Memory-Efficient Fine-Tuning of Stable Diffusion XL

Parameter-Efficient Adaptation for Style Transfer under Hardware Constraints

Harsh Verma
harshv034@gmail.com

December 2025

Abstract

We present a practical approach to fine-tuning Stable Diffusion XL (SDXL), a multi-billion-parameter text-to-image diffusion model, on a single NVIDIA T4 GPU with 16GB VRAM. Through a combination of Low-Rank Adaptation (LoRA), gradient checkpointing, 8-bit optimization, and resolution/architecture choices, we substantially reduce the effective memory footprint while maintaining model quality. Using this setup, we adapt SDXL to generate images in the Naruto anime art style on the `lambdalabs/naruto-blip-captions` dataset, showing that large-scale diffusion models can be fine-tuned under realistic, free-tier Colab hardware constraints.

1 Introduction

1.1 Problem Statement

Stable Diffusion XL (SDXL) represents the state-of-the-art in text-to-image generation, with a multi-billion-parameter architecture and native 1024×1024 resolution. However, standard full-model fine-tuning has very high memory requirements. A naive configuration that keeps all components trainable at full resolution can require tens of gigabytes of VRAM for:

- Model weights (UNet, text encoders, VAE)
- Optimizer states for all parameters
- Gradients for all parameters
- Activations for high-resolution feature maps

This makes straightforward full-model fine-tuning infeasible on a single 16GB GPU.

1.2 Our Solution

We combine multiple orthogonal optimization techniques to substantially reduce memory footprint while preserving model quality:

1. **LoRA**: Train only 0.9% of parameters
2. **Resolution Reduction**: 512×512 instead of 1024×1024
3. **Gradient Checkpointing**: Recompute activations
4. **8-bit Optimization**: Quantize optimizer states
5. **XFormers Attention**: Efficient attention computation
6. **FP32 VAE**: Numerical stability for latents

2 Background

2.1 Diffusion Models

Diffusion models learn to denoise images through a Markov chain. Given a data distribution $q(\mathbf{x}_0)$, the forward process adds Gaussian noise:

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \tag{1}$$

The reverse process learns to predict the noise:

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t)) \tag{2}$$

The training objective minimizes:

$$\mathcal{L} = \mathbb{E}_{t,\mathbf{x}_0,\boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t, \mathbf{c})\|^2\right] \tag{3}$$

where $\mathbf{c}$ is the text conditioning, $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$, and $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}$.

2.2 Stable Diffusion XL Architecture

SDXL operates in latent space using a VAE encoder $\mathcal{E}$ and decoder $\mathcal{D}$:

$$\mathbf{z} = \mathcal{E}(\mathbf{x}), \quad \mathbf{x}' = \mathcal{D}(\mathbf{z}) \tag{4}$$

The latent space has dimension $\mathbf{z} \in \mathbb{R}^{h/8 \times w/8 \times 4}$, providing $8\times$ spatial compression.
Key components:

- **UNet**: $\epsilon_\theta(\mathbf{z}_t, t, \mathbf{c})$ - 2.6B parameters
- **Text Encoders**: Dual CLIP (OpenCLIP-ViT-G, OpenCLIP-ViT-L) - 817M parameters
- **VAE**: Encoder/Decoder - 83M parameters

3 Methodology

3.1 Low-Rank Adaptation (LoRA)

3.1.1 Theoretical Foundation

LoRA [1] hypothesizes that weight updates during fine-tuning have low intrinsic rank. For a pre-trained weight matrix $\mathbf{W}_0 \in \mathbb{R}^{d\times k}$, instead of learning $\Delta\mathbf{W} \in \mathbb{R}^{d\times k}$, we learn:

$$\mathbf{W} = \mathbf{W}_0 + \Delta\mathbf{W} = \mathbf{W}_0 + \mathbf{BA} \tag{5}$$

where $\mathbf{B} \in \mathbb{R}^{d\times r}$, $\mathbf{A} \in \mathbb{R}^{r\times k}$, and $r \ll \min(d, k)$.

3.1.2 Forward Pass

$$\mathbf{h} = \mathbf{W}_0\mathbf{x} + \frac{\alpha}{r}\mathbf{BAx} \tag{6}$$

where α is a scaling hyperparameter. We use $r = 16$ and $\alpha = 32$.

3.1.3 Parameter Reduction

Original parameters: $N_0 = d \times k$

LoRA parameters: $N_{LoRA} = r(d + k)$

Reduction ratio:

$$\rho = \frac{N_{LoRA}}{N_0} = \frac{r(d+k)}{dk} \approx \frac{2r}{d} \quad (\text{for } d \approx k) \tag{7}$$

For SDXL UNet with $d = 1280$ and $r = 16$:

$$\rho \approx \frac{2 \times 16}{1280} = 0.025 = 2.5\% \tag{8}$$

3.1.4 Applied Layers

We apply LoRA to attention layers only:

- Query projection: $\mathbf{W}_Q$
- Key projection: $\mathbf{W}_K$
- Value projection: $\mathbf{W}_V$
- Output projection: $\mathbf{W}_O$

Total trainable parameters are on the order of tens of millions, which is a small fraction of the full model.

3.2 Resolution Reduction

3.2.1 Memory Scaling

Memory consumption scales quadratically with resolution for latents and activations:

$$M(h, w) \propto (h/8)^2 \times (w/8)^2 \times C \tag{9}$$

For resolution reduction $512^2 \to 1024^2$:

$$\frac{M_{512}}{M_{1024}} = \left(\frac{512}{1024}\right)^2 = \frac{1}{4} \tag{10}$$

Memory savings: 75% for latent-dependent components.

3.2.2 Latent Dimensions

$$1024\times1024: \quad \mathbf{z} \in \mathbb{R}^{128\times128\times4} \quad (65,536 \text{ spatial locations}) \tag{11}$$

$$512\times512: \quad \mathbf{z} \in \mathbb{R}^{64\times64\times4} \quad (16,384 \text{ spatial locations}) \tag{12}$$

3.3 Gradient Checkpointing

3.3.1 Activation Memory Problem

For a network with L layers and batch size B:

$$M_{activations} = \sum_{i=1}^{L} B \cdot d_i \tag{13}$$

For SDXL UNet with 70+ layers and $d_i \sim 10^6$, this exceeds available memory.

3.3.2 Checkpointing Strategy

Instead of storing all activations, we:

1. Store activations only at checkpoint boundaries
2. Recompute intermediate activations during backward pass

$$M_{checkpointed} = \frac{M_{activations}}{\sqrt{L}} + O(1) \tag{14}$$

For $L = 70$: reduction factor $\approx \sqrt{70} \approx 8.4$
Trade-off: Increases computation by ∼20% but reduces memory by 60-80%.

3.4 8-bit Optimization

3.4.1 Adam Optimizer States

Standard Adam maintains two states per parameter:

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\mathbf{g}_t \quad \text{(first moment)} \tag{15}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)\mathbf{g}_t^2 \quad \text{(second moment)} \tag{16}$$

Update rule:

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} \tag{17}$$

Memory for FP32 Adam: $3 \times 4N$ bytes (N = number of parameters)

3.4.2 8-bit Quantization

Quantize $\mathbf{m}_t$ and $\mathbf{v}_t$ to INT8:

$$\mathbf{m}_t^q = \text{round}\left(\frac{\mathbf{m}_t - \min(\mathbf{m}_t)}{\max(\mathbf{m}_t) - \min(\mathbf{m}_t)} \times 255\right) \tag{18}$$

Store scale factors: $s_m = \frac{\max(\mathbf{m}_t) - \min(\mathbf{m}_t)}{255}$
Memory for 8-bit Adam: $N + 1N$ bytes (75% reduction)
For 35M LoRA parameters:

$$\text{FP32 Adam:} \quad 3 \times 35M \times 4 = 420 \text{ MB} \tag{19}$$

$$\text{8-bit Adam:} \quad 35M \times 2 = 70 \text{ MB} \tag{20}$$

3.5 Gradient Accumulation

Simulates larger batch size without memory increase:
Effective batch size: $B_{eff} = K \times B_{physical}$
Our configuration: $K = 4$, $B_{physical} = 1 \Rightarrow B_{eff} = 4$

Algorithm 1 Gradient Accumulation

Initialize $\nabla_\theta \mathcal{L} \leftarrow 0$
for $i = 1$ to K **do**
    Sample mini-batch $\mathcal{B}_i$
    Compute loss $\mathcal{L}_i$ on $\mathcal{B}_i$
    $\nabla_\theta \mathcal{L} \leftarrow \nabla_\theta \mathcal{L} + \frac{1}{K} \nabla_\theta \mathcal{L}_i$
end for
Update: $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$

3.6 XFormers Attention

3.6.1 Standard Attention Memory

Standard scaled dot-product attention:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V} \tag{21}$$

Memory for attention matrix: $O(N^2 \times B)$ where N is sequence length.
For 512×512 latents: $N = 64 \times 64 = 4096$
Attention matrix size: $4096^2 \times 4 = 67$ MB per head, per batch

3.6.2 XFormers Optimization

Uses flash attention and block-sparse patterns:

$$M_{xformers} = O(N \times B) \quad \text{vs.} \quad M_{standard} = O(N^2 \times B) \tag{22}$$

Memory reduction: ∼60% for attention layers

3.7 Full Precision VAE

The VAE operates with:

$$\mathbf{z} = \mu + \sigma \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I}) \tag{23}$$

Numerical issues in FP16:

- Underflow in σ values
- Overflow in exponentials during reparameterization
- NaN propagation in gradients (though VAE is frozen)

Solution: Keep VAE in FP32 for stable latent encoding.

4 Training Configuration

4.1 Hyperparameters

4.2 Learning Rate Selection

LoRA typically requires lower learning rates than full fine-tuning:

$$\eta_{LoRA} = \frac{\alpha}{r} \cdot \eta_{base} = \frac{32}{16} \cdot 2.5 \times 10^{-5} = 5 \times 10^{-5} \tag{24}$$

This accounts for the LoRA scaling factor in the forward pass.

Parameter	Value
Base Model	stabilityai/stable-diffusion-xl-base-1.0
Dataset	lambdalabs/naruto-blip-captions (439 images)
Resolution	512×512
Batch Size (physical)	1
Gradient Accumulation Steps	4
Effective Batch Size	4
Learning Rate	5×10^{-5}
Optimizer	AdamW (8-bit)
LoRA Rank (r)	16
LoRA Alpha (α)	32
Max Gradient Norm	0.5
Training Steps	1000
Warmup Steps	$100 \times$ Gradient Accumulation (400)
LR Schedule	Constant (post-warmup)
Checkpoint Frequency	Every 200 steps
Random Seed	42

Table 1: Training hyperparameters

4.3 Data Augmentation

Applied transformations:

- Resize to 512 (bilinear interpolation)
- Center crop 512×512
- Random horizontal flip (p=0.5)
- Normalize: $\tilde{\mathbf{x}} = \frac{\mathbf{x}-0.5}{0.5} \in [-1, 1]$

5 Memory Budget Discussion

Exact VRAM usage depends on the PyTorch/CUDA version and Colab runtime, but the combination of techniques in this work keeps peak memory within the 16GB budget of a T4. Qualitatively:

- Only LoRA adapter weights require gradients and optimizer states; the large base weights are frozen.
- 8-bit Adam reduces optimizer state memory for LoRA parameters compared to standard FP32 Adam.
- Gradient checkpointing and 512×512 resolution keep activation memory manageable.
- VAE and text encoders are used in FP32 but are frozen, so they do not add optimizer or gradient overhead.

Empirically, this configuration runs on a Colab T4 without out-of-memory errors while training at 512×512 with batch size 1 and gradient accumulation.

Algorithm 2 SDXL LoRA Fine-tuning with Memory Optimization

Input: Dataset $\mathcal{D}$, base model θ_0, LoRA rank r
Initialize: LoRA matrices $\mathbf{B}, \mathbf{A}$ with Gaussian noise
Freeze θ_0, VAE, text encoders
Enable gradient checkpointing on UNet
Initialize 8-bit AdamW optimizer
for step $= 1$ to N_{steps} **do**
    Zero accumulated gradients
    for $i = 1$ to K_{accum} **do**
        Sample batch $(\mathbf{x}, \mathbf{c}) \sim \mathcal{D}$
        $\mathbf{z} \leftarrow \text{VAE.encode}(\mathbf{x})$ ▷ FP32
        $\mathbf{z} \leftarrow \mathbf{z} \times 0.18215$ ▷ Scaling factor
        Sample $t \sim \text{Uniform}(1, T)$
        Sample $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$
        $\mathbf{z}_t \leftarrow \sqrt{\bar{\alpha}_t}\mathbf{z} + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}$
        $\mathbf{e}_1, \mathbf{e}_2 \leftarrow \text{TextEncoders}(\mathbf{c})$
        $\mathbf{e} \leftarrow \text{concat}([\mathbf{e}_1, \mathbf{e}_2], \text{dim} = -1)$
        $\hat{\boldsymbol{\epsilon}} \leftarrow \text{UNet}(\mathbf{z}_t, t, \mathbf{e})$ ▷ With LoRA
        $\mathcal{L} \leftarrow \|\hat{\boldsymbol{\epsilon}} - \boldsymbol{\epsilon}\|^2$
        Accumulate: $\nabla\mathcal{L} \leftarrow \nabla\mathcal{L} + \frac{1}{K_{accum}}\nabla_{\mathbf{B},\mathbf{A}}\mathcal{L}$
    end for
    Clip gradients: $\|\nabla\mathcal{L}\| \leftarrow \min(\|\nabla\mathcal{L}\|, \gamma)$
    Update LoRA: $\mathbf{B}, \mathbf{A} \leftarrow \text{AdamW}(\mathbf{B}, \mathbf{A}, \nabla\mathcal{L})$
    if step mod $N_{checkpoint} = 0$ **then**
        Save LoRA weights
    end if
end for
Return: Fine-tuned LoRA adapters

6 Training Algorithm

7 Observed Behaviour

7.1 Loss Trajectory

With the configuration in Table 1 (LoRA on UNet, 512×512, effective batch size 4, 1000 steps), the training loss decreases over time and remains finite (no NaN or Inf values) thanks to the stability checks in the implementation. The exact numerical trajectory depends on random seed and Colab runtime, but qualitatively:

- The loss starts relatively high in the early steps and steadily decreases as the model adapts to the Naruto style.
- Occasional noisy steps do not derail training due to gradient clipping and NaN/Inf checks.
- After several hundred steps, checkpoints already show visible style transfer in generated images.

7.2 Training Time

On a Colab T4 GPU, wall-clock training time for 1000 steps with gradient accumulation can vary depending on runtime load and background processes. In practice:

- Per-step time (forward + backward) is on the order of a few seconds.
- Running 1000 optimizer updates with accumulation fits comfortably within typical free-tier Colab GPU session limits (a few hours), allowing the full run to complete without exhausting the allowed time.

8 Inference

8.1 Loading Fine-tuned Model

```
from diffusers import DiffusionPipeline
import torch

# Load base pipeline
pipe = DiffusionPipeline.from_pretrained(
    "stabilityai/stable-diffusion-xl-base-1.0",
    torch_dtype=torch.float16
).to("cuda")

# Inject LoRA weights
pipe.load_lora_weights("./sdxl-naruto-lora")

# Generate
image = pipe(
    "Naruto Uzumaki eating ramen",
    num_inference_steps=30,
    guidance_scale=7.5
).images[0]
```

9 Limitations and Future Work

9.1 Current Limitations

1. **Resolution**: Training at 512×512 may limit fine detail capture
2. **Dataset Size**: 439 images may cause overfitting
3. **LoRA Rank**: $r = 16$ may be insufficient for complex styles
4. **Training Duration**: 1000 steps may be suboptimal

9.2 Potential Improvements

9.2.1 Progressive Resolution Training

Train in stages:

$$512^2 \xrightarrow{\text{500 steps}} 768^2 \xrightarrow{\text{500 steps}} 1024^2 \tag{25}$$

9.2.2 Dynamic LoRA Rank

Adaptive rank based on layer importance:

$$r_l = r_{base} \times \text{importance}_l \tag{26}$$

9.2.3 Offset Noise Training

Add constant offset to improve dark/light image generation:

$$\boldsymbol{\epsilon}' = \boldsymbol{\epsilon} + \delta, \quad \delta \sim \mathcal{N}(0, 0.1) \tag{27}$$

9.2.4 Mixed Precision Re-enablement

Carefully tune loss scaling for FP16 training:

$$\mathcal{L}_{scaled} = \mathcal{L} \times s, \quad s = 2^{10} \text{ to } 2^{15} \tag{28}$$

10 Conclusion

We demonstrated that SDXL, a large text-to-image model with billions of parameters, can be successfully fine-tuned on a single 16GB GPU through an appropriate combination of:

- Parameter-efficient methods (LoRA): training only a small fraction of parameters
- Memory-efficient training (checkpointing): reducing activation memory at the cost of modest extra compute
- Quantized optimization (8-bit Adam): lowering optimizer memory footprint
- Resolution adaptation: reducing latent and activation sizes by training at 512×512 instead of 1024×1024

This enables democratized access to large-scale diffusion model fine-tuning, making state-of-the-art image generation accessible on consumer hardware.

References

[1] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., ... & Chen, W. (2021). *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv preprint arXiv:2106.09685.

[2] Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. (2022). *High-Resolution Image Synthesis with Latent Diffusion Models*. CVPR 2022.

[3] Podell, D., English, Z., Lacey, K., Blattmann, A., Dockhorn, T., Müller, J., ... & Rombach, R. (2023). *SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis*. arXiv preprint arXiv:2307.01952.

[4] Chen, T., Xu, B., Zhang, C., & Guestrin, C. (2016). *Training Deep Nets with Sublinear Memory Cost*. arXiv preprint arXiv:1604.06174.

[5] Dettmers, T., Lewis, M., Belkada, Y., & Zettlemoyer, L. (2022). *8-bit Optimizers via Block-wise Quantization*. ICLR 2022.

[6] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., & Ré, C. (2022). *FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness*. NeurIPS 2022.

[7] Ruiz, N., Li, Y., Jampani, V., Aberman, K., ... & Rubinstein, M. (2022). *DreamBooth: Fine Tuning Text-to-Image Diffusion Models for Subject-Driven Generation*. arXiv preprint arXiv:2208.12242.

[8] Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., ... & Gelly, S. (2019). *Parameter-Efficient Transfer Learning for NLP*. In Proceedings of the 36th International Conference on Machine Learning (ICML).

[9] Lester, B., Al-Rfou, R., & Constant, N. (2021). *The Power of Scale for Parameter-Efficient Prompt Tuning*. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP).