

DiffusionPipeline 类方法详细说明

目录

- 类属性
- 核心方法
- 设备和内存管理
- 注意力优化
- VAE优化
- 工具方法
- 高级功能

类属性

基础属性

属性名	类型	默认值	原始描述	中文说明
config_name	str	"model_index.json"	The configuration filename that stores the class and module names of all the diffusion pipeline's components.	存储扩散 Pipeline 所有 组件的类名 和模块名的 配置文件名
model_cpu_offload_seq	Optional[str]	None	-	模型CPU卸载序列，定义模型卸载顺序
hf_device_map	Optional[Dict]	None	-	HuggingFace 设备映射配置

属性名	类型	默认值	原始描述	中文说明
_optional_components	List[str]	[]	List of all optional components that don't have to be passed to the pipeline to function (should be overridden by subclasses).	所有可选组件的列表，这些组件不必传递给 Pipeline 即可运行（应由子类重写）
_exclude_from_cpu_offload	List[str]	[]	-	从CPU卸载中排除的组件列表
_load_connected_pipes	bool	False	-	是否加载连接的 Pipeline
_is_onnx	bool	False	-	是否为ONNX Pipeline

核心方法

1. `__init__` 方法

```
def __init__(self, *args, **kwargs)
```

描述: Pipeline的初始化方法，由子类实现具体的组件初始化。

参数:

- `*args`: 位置参数，通常包含Pipeline的各个组件
- `**kwargs`: 关键字参数，用于传递额外配置

2. `register_modules` 方法

```
def register_modules(self, **kwargs)
```

描述: 注册Pipeline的模块组件，将组件信息保存到配置中。

参数:

- `**kwargs`: 要注册的模块，键为模块名，值为模块对象

功能:

- 自动检测模块的库和类名
- 将模块信息注册到配置中
- 设置模块为Pipeline的属性

示例:

```
pipeline.register_modules(  
    unet=unet_model,  
    vae=vae_model,  
    scheduler=scheduler  
)
```

3. `save_pretrained` 方法

```
def save_pretrained(  
    self,  
    save_directory: Union[str, os.PathLike],  
    safe_serialization: bool = True,  
    variant: Optional[str] = None,  
    max_shard_size: Optional[Union[int, str]] = None,  
    push_to_hub: bool = False,  
    **kwargs  
)
```

原始文档: Save all saveable variables of the pipeline to a directory. A pipeline variable can be saved and loaded if its class implements both a save and loading method. The pipeline is easily reloaded using the [~DiffusionPipeline.from_pretrained] class method.

中文描述: 将Pipeline的所有可保存变量保存到目录中。如果Pipeline变量的类实现了保存和加载方法，则可以保存和加载该变量。可以使用[~DiffusionPipeline.from_pretrained]类方法轻松重新加载Pipeline。

参数详解:

- `save_directory (str or os.PathLike)`: Directory to save a pipeline to. Will be created if it doesn't exist. | 保存Pipeline的目录，如果不存在将被创建
- `safe_serialization (bool, optional, defaults to True)`: Whether to save the model using `safetensors` or the traditional PyTorch way with `pickle`. | 是否使用 `safetensors` 保存模型，或使用传统的PyTorch方式与 `pickle`
- `variant (str, optional)`: If specified, weights are saved in the format `pytorch_model.<variant>.bin`. | 如果指定，权重将以 `pytorch_model.<variant>.bin` 格式保存
- `max_shard_size (int or str, defaults to None)`: The maximum size for a checkpoint before being sharded. | 分片前检查点的最大大小
- `push_to_hub (bool, optional, defaults to False)`: Whether or not to push your model to the Hugging Face model hub after saving it. | 保存后是否将模型推送到Hugging Face模型中心

- `**kwargs (Dict[str, Any], optional)`: Additional keyword arguments passed along to the `[~utils.PushToHubMixin.push_to_hub]` method. | 传递给`[~utils.PushToHubMixin.push_to_hub]`方法的额外关键字参数

示例:

```
# 基础保存
pipeline.save_pretrained("./my_pipeline")

# 保存fp16变体并推送到Hub
pipeline.save_pretrained(
    "./my_pipeline",
    variant="fp16",
    push_to_hub=True,
    repo_id="my_username/my_pipeline"
)
```

4. `from_pretrained` 类方法

```
@classmethod
def from_pretrained(
    cls,
    pretrained_model_name_or_path: Optional[Union[str, os.PathLike]],
    **kwargs
) -> Self
```

描述: 从预训练权重实例化PyTorch扩散Pipeline。

核心参数:

- `pretrained_model_name_or_path (str | os.PathLike)`:
 - Hub仓库ID (如 "CompVis/ldm-text2im-large-256")
 - 本地目录路径 (如 "./my_pipeline_directory/")
 - DDUF文件路径

数据类型参数:

- `torch_dtype (torch.dtype | dict)`:
 - 单一类型: `torch.float16`
 - 组件特定: `{'transformer': torch.bfloat16, 'vae': torch.float16}`
 - 带默认值: `{'transformer': torch.bfloat16, 'default': torch.float16}`

自定义Pipeline参数:

- `custom_pipeline (str)`:
 - Hub仓库ID: "hf-internal-testing/diffusers-dummy-pipeline"
 - 社区Pipeline名: "clip_guided_stable_diffusion"

- 本地目录: `"./my_pipeline_directory/"`

下载控制参数:

- `force_download (bool, 默认 False)`: 强制重新下载
- `cache_dir (str)`: 缓存目录路径
- `local_files_only (bool, 默认 False)`: 仅使用本地文件
- `token (str | bool)`: HuggingFace访问令牌
- `revision (str, 默认 "main")`: 模型版本分支/标签/提交ID
- `custom_revision (str)`: 自定义Pipeline的版本

网络参数:

- `proxies (Dict[str, str])`: 代理服务器配置
- `mirror (str)`: 镜像源地址 (中国用户)

设备映射参数:

- `device_map (str)`: 设备映射策略 · 目前支持"balanced"
- `max_memory (Dict)`: 每个设备的最大内存限制
- `offload_folder (str)`: 磁盘卸载目录
- `offload_state_dict (bool)`: 是否临时卸载CPU状态字典

内存优化参数:

- `low_cpu_mem_usage (bool)`: 低CPU内存使用模式
- `use_safetensors (bool)`: 是否使用safetensors格式
- `use_onnx (bool)`: 是否使用ONNX权重
- `variant (str)`: 权重变体 · 如"fp16"
- `dduf_file (str)`: DDUF文件路径

其他参数:

- `output_loading_info (bool, 默认 False)`: 返回加载信息
- `quantization_config`: 量化配置
- `**kwargs`: 覆盖Pipeline组件的参数

示例:

```
# 基础加载
pipeline = DiffusionPipeline.from_pretrained("stable-diffusion-v1-5/stable-diffusion-v1-5")

# 指定数据类型和设备
pipeline = DiffusionPipeline.from_pretrained(
    "stabilityai/stable-diffusion-xl-base-1.0",
    torch_dtype=torch.float16,
    device_map="balanced",
    use_safetensors=True
)

# 组件特定数据类型
```

```
pipeline = DiffusionPipeline.from_pretrained(
    "black-forest-labs/FLUX.1-schnell",
    torch_dtype={
        'transformer': torch.bfloat16,
        'vae': torch.float16,
        'default': torch.float32
    }
)

# 使用自定义调度器
from diffusers import LMSDiscreteScheduler
pipeline = DiffusionPipeline.from_pretrained(
    "stable-diffusion-v1-5/stable-diffusion-v1-5",
    scheduler=LMSDiscreteScheduler.from_config(scheduler_config)
)
```

5. `to` 方法

```
def to(self, *args, **kwargs) -> Self
```

描述: 执行Pipeline的数据类型和/或设备转换。

调用方式:

- `to(dtype)`: 转换到指定数据类型
- `to(device)`: 转换到指定设备
- `to(device, dtype)`: 同时转换设备和数据类型
- `to(device=None, dtype=None, silence_dtype_warnings=False)`: 关键字参数方式

参数:

- `dtype (torch.dtype, 可选)`: 目标数据类型
- `device (torch.device, 可选)`: 目标设备
- `silence_dtype_warnings (bool, 默认 False)`: 是否静默数据类型警告

返回值: 转换后的Pipeline实例 (如果已经是目标类型则返回自身)

示例:

```
# 移动到GPU
pipeline = pipeline.to("cuda")

# 转换数据类型
pipeline = pipeline.to(torch.float16)

# 同时转换设备和数据类型
pipeline = pipeline.to("cuda", torch.float16)
```

```
# 静默警告
pipeline = pipeline.to("cuda", silence_dtype_warnings=True)
```

6. download 类方法

```
@classmethod
def download(cls, pretrained_model_name, **kwargs) -> Union[str, os.PathLike]
```

描述: 下载并缓存PyTorch扩散Pipeline，但不实例化。

参数:

- `pretrained_model_name (str)`: Hub仓库ID
- `custom_pipeline (str, 可选)`: 自定义Pipeline
- `force_download (bool, 默认 False)`: 强制重新下载
- `proxies (Dict[str, str], 可选)`: 代理配置
- `output_loading_info (bool, 默认 False)`: 返回加载信息
- `local_files_only (bool, 默认 False)`: 仅使用本地文件
- `token (str | bool, 可选)`: 访问令牌
- `revision (str, 默认 "main")`: 版本

返回值: 下载的模型路径

示例:

```
# 下载模型到缓存
model_path = DiffusionPipeline.download("stable-diffusion-v1-5/stable-diffusion-
v1-5")
print(f"模型下载到: {model_path}")
```

属性访问器

7. device 属性

```
@property
def device(self) -> torch.device
```

描述: 返回Pipeline所在的torch设备。

返回值: Pipeline当前所在的设备

示例:

```
print(f"Pipeline在设备: {pipeline.device}")
# 输出: Pipeline在设备: cuda:0
```

8. `dtype` 属性

```
@property
def dtype(self) -> torch.dtype
```

描述: 返回Pipeline的torch数据类型。

返回值: Pipeline的数据类型

示例:

```
print(f"Pipeline数据类型: {pipeline.dtype}")
# 输出: Pipeline数据类型: torch.float16
```

9. `components` 属性

```
@property
def components(self) -> Dict[str, Any]
```

描述: 返回Pipeline的所有组件字典，用于在不同Pipeline间共享权重和配置。

返回值: 包含所有Pipeline组件的字典

示例:

```
components = pipeline.components
print("Pipeline组件:", list(components.keys()))
# 输出: Pipeline组件: ['vae', 'text_encoder', 'tokenizer', 'unet', 'scheduler']

# 使用组件创建新Pipeline
new_pipeline = AnotherPipeline(**components)
```

10. `name_or_path` 属性

```
@property
def name_or_path(self) -> str
```

描述: 返回Pipeline的名称或路径。

返回值: Pipeline的原始名称或路径

设备和内存管理

11. enable_model_cpu_offload 方法

```
def enable_model_cpu_offload(self, gpu_id: Optional[int] = None, device: Union[torch.device, str] = None)
```

描述: 使用accelerate将所有模型卸载到CPU，以较低的性能影响减少内存使用。与enable_sequential_cpu_offload相比，此方法在调用forward方法时将整个模型移动到加速器，模型保持在加速器上直到下一个模型运行。

参数:

- `gpu_id (int, 可选):` 推理时使用的加速器ID，默认为0
- `device (torch.device | str, 可选):` 推理时使用的PyTorch设备类型，自动检测可用加速器

特点:

- 内存节省低于enable_sequential_cpu_offload
- 性能影响较小，因为UNet的迭代执行
- 适合需要平衡内存和性能的场景

示例:

```
# 使用默认GPU
pipeline.enable_model_cpu_offload()

# 指定GPU ID
pipeline.enable_model_cpu_offload(gpu_id=1)

# 指定设备
pipeline.enable_model_cpu_offload(device="cuda:1")
```

12. enable_sequential_cpu_offload 方法

```
def enable_sequential_cpu_offload(self, gpu_id: Optional[int] = None, device: Union[torch.device, str] = None)
```

描述: 使用**⚡ Accelerate**将所有模型卸载到CPU，显著减少内存使用。所有**torch.nn.Module**组件的状态字典保存到CPU，然后移动到**torch.device('meta')**，仅在特定子模块调用forward方法时加载到加速器。

参数:

- `gpu_id` (`int`, 可选): 推理时使用的加速器ID，默认为0
- `device` (`torch.device` | `str`, 可选): 推理时使用的PyTorch设备类型

特点:

- 内存节省高于`enable_model_cpu_offload`
- 性能影响较大，因为子模块级别的卸载
- 适合内存极度受限的场景

示例:

```
# 启用顺序CPU卸载
pipeline.enable_sequential_cpu_offload()

# 指定设备
pipeline.enable_sequential_cpu_offload(device="cuda:0")
```

13. `remove_all_hooks` 方法

```
def remove_all_hooks(self)
```

描述: 移除使用`enable_sequential_cpu_offload`或`enable_model_cpu_offload`时添加的所有钩子。

功能:

- 清理所有accelerate钩子
- 恢复模型到正常状态
- 释放钩子占用的资源

示例:

```
# 移除所有卸载钩子
pipeline.remove_all_hooks()
```

14. `maybe_free_model_hooks` 方法

```
def maybe_free_model_hooks(self)
```

描述: 执行以下操作的方法：

- 卸载所有组件

- 移除使用`enable_model_cpu_offload`时添加的所有模型钩子，然后重新应用
- 重置去噪器组件的有状态扩散钩子

用途: 在Pipeline的`__call__`函数末尾添加此函数，确保在应用`enable_model_cpu_offload`时正常工作。

示例:

```
# 通常在Pipeline内部调用
def __call__(self, *args, **kwargs):
    # ... Pipeline逻辑 ...
    self.maybe_free_model_hooks()
    return result
```

15. `reset_device_map` 方法

```
def reset_device_map(self)
```

描述: 将设备映射（如果有）重置为None。

功能:

- 移除所有钩子
- 将所有组件移动到CPU
- 清空设备映射配置

示例:

```
# 重置设备映射
pipeline.reset_device_map()
```

注意力优化

16. `enable_xformers_memory_efficient_attention` 方法

```
def enable_xformers_memory_efficient_attention(self, attention_op:
Optional[Callable] = None)
```

描述: 启用来自xFormers的内存高效注意力。启用此选项时，应该观察到更低的GPU内存使用和推理期间的潜在加速。

参数:

- `attention_op` (`Callable`, 可选): 覆盖默认的`None`操作符，用作xFormers的`memory_efficient_attention()`函数的`op`参数

注意事项:

- ⚠️ 当内存高效注意力和切片注意力都启用时，内存高效注意力优先
- 训练期间的加速不保证

示例:

```
# 启用xFormers内存高效注意力
pipeline.enable_xformers_memory_efficient_attention()

# 使用特定操作符
from xformers.ops import MemoryEfficientAttentionFlashAttentionOp
pipeline.enable_xformers_memory_efficient_attention(
    attention_op=MemoryEfficientAttentionFlashAttentionOp
)

# VAE的Flash Attention变通方案
pipeline.vae.enable_xformers_memory_efficient_attention(attention_op=None)
```

17. `disable_xformers_memory_efficient_attention` 方法

```
def disable_xformers_memory_efficient_attention(self)
```

描述: 禁用来自xFormers的内存高效注意力。

示例:

```
# 禁用xFormers内存高效注意力
pipeline.disable_xformers_memory_efficient_attention()
```

18. `enable_attention_slicing` 方法

```
def enable_attention_slicing(self, slice_size: Optional[Union[str, int]] = "auto")
```

描述: 启用切片注意力计算。启用此选项时，注意力模块将输入张量分割成切片，分几个步骤计算注意力。对于多个注意力头，计算在每个头上顺序执行。

参数:

- `slice_size (str | int, 默认 "auto"):`
 - `"auto"`: 将注意力头的输入减半，注意力将分两步计算
 - `"max"`: 通过一次只运行一个切片来节省最大内存

- 数字: 使用`attention_head_dim // slice_size`个切片 · `attention_head_dim`必须是`slice_size`的倍数

注意事项:

- ⚠ 如果已经使用PyTorch 2.0的`scaled_dot_product_attention`(SDPA)或xFormers · 不要启用注意力切片
- 这些注意力计算已经非常内存高效 · 启用切片可能导致严重减速

示例:

```
# 自动切片
pipeline.enable_attention_slicing()

# 最大内存节省
pipeline.enable_attention_slicing("max")

# 自定义切片大小
pipeline.enable_attention_slicing(4)
```

19. `disable_attention_slicing` 方法

```
def disable_attention_slicing(self)
```

描述: 禁用切片注意力计算。如果之前调用了`enable_attention_slicing` · 注意力将在一步中计算。

示例:

```
# 禁用注意力切片
pipeline.disable_attention_slicing()
```

20. `set_attention_slice` 方法

```
def set_attention_slice(self, slice_size: Optional[int])
```

描述: 设置注意力切片大小的内部方法。

参数:

- `slice_size` (`int`, 可选): 切片大小 · `None`表示禁用切片

功能: 遍历所有支持注意力切片的模块并设置切片大小

VAE优化

21. `enable_vae_slicing` 方法

```
def enable_vae_slicing(self)
```

描述: 启用切片VAE解码。启用此选项时，VAE将输入张量分割成切片，分几个步骤计算解码。这对节省内存和允许更大的批次大小很有用。

特点:

- 内存节省：中等
- 性能影响：很小
- 适用场景：需要处理大批次或高分辨率图像

示例:

```
# 启用VAE切片
pipeline.enable_vae_slicing()
```

22. `disable_vae_slicing` 方法

```
def disable_vae_slicing(self)
```

描述: 禁用切片VAE解码。如果之前启用了`enable_vae_slicing`，此方法将回到一步计算解码。

示例:

```
# 禁用VAE切片
pipeline.disable_vae_slicing()
```

23. `enable_vae_tiling` 方法

```
def enable_vae_tiling(self)
```

描述: 启用平铺VAE解码。启用此选项时，VAE将输入张量分割成瓦片，分几个步骤计算解码和编码。这对节省大量内存和允许处理更大图像非常有用。

特点:

- 内存节省：高
- 性能影响：轻微
- 适用场景：处理超高分辨率图像或内存严重受限

示例：

```
# 启用VAE平铺  
pipeline.enable_vae_tiling()
```

24. disable_vae_tiling 方法

```
def disable_vae_tiling(self)
```

描述: 禁用平铺VAE解码。如果之前启用了enable_vae_tiling，此方法将回到一步计算解码。

示例：

```
# 禁用VAE平铺  
pipeline.disable_vae_tiling()
```

工具方法

25. progress_bar 方法

```
def progress_bar(self, iterable=None, total=None)
```

描述: 创建进度条用于显示推理进度。

参数：

- **iterable** (可选): 可迭代对象，用于包装进度条
- **total** (可选): 总步数，用于创建手动更新的进度条

返回值: tqdm进度条对象

注意: iterable和total必须提供其中一个

示例：

```
# 包装可迭代对象  
for step in pipeline.progress_bar(range(50)):  
    # 执行推理步骤  
    pass
```

```
# 手动进度条
pbar = pipeline.progress_bar(total=50)
for i in range(50):
    # 执行操作
    pbar.update(1)
```

26. set_progress_bar_config 方法

```
def set_progress_bar_config(self, **kwargs)
```

描述: 设置进度条配置参数。

参数:

- ****kwargs**: tqdm进度条的配置参数

常用配置:

- **disable (bool)**: 是否禁用进度条
- **desc (str)**: 进度条描述
- **leave (bool)**: 完成后是否保留进度条
- **position (int)**: 进度条位置

示例:

```
# 禁用进度条
pipeline.set_progress_bar_config(disable=True)

# 自定义进度条
pipeline.set_progress_bar_config(
    desc="生成图像",
    leave=False,
    position=0
)
```

27. numpy_to_pil 静态方法

```
@staticmethod
def numpy_to_pil(images)
```

描述: 将NumPy图像或图像批次转换为PIL图像。

参数:

- `images`: NumPy数组格式的图像

返回值: PIL图像列表

示例:

```
import numpy as np

# 转换NumPy图像为PIL
numpy_images = np.random.rand(2, 512, 512, 3)
pil_images = DiffusionPipeline.numpy_to_pil(numpy_images)
```

28. `from_pipe` 类方法

```
@classmethod
def from_pipe(cls, pipeline, **kwargs)
```

描述: 从给定Pipeline创建新Pipeline。此方法对于从现有Pipeline组件创建新Pipeline而不重新分配额外内存很有用。

参数:

- `pipeline` (`DiffusionPipeline`): 源Pipeline
- `**kwargs`: 覆盖组件或配置的参数

返回值: 具有相同权重和配置的新Pipeline

特点:

- 重用现有组件，节省内存
- 可以覆盖特定组件
- 保持原始配置

示例:

```
from diffusers import StableDiffusionPipeline, StableDiffusionSAGPipeline

# 创建基础Pipeline
base_pipe = StableDiffusionPipeline.from_pretrained("stable-diffusion-v1-5/stable-
diffusion-v1-5")

# 从基础Pipeline创建SAG Pipeline
sag_pipe = StableDiffusionSAGPipeline.from_pipe(base_pipe)

# 覆盖特定组件
new_pipe = StableDiffusionPipeline.from_pipe(
    base_pipe,
    scheduler=new_scheduler,
```

```
    torch_dtype=torch.float16  
)
```

高级功能

29. enable_freeeu 方法

```
def enable_freeeu(self, s1: float, s2: float, b1: float, b2: float)
```

描述: 启用FreeU机制，如论文<https://huggingface.co/papers/2309.11497>所述。缩放因子后的后缀表示应用的阶段。

参数:

- **s1 (float):** 阶段1的缩放因子，用于减弱跳跃特征的贡献，缓解增强去噪过程中的“过度平滑效应”
- **s2 (float):** 阶段2的缩放因子，用于减弱跳跃特征的贡献
- **b1 (float):** 阶段1的缩放因子，用于放大骨干特征的贡献
- **b2 (float):** 阶段2的缩放因子，用于放大骨干特征的贡献

适用模型: 需要Pipeline具有unet组件

推荐值: 请参考官方仓库获取适用于不同Pipeline的已知有效值组合

示例:

```
# Stable Diffusion v1.5推荐值  
pipeline.enable_freeeu(s1=0.9, s2=0.2, b1=1.2, b2=1.4)  
  
# Stable Diffusion XL推荐值  
pipeline.enable_freeeu(s1=0.6, s2=0.4, b1=1.1, b2=1.2)
```

30. disable_freeeu 方法

```
def disable_freeeu(self)
```

描述: 如果启用了FreeU机制，则禁用它。

示例:

```
# 禁用FreeU  
pipeline.disable_freeeu()
```

31. `fuse_qkv_projections` 方法

```
def fuse_qkv_projections(self, unet: bool = True, vae: bool = True)
```

描述: 启用融合QKV投影。对于自注意力模块，所有投影矩阵（查询、键、值）都被融合。对于交叉注意力模块，键和值投影矩阵被融合。

参数:

- `unet` (bool, 默认 `True`): 是否在UNet上应用融合
- `vae` (bool, 默认 `True`): 是否在VAE上应用融合

注意: 🚧 这是实验性API

限制: VAE融合仅支持AutoencoderKL类型

示例:

```
# 融合UNet和VAE的QKV投影
pipeline.fuse_qkv_projections()

# 仅融合UNet
pipeline.fuse_qkv_projections(unet=True, vae=False)
```

32. `unfuse_qkv_projections` 方法

```
def unfuse_qkv_projections(self, unet: bool = True, vae: bool = True)
```

描述: 如果启用了QKV投影融合，则禁用它。

参数:

- `unet` (bool, 默认 `True`): 是否在UNet上取消融合
- `vae` (bool, 默认 `True`): 是否在VAE上取消融合

注意: 🚧 这是实验性API

示例:

```
# 取消融合QKV投影
pipeline.unfuse_qkv_projections()
```

内存优化策略对比

方法	内存节省	性能影响	适用场景
enable_model_cpu_offload()	中等	轻微	平衡内存和性能
enable_sequential_cpu_offload()	高	中等	内存极度受限
enable_attention_slicing()	中等	轻微	大批次推理
enable_vae_slicing()	低	很轻微	VAE内存优化
enable_vae_tiling()	高	轻微	超高分辨率图像
enable_xformers_memory_efficient_attention()	中等	负影响(加速)	有xFormers环境

最佳实践

内存受限环境

```
# 最大内存节省配置
pipeline.enable_sequential_cpu_offload()
pipeline.enable_attention_slicing("max")
pipeline.enable_vae_slicing()
pipeline.enable_vae_tiling()
```

性能优先环境

```
# 性能优化配置
pipeline.to("cuda", torch.float16)
pipeline.enable_xformers_memory_efficient_attention()
pipeline.enable_model_cpu_offload() # 轻微内存节省
```

高分辨率图像生成

```
# 高分辨率优化
pipeline.enable_vae_tiling()
pipeline.enable_attention_slicing("auto")
pipeline.enable_model_cpu_offload()
```