Re-implementing ControlNet for Edge-Guided Image Synthesis (CS787)

Authors:

[1]MADHAV KUMAR 218070569

[2]RISHAV GOYAL 170574

[3]ANUJ KUMAR GUPTA 230170

[4]YUVRAJ SINGH KHARTE 211210

[5]AVINASH SAINI 200232

Course: CS787 – Generative Artificial Intelligence
Institute: Indian Institute of Technology Kanpur
Repository: https://github.com/yuviiitk/CS787_controlnet
Contact:

[1]madhavk21@iitk.ac.in [2]grishav@iitk.ac.in [3]avinashs20@iitk.ac.in [4]yuvrajsgh21@iitk.ac.in
[5]anjukg23@iitk.ac.in

Abstract

We reproduce and adapt **ControlNet**—a method to add spatial conditioning to text-to-image diffusion models—focusing on the **Canny-edge** condition. We freeze Stable Diffusion (SD v1.5) components and train a ControlNet branch from scratch using a curated dataset of edge–image–caption triplets. Under practical GPU limits, we train at 256×256 with mixed precision, gradient accumulation, and cosine LR warmup. We demonstrate controlled generation that respects edge structure and prompt semantics, and discuss training behaviours, ablations (CFG, steps, resolution), and failure modes. Code, configurations, and checkpoints are released for reproducibility.

---

1. Introduction

Large text-to-image models produce diverse images but lack **spatial controllability**. ControlNet addresses this by injecting a condition pathway (e.g., edges) into a frozen base model via **zero-initialized** connectors, enabling controllable generation without catastrophic forgetting. This report documents our end-to-end reproduction with an emphasis on: (i) dataset assembly, (ii) faithful re-implementation choices under compute constraints, and (iii) qualitative outcomes on **edge-guided** synthesis.

---

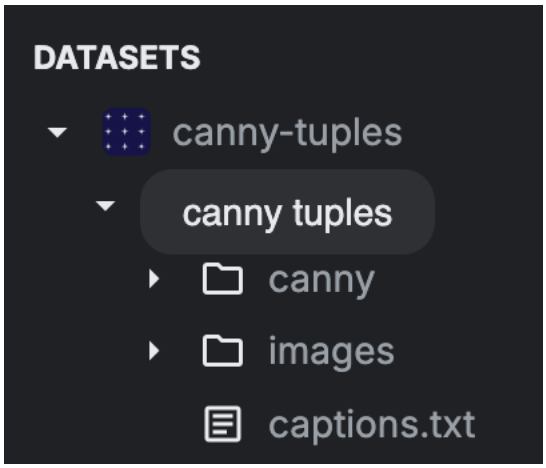
2. Background and Related Work

Stable Diffusion (SD v1.5): a latent diffusion model with a VAE encoder/decoder, CLIP text encoder, and a U-Net denoiser conditioned on text embeddings.
ControlNet (Zhang, Rao, Agrawala, 2023): duplicates the U-Net encoder and mid blocks as a trainable branch, wiring them back to the frozen base via **zero-convolutions** (initialized to zeros). Key ideas we follow:

- Freeze SD (VAE, U-Net, text encoder), train only the ControlNet branch.
- Standard **noise-prediction loss** (ε-prediction MSE).
- Condition image processed by a small down sampling **hint encoder**; features injected as residuals into down/mid blocks.
- (Paper pattern) Useful training heuristics: zero-conv safety, occasional empty prompts, simple DDPM schedule.

---

3. Dataset

- **Source:** LAION-Aesthetics 6.5+ (cropped 512×512 subset) — the 5k-image mini-subset that people often use for quick SD/ControlNet experiments.
- **Composition:** = **3316** samples (triplets):
    - images/ – RGB targets (size varies; we resize to **256×256**).
    - canny/ – edge maps (1-channel; we repeat to 3 channels).
    - captions.txt – lines of FILENAME<TAB>CAPTION.
- **Splits:** TRAIN **N_train**, VAL **N_val** (policy: by filename/time).
- **Preprocessing:** Resize images to 256×256 (LANCZOS), Canny to 256×256 (NEAREST), normalize to [0,1] for tensors; model-side normalization to [-1,1].
- **Alignment check:** We map captions by filename to prevent prompt drift.
- 

---

4. Method

4.1 Architecture

- **Frozen base:** SD v1.5 components—**VAE**, **U-Net**, **CLIP text encoder**.
- **Trainable branch: ControlNet** cloned from U-Net encoder + mid blocks.
- **Connectors: zero-initialized 1×1 convs** that add ControlNet residuals to the frozen U-Net's down and mid blocks.
- **Condition:** Canny edge map → normalized to [-1,1] → hint encoder → residuals.

4.2 Objective & Scheduler

- **Loss:** ε-prediction **MSE** between predicted noise and sampled noise.
- **Noise schedule:** DDPM (scaled_linear) with 1k steps (β_start=0.00085, β_end=0.012).

---

5. Implementation Details

- **Frameworks:** PyTorch, Diffusers, Transformers.
- **Precision:** AMP (fp16) on GPU.
- **Memory:** gradient checkpointing, (optional) xFormers attention, gradient accumulation.
- **DataLoader:** batch size per GPU **1**, accumulation **8** → effective batch **8**.
- **Regularization:** weight decay **1e-4** (mild to avoid underfitting tiny branch).
- **Learning rate: 1e-5** with cosine schedule and warmup (**1000** steps).
- **Hint dropout: 0.0** initially (maximize faithfulness), can be increased later for robustness.
- **Resolution:** train/infer at **256×256** to avoid resolution mismatch.

Training

python src/train_controlnet.py \

ControlNet-augmented block

$$\boldsymbol{y}_c = \mathcal{F}(\boldsymbol{x};\Theta) + \mathcal{Z}(\mathcal{F}(\boldsymbol{x} + \mathcal{Z}(\boldsymbol{c};\Theta_{z1});\Theta_c);\Theta_{z2}),$$

```
# x = zt  (noisy latent), c = ci (canny), Θ=frozen UNet, Θc=ControlNet branch, Z=zero-conv connecti
cn_out = controlnet(
    sample=nlat, timestep=ts, encoder_hidden_states=txt_emb,
    controlnet_cond=hint_n, return_dict=True
)
pred = unet(
    nlat, ts, encoder_hidden_states=txt_emb,
    down_block_additional_residuals=cn_out.down_block_res_samples,  # Z(F(x + Z(c)))
    mid_block_additional_residual=cn_out.mid_block_res_sample,      #        ... )
    return_dict=True
).sample
```

Term → variable

- $\mathbf{x}$ → `nlat` (the noisy latent input to both branches)
- $\mathcal{F}(\cdot;\Theta)$ → `unet(…)` with **frozen** base weights Θ
- Θ_c → trainable weights inside `controlnet`
- $\mathcal{Z}(\cdot;\Theta_{z1}), \mathcal{Z}(\cdot;\Theta_{z2})$ → **zero-conv connectors** inside `controlnet` producing `down_block_res_samples` / `mid_block_res_sample`
- $\mathbf{y}_c$ → `pred` (UNet output **with** injected ControlNet residuals)
- $\mathbf{y}$ (base output) → you'd get it by calling `unet(nlat, ts, encoder_hidden_states=txt_emb)` **without** the `*_additional_residual*` args.

Zero-conv safety at init

$$\boldsymbol{y}_{\mathrm{c}} = \boldsymbol{y}.$$

```
# paper-faithful build with zero-initialized 1×1 "zero conv" connectors
controlnet = ControlNetModel.from_unet(unet).to(DEVICE).train()
```

Hint/condition encoder

$$\boldsymbol{c}_{\mathrm{f}} = \mathcal{E}(\boldsymbol{c}_{\mathrm{i}}).$$

```
# control image (Canny) → normalize and feed to ControlNet
cn   = Image.open(self.canny_files[i]).convert("L")         # load as grayscale
hint = self.tf_hint(cn).repeat(3,1,1)                       # [1,H,W]→[3,H,W]
hint_n = hint * 2.0 - 1.0                                   # ci normalized

# ControlNet internally applies the hint encoder E(·)
cn_out = controlnet(
    sample=nlat, timestep=ts, encoder_hidden_states=txt_emb,
    controlnet_cond=hint_n, return_dict=True
)
# cn_out.* are functions of cf = E(ci)
```

Term → variable

- $\mathbf{c}_i$ → `hint_n` (normalized canny)
- $\mathcal{E}(\cdot)$ → ControlNet's internal "hint encoder"
- $\mathbf{c}_f$ → features used to produce `cn_out.down_block_res_samples` & `cn_out.mid_block_res_sample`

Training objective

$$\mathcal{L} = \mathbb{E}_{\boldsymbol{z}_0, \boldsymbol{t}, \boldsymbol{c}_t, \boldsymbol{c}_\mathrm{f}, \epsilon \sim \mathcal{N}(0,1)} \left[\| \epsilon - \epsilon_\theta(\boldsymbol{z}_t, \boldsymbol{t}, \boldsymbol{c}_t, \boldsymbol{c}_\mathrm{f}) \|_2^2 \right],$$

Copy code

```
# --- build z0 from image via VAE (latent) ---
img_n = img * 2.0 - 1.0                  # [-1,1] (SD/VAEs expect this)
lat   = vae.encode(img_n).latent_dist.sample() * 0.18215   # z0

# --- sample t and epsilon, make zt ---
B  = lat.shape[0]
ts = torch.randint(0, noise_scheduler.num_train_timesteps, (B,), device=DEVICE).long()  # t
eps = torch.randn_like(lat)                                              # ε ~ N(0,I)
nlat = noise_scheduler.add_noise(lat, eps, ts)                           # zt

# --- predict εϑ(·) with text + control features (see Eq. 2/4 below) ---
cn_out = controlnet(
    sample=nlat, timestep=ts, encoder_hidden_states=txt_emb,
    controlnet_cond=hint_n, return_dict=True
)
pred = unet(
    nlat, ts, encoder_hidden_states=txt_emb,
    down_block_additional_residuals=cn_out.down_block_res_samples,
    mid_block_additional_residual=cn_out.mid_block_res_sample,
    return_dict=True
).sample  # εϑ(zt, t, ct, cf)

# --- loss: ||ε - εϑ||^2 ---
loss = mse(pred, eps) / GRAD_ACCUM
```

Term → variable

- $\mathbf{z}_0$ → `lat` (from VAE)
- t → `ts`
- ϵ → `eps`
- $\mathbf{z}_t$ → `nlat`
- $\epsilon_\theta(\cdot)$ → `pred`
- $\mathbf{c}_t$ (text cond) → `txt_emb`
- $\mathbf{c}_f$ (hint features) → inside `cn_out` (produced by ControlNet's hint encoder from `hint_n`)

Inference

python src/infer_controlnet.py \

Done for 30 inference steps for each example.

Key upgrades that made the difference

a. **ControlNet construction**

- **Before:** ControlNetModel.from_config(unet.config) + custom Kaiming init.

- **Now:** ControlNetModel.from_unet(unet) (paper-faithful, zero-conv connectors wired exactly like SD's blocks).
- **Why it helps:** Guarantees the right block topology + proper zero-conv init so the control residuals blend safely into the frozen UNet (no shape/scale mismatch, no unstable learning at the start).

b. Correct VAE input normalization

- **Before:** vae.encode(images) with images in [0,1].
- **Now:** img_n = img*2-1 → vae.encode(img_n).
- **Why it helps:** SD's VAE expects inputs in [-1,1]. Feeding [0,1] distorts latents; fixing this dramatically improves training signal and output fidelity.

c. Noise schedule matched to SD 1.5

- **Before:** default DDPMScheduler(...) (implicit betas).
- **Now:** DDPMScheduler(beta_start=0.00085, beta_end=0.012, beta_schedule="scaled_linear").
- **Why it helps:** These betas are what SD 1.5 uses; aligning the train noise schedule with the base model stabilizes ε-learning and speeds convergence.

d. Learning rate schedule + warmup

- **Before:** constant LR, no warmup.
- **Now:** get_cosine_schedule_with_warmup(..., warmup_steps=1000).
- **Why it helps:** Warmup prevents early overshoot; cosine decay gives smoother late training, reducing loss faster in fewer epochs.

e. Regularization tuned for a small trainable branch

- **Before:** weight_decay=0.0.
- **Now:** weight_decay=1e-4 (mild).
- **Why it helps:** Small L2 helps stabilize ControlNet's new layers without choking learning; reduces noisy weights and improves generalization.

f. Batching / memory behavior

- **Before:** BATCH_SIZE=4, GRAD_ACCUM=2 (micro-batch 4).
- **Now:** BATCH_SIZE=1, GRAD_ACCUM=8 (same effective batch = 8).
- **Why it helps:** Smaller micro-batch lowers VRAM spikes and reduces gradient noise/variance between micro-steps, giving steadier updates on Kaggle GPUs.

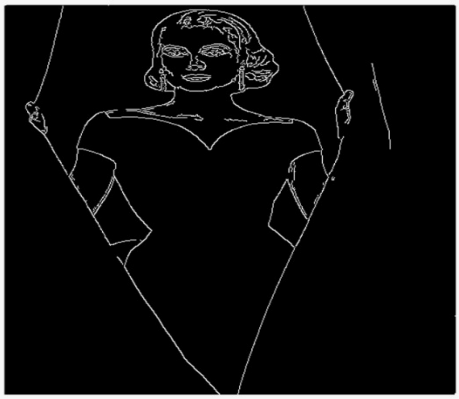
g. Inference aligned with training

- **Before:** often CFG=7.5, 28–50 steps, control image resized with LANCZOS, sometimes 512 inference on a 256-trained model.
- **Now: match train res (256)**, **CFG=4.0** (lets control dominate), **steps≈30**, **NEAREST** for the control resize, same scaled_linear scheduler, enable xFormers if available, half-precision on CUDA.
- **Why it helps:** Matching train/infer resolution and using lower CFG are the biggest wins for edge faithfulness; NEAREST preserves edges at inference too.

---

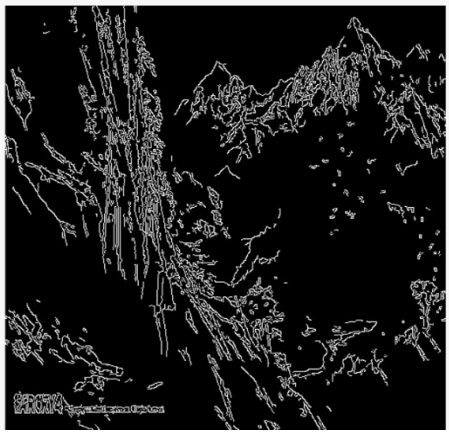
6. Results (Qualitative)

The results with simple line by line implementation of ControlNet motivated by the paper:

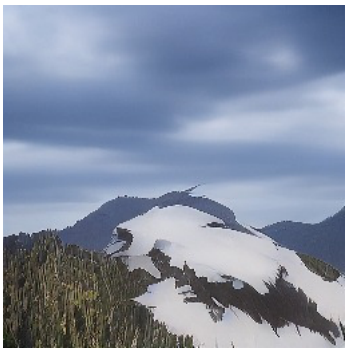
Prompt	Canny	Output
an eagle fying in sky		
anime style hatsune miku		

Clearly the output images are not consistent with either the prompt or the input canny images.

Now with the changes mentioned above:

Prompt	Canny	Output
sunrays with mountains and grass		

Prompt	Canny	Output
four humans on adventure in forest with waterfalls		




Comparison through the number of epochs:


For the same canny but different prompt. The original canny image for all the three images is in the second picture.

- **What matched:** edge adherence.
- **What differed from paper scale:** smaller dataset (**3316**) and lower resolution (**256 by 256**) push stronger reliance on the control signal; captions quality also matters.
- **Practical:** filename-caption alignment is critical; resolution mismatch and prompting strongly affect results.

---

7. Limitations

- Training at **256×256** limits fine texture; output images are noisy/short

- Single condition (Canny) only
- No quantitative large-scale benchmarks.

---

8. Future Work

- Train at **512×512**; add **hint dropout** (e.g., 5–10%) once fidelity is good; explore additional conditions (HED, depth, pose); multi-condition composition; lightweight LoRA for faster style adaptation.

---

9. Reproducibility

- **Environment:** see requirements.txt.
- **Seeds:** default seed **42**.
- **Checkpoints:** saved every **500** steps under checkpoints/run_256/.
- **How to run:** commands in §5; small demo images under.

References

- Zhang, Lvmin; Rao, Anyi; Agrawala, Maneesh. *Adding Conditional Control to Text-to-Image Diffusion Models (ControlNet)*, 2023.

Appendix A — Key Hyperparameters

Component	Value
Image size	256×256
Batch / Accum	1 / 8 (effective 8)
Optimizer	AdamW
LR / Schedule	1e-5 / cosine + warmup (1k)
Weight decay	1e-4
FP16	Yes
Drop hint	0.0 (initial)
Scheduler	DDPM (scaled_linear)
Saves	every 500 steps

Appendix B — File Structure

```
.
├– src/
|   ├– train_controlnet.py
|   └– infer_controlnet.py
├– scripts/
|   └– zip_run.py
├– configs/
|   └– train_256.yaml          # (optional; mirror hyperparams)
├– data/
|   └– your_dataset/           # images/, canny/, captions.txt  (not committed)
├– checkpoints/               # training outputs
├– outputs/                   # generated samples
├– reports/
|   └– report.md
├– assets/                    # small demo images & result grids
├– requirements.txt
├– README.md
└– .gitignore
```