting

- **Include code snippets in the report for every task.** For each task, show the code you wrote alongside the outputs it produced. If the code snippet is not present, you will lose points on that task.
- Place output directly next to the corresponding code block. Avoid placing code and output in separate sections.
- Use figures and tables with captions
- Number equations, figures, and tables
- Use proper citations when referencing papers or DLStudio

4.6 Additional Guidelines

- Convert Jupyter notebooks to .py files. **Do NOT submit .ipynb**
- If submitting late, submit only once
- **Do NOT submit dataset files or model weights**, only code
- Your code and report must be your own work
- Document your compute setup (CPU/GPU, estimated training times) in your README

5 Frequently Asked Questions (FAQ)

Q: Can I reuse code from previous homeworks?

A: Yes. You are encouraged to reuse plotting utilities, dataset loading code from previous homeworks, etc.

Q: DCGAN training on PurdueShapes5GAN is producing garbage images.

A: Make sure you are using the `weights_init` function from DLStudio. If results are poor after 30 epochs, try a different random seed. The DG1 architecture should produce recognizable shapes after 20–30 epochs.

Q: My CelebA GAN produces garbage / mode collapse.

A: This is common with GANs on natural image datasets. Try: (1) Apply `weights_init`. (2) Use the suggested Adam parameters (`lr=2e-4, betas=(0.5, 0.999)`). (3) Normalize images to $[-1, 1]$. (4) Try label

smoothing (use 0.9 instead of 1.0 for real labels). (5) Train for more epochs. Some mode collapse is expected and acceptable. Discuss it in your analysis.

Q: Diffusion training is extremely slow on my machine.

A: A GPU is strongly recommended. If you only have CPU: (1) use Google Colab, (2) reduce iterations (minimum 20,000), or (3) reduce batch size. Document any changes.

Q: The Stable Diffusion model download is very large.

A: Yes, approximately 5 GB for text-to-image and 4 GB for inpainting. Start the downloads early. Google Colab has fast downloads from HuggingFace.

Q: Stable Diffusion is extremely slow on CPU.

A: Expected (10–20 min per image on CPU). You **must** use a GPU for Task 4. Google Colab's free T4 GPU can generate an image in 5–15 seconds. Use `pipe.enable_attention_slicing()` for low VRAM ($\geq$4 GB).

Q: The inpainting model is a separate download?

A: Yes. `runwayml/stable-diffusion-inpainting` is $\sim$4 GB, separate from the text-to-image model. Download both early. You can alternatively use `StableDiffusionInpaintPipelineLegacy` with the base model (single download), but the dedicated inpainting model produces better results.

Q: How do I create good masks for inpainting?

A: Use PIL's `ImageDraw` to draw rectangles or ellipses. Mask size must match input (512×512). White = inpaint, black = keep. For more precise masks, use OpenCV or an image editor.

Q: Can I use a different Stable Diffusion model?

A: Yes. You may use `stabilityai/stable-diffusion-2-1-base` or any other SD variant on HuggingFace. Document which model you used.

Q: Can I modify DLStudio code directly?

A: Copy relevant code from DLStudio's `GenerativeDiffusion` module into your own file and modify it there. Do **not** modify the installed DLStudio package. Cite the source wherever you reuse code.

Q: Can I work in a group?

A: No. This is individual work.

Q: I don't have a GPU. Can I still do this homework?

A: Tasks 1 and Task 2 subtasks a–b (diffusion theory and noise schedule) run fine on CPU. For Task 2c (running GenerativeDiffusion), Task 3 (CelebA GAN + FID), and Task 4 (Stable Diffusion), a GPU is strongly recommended. Use Google Colab's free GPU if needed.

References

[1] Avinash Kak, *DLStudio Platform*. Available at: https://engineering.purdue.edu/kak/distDLS/

[2] Avinash Kak, *Generative Data Modeling with Adversarial Learning and Diffusion*. Available at: https://engineering.purdue.edu/kak/pdf-kak/GenerativeDataModeling.pdf

[3] Avinash Kak, *Transpose Convolutions and the Encoder-Decoder Architectures for Semantic Segmentation of Images*. Available at: https://engineering.purdue.edu/kak/pdf-kak/SemanticSeg.pdf

[4] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, *Generative Adversarial Nets*, NeurIPS, 2014. Available at: https://arxiv.org/abs/1406.2661

[5] Jonathan Ho, Ajay Jain, and Pieter Abbeel, *Denoising Diffusion Probabilistic Models*, NeurIPS, 2020. Available at: https://arxiv.org/abs/2006.11239

[6] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer, *High-Resolution Image Synthesis with Latent Diffusion Models*, CVPR, 2022. Available at: https://arxiv.org/abs/2112.10752

[7] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever, *Learning Transferable Visual Models From Natural Language Supervision*, ICML, 2021. Available at: https://arxiv.org/abs/2103.00020

[8] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter, *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*, NeurIPS, 2017. Available at: https://arxiv.org/abs/1706.08500

[9] Maximilian Seitzer, *pytorch-fid: FID Score for PyTorch*. https://github.com/mseitzer/pytorch-fid, 2020. Version 0.3.0.