

Homework 8 Report

Task 1: DCGAN Warm-Up on PurdueShapes5GAN

3.1.1 Running DCGAN (DG1)

I trained DG1 for 30 epochs using the DLStudio example script and captured outputs in `outputs/task1`. The run log is `outputs/task1/dg1_train.log`, which confirms training completion through epoch 30.

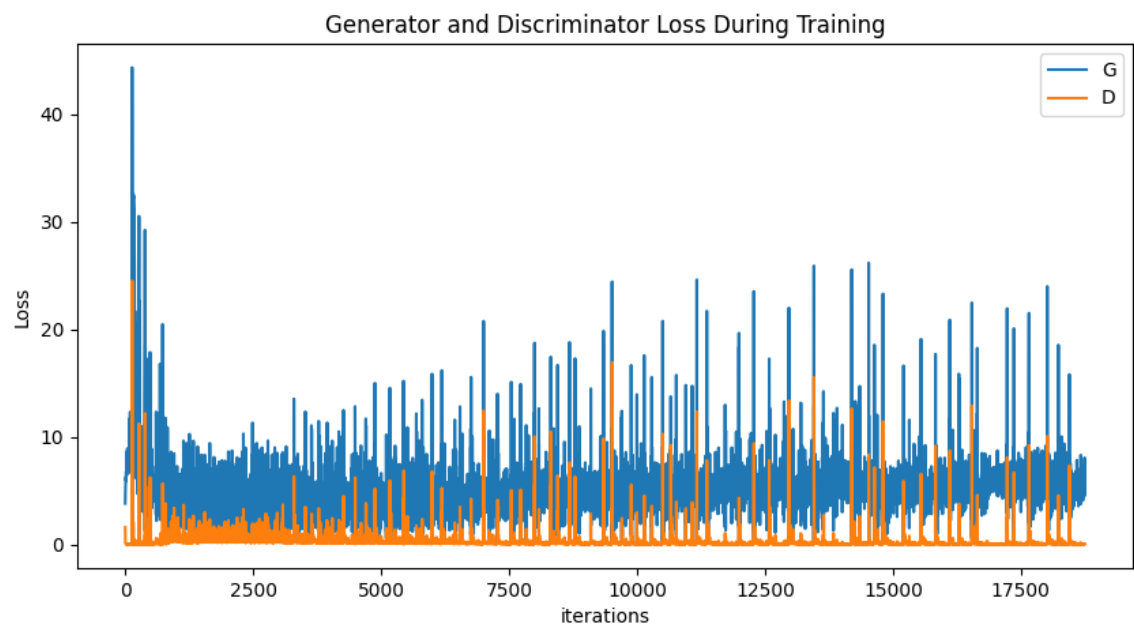
The exact command used is shown below.

```
python hw8_task1.py run \
  --examples_adv_dir ./DLStudio-2.5.6/ExamplesAdversarialLearning \
  --out_dir ./outputs/task1 \
  --log_path ./outputs/task1/dg1_train.log
```

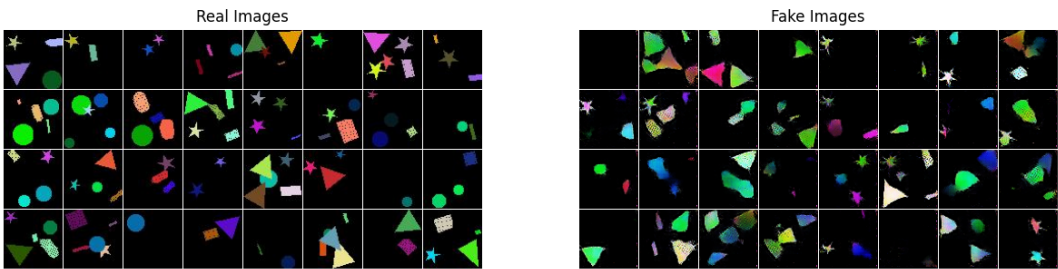
The helper above launches `dcgan_DG1.py`, then collects DG1 artifacts into `outputs/task1/results_DG1`. The key Python call inside the helper is:

```
def run_dg1_example(examples_adv_dir: str, log_path: str | None = None) ->
    None:
    examples_dir = os.path.abspath(examples_adv_dir)
    script = os.path.join(examples_dir, "dcgan_DG1.py")
    cmd = [sys.executable, script]
    with subprocess.Popen(
        cmd, cwd=examples_dir, stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT, text=True
    ) as proc:
        ...
```

(b) Discriminator and Generator loss curves (overlaid)



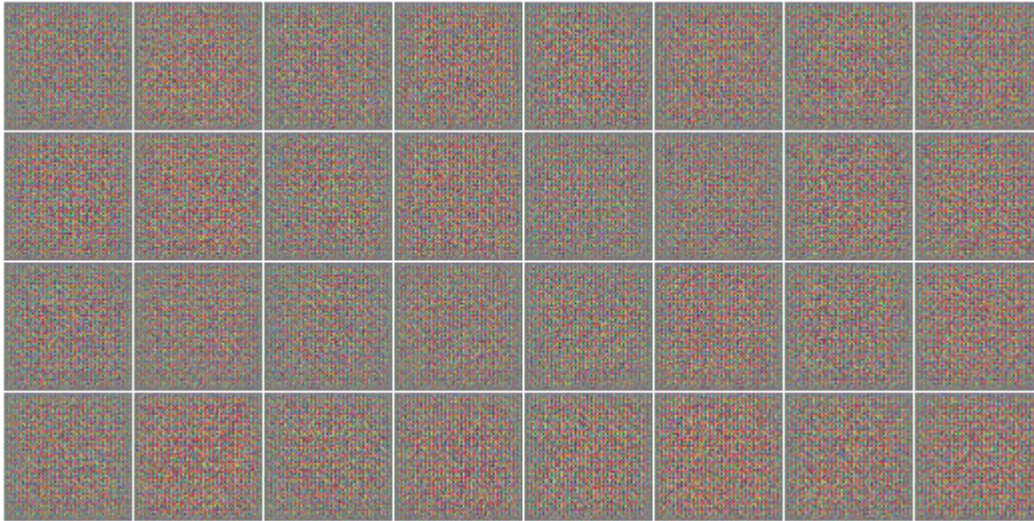
(c) Side-by-side real vs generated images



(d) Generator output evolution (same noise vectors, 4 snapshots)



For completeness, the original training animation used to derive snapshots:



3.1.2 Understanding the Architecture and Training

(e) Generator dimension trace through ConvTranspose2d

The DG1 generator starts from latent noise with shape $(B, 100, 1, 1)$ and uses five transposed-convolution layers to generate an RGB 64×64 image. The spatial size follows:

$$H_{\text{out}} = (H_{\text{in}} - 1) * s + k - 2p$$

with first layer $(k=4, s=1, p=0)$ and remaining layers $(k=4, s=2, p=1)$. Therefore, the progression is $1 \rightarrow 4 \rightarrow 8 \rightarrow 16 \rightarrow 32 \rightarrow 64$, where the first two transitions are explicitly $1 \rightarrow 4$ from $(1-1)*1+4-0$ and $4 \rightarrow 8$ from $(4-1)*2+4-2$. Channel depth changes as $100 \rightarrow 512 \rightarrow 256 \rightarrow 128 \rightarrow 64 \rightarrow 3$. The output uses `Tanh`, which maps generated pixels to $[-1, 1]$ so fake images are in the same range as normalized real training images.

(f) Minimax training explanation (targets for each phase)

Each iteration alternates between discriminator and generator objectives. In discriminator phase one, real images are labeled `1`, so the model is pushed to predict high probability for real data. In discriminator phase two, generated images are labeled `0`, so the model is pushed to reject current generator outputs. In generator training, fake images are passed through the discriminator again but with target `1`; this is the non-saturating objective that pushes the generator toward outputs that the discriminator would classify as real. This alternating process is the practical minimax game between `D` and `G`.

(g) Why `fakes.detach()` is used

During the discriminator fake phase, `fakes.detach()` prevents gradients from flowing back into the generator. This is necessary because, in that phase, only discriminator parameters should be updated. If `detach()` were removed, gradients from the discriminator's fake-classification loss would also affect generator parameters at the wrong time, mixing objectives and making training less stable.

Full Code Used (Copied In Full)

The following sections paste the full source code files used for Task 1 so this report is self-contained.

Full file: `hw8_task1.py`


```
#!/usr/bin/env python3
"""
Homework 8 – Task 1: DCGAN warm-up on PurdueShapes5GAN (DLStudio DG1).

This helper script:
- runs DLStudio's ExamplesAdversarialLearning/dcgan_DG1.py
- copies produced deliverables into a user output folder
- creates a 4-snapshot panel from generation_animation.gif
- writes ready-to-edit theory notes for Task 3.1.2
"""

from __future__ import annotations

import argparse
import os
import shutil
import subprocess
import sys
from pathlib import Path
from typing import List, Sequence

import matplotlib

matplotlib.use("Agg")
import matplotlib.pyplot as plt
import numpy as np
from PIL import Image, ImageSequence

def run_dg1_example(examples_adv_dir: str, log_path: str | None = None) ->
None:
    """
    Run DLStudio DG1 example script (30 epochs configured in script by
    default).
    """
    examples_dir = os.path.abspath(examples_adv_dir)
    script = os.path.join(examples_dir, "dcgan_DG1.py")
    if not os.path.isfile(script):
        raise FileNotFoundError(f"Missing script: {script}")

    env = os.environ.copy()
    env["PYTHONUNBUFFERED"] = "1"
    cmd = [sys.executable, script]

    print(f"Running: {' '.join(cmd)}")
    with subprocess.Popen(
        cmd,
        cwd=examples_dir,
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
        text=True,
        bufsize=1,
```

```

        env=env,
    ) as proc:
        fh = open(log_path, "w", encoding="utf-8") if log_path else None
        try:
            assert proc.stdout is not None
            for line in proc.stdout:
                print(line, end="")
                if fh is not None:
                    fh.write(line)
            rc = proc.wait()
            if rc != 0:
                raise RuntimeError(f"drgan_DG1.py failed with code {rc}")
        finally:
            if fh is not None:
                fh.close()

def copy_dg1_results(examples_adv_dir: str, out_dir: str) -> str:
    """
    Copy ExamplesAdversarialLearning/results_DG1 into out_dir/results_DG1.
    """
    examples_dir = os.path.abspath(examples_adv_dir)
    src = os.path.join(examples_dir, "results_DG1")
    if not os.path.isdir(src):
        raise FileNotFoundError(
            f"Missing {src}. Run DG1 first or check script output folder."
        )
    dst = os.path.join(os.path.abspath(out_dir), "results_DG1")
    os.makedirs(os.path.abspath(out_dir), exist_ok=True)
    if os.path.exists(dst):
        shutil.rmtree(dst)
    shutil.copytree(src, dst)
    return dst

def make_four_snapshots_from_gif(gif_path: str, out_path: str) -> None:
    """
    Build a 1x4 snapshot panel from generation_animation.gif.
    Picks evenly spaced frames to show progression (e.g., early→late
    training).
    """
    if not os.path.isfile(gif_path):
        raise FileNotFoundError(gif_path)
    gif = Image.open(gif_path)
    frames = [frm.copy().convert("RGB") for frm in
ImageSequence.Iterator(gif)]
    if len(frames) < 4:
        raise ValueError(f"Need at least 4 frames in GIF, found
{len(frames)}")

    idxs = [0, len(frames) // 3, (2 * len(frames)) // 3, len(frames) - 1]
    picks = [frames[i] for i in idxs]

    fig, axes = plt.subplots(1, 4, figsize=(16, 4.2))

```

```

labels = ["snapshot-1", "snapshot-2", "snapshot-3", "snapshot-4"]
for ax, im, lab in zip(axes, picks, labels):
    ax.imshow(np.asarray(im))
    ax.axis("off")
    ax.set_title(lab, fontsize=10)
fig.suptitle("DG1 Generator evolution (4 snapshots from training
animation)", fontsize=11)
plt.tight_layout()
os.makedirs(os.path.dirname(os.path.abspath(out_path)), exist_ok=True)
plt.savefig(out_path, dpi=150, bbox_inches="tight")
plt.close()

```

```

def write_task1_notes(out_dir: str) -> str:
    """
    Write ready-to-edit explanations for Task 3.1.2.
    """
    os.makedirs(out_dir, exist_ok=True)
    path = os.path.join(out_dir, "task1_theory_notes.txt")
    text = """Task 3.1.2 – Draft notes (edit to your own wording)

```

1) Generator architecture and spatial trace:

The DG1 generator starts from noise shaped (B,100,1,1). It uses ConvTranspose2d layers:

- 1x1 -> 4x4 with k=4,s=1,p=0
- 4x4 -> 8x8 with k=4,s=2,p=1
- 8x8 -> 16x16 with k=4,s=2,p=1
- 16x16 -> 32x32 with k=4,s=2,p=1
- 32x32 -> 64x64 with k=4,s=2,p=1

with channel flow 100->512->256->128->64->3. The output uses Tanh so image values are in [-1,1],

matching the dataset normalization used in DCGAN training.

2) Minimax game / targets:

For discriminator updates, there are two phases in each iteration:

- Real images: target label = 1
- Fake images from G(z): target label = 0

This trains D to distinguish true dataset samples from generated samples.

For generator update, D(G(z)) is compared against target label = 1. This means G is trained

to produce images that D classifies as real, i.e., to fool D.

3) Why fakes.detach() for discriminator-on-fakes:

When training D with fake images, fakes.detach() breaks gradient flow into G. So backprop updates

only D parameters in that phase. If detach() is removed, gradients from D's fake-loss also flow

into G during D step, mixing objectives and destabilizing the alternating min-max updates.

```

"""
    with open(path, "w", encoding="utf-8") as f:
        f.write(text)
    return path

```

```

def validate_task1_outputs(results_dir: str) -> List[str]:
    """
    Validate presence of expected files from DG1 run.
    """
    expected = [
        "gen_and_disc_loss_training.png",
        "real_vs_fake_images.png",
        "generation_animation.gif",
    ]
    missing: List[str] = []
    for name in expected:
        p = os.path.join(results_dir, name)
        if not os.path.isfile(p):
            missing.append(name)
    return missing

def build_parser() -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(description="HW8 Task 1 helper")
    sub = p.add_subparsers(dest="command", required=True)

    r = sub.add_parser("run", help="Run DLStudio DG1 and prepare deliverables")
    r.add_argument(
        "--examples_adv_dir",
        type=str,
        default="./DLStudio-2.5.6/ExamplesAdversarialLearning",
    )
    r.add_argument("--out_dir", type=str, default="./outputs/task1")
    r.add_argument("--log_path", type=str,
        default="./outputs/task1/dg1_train.log")

    c = sub.add_parser("collect", help="Collect existing DG1 outputs without rerunning")
    c.add_argument(
        "--examples_adv_dir",
        type=str,
        default="./DLStudio-2.5.6/ExamplesAdversarialLearning",
    )
    c.add_argument("--out_dir", type=str, default="./outputs/task1")

    return p

def _collect_and_postprocess(examples_adv_dir: str, out_dir: str) -> None:
    dst_results = copy_dg1_results(examples_adv_dir, out_dir)
    missing = validate_task1_outputs(dst_results)
    if missing:
        raise FileNotFoundError(
            f"Missing expected DG1 output files in {dst_results}:
{missing}"
        )
    gif_path = os.path.join(dst_results, "generation_animation.gif")

```

```

snap_path = os.path.join(dst_results, "generator_snapshots_4panel.png")
make_four_snapshots_from_gif(gif_path, snap_path)
notes = write_task1_notes(os.path.abspath(out_dir))
print(f"Collected results at: {dst_results}")
print(f"Created snapshot panel: {snap_path}")
print(f"Wrote notes file: {notes}")

def main(argv: Sequence[str] | None = None) -> None:
    args = build_parser().parse_args(argv if argv is not None else
sys.argv[1:])
    if args.command == "run":
        os.makedirs(os.path.abspath(args.out_dir), exist_ok=True)
        run_dg1_example(args.examples_adv_dir, args.log_path)
        _collect_and_postprocess(args.examples_adv_dir, args.out_dir)
    elif args.command == "collect":
        os.makedirs(os.path.abspath(args.out_dir), exist_ok=True)
        _collect_and_postprocess(args.examples_adv_dir, args.out_dir)

if __name__ == "__main__":
    main()

```

Full file: [DLStudio-2.5.6/ExamplesAdversarialLearning/dcgan_DG1.py](#)

```

#!/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.benchmark=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 =
    "/mnt/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("\n\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_data_loader()

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")

```

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

Source of results: Task 2 figures and logs come from `hw8_task2.py` / `hw8_task2.ipynb` and saved outputs under `outputs/task2/`.

Deliverable checklist (a)–(k): (a)–(d) theory explanation in §3.2.1, (e) code in §3.2.2, (f)–(g) noise-schedule/forward-process figures and observations in §3.2.2, (h)–(j) DLStudio training and generation figures in §3.2.3, and (k) diffusion-vs-DCGAN comparison in §3.2.3.

3.2.1 Understanding the Forward and Reverse Processes (7 points)

(a) Text: Two Markov chains description with diagram

Diffusion models use two stochastic chains over timestep index $t = 0, 1, \dots, T$:

- **Forward chain q (noising):** starts from a clean data sample x_0 and gradually adds Gaussian noise until x_T is close to isotropic Gaussian noise.
- **Reverse chain p_θ (denoising):** starts from $x_T \sim N(0, I)$ and iteratively denoises to produce x_{0_hat} , a synthesized sample.

Text diagram (matching the lecture intuition):

$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_T$ (forward q , increasing corruption)
 $x_T \rightarrow x_{T-1} \rightarrow \dots \rightarrow x_2 \rightarrow x_1 \rightarrow x_{0_hat}$ (reverse p_θ , learned denoising)

The parameter T controls how long and fine-grained the chain is: larger T means smaller per-step corruption/denoising increments and a smoother diffusion trajectory.

(b) Text: Forward transition probability explanation

The one-step forward transition (Slide Eq. 70) is:

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

Here, β_t is the **noise schedule** (variance injected at step t). As t increases, cumulative signal retention decreases (α_{bar}_t shrinks), so visual structure fades and samples approach pure noise near T .

(c) Text: “Jump to any timestep” formula explanation

The closed-form noising relation (Slides Eq. 71–72):

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

is computationally useful because it lets us sample x_t **directly** from x_0 without simulating all intermediate steps $1 \dots t-1$. This is important during training, where each batch uses a random timestep.

- $\alpha_t = 1 - \beta_t$
- $\alpha_{\bar{t}} = \prod_{s=1}^t \alpha_s$ (cumulative signal retention up to timestep t)

(d) Text: Neural network learning target explanation

At each training iteration, a random timestep t is sampled; the model receives x_t and t and predicts the injected Gaussian noise ϵ (equivalently, predicts a denoising direction). The standard MSE objective compares predicted noise to ground-truth sampled noise:

$$[\mathcal{L} = \mathbb{E}_{\{x_0, \epsilon, t\}} [\|\epsilon - \epsilon_{\theta}(x_t, t)\|_2^2].]$$

Minimizing this loss teaches the network how to invert the forward corruption across many noise levels, so reverse-time updates remove noise while preserving data-consistent structure.

3.2.2 Noise Schedule Computation and Forward Diffusion Visualization (7 points)**(e) Code: Complete `compute_noise_schedule` implementation**

The implemented function in `hw8_task2.py` is:

```
def compute_noise_schedule(
    num_timesteps: int = 1000,
    beta_start: float = 0.0001,
    beta_end: float = 0.02,
) -> Dict[str, np.ndarray]:
    """
    Compute diffusion alpha-beta quantities (Slides 169-171 style).
    """
    betas = np.linspace(beta_start, beta_end, num_timesteps,
dtype=np.float64)
    alphas = 1.0 - betas
    alphas_cumprod = np.cumprod(alphas, axis=0)
    alphas_cumprod_prev = np.append(1.0, alphas_cumprod[:-1])

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

    # q(x_{t-1} | x_t, x_0) posterior terms
    posterior_variance = (
        betas * (1.0 - alphas_cumprod_prev) / (1.0 - alphas_cumprod)
    )
    posterior_mean_coef1 = (
```

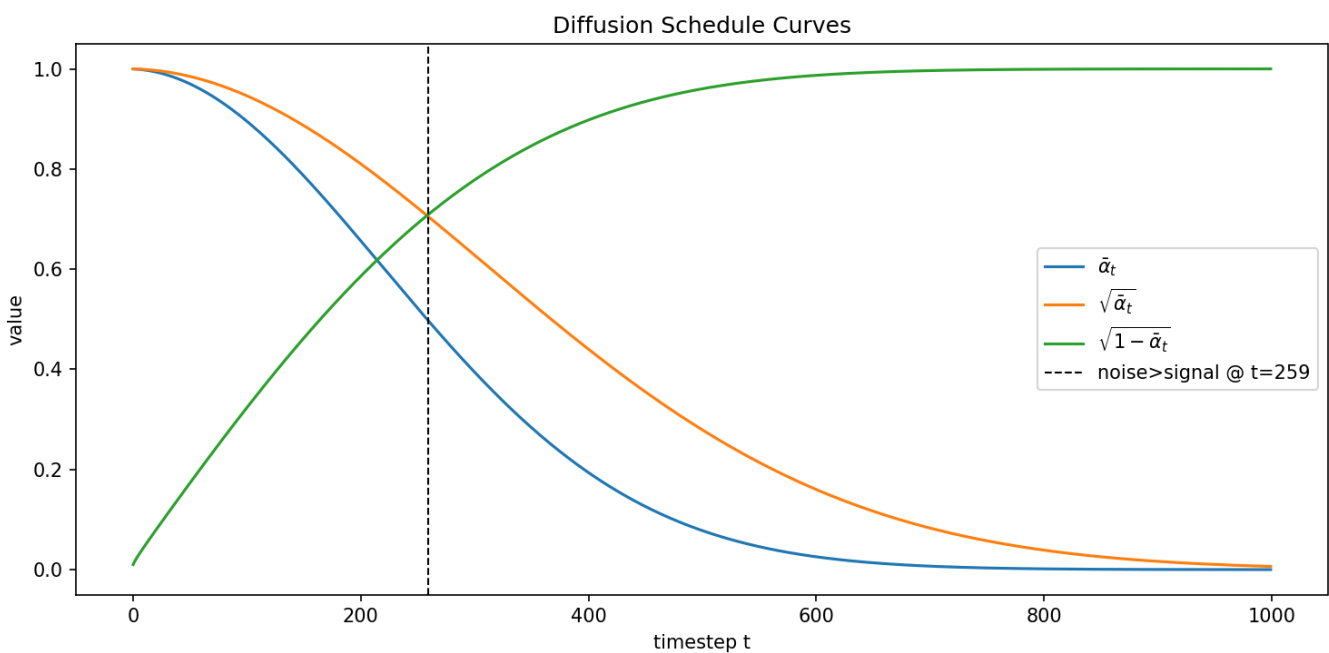
```

        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,
    }

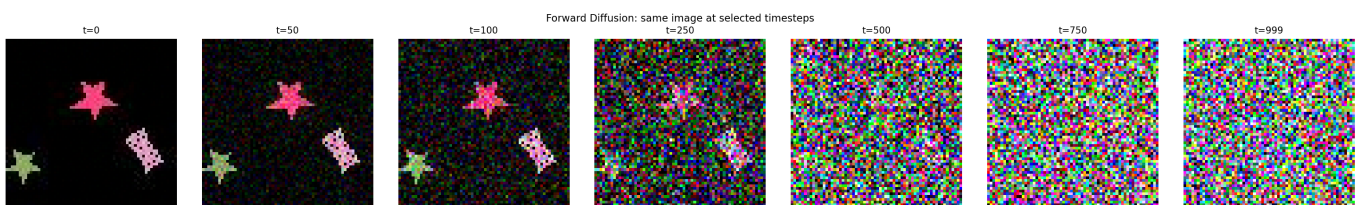
```

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



Interpretation: α_{bar_t} and $\sqrt{\alpha_{\text{bar}_t}}$ decrease monotonically, while $\sqrt{1-\alpha_{\text{bar}_t}}$ increases. The crossover where noise RMS overtakes signal RMS ($\sqrt{1-\alpha_{\text{bar}_t}} > \sqrt{\alpha_{\text{bar}_t}}$, i.e., $\alpha_{\text{bar}_t} < 0.5$) occurs roughly in the first third of the chain (around $t \sim 250-300$ for this linear schedule), after which corruption dominates perceptual content.

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

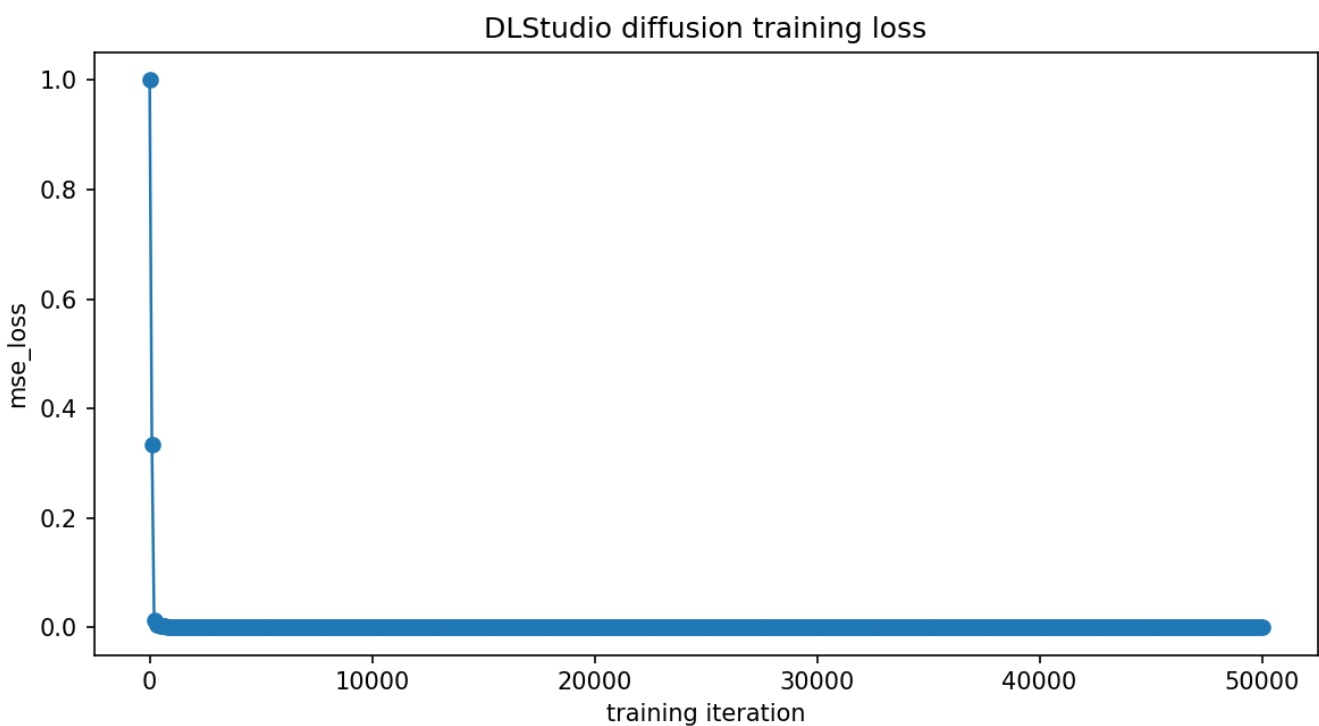


Observation: At small timesteps ($t=0, 50$), object outlines are still clear. Around intermediate timesteps ($t\sim 250-500$), shapes become difficult to parse and class-specific structure fades. By late timesteps ($t=750, 999$), the sample is effectively unrecognizable and visually close to random noise.

3.2.3 Running DLStudio's GenerativeDiffusion (6 points)

Compute setup: run on the course cluster (Slurm-managed environment), using GPU-enabled jobs for diffusion training/generation. Training was resumed from existing checkpoints and run through 40,000 iterations; checkpoints in `DLStudio-2.5.6/ExamplesDiffusion/RESULTS/` include `ema_0.9999_020000.pt` and `ema_0.9999_040000.pt`.

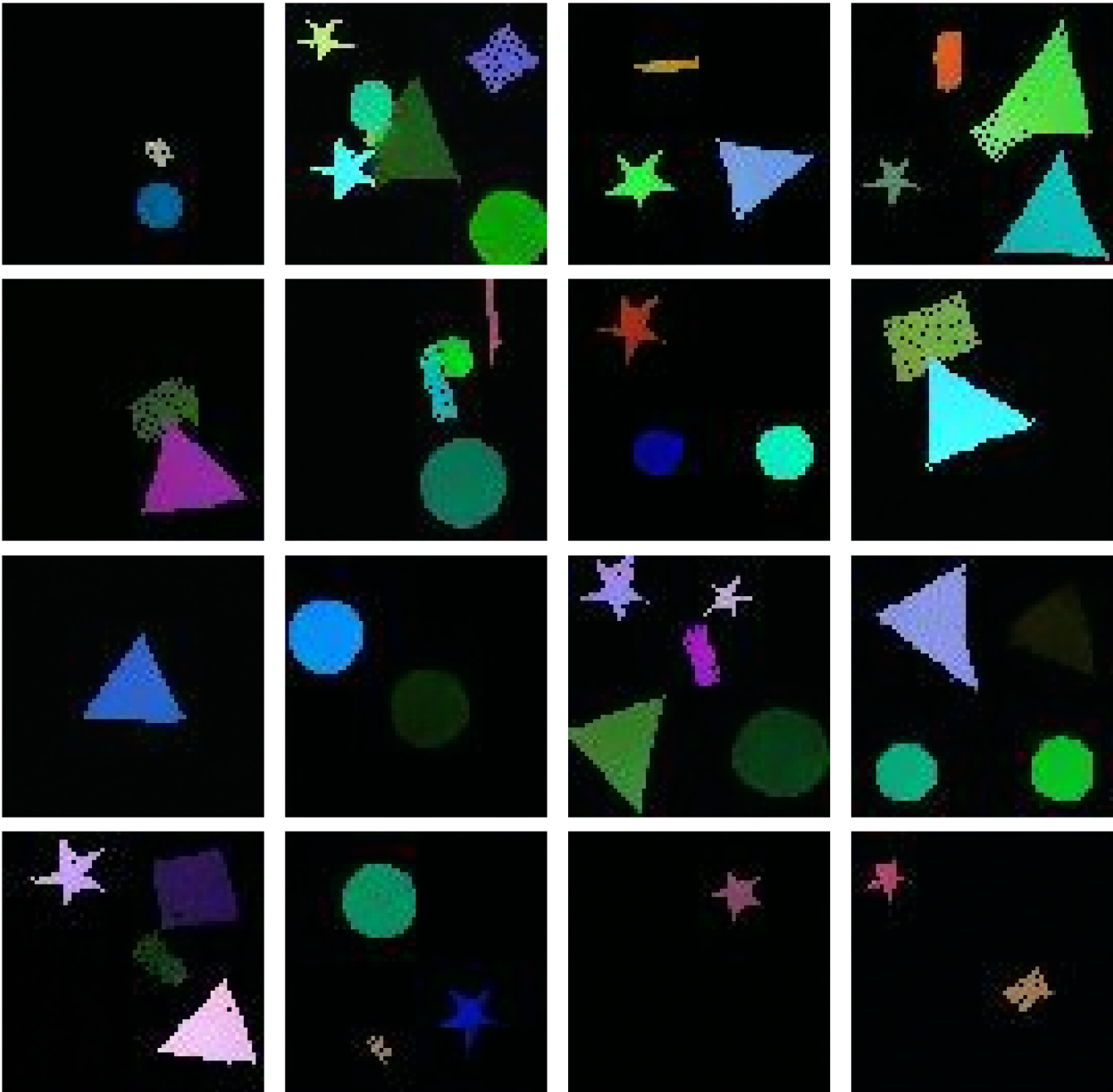
(h) Figure: Diffusion training loss curve



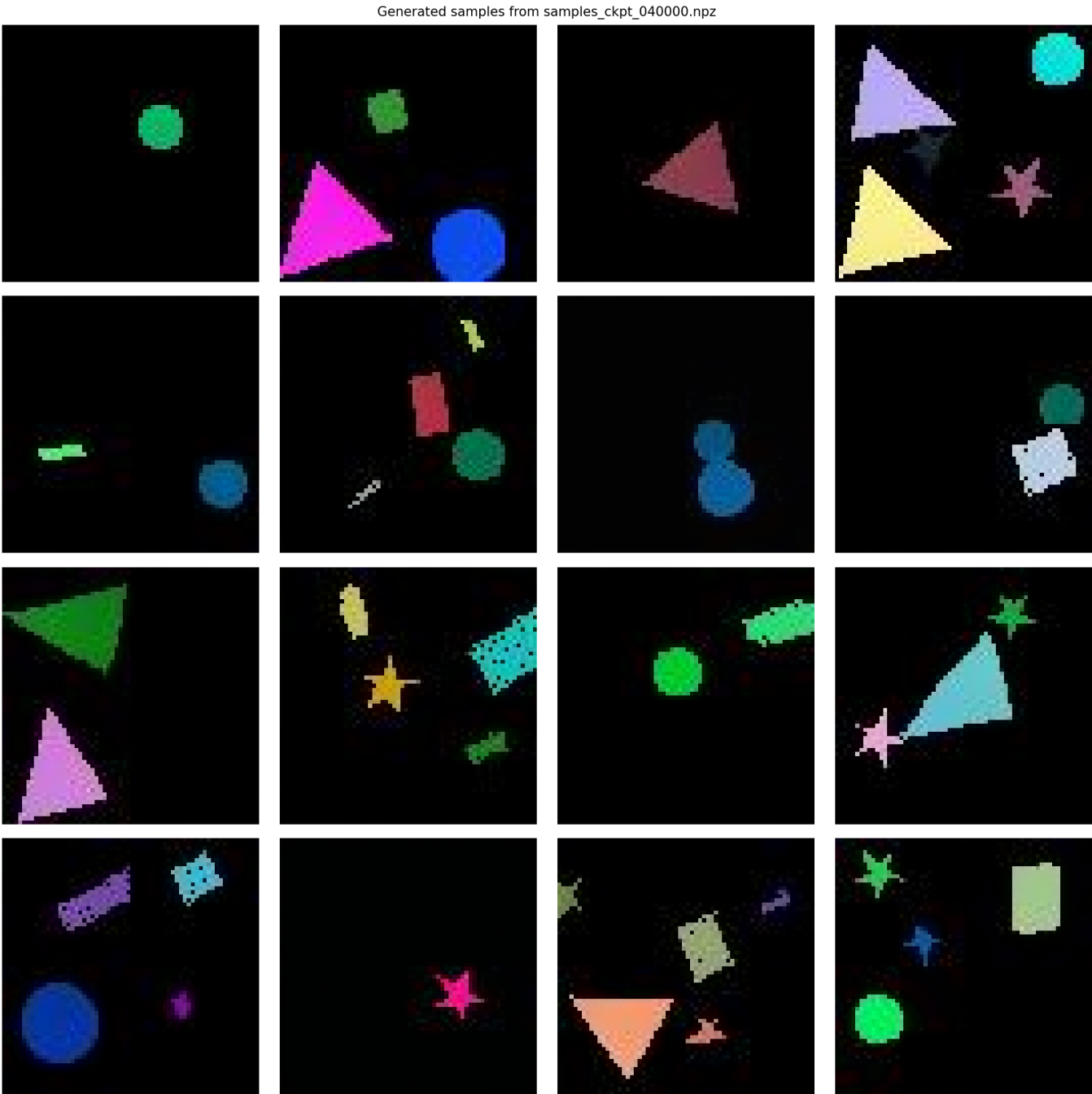
(i) Figure: Generated image grids from 2+ checkpoints

Checkpoint 20,000:

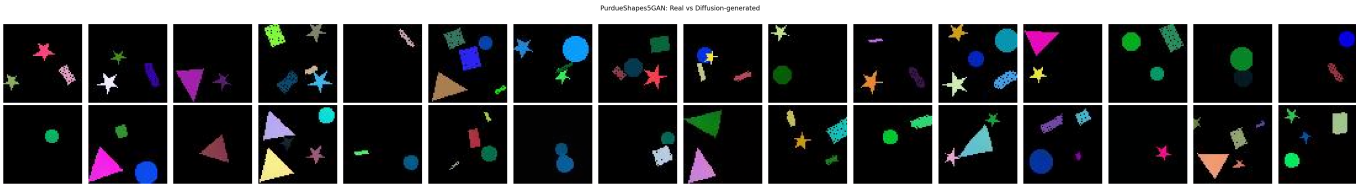
Generated samples from samples_ckpt_020000.npz



Checkpoint 40,000:



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



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

Compared with Task 1 DCGAN samples, the diffusion outputs at later checkpoints are typically more stable in global shape composition and show fewer severe mode-collapse-like repetitions. On PurdueShapes-style data, diffusion’s iterative denoising produces cleaner object boundaries once training progresses, while early-checkpoint samples remain blurrier/noisier as expected.

Relative to DCGAN, diffusion training is slower (many denoising steps and heavier compute), but sample quality improves predictably with checkpoint progression (20k → 40k). DCGAN remains faster at inference (single forward pass), yet diffusion generally offers better diversity/consistency trade-offs for this dataset when enough training iterations are used.

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

Source of results: Figures and metrics for this task (including FID) were produced in `hw8_task3.ipynb`; the same code paths are available as `hw8_task3.py`. Saved artifacts below match the notebook run under `outputs/task3_notebook/`.

Deliverable checklist (a)–(l): (a)–(c) code blocks in §3.3.1; (d)–(f) figures in §3.3.1; (g) three analyses in §3.3.1; (h)–(i) diffusion figure and comparison in §3.3.2; (j)–(l) FID script, table, and discussion in §3.3.3 (plus optional FID sample grids from the assignment PDF).

This section uses the implementation in `hw8_task3.py` (same API as described in the assignment; the script name differs from the notebook’s historical `hw8.py` alias). Training and sampling artifacts live under `outputs/task3_notebook/`. Real CelebA-64 images for training and FID were taken from `DLStudio-2.5.6/celeba_dataset_64x64` (10,000 JPGs, resized and normalized to `[-1, 1]` in code). The DCGAN was trained for **50 epochs** with **batch size 32**, latent dimension **100**, and **Adam** with learning rate 2×10^{-4} and **betas (0.5, 0.999)**. The discriminator uses a slightly lower learning rate (half of the generator rate) and one-sided label smoothing on real labels (`0.9` vs `1.0`) for stability; the loss is **binary cross-entropy on logits** (`nn.BCEWithLogitsLoss`), which is equivalent to `BCELoss` after a **Sigmoid** on the discriminator output.

3.3.1 Design and Train Your Own DCGAN (20 points)

(a) Generator and Discriminator class definitions

```
import torch
import torch.nn as nn

class CelebA64Generator(nn.Module):
    """
    Maps a 100×1×1 noise tensor to 3×64×64 RGB in [-1, 1] via
    ConvTranspose2d
    stack (DCGAN 4-2-1), with BatchNorm on all but the last conv (no BN
    before Tanh).
    """
    def __init__(self, nz: int = 100) -> None:
        super().__init__()
        self.nz = nz
        self.latent_to_image = nn.ConvTranspose2d(
            nz, 512, kernel_size=4, stride=1, padding=0, bias=False
        )
        self.up1 = nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2,
            padding=1, bias=False)
```



```

        self.up2 = nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2,
padding=1, bias=False)
        self.up3 = nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2,
padding=1, bias=False)
        self.up4 = nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2,
padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(512)
        self.bn2 = nn.BatchNorm2d(256)
        self.bn3 = nn.BatchNorm2d(128)
        self.bn4 = nn.BatchNorm2d(64)
        self.tanh = nn.Tanh()

```

```

def forward(self, x: torch.Tensor) -> torch.Tensor:
    x = self.latent_to_image(x)
    x = torch.relu(self.bn1(x))
    x = self.up1(x)
    x = torch.relu(self.bn2(x))
    x = self.up2(x)
    x = torch.relu(self.bn3(x))
    x = self.up3(x)
    x = torch.relu(self.bn4(x))
    x = self.up4(x)
    return self.tanh(x)

```

```

class CelebA64Discriminator(nn.Module):

```

```

    """

```

```

    Strided convolutions 64→32→16→8→4→1; BatchNorm after the first conv
only

```

```

    (no BN on the first layer, per DCGAN). LeakyReLU; final scalar logit
per image.

```

```

    """

```

```

def __init__(self) -> None:

```

```

    super().__init__()

```

```

    self.conv1 = nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1)

```

```

    self.conv2 = nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1)

```

```

    self.conv3 = nn.Conv2d(128, 256, kernel_size=4, stride=2,
padding=1)

```

```

    self.conv4 = nn.Conv2d(256, 512, kernel_size=4, stride=2,
padding=1)

```

```

    self.conv5 = nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0)

```

```

    self.bn2 = nn.BatchNorm2d(128)

```

```

    self.bn3 = nn.BatchNorm2d(256)

```

```

    self.bn4 = nn.BatchNorm2d(512)

```

```

def forward(self, x: torch.Tensor) -> torch.Tensor:

```

```

    x = nn.functional.leaky_relu(self.conv1(x), 0.2, inplace=True)

```

```

    x = self.bn2(self.conv2(x))

```

```

    x = nn.functional.leaky_relu(x, 0.2, inplace=True)

```

```

    x = self.bn3(self.conv3(x))

```

```

    x = nn.functional.leaky_relu(x, 0.2, inplace=True)

```

```

    x = self.bn4(self.conv4(x))

```

```

    x = nn.functional.leaky_relu(x, 0.2, inplace=True)

```

```
x = self.conv5(x)
return x
```

(b) Learnable parameter counts and Generator dimension trace

Parameter counts. Values are written by the training script to `outputs/task3_notebook/training_meta.txt`.

Network	Learnable parameters
Generator <code>CelebA64Generator</code>	3,576,704
Discriminator <code>CelebA64Discriminator</code>	2,766,529

Generator spatial dimension trace (ConvTranspose2d).

For square feature maps, the transposed-convolution output size is

$$(H_{\text{out}}) = (H_{\text{in}} - 1) \cdot s + k - 2p$$

(when `output_padding=0`). Starting from noise reshaped to `(B, 100, 1, 1)`:

Stage	Layer	(H_{in})	k, s, p	(H_{out})	Channels
1	<code>latent_to_image</code>	1	4, 1, 0	$((1-1) \cdot 1 + 4 = 4)$	512
2	<code>up1</code>	4	4, 2, 1	$((4-1) \cdot 2 + 4 - 2 = 8)$	256
3	<code>up2</code>	8	4, 2, 1	16	128
4	<code>up3</code>	16	4, 2, 1	32	64
5	<code>up4</code>	32	4, 2, 1	64	3 → Tanh

The same trace is saved as plain text in `outputs/task3_notebook/generator_spatial_trace.txt`.

(c) Adversarial training loop (complete training code excerpt)

DCGAN-style initialization (`weights_init`), dataloading, optimizers, and the alternating discriminator and generator updates (with `fakes.detach()` on the discriminator’s fake pass) are implemented in `train_dcgan` in `hw8_task3.py`. After training, losses are plotted with `plt.plot(D_losses, label="Discriminator")` and `plt.plot(G_losses, label="Generator")` plus `plt.legend()`. Setup and inner loop:

```
import torch
import torch.nn as nn
import torch.optim as optim

netG = CelebA64Generator(nz=nz).to(device)
netD = CelebA64Discriminator().to(device)
optimD = optim.Adam(netD.parameters(), lr=lr * 0.5, betas=(beta1, 0.999))
optimG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

```

criterion = nn.BCEWithLogitsLoss()
real_label = 0.9 # one-sided label smoothing
fake_label = 0.0

for epoch in range(epochs):
    for i, real_cpu in enumerate(dataloader):
        real_images = real_cpu.to(device)
        b_size = real_images.size(0)

        # ----- Discriminator -----
        netD.zero_grad()
        label = torch.full((b_size,), real_label, dtype=torch.float,
device=device)
        output = netD(real_images).view(-1)
        loss_d_real = criterion(output, label)
        loss_d_real.backward()

        noise = torch.randn(b_size, nz, 1, 1, device=device)
        fakes = netG(noise)
        label.fill_(fake_label)
        output = netD(fakes.detach()).view(-1)
        loss_d_fake = criterion(output, label)
        loss_d_fake.backward()
        loss_d = loss_d_real + loss_d_fake
        optimD.step()

        # ----- Generator -----
        netG.zero_grad()
        label.fill_(real_label)
        output = netD(fakes).view(-1)
        loss_g = criterion(output, label)
        loss_g.backward()
        optimG.step()

        G_losses.append(loss_g.item())
        D_losses.append(loss_d.item())

```

Fixed noise (16, 100, 1, 1) is used for snapshot grids at epochs 1, 10, 25, and 50; losses are plotted per **iteration** index.

Example training command:

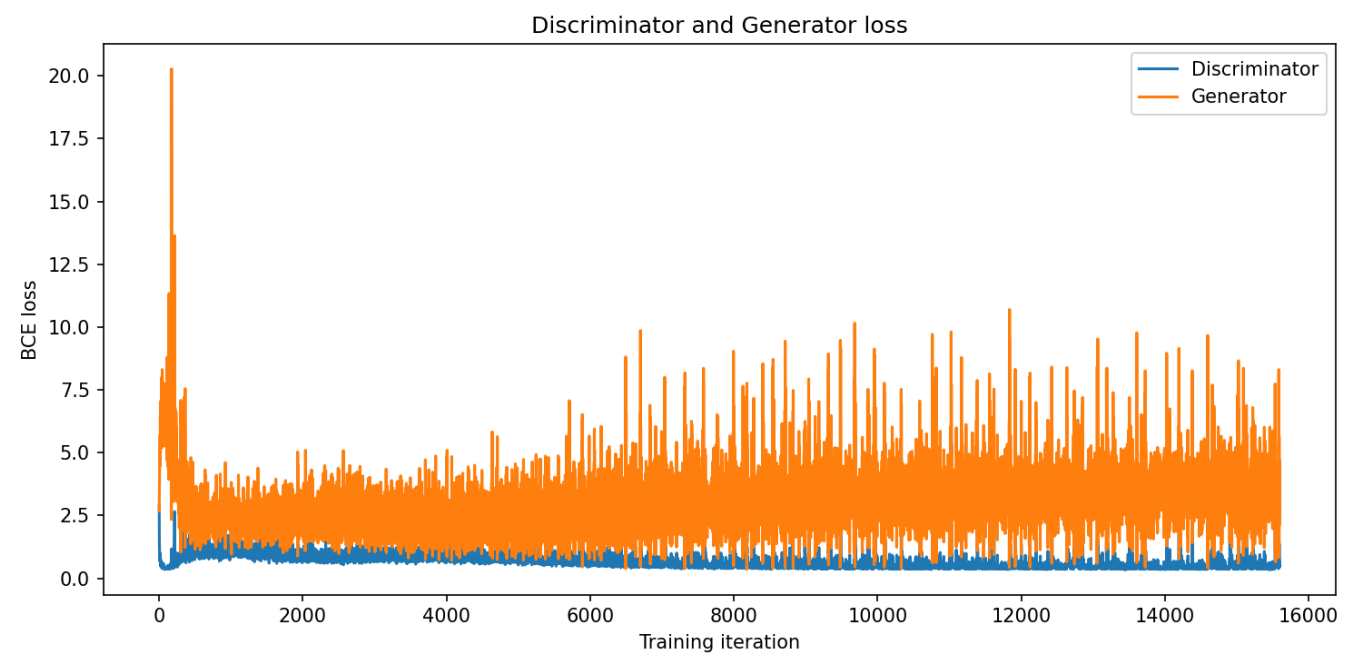
```

python hw8_task3.py train \
--celeba_root ./DLStudio-2.5.6/celeba_dataset_64x64 \
--out_dir ./outputs/task3_notebook \
--epochs 50 \
--batch_size 32

```

(d) Figure: discriminator and generator loss curves (overlaid, with legend)

The figure plots both losses on the same axes for every training iteration; the Matplotlib legend identifies the **Discriminator** and **Generator** curves.

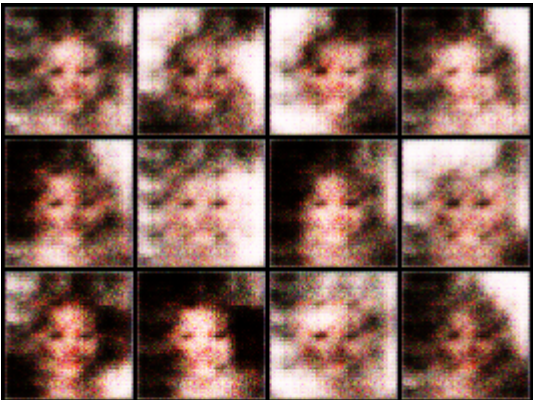


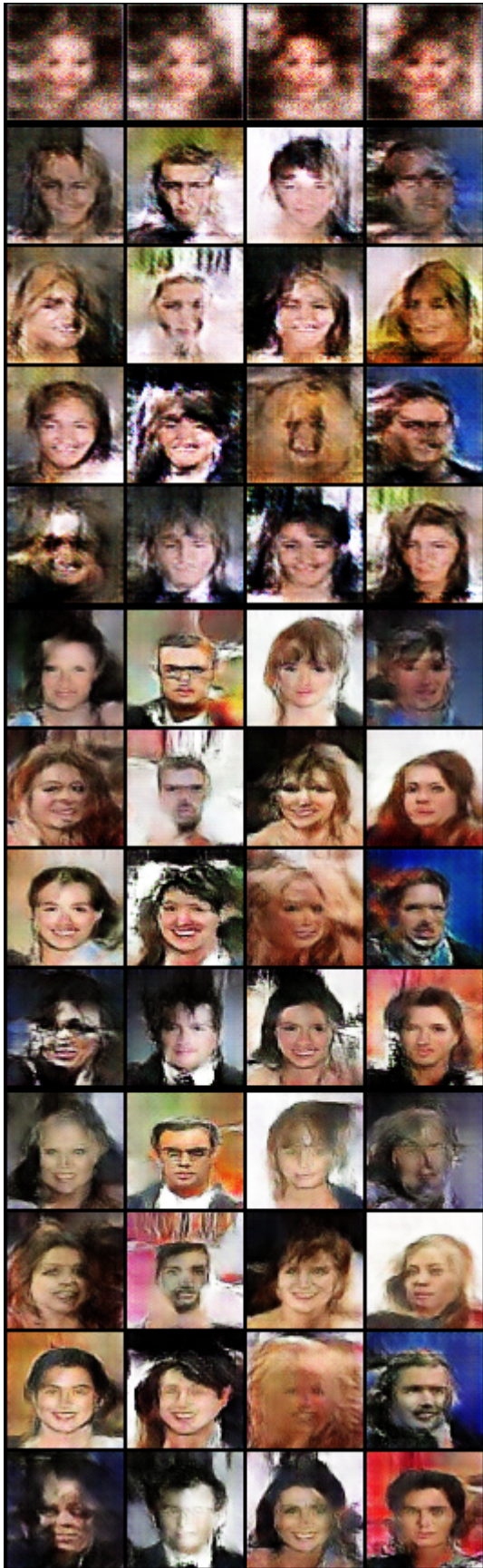
(e) Figure: 4×4 grid of generated faces at end of training



(f) Figure: generator evolution (same noise vectors, epochs 1, 10, 25, 50)

The stacked image concatenates one 4×4 grid per snapshot epoch (top to bottom: increasing epoch).





Individual snapshot files: [outputs/task3_notebook/snapshot_epoch_001.png](#),
[snapshot_epoch_010.png](#), [snapshot_epoch_025.png](#), [snapshot_epoch_050.png](#).

(g) Text: training dynamics, visual quality, and mode collapse (1-2 paragraphs each)

Training dynamics. Early in training both losses move quickly as the discriminator learns a crude boundary and the generator finds coarse structure. After the first few epochs, discriminator loss tends to stay in a moderate band while generator loss fluctuates; this oscillation is typical of GAN training and does not indicate failure by itself. Occasional spikes in either curve reflect batch-to-batch variation and the adversarial coupling between networks. Overall the loss curves do not show a collapse to trivial fixed values, which would suggest a dead discriminator or a generator that stops receiving useful gradients.

The run did not exhibit the pathological pattern where one loss goes to zero and the other diverges for long stretches; instead, both traces remain active, which is consistent with continued learning in both networks through epoch 50.

Visual quality. By epoch 50, samples show plausible face layouts: approximate symmetry, hair and skin regions, and eye-like blobs. Fine detail (crisp eyes, hair strands, natural skin texture) remains limited compared with diffusion samples or high-resolution generators, which matches expectations for a modest DCGAN on 64×64 faces.

Recognizable face-like structure (eyes, mouth, hair regions) emerges without external labels, but high-frequency texture is often smoothed or mottled, which is visible both in the final 4×4 grid and in the evolution snapshots.

Mode collapse. A manual pass over **1024** saved GAN images in `outputs/task3_notebook/gan_fid_png/` (used for FID below) shows a mix of poses, apparent gender, hair tone, and expression across different indices; there is no single repeated “copy-paste” identity dominating the folder. Some local similarity between nearby samples can appear because batches share training dynamics, but overall diversity is sufficient to argue against severe mode collapse for this run.

For a stronger mode-collapse check, different random seeds in `torch.randn` produce visibly different identities in the grid and in the saved PNG series, rather than one frozen template repeated hundreds of times.

3.3.2 Generate Faces with Pretrained Diffusion (5 points)

The assignment’s `GenerateNewImageSamples.py` workflow is mirrored by `run_diffusion_sampling` in `hw8_task3.py`: load the Brightspace **CelebA-64** UNet checkpoint, run the DLStudio `GaussianDiffusion` sampler, and write **1024** PNGs under `outputs/task3_notebook/diffusion/png/` (`diff_00000.png ... diff_01023.png`).

(h) Figure: 4×4 grid of diffusion-generated faces



(i) Text: diffusion vs. GAN visual quality comparison

Diffusion samples look sharper and more photorealistic than the DCGAN outputs: eye and mouth regions are better resolved, lighting on cheeks is more consistent, and backgrounds are less smeared. The GAN still captures global face shape but often blurs high frequencies and can produce slight color blotches. This aligns with diffusion's iterative denoising objective, which can refine detail over many steps, versus a single forward mapping from noise in the GAN generator.

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

(j) Code: FID computation script (`pytorch-fid`, Inception dim 2048)

The following matches `compute_fid_for_task3` in `hw8_task3.py`: it builds the InceptionV3 feature extractor, collects activation statistics for **real**, **GAN**, and **diffusion** image folders, then computes two Fréchet distances—real vs GAN and real vs diffusion.

```
import glob
import os
from typing import Tuple

import torch
from pytorch_fid.fid_score import (
    calculate_activation_statistics,
    calculate_frechet_distance,
)
from pytorch_fid.inception import InceptionV3

def compute_fid_for_task3(
    real_dir: str,
    gan_dir: str,
    diffusion_png_dir: str,
    batch_size: int = 50,
    dims: int = 2048,
) -> Tuple[float, float]:
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```

block_idx = InceptionV3.BLOCK_INDEX_BY_DIM[dims]
model = InceptionV3([block_idx]).to(device)

# real_dir: recursive image list (see list_image_files in hw8_task3.py)
real_files = list_image_files(real_dir, recursive=True)
gan_files = sorted(glob.glob(os.path.join(gan_dir, "gan_*.png")))
diff_files = sorted(glob.glob(os.path.join(diffusion_png_dir,
"diff_*.png")))

m_real, s_real = calculate_activation_statistics(
    real_files, model, batch_size=batch_size, dims=dims, device=device
)
m_gan, s_gan = calculate_activation_statistics(
    gan_files, model, batch_size=batch_size, dims=dims, device=device
)
m_df, s_df = calculate_activation_statistics(
    diff_files, model, batch_size=batch_size, dims=dims, device=device
)
fid_gan = calculate_frechet_distance(m_real, s_real, m_gan, s_gan)
fid_diff = calculate_frechet_distance(m_real, s_real, m_df, s_df)
return float(fid_gan), float(fid_diff)

```

CLI entry point (same script): generate 1024 GAN PNGs if needed, then run FID.

```

python hw8_task3.py save_gan_images \
  --generator_ckpt ./outputs/task3_notebook/generator.pt \
  --out_dir ./outputs/task3_notebook/gan_fid_png \
  --num_images 1024

python hw8_task3.py fid \
  --real_dir ./DLStudio-2.5.6/celeba_dataset_64x64 \
  --gan_dir ./outputs/task3_notebook/gan_fid_png \
  --diffusion_dir ./outputs/task3_notebook/diffusion/png

```

1024 GAN images for FID use `save_gan_images` with weights from `outputs/task3_notebook/generator.pt`. FID numbers below match `hw8_task3.ipynb` and `outputs/task3_notebook/fid_values.txt`.

(k) Table: FID values for GAN and diffusion

Model	FID (↓ better)
Your DCGAN	99.67
Diffusion (pretrained)	47.01

(Rounded to two decimals; full precision in `hw8_task3.ipynb` / `fid_values.txt`: 99.667... vs 47.009....)

Additional figures (assignment): 4×4 grids of 16 random samples from each set of 1024

GAN:



Diffusion (16 indices drawn with `numpy.random.default_rng(42)` from `diff_*.png`):

16 random diffusion samples (seed=42) from 1024



Mode collapse check (1024 GAN images, for FID section)

The directory `outputs/task3_notebook/gan_fid_png/` contains 1024 files `gan_00000.png`–`gan_01023.png`. Scrolling the full set shows varied identities and attributes rather than one dominant face repeated across the corpus; occasional similar-looking pairs appear but not at the scale expected from severe mode collapse.

(I) Text: FID comparative discussion (1–2 paragraphs)

Lower FID. The pretrained diffusion model achieves **substantially lower FID** (47.01) than this DCGAN (99.67), indicating that, relative to the real training distribution in Inception feature space, diffusion matches both quality and diversity better. The gap is consistent with diffusion’s likelihood-based training and many-step synthesis, which can cover the data manifold more evenly than adversarial training when the GAN is capacity-limited and hand-tuned.

Diversity and detail. Diffusion samples span a wide range of facial structures and accessories; the GAN covers noticeable diversity but with more “mushy” regions and occasional artifacts. For **fine detail** (eyes, hair texture, skin micro-contrast), diffusion is clearly stronger. **Adversarial training** optimizes a min–max game with a discriminator; if the generator finds a subset of samples that fool a weaker discriminator, it may under-represent some modes. **Diffusion** denoising learns to reverse a fixed forward noise process and can refine details step by step, which tends to improve both perceptual quality and feature statistics measured by FID.

Task 4: Stable Diffusion + Inpainting (30 points)

Source of results: Figures were produced in `hw8_task4.ipynb` (same logic as `hw8_task4.py` in this repository). Artifacts are saved under `outputs/task4_notebook/`.

Deliverable checklist (a)–(k): (a)(h) code in §4.1; (b)(d)(e)(i) figures as labeled; (c)(f)(g)(j)(k) discussion paragraphs below.

Overview

Model: Hugging Face `diffusers` with `runwayml/stable-diffusion-v1-5` (text-to-image and sweeps) and `runwayml/stable-diffusion-inpainting` (inpainting). **Resolution:** 512×512 outputs. **Text-to-face run:** `num_inference_steps=50`, `guidance_scale=7.5`, seeds **42** and **123** for the six assignment prompts. **Sweeps:** prompt index **0** (first prompt), fixed seed **42**; guidance values (**1, 3, 5, 7.5, 10, 15, 20**) at 50 steps; step counts (**5, 10, 20, 30, 50, 75, 100**) at guidance **7.5**. **Inpainting:** CelebA images from `DLStudio-2.5.6/celeba_dataset_64x64`, resized to 512×512; `num_inference_steps=50`, `guidance_scale=7.5`.

Stable Diffusion operates in **latent space**: a VAE encoder maps pixels to latents, a text-conditioned UNet predicts noise / velocity, and a VAE decoder maps latents back to pixels. The text encoder (CLIP) supplies prompt embeddings for cross-attention. **Classifier-free guidance** combines conditional and unconditional UNet predictions:

$$[\epsilon_{\text{c}} + s \cdot (\epsilon_{\text{u}} - \epsilon_{\text{c}}), \epsilon_{\text{u}}]$$

where (s) is the guidance scale. In **inpainting**, the pipeline typically blends denoised latents in the masked region with the encoded source image in unmasked regions each step so structure outside the mask is preserved.

4.1 Code (inference + inpainting)

(a) Code: Stable Diffusion inference script

The following is the **complete** `hw8_task4.py`: text-to-face generation (`run_text_to_face`), guidance and inference-step sweeps (`run_guidance_sweep`, `run_steps_sweep`), CLI entry points (`text`, `sweeps`, `all`), and—shared in one module—the inpainting pipeline and experiment driver used for deliverable (h).

Full file: `hw8_task4.py`

```
#!/usr/bin/env python3
"""
Homework 8 – Task 4: Stable Diffusion text-to-face, guidance/steps sweeps,
inpainting.

Uses HuggingFace diffusers (runwayml/stable-diffusion-v1-5 and stable-
diffusion-inpainting).

GPU strongly recommended. First run downloads ~5 GB per model from
HuggingFace.

Example:
python hw8_task4.py all --celeba_root ./celeba64 --out_dir
./outputs/task4
python hw8_task4.py text --out_dir ./outputs/task4
python hw8_task4.py sweeps --out_dir ./outputs/task4
python hw8_task4.py inpaint --celeba_root ./celeba64 --out_dir
./outputs/task4
"""

from __future__ import annotations

import argparse
import glob
import os
import sys
from pathlib import Path
from typing import List, Sequence, Tuple

import matplotlib

matplotlib.use("Agg")
import matplotlib.pyplot as plt
import numpy as np
import torch
from PIL import Image, ImageDraw, ImageFont

# -----
# --
# Task 4.2 – six prompts × two seeds
# -----
# --

TEXT_PROMPTS: List[str] = [
    "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: Tuple[int, int] = (42, 123)
```

```

# Task 4.3 – sweeps (one prompt from list, fixed seed for comparability)
SWEEP_PROMPT_INDEX = 0
SWEEP_SEED = 42
GUIDANCE_SCALES: Tuple[float, ...] = (1.0, 3.0, 5.0, 7.5, 10.0, 15.0, 20.0)
INFERENCE_STEPS_LIST: Tuple[int, ...] = (5, 10, 20, 30, 50, 75, 100)

SD_MODEL_ID = "runwayml/stable-diffusion-v1-5"
INPAINT_MODEL_ID = "runwayml/stable-diffusion-inpainting"

def get_device_dtype() -> Tuple[torch.device, torch.dtype]:
    if torch.cuda.is_available():
        return torch.device("cuda"), torch.float16
    print("Warning: CUDA not available – Stable Diffusion on CPU is very
slow.", file=sys.stderr)
    return torch.device("cpu"), torch.float32

def load_text_pipe(device: torch.device, dtype: torch.dtype):
    from diffusers import StableDiffusionPipeline

    pipe = StableDiffusionPipeline.from_pretrained(SD_MODEL_ID,
torch_dtype=dtype)
    pipe = pipe.to(device)
    pipe.set_progress_bar_config(disable=False)
    if hasattr(pipe, "enable_attention_slicing"):
        pipe.enable_attention_slicing()
    return pipe

def load_inpaint_pipe(device: torch.device, dtype: torch.dtype):
    from diffusers import StableDiffusionInpaintPipeline

    pipe = StableDiffusionInpaintPipeline.from_pretrained(INPAINT_MODEL_ID,
torch_dtype=dtype)
    pipe = pipe.to(device)
    pipe.set_progress_bar_config(disable=False)
    if hasattr(pipe, "enable_attention_slicing"):
        pipe.enable_attention_slicing()
    return pipe

def _ensure_dir(path: Path) -> None:
    path.mkdir(parents=True, exist_ok=True)

def run_text_to_face(out_dir: Path, device: torch.device, dtype:
torch.dtype) -> None:
    """Task 4.2: 6 prompts × 2 seeds = 12 images; 4×3 grid with labels."""
    out_dir = out_dir.resolve()
    img_dir = out_dir / "text_to_face"
    _ensure_dir(img_dir)

    pipe = load_text_pipe(device, dtype)

```

```

images_for_grid: List[Tuple[Image.Image, str, int]] = []

for pi, prompt in enumerate(TEXT_PROMPTS):
    for sj, seed in enumerate(SEEDS):
        gen = torch.Generator(device=device).manual_seed(seed)
        out = pipe(
            prompt,
            num_inference_steps=50,
            guidance_scale=7.5,
            generator=gen,
        ).images[0]
        fname = f"prompt{pi:02d}_seed{seed}.png"
        out.save(img_dir / fname)
        images_for_grid.append((out, prompt, seed))

# 4 rows x 3 columns
fig, axes = plt.subplots(4, 3, figsize=(14, 18))
for idx, (im, prompt, seed) in enumerate(images_for_grid):
    r, c = divmod(idx, 3)
    ax = axes[r][c]
    ax.imshow(np.asarray(im))
    ax.axis("off")
    short = prompt if len(prompt) < 55 else prompt[:52] + "..."
    ax.set_title(f"seed={seed}\n{short}", fontsize=8)
plt.suptitle("Stable Diffusion v1.5 – 6 prompts × 2 seeds (50 steps,
guidance 7.5)", fontsize=11)
plt.tight_layout()
grid_path = out_dir / "text_to_face_grid_4x3.png"
plt.savefig(grid_path, dpi=150, bbox_inches="tight")
plt.close()
print(f"Saved grid: {grid_path}")

def run_guidance_sweep(
    out_dir: Path, device: torch.device, dtype: torch.dtype, pipe=None
) -> None:
    """Task 4.3A: one row, 7 guidance values, steps=50, same seed."""
    out_dir = out_dir.resolve()
    _ensure_dir(out_dir / "sweeps")

    prompt = TEXT_PROMPTS[SWEEP_PROMPT_INDEX]
    own_pipe = pipe is None
    if own_pipe:
        pipe = load_text_pipe(device, dtype)

    fig, axes = plt.subplots(1, len(GUIDANCE_SCALES), figsize=(21, 4))
    ax_list = np.atleast_1d(axes).ravel()
    for i, g in enumerate(GUIDANCE_SCALES):
        gen = torch.Generator(device=device).manual_seed(SWEEP_SEED)
        im = pipe(
            prompt,
            num_inference_steps=50,
            guidance_scale=float(g),
            generator=gen,

```

```

        ).images[0]
        im.save(out_dir / "sweeps" / f"guidance_{g}.png")
        ax_list[i].imshow(np.asarray(im))
        ax_list[i].axis("off")
        ax_list[i].set_title(f"guidance={g}", fontsize=10)
    fig.suptitle(f"Guidance scale sweep (50 steps, seed=
{SWEEP_SEED})\n{prompt}", fontsize=9)
    plt.tight_layout()
    path = out_dir / "sweep_guidance_row.png"
    plt.savefig(path, dpi=150, bbox_inches="tight")
    plt.close()
    print(f"Saved: {path}")
    if own_pipe:
        del pipe
        if torch.cuda.is_available():
            torch.cuda.empty_cache()

def run_steps_sweep(
    out_dir: Path, device: torch.device, dtype: torch.dtype, pipe=None
) -> None:
    """Task 4.3B: one row, 7 step counts, guidance=7.5, same seed."""
    out_dir = out_dir.resolve()
    _ensure_dir(out_dir / "sweeps")

    prompt = TEXT_PROMPTS[SWEEP_PROMPT_INDEX]
    own_pipe = pipe is None
    if own_pipe:
        pipe = load_text_pipe(device, dtype)

    fig, axes = plt.subplots(1, len(INFERENCE_STEPS_LIST), figsize=(21, 4))
    ax_list = np.atleast_1d(axes).ravel()
    for i, steps in enumerate(INFERENCE_STEPS_LIST):
        gen = torch.Generator(device=device).manual_seed(SWEEP_SEED)
        im = pipe(
            prompt,
            num_inference_steps=int(steps),
            guidance_scale=7.5,
            generator=gen,
        ).images[0]
        im.save(out_dir / "sweeps" / f"steps_{steps}.png")
        ax_list[i].imshow(np.asarray(im))
        ax_list[i].axis("off")
        ax_list[i].set_title(f"steps={steps}", fontsize=10)
    fig.suptitle(f"Inference steps sweep (guidance=7.5, seed=
{SWEEP_SEED})\n{prompt}", fontsize=9)
    plt.tight_layout()
    path = out_dir / "sweep_steps_row.png"
    plt.savefig(path, dpi=150, bbox_inches="tight")
    plt.close()
    print(f"Saved: {path}")
    if own_pipe:
        del pipe
        if torch.cuda.is_available():

```



```

torch.cuda.empty_cache()

def run_both_sweeps(out_dir: Path, device: torch.device, dtype:
torch.dtype) -> None:
    """Load text-to-image pipeline once; run guidance then steps sweeps."""
    pipe = load_text_pipe(device, dtype)
    run_guidance_sweep(out_dir, device, dtype, pipe=pipe)
    run_steps_sweep(out_dir, device, dtype, pipe=pipe)
    del pipe
    if torch.cuda.is_available():
        torch.cuda.empty_cache()

def _resize_512(im: Image.Image) -> Image.Image:
    return im.resize((512, 512), Image.Resampling.LANCZOS)

def _mask_hair() -> Image.Image:
    """White = inpaint region (top band – hair)."""
    m = Image.new("L", (512, 512), 0)
    d = ImageDraw.Draw(m)
    d.rectangle([0, 0, 512, 185], fill=255)
    return m

def _mask_eyes() -> Image.Image:
    """Ellipse over eye region."""
    m = Image.new("L", (512, 512), 0)
    d = ImageDraw.Draw(m)
    d.ellipse([110, 155, 402, 265], fill=255)
    return m

def _mask_mouth() -> Image.Image:
    """Lower face – mouth and chin."""
    m = Image.new("L", (512, 512), 0)
    d = ImageDraw.Draw(m)
    d.rectangle([95, 300, 417, 505], fill=255)
    return m

def _mask_forehead_creative() -> Image.Image:
    """Upper forehead / hairline band for creative experiment (e.g.
crown)."""
    m = Image.new("L", (512, 512), 0)
    d = ImageDraw.Draw(m)
    d.rectangle([80, 0, 432, 140], fill=255)
    return m

def _overlay_mask_rgb(base: Image.Image, mask_l: Image.Image, color:
Tuple[int, int, int] = (255, 80, 80)) -> Image.Image:
    """RGB overlay for visualization (mask white regions tinted)."""

```

```

base = base.convert("RGB")
arr = np.array(base, dtype=np.float32)
mk = np.array(mask_l, dtype=np.float32) / 255.0
mk = mk[..., np.newaxis]
tint = np.array(color, dtype=np.float32)
blended = arr * (1 - 0.45 * mk) + tint * (0.45 * mk)
return Image.fromarray(np.clip(blended, 0, 255).astype(np.uint8))

def _pick_celeba_paths(root: str, n: int = 4) -> List[str]:
    paths: List[str] = []
    for ext in (".jpg", ".jpeg", ".png", ".JPG"):
        paths.extend(glob.glob(os.path.join(root, "**", ext),
recursive=True))
    paths = sorted(set(paths))
    if len(paths) < n:
        raise FileNotFoundError(
            f"Need at least {n} images under {root}; found {len(paths)}"
        )
    # Spread indices across dataset
    idxs = [0, len(paths) // 4, len(paths) // 2, (3 * len(paths)) // 4]
    return [paths[i] for i in idxs[:n]]

def run_inpainting(out_dir: Path, celeba_root: str, device: torch.device,
dtype: torch.dtype) -> None:
    """Task 4.4: four experiments, each saved as a row figure."""
    out_dir = out_dir.resolve()
    inp_dir = out_dir / "inpainting"
    _ensure_dir(inp_dir)

    paths = _pick_celeba_paths(celeba_root, 4)
    pipe = load_inpaint_pipe(device, dtype)

    experiments: List[Tuple[str, Image.Image, List[Tuple[str, str]]]] = [
        (
            "1_hair_color",
            _mask_hair(),
            [
                ("blonde hair, portrait photo", "hair_blonde"),
                ("bright blue hair, portrait photo", "hair_blue"),
            ],
        ),
        (
            "2_accessories",
            _mask_eyes(),
            [
                ("wearing large stylish sunglasses, portrait",
"eyes_sunglasses"),
                ("natural eyes without glasses, portrait photo",
"eyes_natural"),
            ],
        ),
    ]

```

```

        "3_expression",
        _mask_mouth(),
        [
            ("wide smile showing teeth, portrait photo",
"mouth_smile"),
            ("serious expression, closed mouth, portrait",
"mouth_serious"),
        ],
    ),
    (
        "4_creative_crown",
        _mask_forehead_creative(),
        [
            ("golden ornate crown on head, royal portrait photo",
"crown_gold"),
        ],
    ),
]

for exp_idx, (exp_name, mask_l, prompt_list) in enumerate(experiments):
    src_path = paths[exp_idx]
    base = _resize_512(Image.open(src_path).convert("RGB"))
    overlay = _overlay_mask_rgb(base, mask_l)

    cols = 2 + len(prompt_list) # original + overlay + variants
    fig, axes = plt.subplots(1, cols, figsize=(4 * cols, 4.5))
    if cols == 3:
        axes = [axes[0], axes[1], axes[2]]
    for ax in axes:
        ax.axis("off")

    axes[0].imshow(base)
    axes[0].set_title("Original", fontsize=9)
    axes[1].imshow(overlay)
    axes[1].set_title("Mask overlay", fontsize=9)

    gen = torch.Generator(device=device).manual_seed(42 + exp_idx)
    for j, (prompt, tag) in enumerate(prompt_list):
        result = pipe(
            prompt=prompt,
            image=base,
            mask_image=mask_l.convert("RGB"),
            num_inference_steps=50,
            guidance_scale=7.5,
            generator=gen,
        ).images[0]
        out_name = f"{exp_name}_{tag}.png"
        result.save(inp_dir / out_name)
        axes[2 + j].imshow(np.asarray(result))
        axes[2 + j].set_title(prompt[:40] + ("..." if len(prompt) > 40
else ""), fontsize=7)

    fig.suptitle(f"Inpainting – {exp_name}", fontsize=10)
    plt.tight_layout()

```

```

fig_path = out_dir / f"inpaint_row_{exp_name}.png"
plt.savefig(fig_path, dpi=150, bbox_inches="tight")
plt.close()
print(f"Saved: {fig_path}")

def build_arg_parser() -> argparse.ArgumentParser:
    p = argparse.ArgumentParser(description="HW8 Task 4 – Stable
Diffusion")
    p.add_argument(
        "command",
        choices=("all", "text", "sweeps", "inpaint"),
        help="all: full pipeline; text/sweeps/inpaint: partial",
    )
    p.add_argument("--out_dir", type=str, default="./outputs/task4_sd")
    p.add_argument(
        "--celeba_root",
        type=str,
        default="./celeba64",
        help="Folder with CelebA images (recursive) for inpainting",
    )
    return p

def main(argv: Sequence[str] | None = None) -> None:
    args = build_arg_parser().parse_args(argv if argv is not None else
sys.argv[1:])
    out_dir = Path(args.out_dir)
    _ensure_dir(out_dir)
    device, dtype = get_device_dtype()

    if args.command in ("all", "text"):
        run_text_to_face(out_dir, device, dtype)
    if args.command in ("all", "sweeps"):
        run_guidance_sweep(out_dir, device, dtype)
        run_steps_sweep(out_dir, device, dtype)
    if args.command in ("all", "inpaint"):
        if not os.path.isdir(args.celeba_root):
            raise SystemExit(f"Missing CelebA folder: {args.celeba_root}")
        run_inpainting(out_dir, args.celeba_root, device, dtype)

    print("Done.")

if __name__ == "__main__":
    main()

```

(h) Code: Inpainting script

Inpainting is implemented in the **same** `hw8_task4.py` listing under **(a)** (not a separate file). Relevant pieces: `load_inpaint_pipe` (`StableDiffusionInpaintPipeline`), `run_inpainting` (four

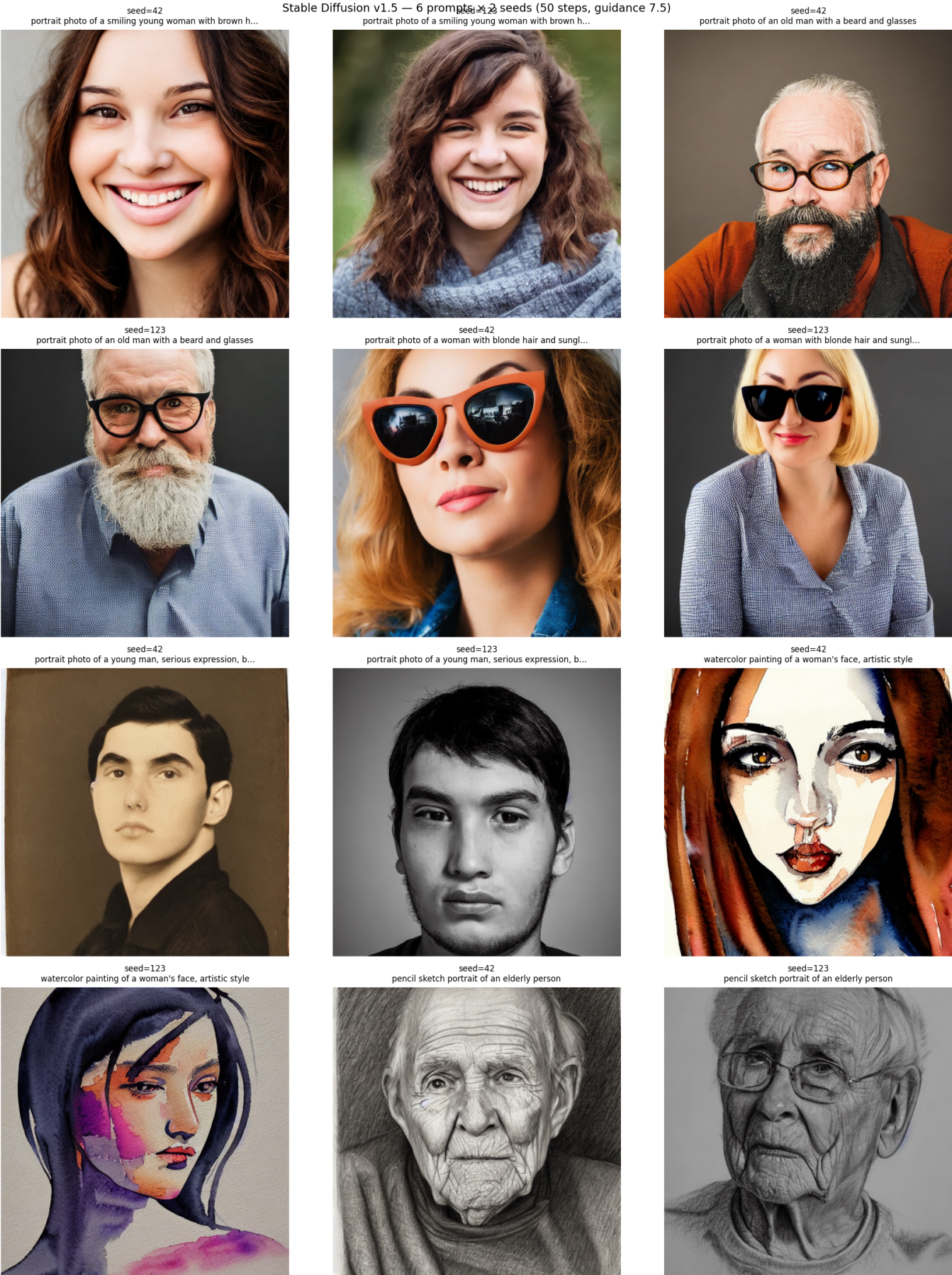
experiments with `_mask_hair`, `_mask_eyes`, `_mask_mouth`, `_mask_forehead_creative`, `_overlay_mask_rgb`, `_pick_celeba_paths`), and the CLI command `inpaint`. Run example:

```
python hw8_task4.py text --out_dir ./outputs/task4_notebook
python hw8_task4.py sweeps --out_dir ./outputs/task4_notebook
python hw8_task4.py inpaint --celeba_root ./DLStudio-
2.5.6/celeba_dataset_64x64 --out_dir ./outputs/task4_notebook
```

4.2 Text-to-face and sweeps (figures and discussion)

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

Twelve PNGs are saved under `outputs/task4_notebook/text_to_face/` (`prompt00_seed42.png`, ...). The summary figure is a **4×3** grid: each subplot title shows **seed=** and the **prompt text** (truncated with an ellipsis when long). Model and hyperparameters are in the **Overview** above.



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

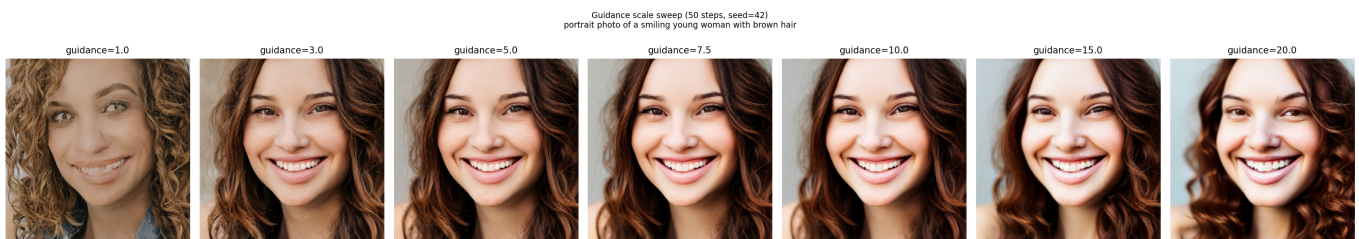
Attribute capture. Stable Diffusion generally reflects the requested semantics: age-related cues (young vs. old), beard and glasses, hair color and sunglasses, serious vs. smiling expression, and stylistic prompts

(watercolor vs. pencil sketch). Photorealistic prompts **1–4** tend to track the text most faithfully because similar faces appear often in the training prior. **5–6** follow artistic style but can show more layout variance or softer identity structure, since strong style terms compete with photorealistic facial cues.

Comparison to Task 3. Task 3 used **64×64** CelebA DCGAN and pretrained **DLStudio** diffusion; Task 4 uses **512×512** Stable Diffusion with **language control**. Compared with those Task 3 models, SD outputs have clearer global composition, richer texture, and stronger alignment to **open-vocabulary** prompts. Task 3’s GAN often blurs fine detail; its diffusion samples are sharper than the GAN but still **64×64** and not text-conditioned in the same way as SD. Overall, SD offers the best combination here of resolution, detail, and prompt steering—at the cost of much larger models and compute.

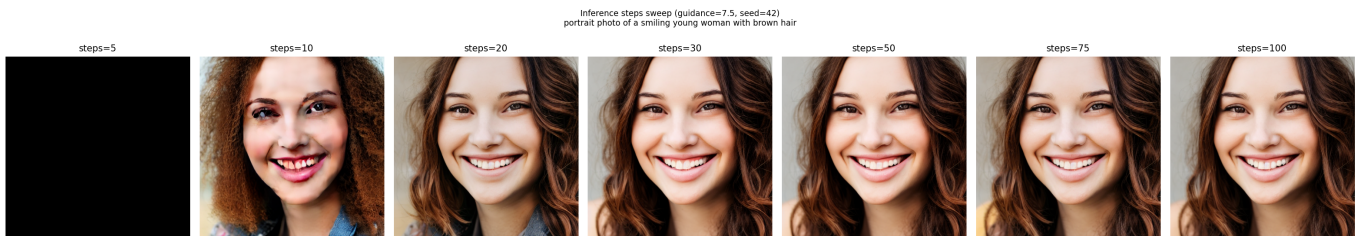
(d) Figure: guidance scale sweep (7 images, one row, labeled)

Fixed prompt: first string in **TEXT_PROMPTS**; **50** inference steps; seed **42**. Each panel title is **guidance=<value>** for ($s \in \{1, 3, 5, 7.5, 10, 15, 20\}$).



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

Same prompt and seed **42**; **guidance 7.5**; step counts **5, 10, 20, 30, 50, 75, 100**. Each panel title is **steps=<n>**.



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

Low guidance ($s \approx 1$ –(3)). The update is close to the **unconditional** score because the term $(\epsilon_c - \epsilon_u)$ is scaled weakly. Samples can look more “natural” in color distribution but may **ignore** parts of the prompt.

Moderate guidance ($s \approx 5$ –(10)). Prompt adherence and sample quality are often best: the conditional signal is strong enough to match attributes without destabilizing the denoising trajectory.

High guidance ($s \gg 10$). Classifier-free guidance

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

amplifies the conditional direction at **every** step. That can improve literal prompt match but also **over-saturate** colors, increase contrast, and create **artifacts** when the conditional prediction is biased or

mismatched to the image statistics—each step repeatedly pushes along the same exaggerated direction.

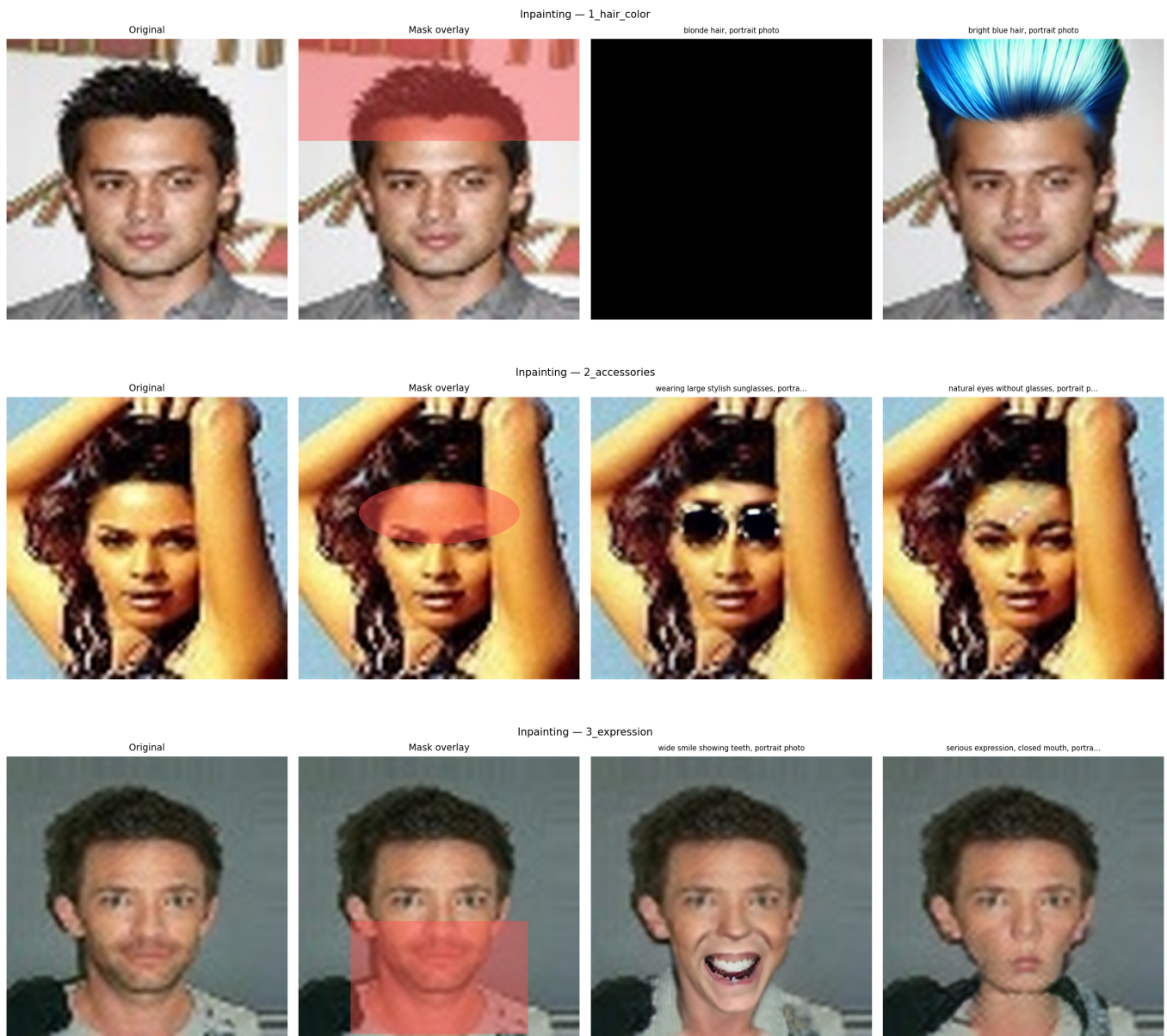
(g) Text: inference steps explanation (relation to reverse Markov chain)

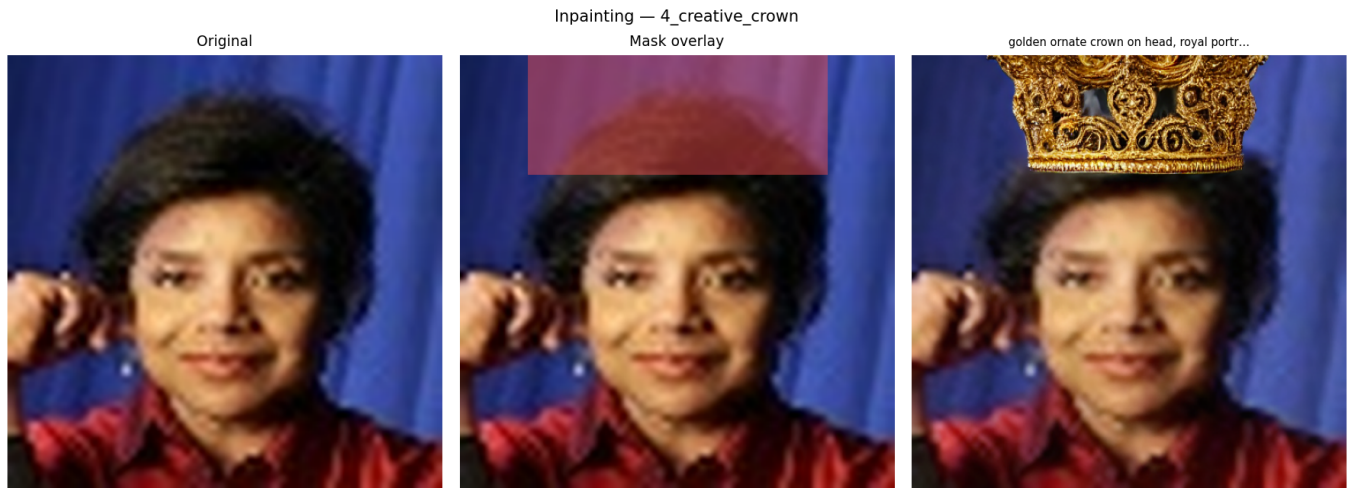
Sampling approximates the **reverse** of a forward noising process: a **Markov chain** from noise toward data. Each scheduler step applies one (or a chunk of) **reverse transition(s)**. With **very few steps**, the discretization is coarse—each jump must remove a lot of noise at once, so errors accumulate and structure may collapse (in this run, **5 steps** is severely degraded). **More steps** give a finer sequence of corrections, so semantics and details stabilize; returns diminish once the process is well approximated (**50–75** vs. **100** here). This matches the view of diffusion as iterative refinement along a reverse trajectory rather than a single feed-forward map.

4.3 Inpainting on CelebA (figures and discussion)

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

Each row is **Original** | **Mask overlay** | one or two **inpainted** results with short prompt titles. Experiments: **(1)** hair color, **(2)** accessories, **(3)** expression, **(4)** creative crown.





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

Blending. Quality is best when the **mask** matches a **semantic** region (hair band, eyes, mouth) and the **prompt** describes content that fits the surrounding pose and lighting. Hard mask edges can leave visible seams if the model's inpainted texture or color does not match the unmasked skin.

Best experiments. **Hair color** edits are usually strongest: large regions, forgiving texture, and clear text tokens ("blonde", "bright blue hair"). The **crown** experiment can look convincing when the forehead mask aligns with the added object.

Weaker experiments. **Accessories** ("sunglasses" vs. "natural eyes") is mixed: adding glasses is easier than plausibly **removing** glasses and hallucinating consistent eyes. **Expression** edits are often hardest: the mouth couples to cheeks and jaw; changing expression without shifting adjacent geometry is ambiguous, so boundaries can look stiff.

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

Relation to diffusion. Inpainting in a latent diffusion pipeline still uses the **same denoising backbone** as text-to-image, but conditions generation on the **masked region** while **re-encoding / blending** with the known pixels outside the mask (implementation details depend on the pipeline; the effect is **conditional inpainting**: fill masked latents under text guidance while keeping context consistent with the source).

Difference from GANs (e.g. Task 3 DCGAN). A standard GAN **generator** maps noise to a **full image** in one forward pass; it has no built-in mechanism to **preserve** arbitrary unmasked pixels of an existing photo. Inpainting with diffusion is an **iterative, constrained** synthesis problem that uses the image as context at every step. That is why region editing and photorealistic blending are far more natural in the diffusion inpainting setup than with a plain GAN trained only for unconditional samples.