

小作业-视觉 LLM 的理解和生成

杨天行 2024310659

2025 年 1 月 9 日

1 实验任务

本实验的主要任务是基于扩散模型 (Diffusion Models) 的原理和实现，手动完成 Stable Diffusion 模型的风格微调，以加强对扩散模型工作原理和微调过程的深入理解。具体任务分为以下几个方面：

理论调研。研究当前扩散模型的基本结构，包括但不限于网络结构、噪声调度机制以及采样算法。理解 Stable Diffusion 模型的架构特点，为微调提供理论支持。

代码实现。使用 Python 编程语言与 diffusers 框架（或其他基础工具），手动实现 Stable Diffusion 模型的风格微调，允许选择 LoRA（低秩适配）或全参数微调方法。实现模型从通用任务到特定风格图像生成任务的迁移，同时尽可能保留模型的语义理解能力。编写清晰、模块化的代码，便于调试和复用，并在实验报告中详细解释代码的实现原理和细节。

风格微调与应用。准备一个或多个风格数据集，基于数据集对模型进行微调。在微调完成后测试生成模型的性能，使用清晰的视觉化效果（包括图像截图）展示模型在不同风格生成任务上的表现。

模型效果测试。定义和说明具体的测试方法及测试数据集。比较微调前后模型在图像生成质量和风格适应能力方面的差异，验证微调的有效性。

2 Diffusion Model 的原理与算法

2.1 Diffusion Model 基本结构

扩散模型 (Diffusion Model) 是一类基于逐步对数据注入噪声，并从噪声中反向还原数据的生成式模型，在图像生成、语音合成、视频生成等任务上取得了优异表现。其核心思想是：

- **正向扩散过程 (Forward Diffusion)**: 从真实数据分布 $q(\mathbf{x}_0)$ 开始，每一步将随机噪声逐渐加入到数据中，最终会得到近似于各向同性高斯分布的高噪声数据
- **反向去噪过程 (Reverse Diffusion)**: 训练一个模型来学习从高噪声数据逐步去噪的过程，即通过条件概率分布 $p_\theta(\mathbf{x}_{t-1} | \mathbf{x}_t)$ 的迭代采样，将噪声一步步去除，最终还原出清晰图像（或其他形式的数据）

扩散模型的高效性主要来自：

- 其训练目标可与变分下界 (VLB) 联系，确保了理论上的可行性和稳定性
- 正向扩散过程可以被设计为可以解析计算的马尔可夫链，因此我们能够基于高斯分布的性质简化训练和采样
- 反向过程在去噪的过程中能学习到多个尺度上的数据结构，对复杂分布有强大表达能力

在图像扩散模型中，最常见也是最核心的网络结构通常是 **U-Net**。U-Net 最初用于医学图像分割，但在扩散模型中具有如下改动或特征：

- **Downsampling-中间瓶颈层-Upsampling 结构。** Downsampling 逐层将特征图缩小，提取更深层次的语义信息；Bottleneck（中间瓶颈层）在最小分辨率下进行跨层特征融合或者引入自注意力（Self-Attention）模块；Upsampling 通过反卷积或插值等方式逐层恢复分辨率，合并 skip connection 传来的特征，以保留浅层的局部细节。
- **多尺度自注意力。** 为了更好地捕捉全局依赖，扩散模型往往会在 U-Net 的中高层分辨率阶段加入 Attention Layer。与纯卷积结构相比，引入自注意力可以帮助模型在生成图像时具有更强的全局一致性和上下文理解。
- **Time Step 或 Diffusion Step 的条件输入。** 在扩散模型中，每一个去噪步骤的网络输入不仅包含了带噪图像，还包含当前时间步 t 的信息。时间步信息通常通过可学习的位置编码或其他嵌入方式加入到网络中，使网络能够区分不同的去噪阶段。
- **可选的条件输入（如文本、标签等）** 在条件生成任务中（如文本到图像，类别标记到图像等），会额外输入文本编码（如 CLIP 文本编码）、或类别标记等信息。这些条件通常通过 Cross-Attention 或拼接在特征通道中的方式与 U-Net 主干结合，从而使模型在去噪时能够“考虑”条件信息并生成相匹配的图像。

2.2 噪声调度

噪声调度描述了正向扩散时在每一步向样本中注入噪声的方式以及反向扩散时去噪的强弱。常见的噪声调度方式包括：

- 线性噪声调度 (Linear Schedule)：将噪声方差从较小数值逐步线性地增加到较大数值，类似于 β_t 线性递增， $\beta_t \in [\beta_{\min}, \beta_{\max}]$ 。
- 余弦噪声调度 (Cosine Schedule)：使用余弦函数进行噪声量的安排，使得在前期噪声注入较慢，在后期噪声注入更快（或反之），从而可能更好地保持早期图像结构或后期细节。

2.3 采样算法

在训练完扩散模型（即学得了从任意噪声状态到更少噪声状态的映射）之后，推理/生成时需要从纯噪声采样开始，逐步去噪得到清晰图像。常见的采样算法有：

- DDPM (Denoising Diffusion Probabilistic Model) 在第 T 步从高斯噪声出发，每一步使用模型预测的均值和方差进行随机采样。该方法采样速度较慢，需要完整 T 步才能完成一次图像生成。
- DDIM (Denoising Diffusion Implicit Models) 提出了一种确定性采样方式，可以在不降低生成质量的前提下减少采样步数。可以视为 DDPM 的一个特例或变体；在降采样步数时可大幅提升生成速度。
- PLMS (Pseudo Linear Multi-Step) 在采样阶段使用多步线性多项式方法进行去噪，减少所需采样步数。相比 DDIM，PLMS 更稳定，并在一定程度上能提升图像保真度。

现在也有其他改进方法，例如 DPM-Solver、DEIS (DPM Efficient Integrator Sampler) 等，均是对采样阶段的加速和质量平衡做出的改进尝试。

2.4 Stable Diffusion

Stable Diffusion 是由 Stability AI、Runway、LAION 等团队共同开发的文本到图像生成模型。它与传统的扩散模型相比有几个显著特点：

1. Latent Diffusion Model (LDM) 体系

Stable Diffusion 并不直接在像素空间进行扩散，而是将图像首先通过预训练的自动编码器（VAE）压缩到一个相对低维的 latent space 中，然后在这个 latent space 中进行扩散和去噪。具体流程如下：

- 预训练 VAE：训练一个自动编码器（Autoencoder）将图像从像素空间映射到潜空间（Encoder），再从潜空间重建回图像（Decoder）。通过对图像进行压缩，可极大减少需要处理的特征维度，使得后续的扩散过程训练更高效且在硬件上更易实现。
- 在 Latent Space 进行扩散：将压缩后的 latent 表示视为“图像”，通过与之前介绍的扩散模型类似的方式，在潜空间中加入噪声并训练去噪网络（U-Net + Cross-Attention）。相比直接在像素空间进行扩散，更能节约计算资源，同时能保留足够多的图像语义信息。
- 推理阶段解码：生成过程完成后会得到在潜空间中的去噪结果，再通过训练好的 VAE Decoder 将其还原成可视图像。

2. 文本条件与 Cross-Attention

Stable Diffusion 是一个文本到图像的生成模型，通常以“Prompt”的形式输入文本描述。最常见的做法是利用 CLIP 模型或类似文本编码器提取文本语义向量，并将其输入扩散网络中的 Cross-Attention 模块。Cross-Attention 的工作机制是对于 U-Net 的某些层，将潜空间特征作为“查询”（Query），文本编码向量作为“键-值”（Key-Value），使网络能根据文本信息对潜空间特征进行加权、融合。不同层次的 Cross-Attention 能让模型同时掌握文本内容概念和图像局部特征，从而更加精准地生成与 Prompt 对应的图像内容。

3. 采样与推理

Stable Diffusion 推理时，常用的采样算法包括 DDIM、PLMS 等，也可以借助其他改进方法（如 DPM-Solver）。为了加速推理，模型可以减少采样步数（例如从默认的 50 步降到 20 步左右），在生成质量和速度之间取得平衡。

3 Stable Diffusion 微调

3.1 数据集准备

本次实验一共尝试了宝可梦风格图片数据集 svjack/pokemon-blip-captions-en-zh，纽约客杂志插图风格图片数据集 jmhessel/newyorker_caption_contest、动漫图片数据集 lambdalabs/naruto-blip-captions、辛普森动画图片数据集 jinmel/simpsons-blip-captions-pil。这些数据集均为 Hugging Face 上的公开数据集，拥有图片、文本两个模态，可以用于 Stable Diffusion 微调。下面是 4 个数据集中的示例图片：

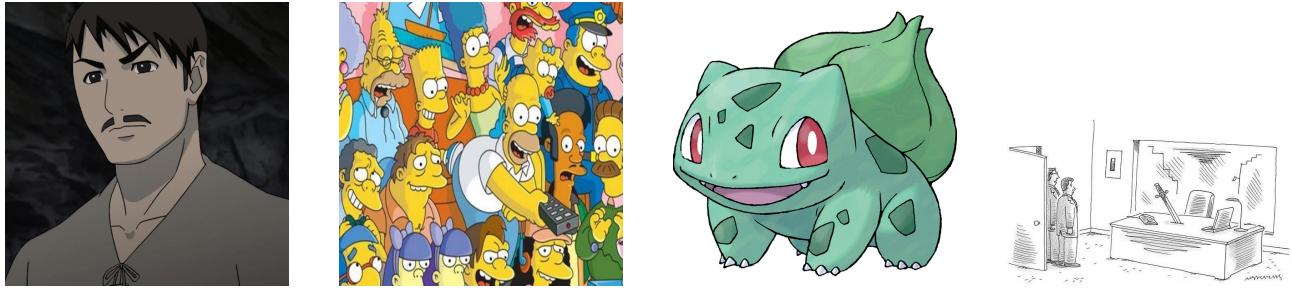


图 1: 图片示例

3.2 微调代码实现

首先我们利用 dataset 库加载 hugging face 上的数据集，设定对应的 image/caption column 进行加载。对于文本模态的数据，使用预训练的 CLIP Text Tokenizer 进行 token 化，图片模态的数据则是按照固定的 transform pipeline 进行（将图片进行双线性插值、裁剪为正方形、随机翻转、正则化），使用 data_loader 加载。

```

1  dataset_name = "skiracer/simpsons_blip_captions"
2  dataset = load_dataset(dataset_name)
3  dataset_columns = DATASET_NAME_MAPPING.get(dataset_name, None)
4
5  image_column = dataset_columns[0]
6  caption_column = dataset_columns[1]
7
8  def tokenize_captions(examples, is_train=True):
9      captions = []
10     for caption in examples[caption_column]:
11         if isinstance(caption, str):
12             captions.append(caption)
13         elif isinstance(caption, (list, np.ndarray)):
14             # take a random caption if there are multiple
15             captions.append(random.choice(caption) if is_train else caption[0])
16         else:
17             raise ValueError(
18                 f"Caption column '{caption_column}' should contain either strings or
19                 lists of strings."
20             )
21     inputs = tokenizer(
22         captions, max_length=tokenizer.model_max_length, padding="max_length",
23         truncation=True, return_tensors="pt"
24     )
25     return inputs.input_ids
26
27     # Preprocessing the datasets.
28     train_transforms = transforms.Compose([
29         transforms.Resize(args.resolution,

```

```

29         interpolation=transforms.InterpolationMode.BILINEAR),
30         transforms.CenterCrop(args.resolution) if args.center_crop else
31             transforms.RandomCrop(args.resolution),
32         transforms.RandomHorizontalFlip() if args.random_flip else
33             transforms.Lambda(lambda x: x),
34         transforms.ToTensor(),
35         transforms.Normalize([0.5], [0.5]),
36     ]
37 )

```

在主体训练流程中，对于一个 batch 的数据，首先随机采样噪声（shape 与输出的图片相同），在随机的时间步上训练。对于固定的 input prompt，计算对应的 encoder hidden states，使用 unet 模型推理，输入包含噪声信息与文本 prompt 对应的 hidden state，timestamp 信息，计算输出去噪后的图片与目标图片的插值作为损失函数。最后进行反向传播，并在 epoch 累计一定次数时保存模型 checkpoint。

```

1   for epoch in range(first_epoch, args.num_train_epochs):
2       train_loss = 0.0
3       for step, batch in enumerate(train_dataloader):
4           with accelerator.accumulate(unet):
5               # Convert images to latent space
6               latents =
7                   vae.encode(batch["pixel_values"].to(weight_dtype)).latent_dist.sample()
8               latents = latents * vae.config.scaling_factor
9
10              # Sample noise that we'll add to the latents
11              noise = torch.randn_like(latents)
12              if args.noise_offset:
13                  # https://www.crosslabs.org//blog/diffusion-with-offset-noise
14                  noise += args.noise_offset * torch.randn(
15                      (latents.shape[0], latents.shape[1], 1, 1), device=latents.device
16                  )
17              if args.input_perturbation:
18                  new_noise = noise + args.input_perturbation * torch.randn_like(noise)
19                  bsz = latents.shape[0]
20                  # Sample a random timestep for each image
21                  timesteps = torch.randint(0, noise_scheduler.config.num_train_timesteps,
22                                  (bsz,), device=latents.device)
23                  timesteps = timesteps.long()
24
25                  # Add noise to the latents according to the noise magnitude at each
26                  # timestep
27                  # (this is the forward diffusion process)
28                  if args.input_perturbation:
29                      noisy_latents = noise_scheduler.add_noise(latents, new_noise,

```

```
30     # Get the text embedding for conditioning
31     encoder_hidden_states = text_encoder(batch["input_ids"],
32                                         return_dict=False)[0]
33
34     # Get the target for loss depending on the prediction type
35     if args.prediction_type is not None:
36         # set prediction_type of scheduler if defined
37         noise_scheduler.register_to_config(prediction_type=args.prediction_type)
38
39         if noise_scheduler.config.prediction_type == "epsilon":
40             target = noise
41         elif noise_scheduler.config.prediction_type == "v_prediction":
42             target = noise_scheduler.get_velocity(latents, noise, timesteps)
43         else:
44             raise ValueError(f"Unknown prediction type
45                               {noise_scheduler.config.prediction_type}")
46
47     # Predict the noise residual and compute loss
48     model_pred = unet(noisy_latents, timesteps, encoder_hidden_states,
49                       return_dict=False)[0]
50
51     if args.snr_gamma is None:
52         loss = F.mse_loss(model_pred.float(), target.float(),
53                            reduction="mean")
54     else:
55         # Compute loss-weights as per Section 3.4 of
56         # https://arxiv.org/abs/2303.09556.
57         # Since we predict the noise instead of  $x_0$ , the original
58         # formulation is slightly changed.
59         # This is discussed in Section 4.2 of the same paper.
60         snr = compute_snr(noise_scheduler, timesteps)
61         mse_loss_weights = torch.stack([snr, args.snr_gamma *
62                                         torch.ones_like(timesteps)], dim=1).min(
63                                         dim=1
64                                         )[0]
65         if noise_scheduler.config.prediction_type == "epsilon":
66             mse_loss_weights = mse_loss_weights / snr
67         elif noise_scheduler.config.prediction_type == "v_prediction":
68             mse_loss_weights = mse_loss_weights / (snr + 1)
69
70         loss = F.mse_loss(model_pred.float(), target.float(),
71                           reduction="none")
72         loss = loss.mean(dim=list(range(1, len(loss.shape)))) *
73                 mse_loss_weights
74         loss = loss.mean()
75
76     # Gather the losses across all processes for logging (if we use
77     # distributed training).
```

```
68         avg_loss = accelerator.gather(loss.repeat(args.train_batch_size)).mean()
69         train_loss += avg_loss.item() / args.gradient_accumulation_steps
70
71     # Backpropagate
72     accelerator.backward(loss)
73     if accelerator.sync_gradients:
74         accelerator.clip_grad_norm_(unet.parameters(), args.max_grad_norm)
75     optimizer.step()
76     lr_scheduler.step()
77     optimizer.zero_grad()
78
79     # Checks if the accelerator has performed an optimization step behind the
80     # scenes
81     if accelerator.sync_gradients:
82         if args.use_ema:
83             if args.offload_ema:
84                 ema_unet.to(device="cuda", non_blocking=True)
85                 ema_unet.step(unet.parameters())
86             if args.offload_ema:
87                 ema_unet.to(device="cpu", non_blocking=True)
88         progress_bar.update(1)
89         global_step += 1
90         accelerator.log({"train_loss": train_loss}, step=global_step)
91         train_loss = 0.0
92
93         if global_step % args.checkpointing_steps == 0:
94             if accelerator.is_main_process:
95                 # before saving state, check if this save would set us over
96                 # the `checkpoints_total_limit`
97                 if args.checkpoints_total_limit is not None:
98                     checkpoints = os.listdir(args.output_dir)
99                     checkpoints = [d for d in checkpoints if
100                         d.startswith("checkpoint")]
101                     checkpoints = sorted(checkpoints, key=lambda x:
102                         int(x.split("-")[1]))
103
104                     # before we save the new checkpoint, we need to have at
105                     # most `checkpoints_total_limit - 1` checkpoints
106                     if len(checkpoints) >= args.checkpoints_total_limit:
107                         num_to_remove = len(checkpoints) -
108                             args.checkpoints_total_limit + 1
109                         removing_checkpoints = checkpoints[0:num_to_remove]
110
111                         logger.info(
112                             f"{len(checkpoints)} checkpoints already exist,
113                             removing {len(removing_checkpoints)} checkpoints"
114                         )
115                         logger.info(f"removing checkpoints: {",
116
```

```
109
110         '.join(removing_checkpoints)}")
111
112     for removing_checkpoint in removing_checkpoints:
113         removing_checkpoint = os.path.join(args.output_dir,
114                                         removing_checkpoint)
115         shutil.rmtree(removing_checkpoint)
116
117         save_path = os.path.join(args.output_dir,
118                                 f"checkpoint-{global_step}")
119         accelerator.save_state(save_path)
120         logger.info(f"Saved state to {save_path}")
121
122         logs = {"step_loss": loss.detach().item(), "lr":
123                  lr_scheduler.get_last_lr()[0]}
124         progress_bar.set_postfix(**logs)
125
126     if global_step >= args.max_train_steps:
127         break
```

对应的推理代码如下（需要关闭模型自带的 safety_checker，直接使用 diffusers 库提供的 StableDiffusionPipeline）。

```
1 import torch
2 from diffusers import StableDiffusionPipeline
3 import os
4
5 model_id = "newyorker-finetune"
6 folder_name = "img/newyorker"
7 device = "cuda:6"
8
9 pipe = StableDiffusionPipeline.from_pretrained(model_id, torch_dtype=torch.float16)
10 pipe.safety_checker = None
11 pipe.requires_safety_checker = False
12 pipe = pipe.to(device)
13
14
15 prompt = "a photo of an astronaut riding a horse on mars"
16 image = pipe(prompt).images[0]
17 image.save(os.path.join(folder_name, "astronaut_rides_horse.png"))
18
19 prompt = "a photo of a boy kissing with a girl"
20 image = pipe(prompt).images[0]
21 image.save(os.path.join(folder_name, "boy_girl.png"))
22
23 prompt = "a photo of a man shooting a gun"
24 image = pipe(prompt).images[0]
25 image.save(os.path.join(folder_name, "gun.png"))
```

```

27 prompt = "a photo of a woman with light shed on her face"
28 image = pipe(prompt).images[0]
29 image.save(os.path.join(folder_name, "light.png"))

```

3.3 微调训练过程

微调过程中的 loss 以及所需时间如图所示，环境为单卡 H100，在 Stable-Diffusion-V1-4 的基础上进行微调。

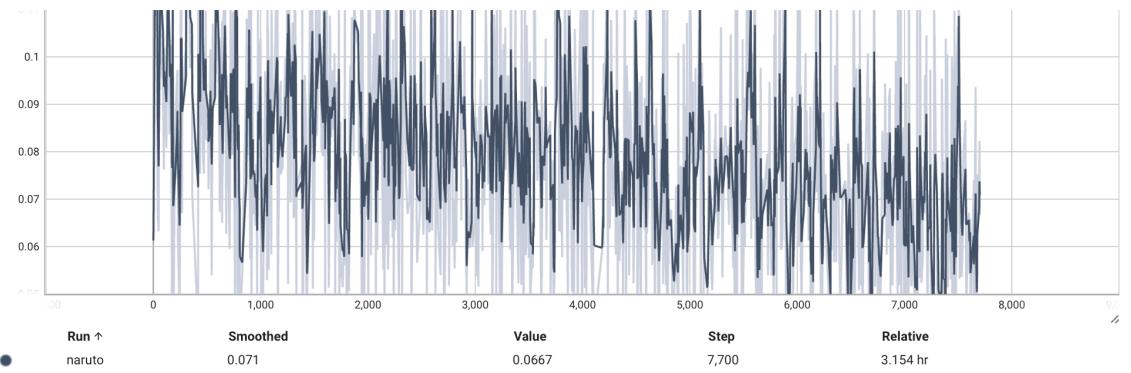


图 2: naturo-loss

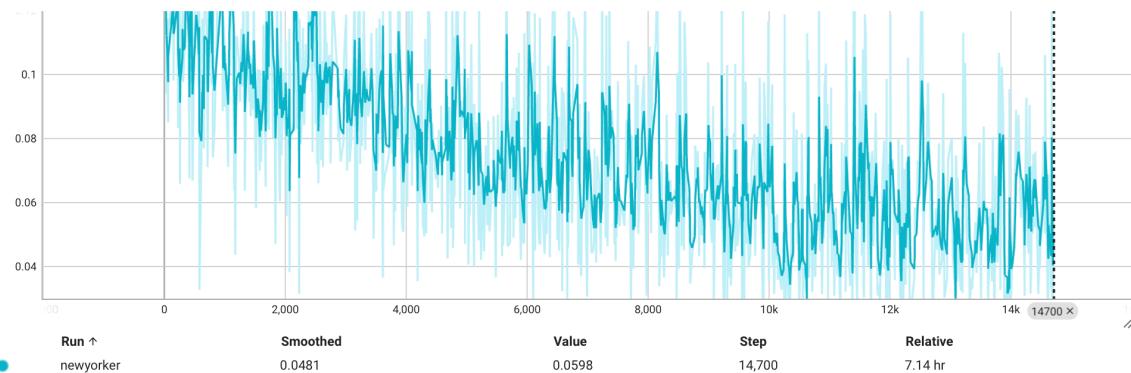


图 3: newyorker-loss

4 实验结果展示

最后我们对比原始模型与微调后模型的输出结果。我们设置了 4 个 prompt (对应图片从左到右)，对比原始模型与在 4 个数据集上微调的模型的输出效果。

- a photo of an astronaut riding a horse on mars
- a photo of a boy kissing with a girl
- a photo of a man shooting a gun

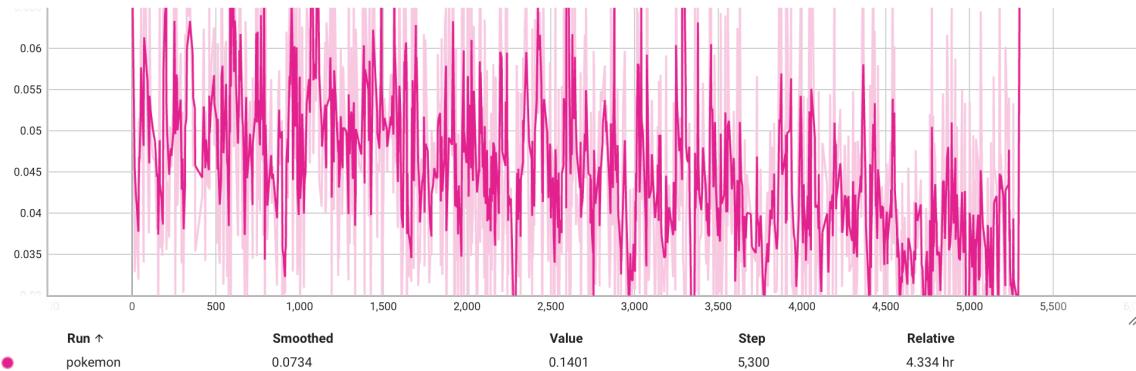


图 4: pokemon-loss

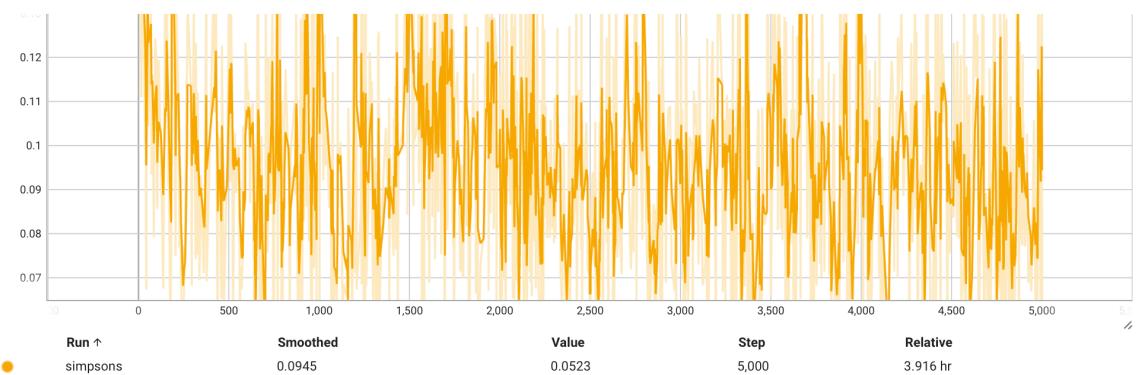


图 5: simpsons-loss

- a photo of a woman with light shed on her face



图 6: 原始模型输出图片



图 7: lambdalabs/naruto-blip-captions 数据集微调模型输出图片



图 8: jmhessel/newyorker_caption_contest 数据集微调模型输出图片



图 9: svjack/pokemon-blip-captions-en-zh 数据集微调模型输出图片



图 10: jinmel/simpsons-blip-captions-pil 数据集微调模型输出图片