A Combined Report on Generative Models for CIFAR-10 and Conditional Sketch-to-Face Translation

SYED TAHA HASAN
FAST NUCES
Email: syed.taha.hasan@example.edu

Course: I211767 — Assignment 2

***Abstract*—This report summarizes the work and results from the assignment's questions (Q1–Q3). We implemented and trained generative models (VAE, VAE-GAN, custom GANs, and a conditional GAN), applied dataset preprocessing, tracked training metrics, and produced sample outputs. The report covers dataset and preprocessing choices, model architectures, training behavior and quantitative/qualitative results (where available), followed by discussion and conclusions. We also document environment-level diagnostics and remediation steps required to run long experiments reproducibly on Windows systems that may contain conflicting native runtimes.**

***Index Terms*—Generative models, GAN, VAE, CIFAR-10, conditional GAN, OpenMP, reproducibility**

I. Introduction

Generative modeling is a central pillar of modern machine learning enabling tasks such as image synthesis, conditional translation, and representation learning. In this project we implemented three experimental questions: (Q1) a Variational Autoencoder (VAE) baseline and diagnostics, (Q2) a custom GAN trained on CIFAR-10 with architectural and training improvements, and (Q3) a conditional GAN for sketch-to-face translation. The objective was to (a) implement working, well-logged training pipelines, (b) stabilize training across realistic Windows development environments, and (c) produce reproducible artifacts (checkpoints, logs, and sample images).

This combined report documents dataset decisions, model architectures, training recipes, results (quantitative and qualitative), system-level diagnostics encountered during development, and remediation recommendations to ensure stable, repeatable training.

II. Related Work

The methods applied build on established generative modeling literature. Variational Autoencoders [**?**] provide a probabilistic latent-variable framework. Generative Adversarial Networks (GANs) [**?**] are known for high-fidelity image synthesis; stabilization improvements used here include Spectral Normalization [**?**] and gradient penalties [**?**]. Implementation leverages PyTorch [**?**] for model and training code, Matplotlib [**?**] for diagnostics, and CIFAR-10 [**?**] as a canonical low-resolution image benchmark.

III. Dataset and Preprocessing

We used CIFAR-10 (32x32 color images, 10 classes) for Q2 experiments and standard sketch/face pairs (where available) for Q3.

Preprocessing steps:

- Convert images to float tensors in range [-1, 1] for GANs (common in transposed-convolution generators).
- For VAE experiments, normalize data to mean 0, std 1 as usual for reconstruction losses.
- Small augmentations (random horizontal flips, random crops) were applied optionally depending on experiment to test robustness.
- Data loaders were implemented with shuffling and multiple workers (worker count reduced in Windows runs if OMP/MKL thread limits were set).

All dataset code is located under `scripts/` and question-specific folders: `question1_vae_gan/`, `question2_custom_gan_cifar/`, and `question3_cgan_sketch2face/`.

IV. Model Architectures

This section summarizes the key architecture choices used across the three questions.

A. VAE (Q1)

A convolutional encoder-decoder with a latent bottleneck and KL divergence regularization:

- Encoder: stacked conv layers with ReLU activations, downsampling by strided convs.
- Latent: mean and log-variance heads, reparameterization trick.
- Decoder: transposed conv layers mirroring the encoder.
- Loss: Reconstruction (MSE or BCE depending on data) + β-weighted KL term (default $\beta = 1$).

B. Custom GAN (Q2)

Generator and discriminator designed for CIFAR-10 sized images:

- Generator: input latent vector (dim increased from 100 to 128), upsampling via ConvTranspose2d, batch normalization, ReLU, final Tanh.

- Discriminator: convolutional downsampling, LeakyReLU activations, and spectral normalization applied to conv layers to stabilize training.
- Additional changes: improved linear-layer initialization, EMA for generator parameters to produce smoother samples during evaluation.

C. Conditional GAN (Q3)

Architectures suitable for sketch-to-face translation:

- Generator: encoder-decoder generator conditioned on input sketch image; skip connections optionally included depending on variant.
- Discriminator: Siamese-style discriminator producing a similarity head to compare sketch and generated face; spectral normalization applied as appropriate.
- Losses: adversarial loss plus optional L1/L2 reconstruction terms and perceptual losses (if available).

V. Training Recipe and Hyperparameters

Common hyperparameters and training patterns used across experiments:

- Optimizer: Adam with tuned betas (commonly $\beta_1 = 0.5$, $\beta_2 = 0.999$ for GANs).
- Learning rates: tuned per model; typical starting values: 2e-4 for generator and discriminator.
- Batch sizes: tuned to fit GPU memory (32–128 depending on GPU).
- EMA: decay factor (e.g., 0.999) for generator weight averaging used for evaluation snapshots.
- Regularization: spectral normalization for discriminator; gradient penalties optionally used (WGAN-GP style) for stability tests.
- Checkpointing: per-epoch or per-N-iterations saving into `checkpoints/`.
- Logging: JSON logs per epoch saved into `logs/` with generator/discriminator losses and a simple quality metric computed from samples (see repository `scripts/eval_metrics.py`).

VI. Engineering Hardening and Environment Diagnostics

During iterative development on Windows, we encountered intermittent fatal native exceptions (Windows exception 0xc06d007f) traced to native libraries during NumPy linear algebra operations and Matplotlib layout computations. The primary findings and mitigations follow.

A. Root cause analysis

Inspection of the base Conda environment revealed co-existing OpenMP runtimes in `Library\bin`: both Intel's `libiomp5md.dll` and LLVM's `libomp.dll` were present. Having multiple OpenMP runtimes on DLL search path can lead to ABI conflicts in BLAS/MKL/OMP-dependent libraries (NumPy, SciPy, and parts of Matplotlib), causing process-level crashes.

B. Temporary mitigations applied

To avoid training aborts while continuing development:

- Force Matplotlib to use a non-interactive backend (`Agg`) before importing `pyplot`.
- Wrap plotting code in try/except to avoid unhandled exceptions aborting the Python process.
- Limit runtime parallel threads: set `OMP_NUM_THREADS=1` and `MKL_NUM_THREADS=1` in the environment when launching training.
- Allow duplicate OpenMP (`KMP_DUPLICATE_LIB_OK=TRUE`) as a last resort in development only (unsafe for production).
- Disable tqdm monitor thread (set environment variable used by tqdm) to avoid additional background threads that can interact badly with native libraries on Windows.

C. Permanent remediation recommendation

Create an isolated Conda environment (or use `mamba`) with a single consistent set of native runtime libraries and a PyTorch build matching the CUDA toolkit (the base install used PyTorch built for CUDA 12.1). Steps:

1) Clean Conda caches: `conda clean -a -y`.
2) Create a new environment and install PyTorch + CUDA-compatible packages (follow PyTorch install selector for the correct CUDA version).
3) Verify only one OpenMP runtime is present in the environment's `Library\bin`.
4) Run import tests (`import numpy, torch, matplotlib`) and a small linear-algebra test (`np.linalg.inv(np.eye(10))`) before long runs.

VII. Results

Here we present representative short-run metrics collected from the GAN experiments (values pulled from JSON logs saved during iterative runs). These are illustrative; a final 200-epoch run in an isolated environment would be needed for complete evaluation (FID/IS).

TABLE I
Representative training metrics from short runs (selected epochs).

Epoch	Generator loss	Discriminator loss	Quality score
5	0.9343	0.4496	0.4080
10	0.04358	0.05811	0.32885
15	0.04474	0.03848	0.39831

Qualitatively, generator samples saved in `generated_images/` show improving textures and structure across epochs, and EMA-smoothed generator checkpoints provided visually sharper outputs.

VIII. Discussion

Key observations and lessons learned:

- Architectural improvements (spectral normalization, larger latent vector, weight initialization) improved training stability and sample quality in short-run tests.
- System-level runtime conflicts (multiple OpenMP DLLs) caused non-deterministic, hard-to-debug crashes. Software-level guards (non-interactive Matplotlib backend, try/except) prevented training from aborting but do not fix the root problem.
- Creating and validating a dedicated environment that isolates native DLLs is a necessary step to reliably run long experiments (full training and final metrics).
- For sketch-to-face translation, adding perceptual loss terms (VGG-based), multi-scale discriminators, and higher-resolution training are clear extensions that should improve realism.

IX. Conclusions

This project implemented and iterated on multiple generative model families (VAE, GAN, conditional GAN) and exposed the interplay between modeling choices and system-level dependencies on Windows. While modeling improvements yielded better short-run results, a permanent fix for native runtime conflicts (single consistent OpenMP runtime via an isolated Conda environment) is required to run large-scale, reproducible experiments. The repository includes trained checkpoints (partial runs), JSON logs, and sample images; with the environment remediation a full 200-epoch training and final evaluation would complete the study.

Acknowledgments

Thank you to course staff and peers for feedback and to the debugging sessions that helped identify runtime issues and remediation strategies.

Prompts and Commands used during development

Below are the practical commands and prompts that were used repeatedly during development and testing. These can be used as a quick start.

Common run command (module mode)

```
# Run training (example)
python -u -m question2_custom_gan_cifar.train --epochs 50 --batch_size 64 --no-plot
```

Environment troubleshooting commands (PowerShell examples)

```
# Clean conda caches
conda clean -a -y

# Create new environment (example)
conda create -n gan_env python=3.11 -y
conda activate gan_env

# Install PyTorch matching CUDA (example via pip or conda per PyTorch site)
# e.g., conda install pytorch torchvision torchaudio pytorch-cuda=12.1 -c pytorch -c nvidia
```

Quick import sanity checks

```
python - <<'PY'
import numpy as np, torch, matplotlib
print("numpy:", np.__version__, "torch:", torch.__versic
print("inv test:", np.linalg.inv(np.eye(3)))
PY
```

Artifacts and repository layout

Important folders:

- `checkpoints/` — model checkpoints created during training.
- `generated_images/` — example outputs saved per epoch.
- `logs/` — per-epoch JSON training logs and metrics.
- `question1_vae_gan/`, `question2_custom_gan_cifar/`, `question3_cgan_sketch2face/` — question-specific implementations.

References

Appendix

A. Recommended next steps to finish the evaluation

1) Ensure a clean Conda environment with CUDA 12.1 and matching PyTorch build is available.
2) Run a short 1–5 epoch smoke test verifying imports and a single training epoch (`-no-plot`) to ensure stability.
3) Run the full training for Q2 (e.g., 200 epochs) and compute FID/IS using saved samples and `scripts/eval_metrics.py`.
4) Add selected sample images into the paper by placing them in `generated_images/` and including with `\includegraphics`.

B. If you want me to add figures to the paper

Tell me which sample files under `generated_images/` you'd like to include (file names), and I will insert `\includegraphics` commands and captions and provide the updated '.tex' ready to compile.