diffusers 项目扩展点分析报告

袁晨圃

2025-10-31

University of Chinese Academy of Sciences

一、 项目介绍

1.1.1 Diffusion Model

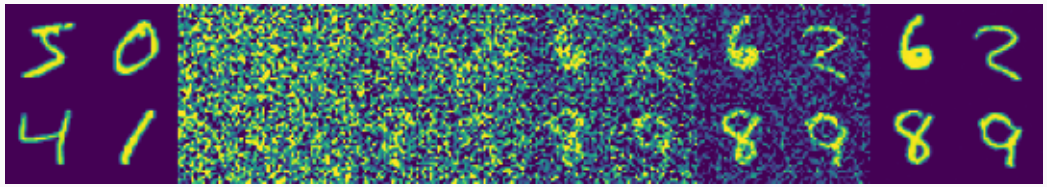


图 1 (Left: MNIST groundtruth. Right: MNIST sampling starting from random noise).

1.1.2 Diffusers

Diffusers 是 Hugging Face 推出的一个开源库， 用于构建、训练和使用扩散模型。 它让开发者能够轻松地进行图像、音频、视频等生成任务。

```
pipe = DiffusionPipeline.from_pretrained(
    "cerspense/zeroscope_v2_576w",
    torch_dtype=torch.float16
)
video_frames = pipe("a cat astronaut floating in space", num_frames=24).frames[0]
export_to_video(video_frames, "cat_astronaut.mp4")
```

- Models

  Models 是神经网络组件,用于去噪处理。所有模型都继承自 `ModelMixin` 和 `ConfigMixin`。

  - 自动编码器: 如 `VAE`,用于潜在空间编码和解码
  - UNet 架构: 如 `UNet2DConditionModel`,用于期望 2D 图像输入并受上下文条件约束的所有 UNet 变体
  - Transformer 模型: 如 `SD3Transformer2DModel`、`FluxTransformer2DModel`

- Schedulers

  Schedulers 控制噪声调度和采样过程。它们负责在推理时控制取噪时间步,以及在训练时定义噪声计划。

  所有调度器都继承自 `SchedulerMixin` 和 `ConfigMixin`。常见的调度器包括:

  - `DDPMScheduler`: DDPM 调度算法
  - `DDIMScheduler`: DDIM 调度算法
  - `EulerDiscreteScheduler`: Euler 离散方法
  - `FlowMatchEulerDiscreteScheduler`: 流匹配方法

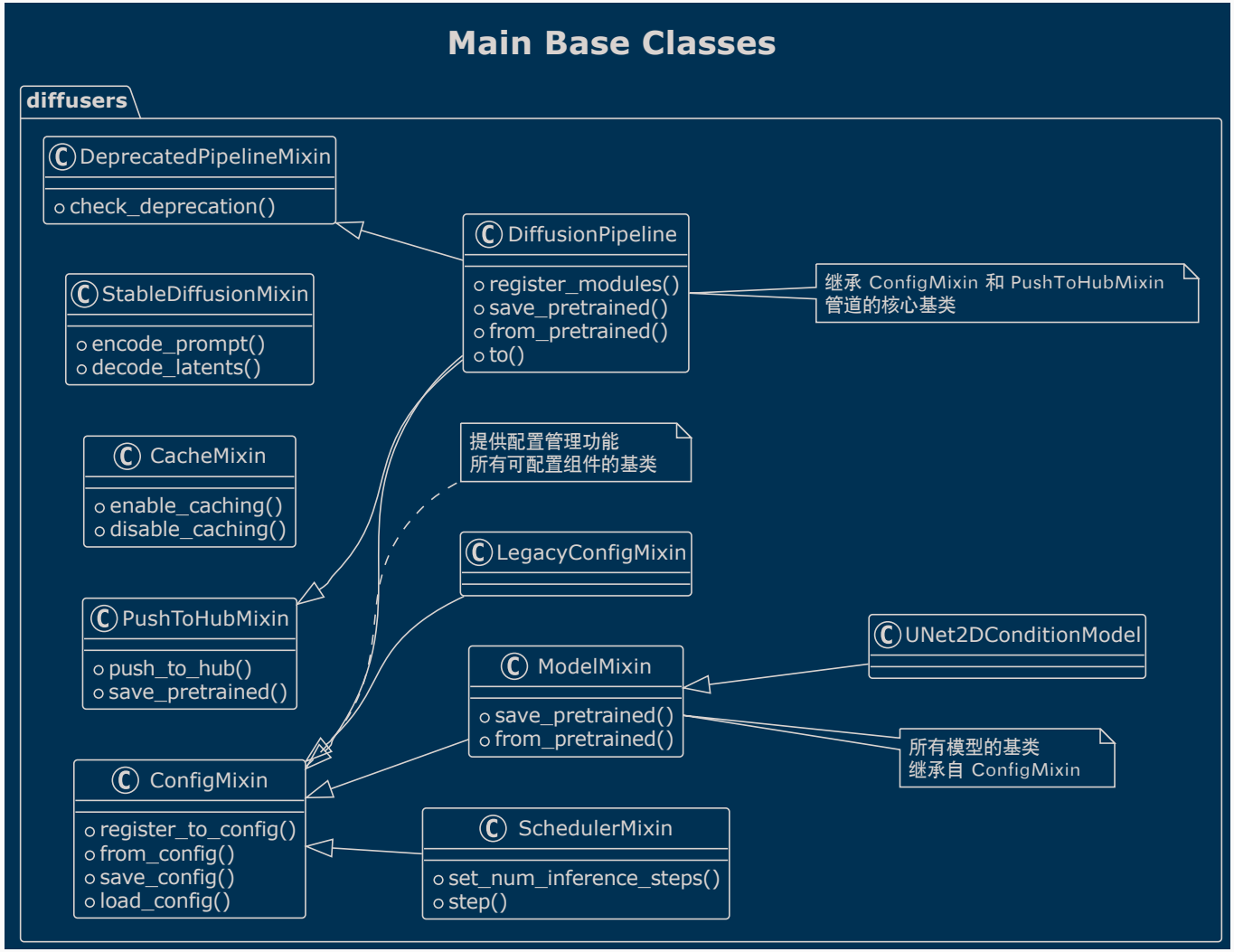
- Pipelines

  所有 pipeline 继承自 `DiffusionPipeline`

  一个 pipeline 的例子 (StableDiffusion):
  - `text_encoder, tokenizer`: 文本处理
  - `unet`: 去噪模型
  - `vae`: 图像编解码
  - `scheduler`: 噪声调度

  主要功能

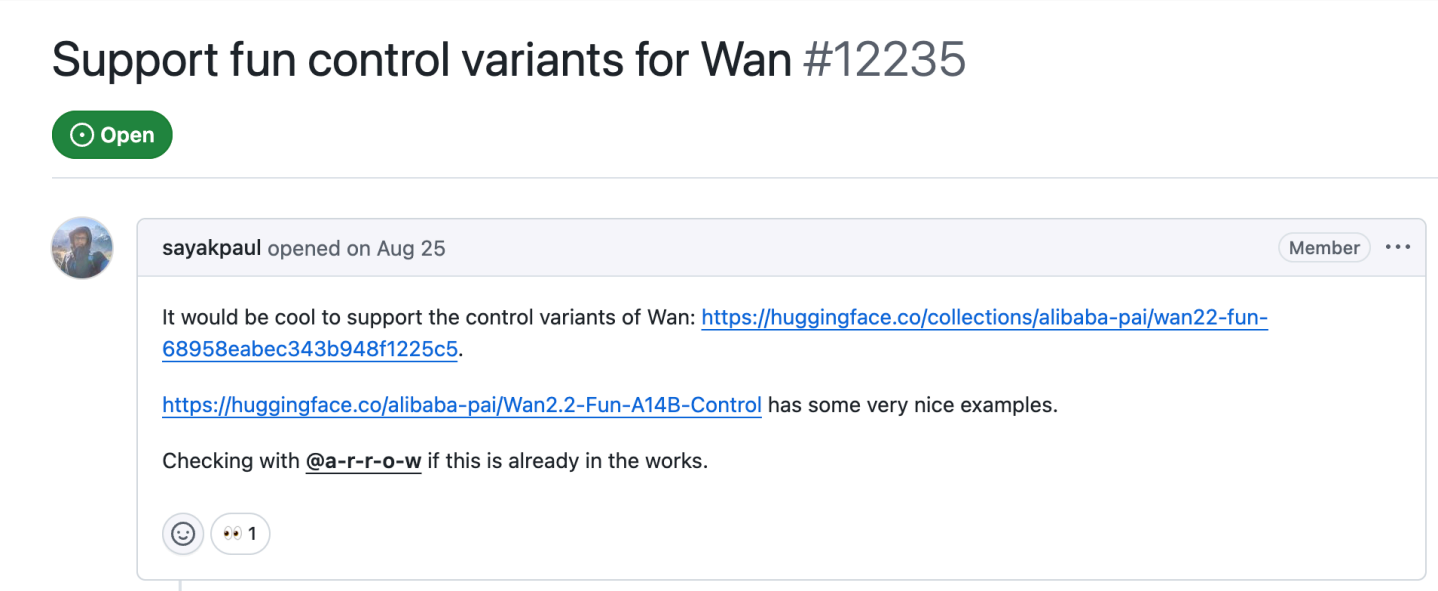
功能	方法
加载	`from_pretrained()`
保存	`save_pretrained()`
推理	`__call__()`
设备管理	`to()` / `enable_model_cpu_offload()`


Main Base Classes
diffusers
DeprecatedPipelineMixin
check_deprecation()
DiffusionPipeline
register_modules()
save_pretrained()
from_pretrained()
to()
继承 ConfigMixin 和 PushToHubMixin
管道的核心基类
StableDiffusionMixin
encode_prompt()
decode_latents()
CacheMixin
enable_caching()
disable_caching()
提供配置管理功能
所有可配置组件的基类
LegacyConfigMixin
PushToHubMixin
push_to_hub()
save_pretrained()
UNet2DConditionModel
ModelMixin
save_pretrained()
from_pretrained()
所有模型的基类
继承自 ConfigMixin
ConfigMixin
register_to_config()
from_config()
save_config()
load_config()
SchedulerMixin
set_num_inference_steps()
step()

二、 需求点分析

适配万相视频生成模型的一个变体 **Wan-Fun-Control**

Wan-Fun-Control 在 Wan 的基础上添加了控制模块，可以输入额外的 Control Image/Video



Support fun control variants for Wan #12235

Open

sayakpaul opened on Aug 25 Member

It would be cool to support the control variants of Wan: https://huggingface.co/collections/alibaba-pai/wan22-fun-68958eabec343b948f1225c5.

https://huggingface.co/alibaba-pai/Wan2.2-Fun-A14B-Control has some very nice examples.

Checking with @a-r-r-o-w if this is already in the works.

👀 1

2.1.1 需求点分析

2.1.1.1 用例名称

实现 WanFunControlPipeline（支持控制视频与相机外参注入的扩散 pipeline）

2.1.1.2 场景

Who: 用户 / 上层服务、`WanFunControlPipeline`（继承自 `DiffusionPipeline`, `WanLoraLoaderMixin`） Where: 推理/部署环境（内存/GPU/CPU），输入：prompt、control video/image、mask、相机外参；输出：图像序列或视频 When: 推理阶段（Inference Time），支持预处理 control latents

2.1.1.3 用例描述

1. 校验输入 (`check_inputs()`)：验证格式、帧数、分辨率、相机外参长度
   - 异常：ValueError(“Invalid input”)
2. 预处理 control video (`preprocess_control_video()`)：解码、resize、normalize、to tensor，可以参考仓库中已经有的 `CogVideoXFunControl` 实现
3. 编码 control latents (`prepare_control_latents()`)：调用 control encoder / vae.encode，支持 mask

- 异常：ValueError / torch.OutOfMemoryError

4. 将相机外参注入 `patchify()`：通过 `control_adapter` 映射并与 patch embedding 融合
   - 异常：RuntimeError("Shape Mismatch")
5. 去噪循环中注入控制信号：concat/add/cross-attn 等注入策略（可配置）
   - 若未定义策略 → fallback 并记录 warning
6. 解码并输出：`vae.decode()` → frames / video_bytes
7. 返回 meta 信息：control_strategy、steps、device

2.1.1.4 约束与接口

保持 `DiffusionPipeline` 接口兼容（`from_pretrained`, `__call__`, `save_pretrained`, `to`）

输入契约：`prompt: str`；`control_video: Tensor(B,T,C,H,W)` 或 path；`camera_params: list`；`mask: Tensor`

输出契约：`frames: list[Tensor]` 或 `video_bytes`；`meta: dict`

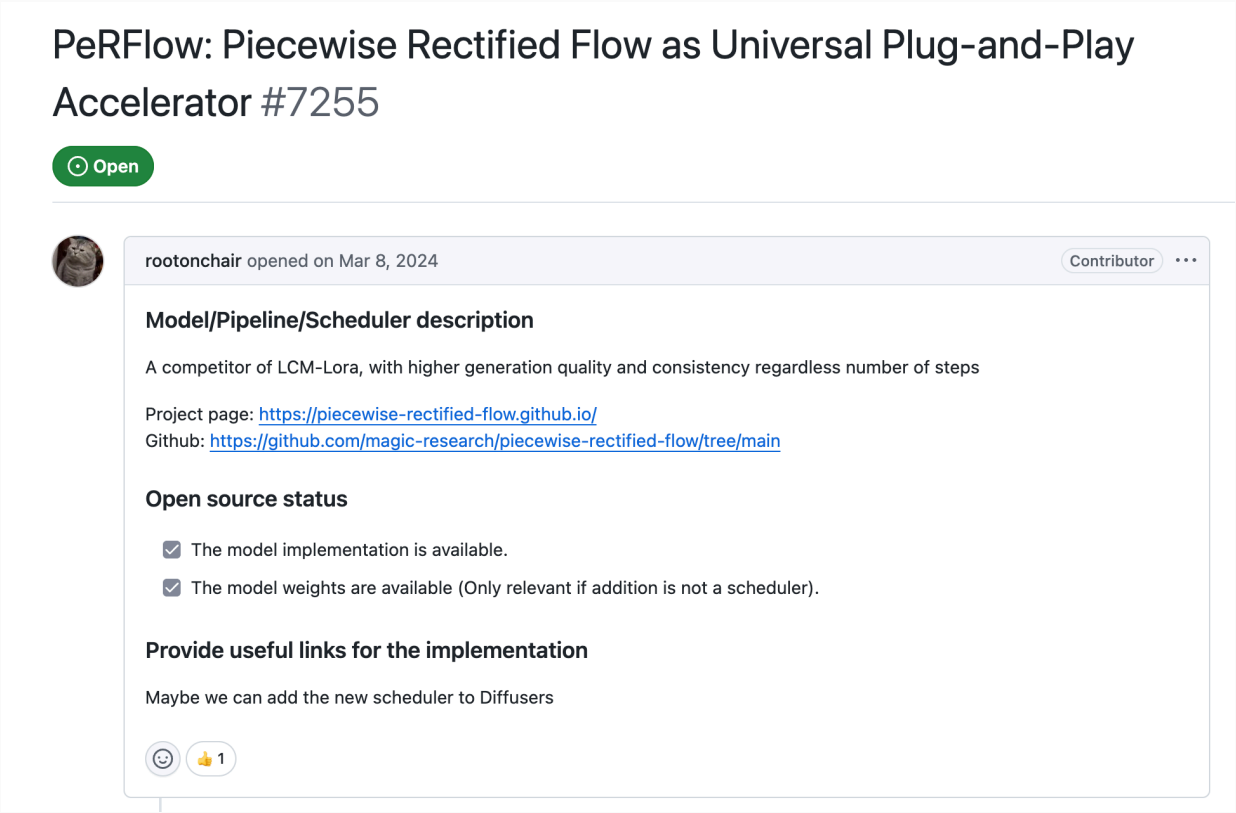
2.1.2 其他项目参考

`DiffSynth-Studio` 的做法：

没有显式的设置不同 pipeline
在模型中设置可选的扩展组件
推理过程中根据组件加载情况选择具体模式

```python
def patchify(self, x: torch.Tensor, control_camera_latents_input: torch.Tensor | None = None):
    x = self.patch_embedding(x)
    if self.control_adapter is not None and control_camera_latents_input is not None:
        y_camera = self.control_adapter(control_camera_latents_input)
        x = [u + v for u, v in zip(x, y_camera)]
    return x
```

分段线性流 PeRFlow



PeRFlow: Piecewise Rectified Flow as Universal Plug-and-Play Accelerator #7255

Open

rootonchair opened on Mar 8, 2024 Contributor

Model/Pipeline/Scheduler description

A competitor of LCM-Lora, with higher generation quality and consistency regardless number of steps

Project page: https://piecewise-rectified-flow.github.io/
Github: https://github.com/magic-research/piecewise-rectified-flow/tree/main

Open source status

- [x] The model implementation is available.
- [x] The model weights are available (Only relevant if addition is not a scheduler).

Provide useful links for the implementation

Maybe we can add the new scheduler to Diffusers

1

2.2.1 原理

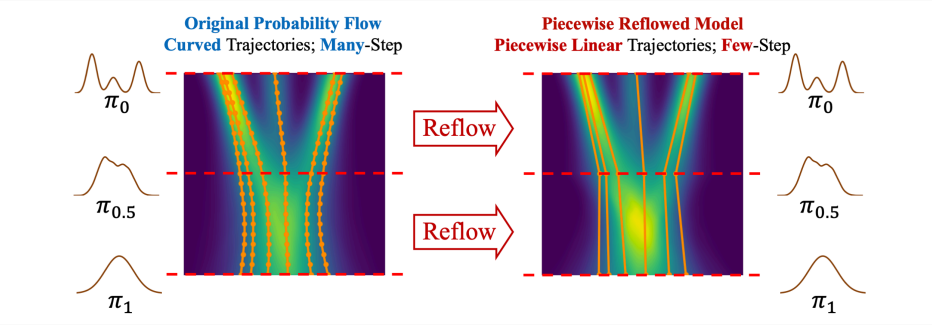
划分一些时间步区间，然后尝试在区间内把 Flow 拉直

$$\min_{\theta} \sum_{k=1}^{K} \mathbb{E}_{z_{t_k} \sim \pi_k} \left[\int_{t_{k-1}}^{t_k} \left\| \frac{z_{t_{k-1}} - z_{t_k}}{t_{k-1} - t_k} - v_\theta(z_t, t) \right\|^2 \mathrm{d}t \right]$$

这样能减少去噪步数

学习到的新参数 θ 相比原有参数的 ϕ 的权重变化 $\Delta W = \theta - \phi$

可以作为即插即用的加速器



2.2.2 需求点分析

2.2.2.1 用例名称

实现 `PerflowScheduler`（分段线性流加速调度器）

2.2.2.2 场景

采样/推理阶段，文件位置 `schedulers/scheduling_perflow.py`，并在 `schedulers/__init__.py` 注册

2.2.2.3 关键步骤

1. `__init__`：读取配置（分段 K、细分、theta 初值等）
2. `set_timesteps()`：构造 timesteps 列表，支持不同采样长度
3. `step()`：基于 Perflow 近似修正预测，返回 `next_latents`；遇数值不稳回退 baseline
4. `state_dict()` / `load_state_dict()`：保存/恢复 learned theta，支持 strict=False 的部分加载
5. 注册至调度器列表，支持 `from_pretrained(..., subfolder="scheduler")`

6. 实现 Loader
   - 支持 `from_config()` 与 `from_pretrained(..., subfolder="scheduler")`
   - `from_pretrained` 行为：读取 `scheduler/config.json` → `from_config` 创建实例 → 若存在 `scheduler` 权重文件则 `load_state_dict`
   - `state_dict` 至少包含 `"perflow/theta"` 和 `"config"`，加载失败时可部分加载并记录 warning

2.2.2.4 约束与接口

必须兼容 `SchedulerMixin` 签名，输入 `(latents, timestep, model_output)` → 输出 `next_latents`，包含数值稳定保护

2.2.2.5 验收标准

`set_timesteps()/step()` 边界测试通过；`state_dict` roundtrip 测试通过；`from_pretrained` 能加载有/无 learned theta 的仓库；与 baseline 做质量/速度对比并达成目标

2.2.2.6 结构

```
src/diffusers/
├── schedulers/
│   ├── __init__.py                    # Register scheduler here
│   └── scheduling_perflow.py          # New scheduler implementation
├── loaders/
│   ├── __init__.py                    # Register loader here
│   └── perflow_loader.py              # New loader implementation (if needed)
└── __init__.py                        # Main package exports

tests/
└── schedulers/
    └── test_scheduler_perflow.py      # Scheduler tests
```

三、 其他内容

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

The Zen of Python

3.1.1 可用性优先于性能（Usability over Performance）

- 模型默认以最高精度（float32）在 CPU 上加载，确保跨平台可用性。
- 保持库的轻量化：强制依赖极少，可选依赖灵活（`accelerate`、`onnx` 等）。
- 追求清晰、可解释的代码，而非晦涩“魔法”写法。
- 优先让库“能用”，再考虑“更快”。

3.1.2 简单优于容易（Simple over Easy）

- 遵循 PyTorch 原则：显式 > 隐式，简单 > 复杂。
- 明确错误优于自动修正——帮助用户理解模型行为。
- 模型与调度器分离，暴露核心逻辑，提升调试与定制能力。
- 管道组件（文本编码器、UNet、VAE）独立实现，便于扩展与训练（如 DreamBooth、Textual Inversion）。

3.1.3 易调试、易贡献优于过度抽象（Tweakable over Abstraction）

- 采用 单文件策略（single-file policy）：一个类/算法尽量在一个独立文件中实现。
- 宁可复制代码，也不仓促抽象。
- 原因：
  - 机器学习领域变化快，抽象层易过时；
  - 研究者更喜欢可直接修改的自包含代码；
  - 降低社区贡献门槛，避免复杂依赖。
- 这一策略源自 Transformers 库的成功经验。

Thanks!