Diffusion Model相关代码学习

Diffusers[Huggingface]

代码在：https://github.com/huggingface/diffusers，也就是后文一部分介绍所使用的代码库，但他很多算法收录不全，而且bug贼多。

SCORE-BASED GENERATIVE MODELING THROUGHSTOCHASTIC DIFFERENTIAL EQUATIONS[ICLR2021]

PC算法（DEMO介绍）：

代码在: score_sde_pytorch 里，但是可以直接查看diffusers，因此，安装如下包:

```
pip install diffusers accelerate # 在torch环境下
pip install -U rich
```

SDE的工作在代码层面主要是以下几点：

- 提出了VP-SDE,VE-SDE,SUB-VP-SDE。
- 兼备了离散和连续版本。
- 提出了PC和ODE两种sampling形式。
- 更换了原来离散的训练loss。

其次，SDE的采样算法属于PC算法，算法图如下：

Algorithm 1 Predictor-Corrector (PC) sampling

Require:

N: Number of discretization steps for the reverse-time SDE

M: Number of corrector steps

Initialize $\mathbf{x}_N \sim p_T(\mathbf{x})$

for $i = N-1$ **to** 0 **do**

  $\mathbf{x}_i \leftarrow \text{Predictor}(\mathbf{x}_{i+1})$

  for $j = 1$ **to** M **do**

    $\mathbf{x}_i \leftarrow \text{Corrector}(\mathbf{x}_i)$

return $\mathbf{x}_0$

对于VE和VP两种采样形式，算法图如下：

Algorithm 2 PC sampling (VE SDE)

$\mathbf{x}_N \sim \mathcal{N}(\mathbf{0}, \sigma_{\max}^2 \mathbf{I})$
for $i = N-1$ **to** 0 **do**
  $\mathbf{x}'_i \leftarrow \mathbf{x}_{i+1} + (\sigma_{i+1}^2 - \sigma_i^2)\mathbf{s}_{\boldsymbol{\theta}^*}(\mathbf{x}_{i+1}, \sigma_{i+1})$
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
  $\mathbf{x}_i \leftarrow \mathbf{x}'_i + \sqrt{\sigma_{i+1}^2 - \sigma_i^2}\,\mathbf{z}$
  for $j = 1$ **to** M **do**
    $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
    $\mathbf{x}_i \leftarrow \mathbf{x}_i + \epsilon_i \mathbf{s}_{\boldsymbol{\theta}^*}(\mathbf{x}_i, \sigma_i) + \sqrt{2\epsilon_i}\,\mathbf{z}$
return $\mathbf{x}_0$

Algorithm 3 PC sampling (VP SDE)

$\mathbf{x}_N \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for $i = N-1$ **to** 0 **do**
  $\mathbf{x}'_i \leftarrow (2 - \sqrt{1-\beta_{i+1}})\mathbf{x}_{i+1} + \beta_{i+1}\mathbf{s}_{\boldsymbol{\theta}^*}(\mathbf{x}_{i+1}, i+1)$
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
  $\mathbf{x}_i \leftarrow \mathbf{x}'_i + \sqrt{\beta_{i+1}}\,\mathbf{z}$ Predictor
  for $j = 1$ **to** M **do** Corrector
    $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
    $\mathbf{x}_i \leftarrow \mathbf{x}_i + \epsilon_i \mathbf{s}_{\boldsymbol{\theta}^*}(\mathbf{x}_i, i) + \sqrt{2\epsilon_i}\,\mathbf{z}$
return $\mathbf{x}_0$

Algorithm 4 Corrector algorithm (VE SDE).

Require: $\{\sigma_i\}_{i=1}^N, r, N, M.$
$\mathbf{x}_N^0 \sim \mathcal{N}(\mathbf{0}, \sigma_{\max}^2 \mathbf{I})$
for $i \leftarrow N$ to 1 **do**
  for $j \leftarrow 1$ to M **do**
    $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
    $\mathbf{g} \leftarrow \mathbf{s}_{\boldsymbol{\theta}^*}(\mathbf{x}_i^{j-1}, \sigma_i)$
    $\epsilon \leftarrow 2(r\,\|\mathbf{z}\|_2 / \|\mathbf{g}\|_2)^2$
    $\mathbf{x}_i^j \leftarrow \mathbf{x}_i^{j-1} + \epsilon\,\mathbf{g} + \sqrt{2\epsilon}\,\mathbf{z}$
  $\mathbf{x}_{i-1}^0 \leftarrow \mathbf{x}_i^M$
return $\mathbf{x}_0^0$

Algorithm 5 Corrector algorithm (VP SDE).

Require: $\{\beta_i\}_{i=1}^N, \{\alpha_i\}_{i=1}^N, r, N, M.$
$\mathbf{x}_N^0 \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for $i \leftarrow N$ to 1 **do**
  for $j \leftarrow 1$ to M **do**
    $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
    $\mathbf{g} \leftarrow \mathbf{s}_{\boldsymbol{\theta}^*}(\mathbf{x}_i^{j-1}, i)$
    $\epsilon \leftarrow 2\alpha_i(r\,\|\mathbf{z}\|_2 / \|\mathbf{g}\|_2)^2$
    $\mathbf{x}_i^j \leftarrow \mathbf{x}_i^{j-1} + \epsilon\,\mathbf{g} + \sqrt{2\epsilon}\,\mathbf{z}$
  $\mathbf{x}_{i-1}^0 \leftarrow \mathbf{x}_i^M$
return $\mathbf{x}_0^0$

VE SDE（contiguous）

首先介绍一下SDE官方提供的demo，为：

```
from diffusers import DiffusionPipeline
model_id = ”google/ncsnpp–ffhq–1024”
sde_ve = DiffusionPipeline.from_pretrained(model_id)
image = sde_ve().images[0]
image[0].save(”sde_ve_generated_image.png”)
```

然后介绍huggingface的代码库，SDE的代码主要存在于pipeline中score_sde_ve代码目录下，以及schedulers目录下（毕竟也是采样算法），可以查看ScoreSdeVePipeline对于unet和scheduler的初始化，如下：

```
    unet: UNet2DModel
    scheduler: ScoreSdeVeScheduler
```

很明显，UNet2DModel还是老模型，而scheduler则是SDE的精华，即ScoreSdeVeScheduler，huggingface官网说ScoreSdeVpScheduler还在构建中，因此，首先查看ScoreSdeVePipeline的__call__方法：

```
sample = torch.randn(*shape, generator=generator) * self.scheduler.init_noise_sigma
sample = sample.to(self.device)

self.scheduler.set_timesteps(num_inference_steps)
self.scheduler.set_sigmas(num_inference_steps)

for i, t in enumerate(self.progress_bar(self.scheduler.timesteps)):
    sigma_t = self.scheduler.sigmas[i] * torch.ones(shape[0], device=self.device)
```

```
    # correction step
    for _ in range(self.scheduler.config.correct_steps):
        model_output = self.unet(sample, sigma_t).sample
        sample = self.scheduler.step_correct(model_output, sample, generator=generator).prev_sample

    # prediction step
    model_output = model(sample, sigma_t).sample
    output = self.scheduler.step_pred(model_output, t, sample, generator=generator)

    sample, sample_mean = output.prev_sample, output.prev_sample_mean

sample = sample_mean.clamp(0, 1)
sample = sample.cpu().permute(0, 2, 3, 1).numpy()
if output_type == "pil":
    sample = self.numpy_to_pil(sample)

if not return_dict:
    return (sample,)

return ImagePipelineOutput(images=sample)
```

一些不太重要的代码已经在这里删去，第一步先设置sample,这里的init_noise_sigma就是论文原文的$\sigma_{\max}$,默认设置为1348，而$\sigma_{\min}$设置为0.01：

```
sample = torch.randn(*shape, generator=generator) * self.scheduler.init_noise_sigma
sample = sample.to(self.device)
```

第二步设置scheduler一些参数：

```
        self.scheduler.set_timesteps(num_inference_steps)
        self.scheduler.set_sigmas(num_inference_steps)
```

对于set_timesteps，代码为：

```
    def set_timesteps(
        self, num_inference_steps: int, sampling_eps: float = None, device: Union[str, torch.device] = None
    ):
        """
        Sets the continuous timesteps used for the diffusion chain.
        Supporting function to be run before inference.
        """
        sampling_eps = sampling_eps if sampling_eps is not None else \
        self.config.sampling_eps

        self.timesteps = torch.linspace(1, sampling_eps, num_inference_steps,
                                        device=device)
```

这里的sampling_eps，默认设置为1e-5。然后对于方法set_sigmas，代码为：

```
    def set_sigmas(
        self, num_inference_steps: int, sigma_min: float = None, sigma_max: float = None, sampling_eps: float = None
    ):
        """
        Sets the noise scales used for the diffusion chain.
        Supporting function to be run before inference.
        The sigmas control the weight of the `drift` and `diffusion`
```

```
        components of sample update.
        """
        sigma_min = sigma_min if sigma_min is not None else self.config.sigma_min
        sigma_max = sigma_max if sigma_max is not None else self.config.sigma_max
        sampling_eps = sampling_eps if sampling_eps is not None else self.config.sampling_eps
        if self.timesteps is None:
            self.set_timesteps(num_inference_steps, sampling_eps)
        self.sigmas = sigma_min * (sigma_max / sigma_min) ** \
        (self.timesteps / sampling_eps)
        self.discrete_sigmas = torch.exp(torch.linspace(math.log(sigma_min),
                                math.log(sigma_max), num_inference_steps))
        self.sigmas = torch.tensor([sigma_min * (sigma_max / sigma_min) ** t \
                                    for t in self.timesteps])
```

这里的self.timesteps/sampling_eps应该就是步数（离散），可以理解为[1e5,1e5-1,...,1]这样一个序列。那么(sigma_max/sigma_min)就是1348/0.01，这样再以上面提到的序列作为指数项肯定爆炸了，我debug输出一下，果然（但这确实是官方的demo）：

```
tensor([      inf,       inf,       inf,  ...,       inf,       inf,
        1347.7513])
```

然后这里的self.discrete_sigmas，其实就是非线性的采样从$\sigma_{\min}$到$\sigma_{\max}$。最后由重新定义self.sigmas，挺离谱的，huggingface在维护什么呀，脑子有问题？这次没有除去sampling_eps，因此不会变成inf:

```
tensor([1.3480e+03, 1.3401e+03, 1.3322e+03,  ..., 1.0120e-02, 1.0060e-02,
        1.0001e-02])  # sigma
tensor([1.0000e-02, 1.0059e-02, 1.0119e-02,  ..., 1.3322e+03, 1.3401e+03,
        1.3480e+03])  # discrete_sigma
```

这里可以发现两个序列似乎刚刚是相反的次序，公式推导一下，设当前self.timesteps序列对应的值为t,那么对应的时间步为$\frac{(t-1)T}{\epsilon-1}$，然后第一个式子的对应项为$\exp(\log(\sigma_{\min})+(t-1)\frac{\log(\sigma_{\max})-\log(\sigma_{\min})}{\epsilon-1})$,而第二个式子则是$\sigma_{\min}(\frac{\sigma_{\max}}{\sigma_{\min}})^t$。第一个式子可以化简，变成$\sigma_{\min}(\frac{\sigma_{\max}}{\sigma_{\min}})^{\frac{t-1}{\epsilon-1}}$。由于$t$是线性递减的，因此如果这两个序列反序，也就意味着当self.timesteps的值为t和$1+\epsilon-t$时，两个对应的值应该相等，代入得：即需要保证：$(1+\epsilon-t)(\epsilon-1)==t-1$,展开左边得：$\epsilon+\epsilon^2-t\epsilon-1-\epsilon+t$，由于$\epsilon^2$过小可以忽略，因此结果为：$t-t\epsilon-1$,而$t\epsilon\in[\epsilon,\epsilon^2]$,因此也是因为这一项使得看上去略微有些差异。

下一步，便是对由T到1的每个时间步进行采样并执行PC算法，第一步是计算σ_t，由于原来self.sigmas中的σ为一个值，而VE-SDE要求是一个向量，因此进行如下操作：

```
sigma_t = self.scheduler.sigmas[i] * torch.ones(shape[0], device=self.device)
```

第二步是执行corrector算法（这里和SDE论文中是反着来的，确实挺奇怪），如下：

```
# correction step
for _ in range(self.scheduler.config.correct_steps):
    model_output = self.unet(sample, sigma_t).sample
    sample = self.scheduler.step_correct(model_output, sample, generator=generator).prev_sample
```

model_output是对应的unet的输出，其中一般DDPM调用unet的forward的第二个参数一般是t,而这里变成了σ_t,当然这一点和原文一致，即原文的$g\leftarrow s_\theta*(x_t^j,\sigma_t)$, where j=0 in this case。然后执行step_correct方法：

```
    def step_correct(
        self,
```

```
        model_output: torch.FloatTensor,
        sample: torch.FloatTensor,
        generator: Optional[torch.Generator] = None,
        return_dict: bool = True,
    ) -> Union[SchedulerOutput, Tuple]:
        noise = torch.randn(sample.shape, layout=sample.layout, generator=generator).to(sample.device)
        # compute step size from the model_output, the noise, and the snr
        grad_norm = torch.norm(model_output.reshape(model_output.shape[0], -1), dim=-1).mean()
        noise_norm = torch.norm(noise.reshape(noise.shape[0], -1), dim=-1).mean()
        step_size = (self.config.snr * noise_norm / grad_norm) ** 2 * 2
        step_size = step_size * torch.ones(sample.shape[0]).to(sample.device)
        # compute corrected sample: model_output term and noise term
        step_size = step_size.flatten()
        while len(step_size.shape) < len(sample.shape):
            step_size = step_size.unsqueeze(-1)
        prev_sample_mean = sample + step_size * model_output
        prev_sample = prev_sample_mean + ((step_size * 2) ** 0.5) * noise
        return SchedulerOutput(prev_sample=prev_sample)
```

为了方便起见进行了略微化简，这里的noise$\sim \mathcal{N}(0,1)$,也就是原文中的z,这里的snr就是原文的r,所以第12行求step_size都是在计算原文公式：$\epsilon \leftarrow 2(||z||_2/||g||_2)^2$,第13行到第17行全是工程技巧，第18和19行合起来就是执行：

```
prev_sample = sample + step_size * model_output + ((step_size * 2) ** 0.5) * noise
```

即原文的公式：$x_t \leftarrow x_i + \epsilon g + \sqrt{2\epsilon}z$，完成单步的corrector操作。这样的操作执行多步后，需要完成predictor的算法，即如下：

```
            # prediction step
            model_output = model(sample, sigma_t).sample
            output = self.scheduler.step_pred(model_output, t, sample, generator=generator)
```

这里的model就是self.unet,我们查看self.scheduler.step_pred代码如下：

```
    def step_pred(
        self,
        model_output: torch.FloatTensor,
        timestep: int,
        sample: torch.FloatTensor,
        generator: Optional[torch.Generator] = None,
        return_dict: bool = True,
    ) -> Union[SdeVeOutput, Tuple]:
        timestep = timestep * torch.ones(
            sample.shape[0], device=sample.device
        )  # torch.repeat_interleave(timestep, sample.shape[0])
        timesteps = (timestep * (len(self.timesteps) - 1)).long()
        timesteps = timesteps.to(self.discrete_sigmas.device)
        sigma = self.discrete_sigmas[timesteps].to(sample.device)
        adjacent_sigma = self.get_adjacent_sigma(timesteps, timestep).to(sample.device)
        drift = torch.zeros_like(sample)
        diffusion = (sigma**2 - adjacent_sigma**2) ** 0.5
        # equation 6 in the paper: the model_output modeled by the network is grad_x log pt(x)
        # also equation 47 shows the analog from SDE models to ancestral sampling methods
        diffusion = diffusion.flatten()
        while len(diffusion.shape) < len(sample.shape):
            diffusion = diffusion.unsqueeze(-1)
        drift = drift - diffusion**2 * model_output
```

```
    #  equation 6: sample noise for the diffusion term of
    noise = torch.randn(sample.shape, layout=sample.layout, generator=generator).to(sample.device)
    prev_sample_mean = sample - drift  # subtract because `dt` is a small negative timestep
    # TODO is the variable diffusion the correct scaling term for the noise?
    prev_sample = prev_sample_mean + diffusion * noise  # add impact of diffusion field g
    return SdeVeOutput(prev_sample=prev_sample, prev_sample_mean=prev_sample_mean)
```

这里代码也进行了略微简化，第14行前都在取出来σ,只不过是t从1999变到1的迭代过程，因此σ也是由大变小的过程，这里的get_adjacent_sigma代码如下：

```
  def get_adjacent_sigma(self, timesteps, t):
    return torch.where(
        timesteps == 0,
        torch.zeros_like(t.to(timesteps.device)),
        self.discrete_sigmas[timesteps - 1].to(timesteps.device),
    )
```

介绍一下，timesteps是int类型，也就是离散的时间步，t$\in [0,1]$,是一个连续的时间步。因此这个函数做的事情很简单，就是取出σ_{t-1},因此，adjacent_sigma = $\sigma_{discrete,t-1}$,而sigma = $\sigma_{discrete,t}$,下一步就是求出scalar function,如下：

```
    drift = torch.zeros_like(sample)
    diffusion = (sigma**2 - adjacent_sigma**2) ** 0.5
```

diffusion就是原文的$\sqrt{\sigma_{i+1}^2-\sigma_i^2}$,然后计算$-(x_{i+1}-x_i)$,代码为：

```
    diffusion = diffusion.flatten()
    while len(diffusion.shape) < len(sample.shape):
        diffusion = diffusion.unsqueeze(-1)
    drift = drift - diffusion**2 * model_output
```

这里的dirft相当于原文的：$(\sigma_{i+1}^2-\sigma_i^2)s_\theta*(x_{i+1},\sigma_{i+1})$,下一步计算论文中的$x'$,如下：

```
noise = torch.randn(sample.shape, layout=sample.layout, generator=generator).to(sample.device)
prev_sample_mean = sample - drift  # subtract because `dt` is a small negative timestep
# TODO is the variable diffusion the correct scaling term for the noise?
prev_sample = prev_sample_mean + diffusion * noise  # add impact of diffusion field g
```

ev_sample_mean就是原文的x'_i,然后ev_sample则是原文的x_i,公式是一一对应的。调度返回，至此predictor算法完成，最后经过无数步后，通过以下归一化得到生成图像：

```
sample = sample_mean.clamp(0, 1)
sample = sample.cpu().permute(0, 2, 3, 1).numpy()
```

VE SDE（discontiguous）[SMLD]

这个diffusers里面没有，代码还是来自于song的代码库，如下：

```
  if continuous:
    labels = sde.marginal_prob(torch.zeros_like(x), t)[1]
  else:
    # For VE-trained models, t=0 corresponds to the highest noise level
    labels = sde.T - t
    labels *= sde.N - 1
    labels = torch.round(labels).long()
```

在VE SDE中，score function就是unet的预测结果，对于连续状态，查看sde.marginal_prob:

```
def marginal_prob(self, x, t):
  std = self.sigma_min * (self.sigma_max / self.sigma_min) ** t
  mean = x
  return mean, std
```

可以发现和前文的玩意是一摸一样的，即：$\sigma_{\min}(\frac{\sigma_{\max}}{\sigma_{\min}})^t$。然后对于离散状态，T是1，N是步数，其实相当于对应的时间步，因此理解上是相同的。

VP SDE（contiguous）

SDE和huggingface官方没有提供VP SDE的demo,但是不难知道核心代码在：ScoreVpSdeScheduler:

```
def step_pred(self, score, x, t, generator=None):
    log_mean_coeff = (
        -0.25 * t**2 * (self.config.beta_max - self.config.beta_min) - 0.5 * t * self.config.beta_min
    )
    std = torch.sqrt(1.0 - torch.exp(2.0 * log_mean_coeff))
    std = std.flatten()
    while len(std.shape) < len(score.shape):
        std = std.unsqueeze(-1)
    score = -score / std
    # compute
    dt = -1.0 / len(self.timesteps)
    beta_t = self.config.beta_min + t * (self.config.beta_max - self.config.beta_min)
    beta_t = beta_t.flatten()
    while len(beta_t.shape) < len(x.shape):
        beta_t = beta_t.unsqueeze(-1)
    drift = -0.5 * beta_t * x
    diffusion = torch.sqrt(beta_t)
    drift = drift - diffusion**2 * score
    x_mean = x + drift * dt
    # add noise
    noise = torch.randn(x.shape, layout=x.layout, generator=generator).to(x.device)
    x = x_mean + diffusion * math.sqrt(-dt) * noise
    return x, x_mean
```

由于VPSDE算法没有corrector，所以只有predictor，这里的log_mean_coeff = $\frac{-1}{4}t^2(\beta_{max}-\beta_{min})-\frac{1}{2}t\beta_{min}$,std = $\sqrt{1-\exp(2[\frac{-1}{4}t^2(\beta_{max}-\beta_{min})-\frac{1}{2}t\beta_{min}])}$.再通过语句：

```
score = -score / std
```

转化为score function，对应论文中的：$\nabla_x \log p(x)$。由于exp中的项可以化简为：$\frac{-1}{2}((\beta_{max}-\beta_{min})t-\frac{\beta_{min}}{\beta_{max}-\beta_{min}})^2$。然后dt为$-\frac{1}{T}=-dt$,接下来通过线性插值求出当前时刻的$\beta_t$。经过14–

15的工程代码后，第16行则是完成$\text{drift} \leftarrow -\frac{1}{2}\beta_t x_t$，随后通过17-18行代码，完成$\text{drift} \leftarrow -\frac{1}{2}\beta_t x_t - \beta_t \nabla_x \log p(x)$,然后在第19行变成：$\text{x_mean} \leftarrow x_t + [\frac{1}{2}\beta_t x_t + \beta_t \nabla_x \log p(x)]dt$,最后22行变成：$x_{t-1} \leftarrow x_t + [\frac{1}{2}\beta_t x_t + \beta_t \nabla_x \log p(x)]dt + \sqrt{\beta_t dt}\epsilon$，这里的噪声是$\epsilon$,因此就是式子：$-dx = [\frac{1}{2}\beta_t x_t + \beta_t \nabla_x \log p(x)]dt + \sqrt{\beta_t dt}\epsilon$,然后将$\sqrt{dt}\epsilon$视为$dw$，那么便得到：$-dx = [\frac{1}{2}\beta_t x_t + \beta_t \nabla_x \log p(x)]dt + \sqrt{\beta_t}dw$，由于$dw$是标准高斯分布，因此是不是加负号是一致的，所以结果为：$dx = -[\frac{1}{2}\beta_t x_t + \beta_t \nabla_x \log p(x)]dt + \sqrt{\beta_t}dw$，和原文相同，这便是连续场景下（即SDE）的结果。

VP SDE（discontiguous）[DDPM]

这个部分代码不存在于diffusers中，也就是原DDPM的理解方式，这里根据song的代码库的代码再过一遍，下面列出contiguous和discontiguous两种形式的代码：

```
def score_fn(x, t):
  if continuous or isinstance(sde, sde_lib.subVPSDE):
    labels = t * 999
    score = model_fn(x, labels)
    std = sde.marginal_prob(torch.zeros_like(x), t)[1]
  else:
    labels = t * (sde.N - 1)
    score = model_fn(x, labels)
    std = sde.sqrt_1m_alphas_cumprod.to(labels.device)[labels.long()]
```

第一种是contiguous()形式，因此默认远端为999，之前提到过，而离散形式下，则远端就是t(N-1)。两者std的计算存在着差异：

```
# contiguous
  def marginal_prob(self, x, t):
    log_mean_coeff = -0.25 * t ** 2 * (self.beta_1 - self.beta_0) - 0.5 * t * self.beta_0
    mean = torch.exp(log_mean_coeff)[:, None, None, None] * x
    std = 1 - torch.exp(2. * log_mean_coeff)
    return mean, std
```

```
# discontiguous
torch.sqrt(1. - self.alphas_cumprod)[t(N-1)]
```

这边对应的其实是score function的求法，discontiguous()是一种基于Tweedie's Formula (详细看一文解释 经验贝叶斯估计, Tweedie's formula)推导的结果，即：$\nabla_x \log p(x) = -\frac{\epsilon}{\sqrt{1-\bar{\alpha}_t}}$。而contiguous()则是$-\frac{\epsilon}{\sqrt{1-\exp(2[\frac{-1}{4}t^2(\beta_{max}-\beta_{min}) - \frac{1}{2}t\beta_{min}])}}$。

SUB VP SDE（contiguous）

sub vp sde只有contiguous形式，因为本来就是在连续条件下推导的，它也是采用了PC算法，和VP SDE一样，predictor采用euler_maruyama，而corrector采用none。唯一的区别在于调用sde的时刻，如下代码：

```
@register_predictor(name='euler_maruyama')
class EulerMaruyamaPredictor(Predictor):
  def __init__(self, sde, score_fn, probability_flow=False):
    super().__init__(sde, score_fn, probability_flow)

  def update_fn(self, x, t):
    dt = -1. / self.rsde.N
    z = torch.randn_like(x)
    drift, diffusion = self.rsde.sde(x, t)
    x_mean = x + drift * dt
    x = x_mean + diffusion[:, None, None, None] * np.sqrt(-dt) * z
    return x, x_mean
```

核心在第9行，调用self.rsde.sde时，查看vp sde和sub vp sde 的区别：

```
# sub vp sde
  def sde(self, x, t):
    beta_t = self.beta_0 + t * (self.beta_1 - self.beta_0)
    drift = -0.5 * beta_t[:, None, None, None] * x
    discount = 1. - torch.exp(-2 * self.beta_0 * t - (self.beta_1 - self.beta_0) * t ** 2)
    diffusion = torch.sqrt(beta_t * discount)
    return drift, diffusion
# vp sde
  def sde(self, x, t):
    beta_t = self.beta_0 + t * (self.beta_1 - self.beta_0)
    drift = -0.5 * beta_t[:, None, None, None] * x
    diffusion = torch.sqrt(beta_t)
    return drift, diffusion
```

核心其实在于diffusion的求解，sub vp sde的形式为：$\sqrt{\beta_t(1-e^{-2\int_0^t \beta_t dt})}$,而vp sde的形式为$\sqrt{\beta_t}$。这里的$\beta_t$是一个关于$t$的线性函数，即$\beta_t = \beta_{min} + \frac{t}{N}(\beta_{max} - \beta_{min})$,因此积分结果是$t\beta_{min} + \frac{t^2}{2N}(\beta_{max} - \beta_{min})$，代入$0$和$t$,发现完全符合。

ODE算法（DEMO介绍）：

ODE算法可以嵌入于任何SDE所采用的采样形式，即VE SDE, VP SDE, SUB VP SDE。这里只讲其核心，即采样流程：

```
  def ode_sampler(model, z=None):
    with torch.no_grad():
      x = sde.prior_sampling(shape).to(device)
      def ode_func(t, x):
        x = from_flattened_numpy(x, shape).to(device).type(torch.float32)
        vec_t = torch.ones(shape[0], device=x.device) * t
        drift = drift_fn(model, x, vec_t)
        return to_flattened_numpy(drift)
      solution = integrate.solve_ivp(ode_func, (sde.T, eps), to_flattened_numpy(x),
                                     rtol=rtol, atol=atol, method=method)
      nfe = solution.nfev
      x = torch.tensor(solution.y[:, -1]).reshape(shape).to(device).type(torch.float32)
      if denoise:
        x = denoise_update_fn(model, x)
      x = inverse_scaler(x)
      return x, nfe
```

相比于SDE，这里没有PC了，核心就是该算法：

$$dx=\{\mathbf{f}_i(x,t)-\frac{1}{2}[\nabla\cdot[\mathbf{G}(x,t)\mathbf{G}(x,t)^T]+\mathbf{G}(x,t)\mathbf{G}(x,t)^T\nabla_x\ log\ p_t(x)]\}dt$$

,这里的$f(t)$和$G(t)$就是dirft和diffusion。代码中的第三行取出随机生成的噪声x，然后，调用ode_func计算dirft:

```
def drift_fn(model, x, t):
  """Get the drift function of the reverse-time SDE."""
  score_fn = get_score_fn(sde, model, train=False, continuous=True)
  rsde = sde.reverse(score_fn, probability_flow=True)
  return rsde.sde(x, t)[0]
```

这里的sde.reverse调用了SDE基类的方法，如下：

```
def reverse(self, score_fn, probability_flow=False):
  """Create the reverse-time SDE/ODE.

  Args:
    score_fn: A time-dependent score-based model that takes x and t and returns the score.
    probability_flow: If `True`, create the reverse-time ODE used for probability flow sampling.
  """
  N = self.N
  T = self.T
  sde_fn = self.sde
  discretize_fn = self.discretize

  # Build the class for reverse-time SDE.
  class RSDE(self.__class__):
    def __init__(self):
      self.N = N
      self.probability_flow = probability_flow
    @property
    def T(self):
      return T
    def sde(self, x, t):
      """Create the drift and diffusion functions for the reverse SDE/ODE."""
      drift, diffusion = sde_fn(x, t)
      score = score_fn(x, t)
      drift = drift - diffusion[:, None, None, None] ** 2 * score * (0.5 if self.probability_flow else 1.)
      # Set the diffusion function to zero for ODEs.
      diffusion = 0. if self.probability_flow else diffusion
      return drift, diffusion
    def discretize(self, x, t):
      """Create discretized iteration rules for the reverse diffusion sampler."""
      f, G = discretize_fn(x, t)
      rev_f = f - G[:, None, None, None] ** 2 * score_fn(x, t) * (0.5 if self.probability_flow else 1.)
      rev_G = torch.zeros_like(G) if self.probability_flow else G
      return rev_f, rev_G
  return RSDE()
```

调用的是RSDE中的sde方法，首先调用sde_fn来得到前向流的drift和diffusion,然后计算score,新的drift为：

$$f'(x,t)=f(x,t)-\frac{1}{2}G(x,t)^2\nabla_x\log p(x)$$

,相当于上面公式中第一项和第三项相加的结果。由于当前$G(x,t)$是一个值，所以求导直接为0，第二项没有，不存在，因此这里的ode_func完成了$f'(x,t)$的计算，即ODE的drift，然后调用库函数integrate.solve_ivp解ODE方程：

```
def solve_ivp(fun, t_span, y0, method='RK45', t_eval=None, dense_output=False,
              events=None, vectorized=False, args=None, **options):
    """Solve an initial value problem for a system of ODEs.

    This function numerically integrates a system of ordinary differential
    equations given an initial value::

        dy / dt = f(t, y)
        y(t0) = y0

    Here t is a 1-D independent variable (time), y(t) is an
    N-D vector-valued function (state), and an N-D
    vector-valued function f(t, y) determines the differential equations.
    The goal is to find y(t) approximately satisfying the differential
    equations, given an initial value y(t0)=y0.

    Some of the solvers support integration in the complex domain, but note
    that for stiff ODE solvers, the right-hand side must be
    complex-differentiable (satisfy Cauchy-Riemann equations [11]_).
    To solve a problem in the complex domain, pass y0 with a complex data type.
    Another option always available is to rewrite your problem for real and
    imaginary parts separately.
```

得到函数在T时的解。最后一些工程技巧后返回。

LOSS FUNCTION：

loss function的获取代码如下：

```
  if continuous:
    loss_fn = get_sde_loss_fn(sde, train, reduce_mean=reduce_mean,
                              continuous=True, likelihood_weighting=likelihood_weighting)
  else:
    assert not likelihood_weighting, "Likelihood weighting is not supported for original SMLD/DDPM training."
    if isinstance(sde, VESDE):
      loss_fn = get_smld_loss_fn(sde, train, reduce_mean=reduce_mean)
    elif isinstance(sde, VPSDE):
      loss_fn = get_ddpm_loss_fn(sde, train, reduce_mean=reduce_mean)
```

get_smld_loss_fn代码如下：

```
  def loss_fn(model, batch):
    smld_sigma_array = torch.flip(torch.exp(torch.linspace(np.log(self.sigma_min),
                                                           np.log(self.sigma_max), N)),
                                  dims=(0,))
    model_fn = mutils.get_model_fn(model, train=train)
    labels = torch.randint(0, vesde.N, (batch.shape[0],), device=batch.device)
    sigmas = smld_sigma_array.to(batch.device)[labels]
    noise = torch.randn_like(batch) * sigmas[:, None, None, None]
    perturbed_data = noise + batch
    score = model_fn(perturbed_data, labels)
    target = -noise / (sigmas ** 2)[:, None, None, None]
    losses = torch.square(score - target)
```

```
    losses = reduce_op(losses.reshape(losses.shape[0], -1), dim=-1) * sigmas ** 2
    loss = torch.mean(losses)
    return loss
```

这实际上就是训练代码，算法为计算以下loss：

$$||x_0 + \epsilon * \exp(\log(\sigma_{min}) + \frac{t}{T}(\log(\sigma_{max}) - \log(\sigma_{min}))) + \frac{\epsilon}{\exp(\log(\sigma_{min}) + \frac{t}{T}(\log(\sigma_{max}) - \log(\sigma_{min})))}||_2$$

get_ddpm_loss_fn代码如下：

```
  def loss_fn(model, batch):
    model_fn = mutils.get_model_fn(model, train=train)
    labels = torch.randint(0, vpsde.N, (batch.shape[0],), device=batch.device)
    sqrt_alphas_cumprod = vpsde.sqrt_alphas_cumprod.to(batch.device)
    sqrt_1m_alphas_cumprod = vpsde.sqrt_1m_alphas_cumprod.to(batch.device)
    noise = torch.randn_like(batch)
    perturbed_data = sqrt_alphas_cumprod[labels, None, None, None] * batch + \
                     sqrt_1m_alphas_cumprod[labels, None, None, None] * noise
    score = model_fn(perturbed_data, labels)
    losses = torch.square(score - noise)
    losses = reduce_op(losses.reshape(losses.shape[0], -1), dim=-1)
    loss = torch.mean(losses)
    return loss
```

这个式子非常熟悉，截图一下：

$$\|\boldsymbol{x}_0 - \bar{\boldsymbol{\mu}}(\boldsymbol{x}_t)\|^2 = \frac{\bar{\beta}_t^2}{\bar{\alpha}_t^2}\left\|\boldsymbol{\varepsilon} - \boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\bar{\alpha}_t\boldsymbol{x}_0 + \bar{\beta}_t\boldsymbol{\varepsilon}, t)\right\|^2$$

get_sde_loss_fn代码如下：

```
  def loss_fn(model, batch):
    score_fn = mutils.get_score_fn(sde, model, train=train, continuous=continuous)
    t = torch.rand(batch.shape[0], device=batch.device) * (sde.T - eps) + eps
    z = torch.randn_like(batch)
    mean, std = sde.marginal_prob(batch, t)
    perturbed_data = mean + std[:, None, None, None] * z
    score = score_fn(perturbed_data, t)
    if not likelihood_weighting:
      losses = torch.square(score * std[:, None, None, None] + z)
      losses = reduce_op(losses.reshape(losses.shape[0], -1), dim=-1)
    else:
      g2 = sde.sde(torch.zeros_like(batch), t)[1] ** 2
      losses = torch.square(score + z / std[:, None, None, None])
      losses = reduce_op(losses.reshape(losses.shape[0], -1), dim=-1) * g2
    loss = torch.mean(losses)
    return loss
```

这里sde.marginal_prob代码根据不同的SDE算法而变化，但std前文已经出现过，是计算score function的时候除去的值，事实上，第6行这一步其实是依据于

$$\mu_z = z + \Sigma_z \nabla \log p(z)$$

得到的，这个等式可以看博客

https://zhuanlan.zhihu.com/p/589106222理解。这里不是求μ而是求z。这种情况下估计的score function就可以计算，随后loss为$||s_\theta*(\mu_z+z\sigma_z,t)-(-\frac{z}{\sigma_t})||$,后面一项就是$\nabla_x \log p(x)$。

DENOISING DIFFUSION IMPLICIT MODELS[ICLR2021]

DEMO介绍：

代码在: ddim 里，但是可以直接查看diffusers，因此，安装如下包:

```
pip install diffusers accelerate # 在torch环境下
pip install -U rich
```

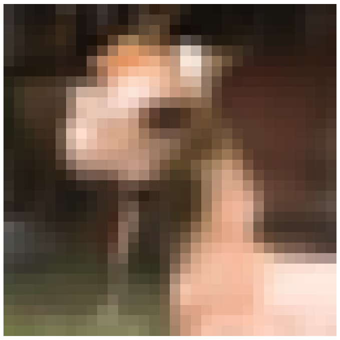
demo代码：

```
from diffusers import DDIMPipeline

model_id = "google/ddpm-cifar10-32"

# load model and scheduler
ddim = DDIMPipeline.from_pretrained(model_id)

# run pipeline in inference (sample random noise and denoise)
image = ddim(num_inference_steps=50).images[0]

# save image
image.save("ddim_generated_image.png")
```

这个代码跑起来会报错，当然原因是因为这个ddim.scheduler使用了DDPMScheduler，因此要进行设置为DDIMScheduler，修改后的demo如下：

```
from diffusers import DDIMPipeline

model_id = "google/ddpm-cifar10-32"

# load model and scheduler
ddim = DDIMPipeline.from_pretrained(model_id)

from diffusers import DDIMScheduler

scheduler = DDIMScheduler.from_config(ddim.scheduler.config)
ddim.scheduler = scheduler

# run pipeline in inference (sample random noise and denoise)
image = ddim(num_inference_steps=50).images[0]

# save image
image.save("ddim_generated_image.png")
```

最后得到的图片为（分辨率32x32）：



DDIM对训练不存在改动，仅仅改动了推理部分，代码如下：

```
@torch.no_grad()
  def __call__(
      self,
      batch_size: int = 1,
      generator: Optional[Union[torch.Generator, List[torch.Generator]]] = None,
      eta: float = 0.0,
      num_inference_steps: int = 50,
      use_clipped_model_output: Optional[bool] = None,
      output_type: Optional[str] = "pil",
      return_dict: bool = True,
  ) -> Union[ImagePipelineOutput, Tuple]:
      if (
          generator is not None
          and isinstance(generator, torch.Generator)
          and generator.device.type != self.device.type
          and self.device.type != "mps"
      ):
          message = (
              f"The `generator` device is `{generator.device}` and does not match the pipeline "
              f"device `{self.device}`, so the `generator` will be ignored. "
              f'Please use `generator=torch.Generator(device="{self.device}")` instead.'
          )
          deprecate(
              "generator.device == 'cpu'",
              "0.12.0",
              message,
          )
          generator = None

      # Sample gaussian noise to begin loop
      if isinstance(self.unet.sample_size, int):
          image_shape = (batch_size, self.unet.in_channels, self.unet.sample_size, self.unet.sample_size)
      else:
          image_shape = (batch_size, self.unet.in_channels, *self.unet.sample_size)

      if isinstance(generator, list) and len(generator) != batch_size:
          raise ValueError(
              f"You have passed a list of generators of length {len(generator)}, but requested an effective batch"
              f" size of {batch_size}. Make sure the batch size matches the length of the generators."
          )

      rand_device = "cpu" if self.device.type == "mps" else self.device
      if isinstance(generator, list):
          shape = (1,) + image_shape[1:]
```

```
            image = [
                torch.randn(shape, generator=generator[i], device=rand_device, dtype=self.unet.dtype)
                for i in range(batch_size)
            ]
            image = torch.cat(image, dim=0).to(self.device)
        else:
            image = torch.randn(image_shape, generator=generator, device=rand_device, dtype=self.unet.dtype)
            image = image.to(self.device)

        # set step values
        self.scheduler.set_timesteps(num_inference_steps)

        for t in self.progress_bar(self.scheduler.timesteps):
            # 1. predict noise model_output
            model_output = self.unet(image, t).sample

            # 2. predict previous mean of image x_t-1 and add variance depending on eta
            # eta corresponds to η in paper and should be between [0, 1]
            # do x_t -> x_t-1
            image = self.scheduler.step(
                model_output, t, image, eta=eta, use_clipped_model_output=use_clipped_model_output,
generator=generator
            ).prev_sample

        image = (image / 2 + 0.5).clamp(0, 1)
        image = image.cpu().permute(0, 2, 3, 1).numpy()
        if output_type == "pil":
            image = self.numpy_to_pil(image)

        if not return_dict:
            return (image,)

        return ImagePipelineOutput(images=image)
```

其中这里在55行前都是在做一些准备工作，这里的self.scheduler是一个采样工具，首先需要设置 `num_inference_steps` 这个参数，也意味着采样的步数，查看方法 `self.scheduler.set_timesteps` ，代码为：

```
    def set_timesteps(self, num_inference_steps: int, device: Union[str, torch.device] = None):
        self.num_inference_steps = num_inference_steps
        step_ratio = self.config.num_train_timesteps // self.num_inference_steps
        timesteps = (np.arange(0, num_inference_steps) * step_ratio).round()[::-1].copy().astype(np.int64)
        self.timesteps = torch.from_numpy(timesteps).to(device)
        self.timesteps += self.config.steps_offset
```

这里目的很简单，会创建一个list为 `[step_ratio,2*step_ratio,...,(num_inference_steps-1)*step_ratio]` ，这个便是SDE的采样时间步，从属于 `[0,1]` 。

下一步便是在循环内完成正式的inference,而第一步是得到unet所拟合的单步噪声：

```
model_output = self.unet(image, t).sample
```

这里的unet采用的是 `UNet2DModel` ，最后返回的函数调用为：

```
        # 6. post-process
        sample = self.conv_norm_out(sample)
        sample = self.conv_act(sample)
        sample = self.conv_out(sample)
```

```
        if skip_sample is not None:
            sample += skip_sample

        if self.config.time_embedding_type == "fourier":
            timesteps = timesteps.reshape((sample.shape[0], *([1] * len(sample.shape[1:]))))
            sample = sample / timesteps

        if not return_dict:
            return (sample,)

        return UNet2DOutput(sample=sample)
```

这里的UNet2DOutput只是一个只有一个变量的class，因此在取 `self.unet(image, t).sample` 时其实取的就是UNet的输出，因此对于下一步，便是调用scheduler.step完成真正的条件概率采样，代码为：

```
            image = self.scheduler.step(
                model_output, t, image, eta=eta, use_clipped_model_output=use_clipped_model_output,
generator=generator
            ).prev_sample
```

```
    def step(
        self,
        model_output: torch.FloatTensor,
        timestep: int,
        sample: torch.FloatTensor,
        eta: float = 0.0,
        use_clipped_model_output: bool = False,
        generator=None,
        variance_noise: Optional[torch.FloatTensor] = None,
        return_dict: bool = True,
    ) -> Union[DDIMSchedulerOutput, Tuple]:
        """
        Predict the sample at the previous timestep by reversing the SDE. Core function to propagate the diffusion
        process from the learned model outputs (most often the predicted noise).

        Args:
            model_output (`torch.FloatTensor`): direct output from learned diffusion model.
            timestep (`int`): current discrete timestep in the diffusion chain.
            sample (`torch.FloatTensor`):
                current instance of sample being created by diffusion process.
            eta (`float`): weight of noise for added noise in diffusion step.
            use_clipped_model_output (`bool`): if `True`, compute "corrected" `model_output` from the clipped
                predicted original sample. Necessary because predicted original sample is clipped to [-1, 1] when
                `self.config.clip_sample` is `True`. If no clipping has happened, "corrected" `model_output` would
                coincide with the one provided as input and `use_clipped_model_output` will have not effect.
            generator: random number generator.
            variance_noise (`torch.FloatTensor`): instead of generating noise for the variance using `generator`, we
                can directly provide the noise for the variance itself. This is useful for methods such as
                CycleDiffusion. (https://arxiv.org/abs/2210.05559)
            return_dict (`bool`): option for returning tuple rather than DDIMSchedulerOutput class

        Returns:
            [`~schedulers.scheduling_utils.DDIMSchedulerOutput`] or `tuple`:
            [`~schedulers.scheduling_utils.DDIMSchedulerOutput`] if `return_dict` is True, otherwise a `tuple`. When
```

```
    returning a tuple, the first element is the sample tensor.

"""
```

这个是开头，第一步是取上一步的时间步：

```
prev_timestep = timestep - self.config.num_train_timesteps // self.num_inference_steps
```

计算α和β,如下：

```
alpha_prod_t = self.alphas_cumprod[timestep]
alpha_prod_t_prev = self.alphas_cumprod[prev_timestep] if prev_timestep >= 0 else self.final_alpha_cumprod

beta_prod_t = 1 - alpha_prod_t
```

这里的对应关系是alpha_prod_t = $\overline{\alpha}_t$,alpha_prod_t_prev = $\overline{\alpha}_{t-1}$,beta_prod_t = $\overline{\beta}_t$。下一步根据DDIM原文公式12进行计算，这边直接截图：

$$\boldsymbol{x}_{t-1} = \sqrt{\alpha_{t-1}} \underbrace{\left(\frac{\boldsymbol{x}_t - \sqrt{1-\alpha_t}\epsilon_\theta^{(t)}(\boldsymbol{x}_t)}{\sqrt{\alpha_t}} \right)}_{\text{" predicted } \boldsymbol{x}_0 \text{"}} + \underbrace{\sqrt{1-\alpha_{t-1}-\sigma_t^2} \cdot \epsilon_\theta^{(t)}(\boldsymbol{x}_t)}_{\text{"direction pointing to } \boldsymbol{x}_t \text{"}} + \underbrace{\sigma_t \epsilon_t}_{\text{random noise}} \qquad (12)$$

这里的α_t和β_t其实相当于加上了上横杠的结果，即$\overline{\alpha}_t, \overline{\beta}_t$,因此也就是累乘。下一步预测$x_0$:

```
# 3. compute predicted original sample from predicted noise also called
# "predicted x_0" of formula (12) from https://arxiv.org/pdf/2010.02502.pdf
if self.config.prediction_type == "epsilon":
    pred_original_sample = (sample - beta_prod_t ** (0.5) * model_output) / alpha_prod_t ** (0.5)
elif self.config.prediction_type == "sample":
    pred_original_sample = model_output
elif self.config.prediction_type == "v_prediction":
    pred_original_sample = (alpha_prod_t**0.5) * sample - (beta_prod_t**0.5) * model_output
    # predict V
    model_output = (alpha_prod_t**0.5) * model_output + (beta_prod_t**0.5) * sample
else:
    raise ValueError(
        f"prediction_type given as {self.config.prediction_type} must be one of `epsilon`, `sample`, or"
        " `v_prediction`"
    )
```

先看 `self.config.prediction_type=="epsilon"` 分支，sample = x_t,model_output = $\epsilon_\theta^{(t)}(x_t)$,因此这个判断语句下的结果是 $x_{0,pred} = \frac{x_t - \sqrt{1-\overline{\alpha}_t}\epsilon_\theta^{(t)}(x_t)}{\sqrt{\overline{\alpha}_t}}$,然后看 `self.config.prediction_type=="sample"` 分支，这个判断语句下的结果是 $x_{0,pred} = \epsilon_\theta^{(t)}(x_t)$,最后看 `elif self.config.prediction_type == "v_prediction"` 分支，这个判断条件下的结果是 $x_{0,pred} = \sqrt{\overline{\alpha}_t}x_t - \sqrt{1-\overline{\alpha}_t}\epsilon_\theta^{(t)}(x_t)$。

下一步，对预测的x_0进行clip操作，这个不在论文中，只是工程技巧：

```
# 4. Clip "predicted x_0"
if self.config.clip_sample:
    pred_original_sample = torch.clamp(pred_original_sample, -1, 1)
```

下一步，使用论文中的公式16计算σ_t，之所以进行这个操作是为了使用η控制这个参数，从而可以在DDPM的SDE采样和ODE采样中任意伸缩：

```
# 5. compute variance: "sigma_t(η)" -> see formula (16)
# σ_t = sqrt((1 − α_t−1)/(1 − α_t)) * sqrt(1 − α_t/α_t−1)
variance = self._get_variance(timestep, prev_timestep)
std_dev_t = eta * variance ** (0.5)
```

得到的结果是std_dev_t = σ_t,下一步对unet的输出进行转换，即$\epsilon_{pred} = \frac{x_t - \sqrt{\bar{\alpha}_t} x_{0,pred}}{\sqrt{1-\bar{\alpha}_t}}$：

```
if use_clipped_model_output:
    # the model_output is always re-derived from the clipped x_0 in Glide
    model_output = (sample - alpha_prod_t ** (0.5) * pred_original_sample) / beta_prod_t ** (0.5)
```

这个操作如果按照之前第一个分支预测的x_t就是又变回去unet的预测了，也就是$\epsilon_{pred} = \epsilon_\theta^{(t)}(x_t)$。这应该也是工程技巧，下一步，计算direction pointing to x_t，

```
# 6. compute "direction pointing to x_t" of formula (12) from https://arxiv.org/pdf/2010.02502.pdf
pred_sample_direction = (1 - alpha_prod_t_prev - std_dev_t**2) ** (0.5) * model_output
```

这里的公式表达是：$x_{\text{direction pointing},t} = \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\,\epsilon_{pred}$，和论文相同。然后先计算前两项的和：

t

公式表示就是：$x_{prev} = \sqrt{\bar{\alpha}}_{t-1} x_{0,pred} + x_{\text{direction pointing},t}$。然后就是计随机项：

```
variance_noise = torch.randn(model_output.shape, generator=generator, device=device, dtype=model_output.dtype)
variance = self._get_variance(timestep, prev_timestep) ** (0.5) * eta * variance_noise
```

这里代码和diffusers代码略微不同，进行了简化，公式表达为：$x_{noise} = \sigma_t \epsilon_t$，最后求和，结果为：

```
prev_sample = prev_sample + variance
```

公式表达为：$x_{t-1} = x_{prev} + x_{noise}$,和原文公式12是相同的。完成从$[T_t, T_{t-1}, ..., T_0]$的迭代循环后，得到$x_0$,然后通过归一化操作得到最后的输出：

```
image = (image / 2 + 0.5).clamp(0, 1)
image = image.cpu().permute(0, 2, 3, 1).numpy()
```

至此，demo与DDIM原文公式匹配结束。

ANALYTIC-DPM: AN ANALYTIC ESTIMATE OF THEOPTIMAL REVERSE VARIANCE IN DIFFUSION PROB-ABILISTIC MODELS [ICLR2021]

这篇文章是基于DDIM继续深入的一篇文章，对DDIM所使用的σ_t确定了修正方差的上确界，推导并不复杂，最后得出的结果是：

$$\bar{\sigma}_t^2 = \frac{\bar{\beta}_t^2}{\bar{\alpha}_t^2}\left(1 - \frac{1}{d}\mathbb{E}_{\boldsymbol{x}_t \sim p(\boldsymbol{x}_t)}\left[\|\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\|^2\right]\right) \le \frac{\bar{\beta}_t^2}{\bar{\alpha}_t^2}$$

DEMO介绍：

这里采用苏神的代码进行介绍，地址在：https://github.com/bojone/Keras-DDPM/blob/main/adpm.py 中,核心是复现如下采样形式，首先展示最优均值和方差：

$$\tilde{\boldsymbol{\mu}}_n(\boldsymbol{x}_n, \boldsymbol{x}_0) = \sqrt{\overline{\alpha}_{n-1}}\boldsymbol{x}_0 + \sqrt{\overline{\beta}_{n-1} - \lambda_n^2} \cdot \frac{\boldsymbol{x}_n - \sqrt{\overline{\alpha}_n}\boldsymbol{x}_0}{\sqrt{\overline{\beta}_n}}.$$

$$\boldsymbol{\mu}_n^*(\boldsymbol{x}_n) = \tilde{\boldsymbol{\mu}}_n\left(\boldsymbol{x}_n, \frac{1}{\sqrt{\overline{\alpha}_n}}(\boldsymbol{x}_n + \overline{\beta}_n \nabla_{\boldsymbol{x}_n} \log q_n(\boldsymbol{x}_n))\right),$$

$$\hat{\sigma}^2_{\tau_{k-1}|\tau_k} = \lambda^2_{\tau_{k-1}|\tau_k} + \left(\sqrt{\frac{\overline{\beta}_{\tau_k}}{\alpha_{\tau_k|\tau_{k-1}}}} - \sqrt{\overline{\beta}_{\tau_{k-1}} - \lambda^2_{\tau_{k-1}|\tau_k}}\right)^2 (1 - \overline{\beta}_{\tau_k}\Gamma_{\tau_k}),$$

而采样公式为：$q_\lambda(\boldsymbol{x}_{n-1}|\boldsymbol{x}_n, \boldsymbol{x}_0) = \mathcal{N}(\boldsymbol{x}_{n-1}|\widetilde{\boldsymbol{\mu}}_n, \widehat{\boldsymbol{\sigma}}^2\mathbf{I})$

这里和苏神的结果略有不同，其中$\lambda^2_{\tau_{k-1}|k}$和DDIM中的σ_t含义相同。

执行语句为：

```
sample('test.png', n=8, stride=100, eta=1)
```

首先是一系列准备工作，定义一系列变量等：

```
T = 1000
alpha = np.sqrt(1 - 0.02 * np.arange(1, T + 1) / T)
beta = np.sqrt(1 - alpha**2)
bar_alpha = np.cumprod(alpha)
bar_beta = np.sqrt(1 - bar_alpha**2)
sigma = beta.copy()
# sigma *= np.pad(bar_beta[:-1], [1, 0]) / bar_beta
```

alpha = α_t,beta = β_t,bar_alpha = $\overline{\alpha_t}$,bar_beta = $\sqrt{1 - \overline{\alpha_t}^2}$,然后是Analytic-DPM的变量定义：

```
def sample(path=None, n=4, z_samples=None, stride=1, eta=1):
    """随机采样函数
    注：eta控制方差的相对大小；stride空间跳跃
    """
```

```
# 采样参数
bar_alpha_ = bar_alpha[::stride]
bar_alpha_pre_ = np.pad(bar_alpha_[:-1], [1, 0], constant_values=1)
bar_beta_ = np.sqrt(1 - bar_alpha_**2)
bar_beta_pre_ = np.sqrt(1 - bar_alpha_pre_**2)
alpha_ = bar_alpha_ / bar_alpha_pre_
sigma_ = bar_beta_pre_ / bar_beta_ * np.sqrt(1 - alpha_**2) * eta
epsilon_ = bar_beta_ - alpha_ * np.sqrt(bar_beta_pre_**2 - sigma_**2)
gamma_ = epsilon_ * bar_alpha_pre_ / bar_alpha_  # 增加代码
sigma_ = np.sqrt(sigma_**2 + gamma_**2 * factors[::stride])  # 增加代码
T_ = len(bar_alpha_)
```

这个代码是在DDIM上面改的，因此存在非马尔可夫链以及跳步采样，而stride正是为了控制这个，第7行求前一步的alpha并左右补上0和1，第8行和第9行求出对应的$\overline{\alpha}_t,\overline{\alpha}_{t-1}$序列的$\overline{\beta}_t,\overline{\beta}_{t-1}$,第10行和第11行是原来DDIM的代码，即：$\sigma_t=\eta\frac{\overline{\beta}_{t-1}}{\overline{\beta}_t}\sqrt{1-(\overline{\alpha}_t/\overline{\alpha}_{t-1})^2}$, $\epsilon=\overline{\beta}_t-(\overline{\alpha}_t/\overline{\alpha}_{t-1})\sqrt{1-\overline{\alpha}_{t-1}^2-\sigma_t^2}$,第12和13行就是增加的代码，$\gamma_t=\epsilon\frac{\overline{\alpha}_{t-1}}{\overline{\alpha}_t}=\frac{\overline{\alpha}_{t-1}}{\overline{\alpha}_t}\overline{\beta}_t-\sqrt{1-\overline{\alpha}_{t-1}^2-\sigma_t^2}$, $\sigma_t=\sqrt{\sigma_t^2+\gamma_t^2\text{factors}}$，这里的factors之后会解释。下一步就是生成采样生成，代码如下：

```
# 采样过程
if z_samples is None:
    z_samples = np.random.randn(n**2, img_size, img_size, 3)
else:
    z_samples = z_samples.copy()
for t in tqdm(range(T_), ncols=0):
    t = T_ - t - 1
    bt = np.array([[t * stride]] * z_samples.shape[0])
    z_samples -= epsilon_[t] * model.predict([z_samples, bt])
    z_samples /= alpha_[t]
    z_samples += np.random.randn(*z_samples.shape) * sigma_[t]
x_samples = np.clip(z_samples, -1, 1)
if path is None:
    return x_samples
figure = np.zeros((img_size * n, img_size * n, 3))
for i in range(n):
    for j in range(n):
        digit = x_samples[i * n + j]
        figure[i * img_size:(i + 1) * img_size,
               j * img_size:(j + 1) * img_size] = digit
imwrite(path, figure)
```

核心就是6–11行，第7行求出真实的总共采样次数，第8行求出真实的t在没有跳步的马尔可夫中的位置，第9行在计算：

$$x_{t-1}=x_t-\left[\overline{\beta}_t-(\overline{\alpha}_t/\overline{\alpha}_{t-1})\sqrt{1-\overline{\alpha}_{t-1}-\sigma_t^2}\right]\epsilon_\theta*(x_t)$$,第10行计算：

$$x_{t-1}=(\overline{\alpha}_{t-1}/\overline{\alpha}_t)x_t-\left[\frac{\overline{\alpha}_{t-1}}{\overline{\alpha}_t}\overline{\beta}_t-\sqrt{1-\overline{\alpha}_{t-1}-\sigma_t^2}\right]\epsilon_\theta*(x_t)$$,第11行计算：

$$x_{t-1}=(\overline{\alpha}_{t-1}/\overline{\alpha}_t)x_t-\left[\frac{\overline{\alpha}_{t-1}}{\overline{\alpha}_t}\overline{\beta}_t-\sqrt{1-\overline{\alpha}_{t-1}-\sigma_t^2}\right]\epsilon_\theta*(x_t)+\left[\sqrt{\sigma_t^2+\gamma_t^2\text{factors}}\right]\epsilon_t$$

和DDIM相比，变化来自于3两项，这说明前人的工作在最优均值这一项是正确的，而错误则在最优方差，即第三项，为：

$$\left[\sqrt{\eta\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}[1-(\bar{\alpha}_t/\bar{\alpha}_{t-1})^2]+\gamma_t^2\text{factors}}\right]$$

$$=>\left[\sqrt{\eta\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}[1-(\bar{\alpha}_t/\bar{\alpha}_{t-1})^2]+\left(\frac{\bar{\alpha}_{t-1}}{\bar{\alpha}_t}\bar{\beta}_t-\sqrt{1-\bar{\alpha}_{t-1}^2-\sigma_t^2}\right)^2\text{factors}}\right]$$

$$=>\left[\sqrt{\eta\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}[1-(\bar{\alpha}_t/\bar{\alpha}_{t-1})^2]+\left(\frac{\bar{\beta}_t}{\alpha_t}-\sqrt{\bar{\beta}_{t-1}^2-\sigma_t^2}\right)^2\text{factors}}\right]$$

$$=>\left[\sqrt{\sigma_t^2+\left(\frac{\bar{\beta}_t}{\alpha_t}-\sqrt{\bar{\beta}_{t-1}^2-\sigma_t^2}\right)^2\text{factors}}\right]$$

和原文一模一样，除了factors，事实上这个需要在训练集上进行蒙特卡洛采样才行，factor被定义在如下：

```
factors = [(model.predict(data_generator(t), steps=5)**2).mean()
           for t in tqdm(range(T), ncols=0)]  # 用(batch_size * steps)个样本去估计方差修正项
factors = np.clip(1 – np.array(factors), 0, 1)
```

如同原文公式所展示的：

$$\hat{\sigma}^2_{\tau_{k-1}|\tau_k}=\lambda^2_{\tau_{k-1}|\tau_k}+\left(\sqrt{\frac{\bar{\beta}_{\tau_k}}{\alpha_{\tau_k|\tau_{k-1}}}}-\sqrt{\bar{\beta}_{\tau_{k-1}}-\lambda^2_{\tau_{k-1}|\tau_k}}\right)^2(1-\bar{\beta}_{\tau_k}\Gamma_{\tau_k}),$$

这个factors就是$1-\frac{1}{d}\mathbb{E}_{x_t\sim p(x_t)}[||\epsilon_\theta(x_t,t)||^2]$，即$1-\bar{\beta}_{\tau_k}\Gamma_{\tau_k}$，代入上面的公式中，就是原文估计的新$\sigma_t$,从而完成修正，之所以这边没有了$\bar{\beta}_{\tau_k}$是因为score function除去了$-\bar{\beta}_{\tau_k}$.

DIFFUSION MODELS BEATS GANS ON IMAGE SYNTHESIS [NIPS2021]

Algorithm 1 Classifier guided diffusion sampling, given a diffusion model $(\mu_\theta(x_t), \Sigma_\theta(x_t))$, classifier $p_\phi(y|x_t)$, and gradient scale s.

Input: class label y, gradient scale s
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
for all t from T to 1 **do**
  $\mu, \Sigma \leftarrow \mu_\theta(x_t), \Sigma_\theta(x_t)$
  $x_{t-1} \leftarrow$ sample from $\mathcal{N}(\mu + s\Sigma\, \nabla_{x_t} \log p_\phi(y|x_t), \Sigma)$
end for
return x_0

Algorithm 2 Classifier guided DDIM sampling, given a diffusion model $\epsilon_\theta(x_t)$, classifier $p_\phi(y|x_t)$, and gradient scale s.

Input: class label y, gradient scale s
$x_T \leftarrow$ sample from $\mathcal{N}(0, \mathbf{I})$
for all t from T to 1 **do**
  $\hat{\epsilon} \leftarrow \epsilon_\theta(x_t) - \sqrt{1-\bar{\alpha}_t}\, \nabla_{x_t} \log p_\phi(y|x_t)$
  $x_{t-1} \leftarrow \sqrt{\bar{\alpha}_{t-1}} \left(\frac{x_t - \sqrt{1-\bar{\alpha}_t}\hat{\epsilon}}{\sqrt{\bar{\alpha}_t}} \right) + \sqrt{1-\bar{\alpha}_{t-1}}\hat{\epsilon}$
end for
return x_0

DEMO介绍：

该代码在：https://github.com/openai/guided-diffusion

TRAIN CLASSIFIER:

个人认为guided-based diffusion model的核心在于，训练一个classifier，然后利用pre-trained的diffusion model预训练权重和这个classifier来生成指定格式的图片。

首先是训练一个classifier,代码在：https://github.com/openai/guided-diffusion/blob/main/scripts/classifier_train.py 中，核心是训练：$p_\phi(y|x_t)$,为了简洁地解析代码，我将会省略一些工程细节，首先这个script将会调用main函数，首先定义diffusion模型:

```
model, diffusion = create_classifier_and_diffusion(
    **args_to_dict(args, classifier_and_diffusion_defaults().keys())
)
```

这里是同时定义了分类器和diffusion model,其中model就是分类器，而diffusion就是diffusion model，classifier的实例化如下：

CLASSIFIER：

```
classifier = create_classifier(
    image_size,
    classifier_use_fp16,
    classifier_width,
    classifier_depth,
    classifier_attention_resolutions,
    classifier_use_scale_shift_norm,
```

```
        classifier_resblock_updown,
        classifier_pool,
    )
```

而这个函数定义为：

```
def create_classifier(
    image_size,
    classifier_use_fp16,
    classifier_width,
    classifier_depth,
    classifier_attention_resolutions,
    classifier_use_scale_shift_norm,
    classifier_resblock_updown,
    classifier_pool,
):
    if image_size == 512:
        channel_mult = (0.5, 1, 1, 2, 2, 4, 4)
    elif image_size == 256:
        channel_mult = (1, 1, 2, 2, 4, 4)
    elif image_size == 128:
        channel_mult = (1, 1, 2, 3, 4)
    elif image_size == 64:
        channel_mult = (1, 2, 3, 4)
    else:
        raise ValueError(f"unsupported image size: {image_size}")

    attention_ds = []
    for res in classifier_attention_resolutions.split(","):
        attention_ds.append(image_size // int(res))

    return EncoderUNetModel(
        image_size=image_size,
        in_channels=3,
        model_channels=classifier_width,
        out_channels=1000,
        num_res_blocks=classifier_depth,
        attention_resolutions=tuple(attention_ds),
        channel_mult=channel_mult,
        use_fp16=classifier_use_fp16,
        num_head_channels=64,
        use_scale_shift_norm=classifier_use_scale_shift_norm,
        resblock_updown=classifier_resblock_updown,
        pool=classifier_pool,
    )
```

这个class的定义非常长，直接看forward，如下：

```
    def forward(self, x, timesteps):
        """
        Apply the model to an input batch.
        :param x: an [N x C x ...] Tensor of inputs.
        :param timesteps: a 1-D batch of timesteps.
        :return: an [N x K] Tensor of outputs.
        """
        emb = self.time_embed(timestep_embedding(timesteps, self.model_channels))
```

```
        results = []
        h = x.type(self.dtype)
        for module in self.input_blocks:
            h = module(h, emb)
            if self.pool.startswith("spatial"):
                results.append(h.type(x.dtype).mean(dim=(2, 3)))
        h = self.middle_block(h, emb)
        if self.pool.startswith("spatial"):
            results.append(h.type(x.dtype).mean(dim=(2, 3)))
            h = th.cat(results, axis=-1)
            return self.out(h)
        else:
            h = h.type(x.dtype)
            return self.out(h)
```

所以是用self.out完成了分类，里面有depth-wise的线性层完成了分类任务。因此这个模型其实是一个UNet+classifier的组合。

SAMPLER：

组建采样scheduler，实例化在main函数内：

```
    if args.noised:
        schedule_sampler = create_named_schedule_sampler(
            args.schedule_sampler, diffusion
        )
```

这个玩意可以不用关注，这个就是借鉴了DDIM,DDPM,SDE的scheduler,直接下一个。

TRAINER:

trainer是核心所在，定义了训练常用框架的方法，实例化依旧在main函数中：

```
    mp_trainer = MixedPrecisionTrainer(
        model=model, use_fp16=args.classifier_use_fp16, initial_lg_loss_scale=16.0
    )
```

其定义为：

```
class MixedPrecisionTrainer:
    def __init__(
        self,
        *,
        model,
        use_fp16=False,
        fp16_scale_growth=1e-3,
        initial_lg_loss_scale=INITIAL_LOG_LOSS_SCALE,
    ):
        self.model = model
        self.use_fp16 = use_fp16
        self.fp16_scale_growth = fp16_scale_growth

        self.model_params = list(self.model.parameters())
        self.master_params = self.model_params
        self.param_groups_and_shapes = None
        self.lg_loss_scale = initial_lg_loss_scale
```

```
    if self.use_fp16:
        self.param_groups_and_shapes = get_param_groups_and_shapes(
            self.model.named_parameters()
        )
        self.master_params = make_master_params(self.param_groups_and_shapes)
        self.model.convert_to_fp16()
```

这个就说trainer的定义，首先self.model_params则是取出来classifier的参数list，self.model_params和self.master_params则是为了实现EMA，这个trainer的成员函数其实是一些基本torch训练框架的组件，因此过一下就行，如下：

```
def zero_grad(self):
    zero_grad(self.model_params)

def backward(self, loss: th.Tensor):
    if self.use_fp16:
        loss_scale = 2 ** self.lg_loss_scale
        (loss * loss_scale).backward()
    else:
        loss.backward()

def optimize(self, opt: th.optim.Optimizer):
    if self.use_fp16:
        return self._optimize_fp16(opt)
    else:
        return self._optimize_normal(opt)

def _optimize_fp16(self, opt: th.optim.Optimizer):
    logger.logkv_mean(”lg_loss_scale”, self.lg_loss_scale)
    model_grads_to_master_grads(self.param_groups_and_shapes, self.master_params)
    grad_norm, param_norm = self._compute_norms(grad_scale=2 ** self.lg_loss_scale)
    if check_overflow(grad_norm):
        self.lg_loss_scale -= 1
        logger.log(f”Found NaN, decreased lg_loss_scale to {self.lg_loss_scale}”)
        zero_master_grads(self.master_params)
        return False

    logger.logkv_mean(”grad_norm”, grad_norm)
    logger.logkv_mean(”param_norm”, param_norm)

    for p in self.master_params:
        p.grad.mul_(1.0 / (2 ** self.lg_loss_scale))
    opt.step()
    zero_master_grads(self.master_params)
    master_params_to_model_params(self.param_groups_and_shapes, self.master_params)
    self.lg_loss_scale += self.fp16_scale_growth
    return True

def _optimize_normal(self, opt: th.optim.Optimizer):
    grad_norm, param_norm = self._compute_norms()
    logger.logkv_mean(”grad_norm”, grad_norm)
    logger.logkv_mean(”param_norm”, param_norm)
    opt.step()
    return True

def _compute_norms(self, grad_scale=1.0):
    grad_norm = 0.0
    param_norm = 0.0
```

```
        for p in self.master_params:
            with th.no_grad():
                param_norm += th.norm(p, p=2, dtype=th.float32).item() ** 2
                if p.grad is not None:
                    grad_norm += th.norm(p.grad, p=2, dtype=th.float32).item() ** 2
        return np.sqrt(grad_norm) / grad_scale, np.sqrt(param_norm)

    def master_params_to_state_dict(self, master_params):
        return master_params_to_state_dict(
            self.model, self.param_groups_and_shapes, master_params, self.use_fp16
        )

    def state_dict_to_master_params(self, state_dict):
        return state_dict_to_master_params(self.model, state_dict, self.use_fp16)
```

OPTIMIZER AND DATASET:

定义优化器和dataset，如下，不是特别关键，跳过：

```
data = load_data(
    data_dir=args.data_dir,
    batch_size=args.batch_size,
    image_size=args.image_size,
    class_cond=True,
    random_crop=True,
)
if args.val_data_dir:
    val_data = load_data(
        data_dir=args.val_data_dir,
        batch_size=args.batch_size,
        image_size=args.image_size,
        class_cond=True,
    )
else:
    val_data = None

logger.log(f"creating optimizer...")
opt = AdamW(mp_trainer.master_params, lr=args.lr, weight_decay=args.weight_decay)
```

TRAIN：

训练是一个迭代过程，如下：

```
for step in range(args.iterations - resume_step):
    logger.logkv("step", step + resume_step)
    logger.logkv(
        "samples",
        (step + resume_step + 1) * args.batch_size * dist.get_world_size(),
    )
    if args.anneal_lr:
        set_annealed_lr(opt, args.lr, (step + resume_step) / args.iterations)
    forward_backward_log(data)
```

可以看到，它forward和backward的核心都在forward_backward_log函数中，其定义如下：

```
def forward_backward_log(data_loader, prefix=”train”):
    batch, extra = next(data_loader)
    labels = extra[”y”].to(dist_util.dev())

    batch = batch.to(dist_util.dev())
    # Noisy images
    if args.noised:
        t, _ = schedule_sampler.sample(batch.shape[0], dist_util.dev())
        batch = diffusion.q_sample(batch, t)
    else:
        t = th.zeros(batch.shape[0], dtype=th.long, device=dist_util.dev())

    for i, (sub_batch, sub_labels, sub_t) in enumerate(
        split_microbatches(args.microbatch, batch, labels, t)
    ):
        logits = model(sub_batch, timesteps=sub_t)
        loss = F.cross_entropy(logits, sub_labels, reduction=”none”)

        losses = {}
        losses[f”{prefix}_loss”] = loss.detach()
        losses[f”{prefix}_acc@1”] = compute_top_k(
            logits, sub_labels, k=1, reduction=”none”
        )
        losses[f”{prefix}_acc@5”] = compute_top_k(
            logits, sub_labels, k=5, reduction=”none”
        )
        log_loss_dict(diffusion, sub_t, losses)
        del losses
        loss = loss.mean()
        if loss.requires_grad:
            if i == 0:
                mp_trainer.zero_grad()
            mp_trainer.backward(loss * len(sub_batch) / len(batch))
```

首先是7-15行，这些代码实际上就是通过x_0去获得x_t,其中batch一开始是x_0,而diffusion.q_sample来完成这一目的，q_sample定义如下：

```
def q_sample(self, x_start, t, noise=None):
    ”””
    Diffuse the data for a given number of diffusion steps.
    In other words, sample from q(x_t | x_0).
    :param x_start: the initial data batch.
    :param t: the number of diffusion steps (minus 1). Here, 0 means one step.
    :param noise: if specified, the split-out normal noise.
    :return: A noisy version of x_start.
    ”””
    if noise is None:
        noise = th.randn_like(x_start)
    assert noise.shape == x_start.shape
    return (
        _extract_into_tensor(self.sqrt_alphas_cumprod, t, x_start.shape) * x_start
        + _extract_into_tensor(self.sqrt_one_minus_alphas_cumprod, t, x_start.shape)
        * noise
    )
```

那么在forward_backward_log函数中，第16-17行通过简单的交叉熵计算loss，然后最后反向传播，在main函数中，使用如下代码完成参数更新：

```
mp_trainer.optimize(opt)
```

GUIDED-BASED SAMPLE：

PRE-LOAD:

训练完一个classifier后，我们就需要完成采样，script在https://github.com/openai/guided-diffusion/blob/main/scripts/classifier_sample.py 中，这里不得不吐槽一下，openai的代码写的是真的好，baofan那个代码看的我头疼，当然最后还是找到了他的core code,不过还是苏神写得好。首先是main函数实例化diffusion model和classifier，如下:

```
model, diffusion = create_model_and_diffusion(
    **args_to_dict(args, model_and_diffusion_defaults().keys())
)
model.load_state_dict(
    dist_util.load_state_dict(args.model_path, map_location="cpu")
)
model.to(dist_util.dev())
if args.use_fp16:
    model.convert_to_fp16()
model.eval()

logger.log("loading classifier...")
classifier = create_classifier(**args_to_dict(args, classifier_defaults().keys()))
classifier.load_state_dict(
    dist_util.load_state_dict(args.classifier_path, map_location="cpu")
)
classifier.to(dist_util.dev())
if args.classifier_use_fp16:
    classifier.convert_to_fp16()
classifier.eval()
```

注意这里是create_model_and_diffusion，而不是create_classifier_and_diffusion,因此这里的model是没有分类层的UNet，而classifier才是有分类层的UNet。

SAMPLING:

sampling的核心代码就如下几行：

```
while len(all_images) * args.batch_size < args.num_samples:
    model_kwargs = {}
    classes = th.randint(
        low=0, high=NUM_CLASSES, size=(args.batch_size,), device=dist_util.dev()
    )
    model_kwargs["y"] = classes
    sample_fn = (
        diffusion.p_sample_loop if not args.use_ddim else diffusion.ddim_sample_loop
    )
    def cond_fn(x, t, y=None):
        assert y is not None
        with th.enable_grad():
            x_in = x.detach().requires_grad_(True)
            logits = classifier(x_in, t)
            log_probs = F.log_softmax(logits, dim=-1)
            selected = log_probs[range(len(logits)), y.view(-1)]
            return th.autograd.grad(selected.sum(), x_in)[0] * args.classifier_scale
    def model_fn(x, t, y=None):
```

```
        assert y is not None
        return model(x, t, y if args.class_cond else None)

    sample = sample_fn(
        model_fn,
        (args.batch_size, 3, args.image_size, args.image_size),
        clip_denoised=args.clip_denoised,
        model_kwargs=model_kwargs,
        cond_fn=cond_fn,
        device=dist_util.dev(),
    )
    sample = ((sample + 1) * 127.5).clamp(0, 255).to(th.uint8)
    sample = sample.permute(0, 2, 3, 1)
    sample = sample.contiguous
```

除去循环迭代和归一化操作，核心其实是sample_fn的call function，这里用diffusion.p_sample_loop进行讲解：

```
def p_sample_loop(
    self,
    model,
    shape,
    noise=None,
    clip_denoised=True,
    denoised_fn=None,
    cond_fn=None,
    model_kwargs=None,
    device=None,
    progress=False,
):
    final = None
    for sample in self.p_sample_loop_progressive(
        model,
        shape,
        noise=noise,
        clip_denoised=clip_denoised,
        denoised_fn=denoised_fn,
        cond_fn=cond_fn,
        model_kwargs=model_kwargs,
        device=device,
        progress=progress,
    ):
        final = sample
    return final["sample"]
```

有点懵逼，继续看self.p_sample_loop_progressive方法，如下：

```
def p_sample_loop_progressive(
    self,
    model,
    shape,
    noise=None,
    clip_denoised=True,
    denoised_fn=None,
    cond_fn=None,
    model_kwargs=None,
    device=None,
```

```
        progress=False,
    ):
        """
        Generate samples from the model and yield intermediate samples from
        each timestep of diffusion.
        Arguments are the same as p_sample_loop().
        Returns a generator over dicts, where each dict is the return value of
        p_sample().
        """
        if device is None:
            device = next(model.parameters()).device
        assert isinstance(shape, (tuple, list))
        if noise is not None:
            img = noise
        else:
            img = th.randn(*shape, device=device)
        indices = list(range(self.num_timesteps))[::-1]

        if progress:
            # Lazy import so that we don't depend on tqdm.
            from tqdm.auto import tqdm

            indices = tqdm(indices)

        for i in indices:
            t = th.tensor([i] * shape[0], device=device)
            with th.no_grad():
                out = self.p_sample(
                    model,
                    img,
                    t,
                    clip_denoised=clip_denoised,
                    denoised_fn=denoised_fn,
                    cond_fn=cond_fn,
                    model_kwargs=model_kwargs,
                )
                yield out
                img = out["sample"]
```

所以需要继续调用self.p_sample,代码为如下：

```
        out = self.p_mean_variance(
            model,
            x,
            t,
            clip_denoised=clip_denoised,
            denoised_fn=denoised_fn,
            model_kwargs=model_kwargs,
        )
        noise = th.randn_like(x)
        nonzero_mask = (
            (t != 0).float().view(-1, *([1] * (len(x.shape) - 1)))
        )  # no noise when t == 0
        if cond_fn is not None:
            out["mean"] = self.condition_mean(
                cond_fn, out, x, t, model_kwargs=model_kwargs
            )
```

```
        sample = out["mean"] + nonzero_mask * th.exp(0.5 * out["log_variance"]) * noise
        return {"sample": sample, "pred_xstart": out["pred_xstart"]}
```

到这里已经很明显了，model就是main函数中的内嵌函数model_fn,而cond_fn则就是main函数中的内嵌函数cond_fn,因此noise来自于$\mathcal{N}(0, \mathbf{I})$,而nonzero_mask保证0时刻没有噪声，out["mean"]便是求论文中的$\mu + s\sum\nabla_{x_t}\log p_\psi(y|x_t)$,代码如下：

```
    def condition_mean(self, cond_fn, p_mean_var, x, t, model_kwargs=None):
        gradient = cond_fn(x, self._scale_timesteps(t), **model_kwargs)
        new_mean = (
            p_mean_var["mean"].float() + p_mean_var["variance"] * gradient.float()
        )
        return new_mean
```

gradient则是如下代码：

```
            with th.enable_grad():
                x_in = x.detach().requires_grad_(True)
                logits = classifier(x_in, t)
                log_probs = F.log_softmax(logits, dim=-1)
                selected = log_probs[range(len(logits)), y.view(-1)]
                return th.autograd.grad(selected.sum(), x_in)[0] * args.classifier_scale
```

很明显和$s\nabla_{x_t}\log p_\psi(y|x_t)$一模一样，因此new_mean就是$\mu + s\sum\nabla_{x_t}\log p_\psi(y|x_t)$,最后在一下行中加入噪声项，得到结果$\mu + s\sum\nabla_{x_t}\log p_\psi(y|x_t) + \sqrt{\sum}\epsilon$,和论文相同：

```
sample = out["mean"] + nonzero_mask * th.exp(0.5 * out["log_variance"]) * noise
```

DPM-SOLVER A FASTODE SOLVER FOR DIFFUSION PROBABILISTIC MODEL SAMPLING IN AROUND 10 STEPS [NIPS2022]

DEMO介绍：

luchen的代码仓库在：https://github.com/LuChengTHU/dpm-solver，但对应bring dpm solver to you code，核心的代码文件是：https://github.com/LuChengTHU/dpm-solver/blob/main/dpm_solver_pytorch.py。而这里我讲解的demo在：https://github.com/shaoshitong/diffusion-model-learning/blob/main/demo/uncond_dpm_solver_demo.py

论文中展示的DPM SOLVER算法，如下：

DPM-Solver-1. Given an initial value $\boldsymbol{x}_T$ and $M+1$ time steps $\{t_i\}_{i=0}^{M}$ decreasing from $t_0 = T$ to $t_M = 0$. Starting with $\tilde{\boldsymbol{x}}_{t_0} = \boldsymbol{x}_T$, the sequence $\{\tilde{\boldsymbol{x}}_{t_i}\}_{i=1}^{M}$ is computed iteratively as follows:

$$\tilde{\boldsymbol{x}}_{t_i} = \frac{\alpha_{t_i}}{\alpha_{t_{i-1}}}\tilde{\boldsymbol{x}}_{t_{i-1}} - \sigma_{t_i}(e^{h_i}-1)\boldsymbol{\epsilon}_\theta(\tilde{\boldsymbol{x}}_{t_{i-1}}, t_{i-1}), \quad \text{where } h_i = \lambda_{t_i} - \lambda_{t_{i-1}}. \tag{3.7}$$

For $k \geq 2$, approximating the first k terms of the Taylor expansion needs additional intermediate points between t and s [31]. The derivation is more technical so we defer it to Appendix B. Below we propose algorithms for $k = 2, 3$ and name them as *DPM-Solver-2* and *DPM-Solver-3*, respectively.

Algorithm 1 DPM-Solver-2.

Require: initial value $\boldsymbol{x}_T$, time steps $\{t_i\}_{i=0}^{M}$, model $\boldsymbol{\epsilon}_\theta$

$\tilde{\boldsymbol{x}}_{t_0} \leftarrow \boldsymbol{x}_T$

for $i \leftarrow 1$ to M **do**

$\quad s_i \leftarrow t_\lambda\left(\frac{\lambda_{t_{i-1}}+\lambda_{t_i}}{2}\right)$

$\quad \boldsymbol{u}_i \leftarrow \frac{\alpha_{s_i}}{\alpha_{t_{i-1}}}\tilde{\boldsymbol{x}}_{t_{i-1}} - \sigma_{s_i}\left(e^{\frac{h_i}{2}}-1\right)\boldsymbol{\epsilon}_\theta(\tilde{\boldsymbol{x}}_{t_{i-1}}, t_{i-1})$

$\quad \tilde{\boldsymbol{x}}_{t_i} \leftarrow \frac{\alpha_{t_i}}{\alpha_{t_{i-1}}}\tilde{\boldsymbol{x}}_{t_{i-1}} - \sigma_{t_i}\left(e^{h_i}-1\right)\boldsymbol{\epsilon}_\theta(\boldsymbol{u}_i, s_i)$

end for

return $\tilde{\boldsymbol{x}}_{t_M}$

Algorithm 2 DPM-Solver-3.

Require: initial value $\boldsymbol{x}_T$, time steps $\{t_i\}_{i=0}^{M}$, model $\boldsymbol{\epsilon}_\theta$

$\tilde{\boldsymbol{x}}_{t_0} \leftarrow \boldsymbol{x}_T$, $r_1 \leftarrow \frac{1}{3}$, $r_2 \leftarrow \frac{2}{3}$

for $i \leftarrow 1$ to M **do**

$\quad s_{2i-1} \leftarrow t_\lambda\left(\lambda_{t_{i-1}} + r_1 h_i\right), \quad s_{2i} \leftarrow t_\lambda\left(\lambda_{t_{i-1}} + r_2 h_i\right)$

$\quad \boldsymbol{u}_{2i-1} \leftarrow \frac{\alpha_{s_{2i-1}}}{\alpha_{t_{i-1}}}\tilde{\boldsymbol{x}}_{t_{i-1}} - \sigma_{s_{2i-1}}\left(e^{r_1 h_i}-1\right)\boldsymbol{\epsilon}_\theta(\tilde{\boldsymbol{x}}_{t_{i-1}}, t_{i-1})$

$\quad \boldsymbol{D}_{2i-1} \leftarrow \boldsymbol{\epsilon}_\theta(\boldsymbol{u}_{2i-1}, s_{2i-1}) - \boldsymbol{\epsilon}_\theta(\tilde{\boldsymbol{x}}_{t_{i-1}}, t_{i-1})$

$\quad \boldsymbol{u}_{2i} \leftarrow \frac{\alpha_{s_{2i}}}{\alpha_{t_{i-1}}}\tilde{\boldsymbol{x}}_{t_{i-1}} - \sigma_{s_{2i}}\left(e^{r_2 h_i}-1\right)\boldsymbol{\epsilon}_\theta(\tilde{\boldsymbol{x}}_{t_{i-1}}, t_{i-1}) - \frac{\sigma_{s_{2i}} r_2}{r_1}\left(\frac{e^{r_2 h_i}-1}{r_2 h_i}-1\right)\boldsymbol{D}_{2i-1}$

$\quad \boldsymbol{D}_{2i} \leftarrow \boldsymbol{\epsilon}_\theta(\boldsymbol{u}_{2i}, s_{2i}) - \boldsymbol{\epsilon}_\theta(\tilde{\boldsymbol{x}}_{t_{i-1}}, t_{i-1})$

$\quad \tilde{\boldsymbol{x}}_{t_i} \leftarrow \frac{\alpha_{t_i}}{\alpha_{t_{i-1}}}\tilde{\boldsymbol{x}}_{t_{i-1}} - \sigma_{t_i}\left(e^{h_i}-1\right)\boldsymbol{\epsilon}_\theta(\tilde{\boldsymbol{x}}_{t_{i-1}}, t_{i-1}) - \frac{\sigma_{t_i}}{r_2}\left(\frac{e^{h_i}-1}{h}-1\right)\boldsymbol{D}_{2i}$

end for

return $\tilde{\boldsymbol{x}}_{t_M}$

DDIM和一阶DPM SOLVER等价。

代码文件模块讲解：

首先介绍以下这个代码文件的主要构造，luchen学长真的非常细心，解释的非常到位。因此，首先需要查看类NoiseScheduleVP，解读luchen学长的注解，首先要明白的是DPM solver是支持continuous和discrete两种场景的，而控制这两种场景的方式是设置参数 `schedule` `=` `'discrete'` 来进行。DPM solver论文中提到其为了简化采样是考虑了以下的forward sample场景：

$$q_{t|0}(x_t|x_0) = N(\alpha_t x_0, \sigma_t^2 \mathbf{I})$$

作者论文中提到的reverse sample过程为：

$$dx_t = \left[f(t)x_t - g^2(t)\nabla_x \log q_t(x_t)\right]dt + g(t)dw,$$

根据songyang的ICLR 2021 论文结论，其对应的ODE方程为：

$$dx_t = \left[f(t)x_t - \frac{1}{2}g^2(t)\nabla_x \log q_t(x_t)\right]dt$$,

这是一个一阶非齐次常微分方程，其解可以直接得出：

$$x_t = e^{\int_s^t f(\tau)d\tau}x_s + \int_s^t (e^{\int_\tau^t f(\tau)d\tau}\frac{g^2(\tau)}{2\sigma_\tau}\epsilon_\theta(x_\tau, \tau))d\tau$$,

为了简化这个方程并利用SDE的一些结论，作者定义了：

$$\lambda_t = \log(\alpha_t) - \log(\sigma_t)$$

作者在论文中证明这个函数是一个严格单调的函数，同时在代码层面，这些参数通过如下调用获得：

```
        log_alpha_t = self.marginal_log_mean_coeff(t) # \log(\alpha_t)
        sigma_t = self.marginal_std(t) # \sigma_t
        lambda_t = self.marginal_lambda(t) # \lambda_t
```

λ_t具备一个唯一的逆函数，论文中定义为t_λ,其可以通过如下调用获得：

```
t = self.inverse_lambda(lambda_t)
```

通过转变一阶非齐次常微分方程的解为：

$$x_t = \frac{\alpha_t}{\alpha_s}x_s - \alpha_t\int_{\lambda_s}^{\lambda_t}(e^{\lambda}\hat{\epsilon}_\theta(\hat{x}_\lambda, \lambda))d\lambda$$,

接下来作者在论文中就是利用数值分析的技巧进行近似，暂时不细讲。

luchen特别提醒，离散场景下，代码中时间步为$t_i = (i+1)/N$,此外，$t_0 = 1e-3, \quad t_{N-1} = 1$。该类有一个参数名为alphas_cumprod,这个参数，这个参数的符号在DDPM中定义为$\widehat{\alpha_n}$,DDPM的forward process为：

$$q_{t_n|t_0}(x_{t_n}|x_0) = N(\sqrt{\widehat{\alpha_n}} * x_0, (1-\widehat{\alpha_n}) * \mathbf{I})$$.

在DPM solver中，α_{t_n} 被定义为：

$$\alpha_{t_n} = \sqrt{\widehat{\alpha_n}}, \ \log(\alpha_{t_n}) = 0.5 * \log(\widehat{\alpha_n})$$.

而连续场景下作者没有过多介绍，作者后面给出了example关于如果去使用这个类：

```
# For discrete-time DPMs, given betas (the beta array for n = 0, 1, ..., N - 1):
>>> ns = NoiseScheduleVP('discrete', betas=betas)
# For discrete-time DPMs, given alphas_cumprod (the \hat{alpha_n} array for n = 0, 1, ..., N - 1):
>>> ns = NoiseScheduleVP('discrete', alphas_cumprod=alphas_cumprod)
# For continuous-time DPMs (VPSDE), linear schedule:
>>> ns = NoiseScheduleVP('linear', continuous_beta_0=0.1, continuous_beta_1=20.)
```

然后是类DPM_Solver,是为连续时间扩散ODEs设计的。对于离散时间扩散模型，我们还实现了一个包装函数，在model_wrapper函数中把离散时间扩散模型转换成连续时间扩散模型。这个类的初始化参数如下：

```
class DPM_Solver:
    def __init__(
        self,
        model_fn,
        noise_schedule,
```

```
        algorithm_type="dpmsolver++",
        correcting_x0_fn=None,
        correcting_xt_fn=None,
        thresholding_max_val=1.,
        dynamic_thresholding_ratio=0.995,
    ):
```

这里的algorithm_type可以是dpmsolver和dpmsolver++,这里model_fn和noise_schedule的定义如下：

```
    model_fn: A noise prediction model function which accepts the continuous-time input (t in [epsilon, T]):
        ``
        def model_fn(x, t_continuous):
            return noise
        ``
        The shape of `x` is `(batch_size, **shape)`, and the shape of `t_continuous` is `(batch_size,)`.
    noise_schedule: A noise schedule object, such as NoiseScheduleVP.
```

__init__的初始化如下：

```
        self.model = lambda x, t: model_fn(x, t.expand((x.shape[0])))
        self.noise_schedule = noise_schedule
        assert algorithm_type in ["dpmsolver", "dpmsolver++"]
        self.algorithm_type = algorithm_type
        if correcting_x0_fn == "dynamic_thresholding":
            self.correcting_x0_fn = self.dynamic_thresholding_fn
        else:
            self.correcting_x0_fn = correcting_x0_fn
        self.correcting_xt_fn = correcting_xt_fn
        self.dynamic_thresholding_ratio = dynamic_thresholding_ratio
        self.thresholding_max_val = thresholding_max_val
```

作者提供了一个伪代码来展示调参过程：

```
for algorithm_type in ["dpmsolver", "dpmsolver++"]:
# Optional, for correcting_x0_fn in [None, "dynamic_thresholding"]:
    dpm_solver = DPM_Solver(..., algorithm_type=algorithm_type) # ... means other arguments
    for method in ['singlestep', 'multistep']:
        for order in [2, 3]:
            for steps in [10, 15, 20, 25, 50, 100]:
                sample = dpm_solver.sample(
                    ..., # ... means other arguments
                    method=method,
                    order=order,
                    steps=steps,
                    # optional: skip_type='time_uniform' or 'logSNR' or 'time_quadratic',
                    # optional: denoise_to_zero=True or False
                )
```

主函数就是dpm_solver.sample，由于作者只支持连续场景下ODE的dpm solver，我们将以这个场景进行讲解：

主函数（dpm_solver.sample）：

函数调用接口：

```
    def sample(self, x, steps=20, t_start=None, t_end=None, order=2, skip_type='time_uniform',
        method='multistep', lower_order_final=True, denoise_to_zero=False, solver_type='dpmsolver',
```

```
        atol=0.0078, rtol=0.05, return_intermediate=False,
    ):
```

这里面method参数支持四种形式，详细可见luchen学长的注释：https://github.com/LuChengTHU/dpm-solver/blob/3d31b82a62799e62cbfbb37a109fdef6309f9cd3/dpm_solver_pytorch.py#L1064 ，默认为multistep，解释如下：

```
Multistep DPM-Solver with the order of `order`. The total number of function evaluations (NFE) == `steps`.
We initialize the first `order` values by lower order multistep solvers.
Given a fixed NFE == `steps`, the sampling procedure is:
    Denote K = steps.
    - If `order` == 1:
        - We use K steps of DPM-Solver-1 (i.e. DDIM).
    - If `order` == 2:
        - We firstly use 1 step of DPM-Solver-1, then use (K - 1) step of multistep DP
                M-Solver-2.
    - If `order` == 3:
        - We firstly use 1 step of DPM-Solver-1, then 1 step of multistep DPM-Solver-2
                , then (K - 2) step of multistep DPM-Solver-3.
```

这里只介绍multistep，dpm solver，对于skip_type也有三种，如下：

```
- 'logSNR': uniform logSNR for the time steps. **Recommended for low-resolutional images**
- 'time_uniform': uniform time for the time steps. **Recommended for high-resolutional images**.
- 'time_quadratic': quadratic time for the time steps.
```

默认为time_uniform，这里根据logSNR进行介绍，因为这是论文使用的形式。

sample函数的第一步仍然是设置一系列超参数，首先是x_0, x_t,如下：

```
t_0 = 1. / self.noise_schedule.total_N if t_end is None else t_end
t_T = self.noise_schedule.T if t_start is None else t_start
assert t_0 > 0 and t_T > 0, "Time range needs to be greater than"+\
                                        "0. For discrete-time DPMs, it needs "+\
                                        "to be in [1 / N, 1], where N is the l"+\
                                       "ength of betas array"
```

下面是一个判断语句，如下：

```
with torch.no_grad():
    if method == 'adaptive':
        ...
    elif method == 'multistep':
        ...
    elif method in ['singlestep', 'singlestep_fixed']:
        ...
    else:
        raise ValueError("Got wrong method {}".format(method))
```

这里只介绍multistep分支，首先获取timesteps，如下：

```
timesteps = self.get_time_steps(skip_type=skip_type, t_T=t_T, t_0=t_0, N=steps, device=device)
```

代码如下：

```
if skip_type == 'logSNR':
    lambda_T = self.noise_schedule.marginal_lambda(torch.tensor(t_T).to(device))
```

```
        lambda_0 = self.noise_schedule.marginal_lambda(torch.tensor(t_0).to(device))
        logSNR_steps = torch.linspace(lambda_T.cpu().item(), lambda_0.cpu().item(), N + 1).to(device)
        return self.noise_schedule.inverse_lambda(logSNR_steps)
    elif skip_type == 'time_uniform':
        return torch.linspace(t_T, t_0, N + 1).to(device)
    elif skip_type == 'time_quadratic':
        t_order = 2
        t = torch.linspace(t_T**(1. / t_order), t_0**(1. / t_order), N + 1).pow(t_order).to(device)
        return t
    else:
        raise ValueError
```

计算介绍分支1，self.noise_scheduler.marginal_lambda是一个计算$\lambda_t = \log(\alpha_t) - \log(\sigma_t)$的函数，因此首先求得$\lambda_T$和$\lambda_0$，然后第四句生成序列$[\lambda_T, \lambda_{T-1}, \cdots, \lambda_0]$，最后调用self.noise_schedule.inverse_lambda来通过逆函数求得连续的时间步t,并返回。通过debug，观察到timesteps值为：

```
tensor([1.0000e+00, 9.9039e-01, 9.8069e-01,..., 1.3935e-03, 1.1778e-03, 9.9992e-04],
       device='cuda:0')
```

也就是单调递减，且$\in [1, 0]$。随后，初始化一系列超参数，如下：

```
step = 0
t = timesteps[step]
t_prev_list = [t]
model_prev_list = [self.model_fn(x, t)]
```

这里的x就是一个noise，来源于torch.randn,因此model_prev_list得到的是包含一个经过第一次预测噪声的list。下一步，通过判断语句执行如下代码：

```
if self.correcting_xt_fn is not None:
    x = self.correcting_xt_fn(x, t, step)
```

由于DPM_Solver类一直将其置为None，所以完全不用管，继续。进行采样，如下：

```
for step in range(1, order):
    t = timesteps[step]
    x = self.multistep_dpm_solver_update(x, model_prev_list, t_prev_list, t, step, solver_type=solver_type)
    t_prev_list.append(t)
    model_prev_list.append(self.model_fn(x, t))
```

这里的order就是阶数，可以看出来计算高阶采用的是一个迭代运算，首先step为1，那么t会取出timesteps[1],然后调用self.multistep_dpm_solver_update来计算得到该阶上的下一个x。查看self.multistep_dpm_solver_update代码，如下：

```
    def multistep_dpm_solver_update(self, x, model_prev_list, t_prev_list, t, order, solver_type='dpmsolver'):
        if order == 1:
            return self.dpm_solver_first_update(x, t_prev_list[-1], t, model_s=model_prev_list[-1])
        elif order == 2:
            return self.multistep_dpm_solver_second_update(x, model_prev_list, t_prev_list, t, solver_type=solver_type)
        elif order == 3:
            return self.multistep_dpm_solver_third_update(x, model_prev_list, t_prev_list, t, solver_type=solver_type)
        else:
            raise ValueError("Solver order must be 1 or 2 or 3, got {}".format(order))
```

不同阶采用了不同操作，这里先将其视为黑盒，在后面讲解三种阶的近似计算，每次计算完成指定阶后，t_prev_list会加入t而model_prev_list则会加入noise的预测。

可以发现，上面的计算是计算了第一步的dpm solver估计，那么接下来将完成全部采样，如下：

```
for step in range(order, steps + 1):
    t = timesteps[step]
    # We only use lower order for steps < 10
    if lower_order_final and steps < 10:
        step_order = min(order, steps + 1 - step)
    else:
        step_order = order
    x = self.multistep_dpm_solver_update(x, model_prev_list, t_prev_list, t, step_order, solver_type=solver_type)
    for i in range(order - 1):
        t_prev_list[i] = t_prev_list[i + 1]
        model_prev_list[i] = model_prev_list[i + 1]
    t_prev_list[-1] = t
    # We do not need to evaluate the final model value.
    if step < steps:
        model_prev_list[-1] = self.model_fn(x, t)
```

这里的steps是timesteps.shape[0]–1，也就是区间数，大部分情况，steps不会小于10，且steps+1–step>order是常见的，因此只需要考虑step_order = order这种情况。然后对x进行更新，同时，将2或者3阶情况下的t_prev_list和model_prev_list最后1或者2个值前移1步。更新model_prev_list的最后一个值为新t下的noise。

这一部分代码其实可以看出，作者将t_prev_list和model_prev_list作为了buffer，来计算不同阶的dpm solver，我们可以通过debug来观察这个buffer的变化，先查看steps为11，order为1的情景(我们在 `for step in range(order, steps + 1):` 这一行后加入 `print(step,t_prev_list,[i.norm() for i in model_prev_list])` ）：

```
1 [tensor(1., device='cuda:0')] [tensor(222.0860, device='cuda:0')]
2 [tensor(0.9089, device='cuda:0')] [tensor(222.6178, device='cuda:0')]
3 [tensor(0.8076, device='cuda:0')] [tensor(222.9460, device='cuda:0')]
4 [tensor(0.6920, device='cuda:0')] [tensor(223.0778, device='cuda:0')]
5 [tensor(0.5554, device='cuda:0')] [tensor(223.0060, device='cuda:0')]
6 [tensor(0.3916, device='cuda:0')] [tensor(222.6469, device='cuda:0')]
7 [tensor(0.2201, device='cuda:0')] [tensor(220.4799, device='cuda:0')]
8 [tensor(0.0991, device='cuda:0')] [tensor(215.7282, device='cuda:0')]
9 [tensor(0.0397, device='cuda:0')] [tensor(209.2391, device='cuda:0')]
10 [tensor(0.0144, device='cuda:0')] [tensor(199.0343, device='cuda:0')]
11 [tensor(0.0043, device='cuda:0')] [tensor(183.1858, device='cuda:0')]
```

解析一下，order=1时，其实inital_step 是不会进行的，同样以下语句也不会进行：

```
    for i in range(order - 1):
        t_prev_list[i] = t_prev_list[i + 1]
        model_prev_list[i] = model_prev_list[i + 1]
```

因此，t_prev_list和model_prev_list一直是最新的t和noise。

再查看steps为11，order为2的情景：

```
2 [tensor(1., device='cuda:0'), tensor(0.9089, device='cuda:0')] [tensor(222.8838, device='cuda:0'), tensor(223.5008, device='cuda:0')]
3 [tensor(0.9089, device='cuda:0'), tensor(0.8076, device='cuda:0')] [tensor(223.5008, device='cuda:0'), tensor(223.4914, device='cuda:0')]
4 [tensor(0.8076, device='cuda:0'), tensor(0.6920, device='cuda:0')] [tensor(223.4914, device='cuda:0'), tensor(223.7095, device='cuda:0')]
5 [tensor(0.6920, device='cuda:0'), tensor(0.5554, device='cuda:0')] [tensor(223.7095, device='cuda:0'), tensor(223.5953, device='cuda:0')]
6 [tensor(0.5554, device='cuda:0'), tensor(0.3916, device='cuda:0')] [tensor(223.5953, device='cuda:0'), tensor(223.4862, device='cuda:0')]
```

```
7 [tensor(0.3916, device='cuda:0'), tensor(0.2201, device='cuda:0')] [tensor(223.4862, device='cuda:0'), tensor
(221.1932, device='cuda:0')]
8 [tensor(0.2201, device='cuda:0'), tensor(0.0991, device='cuda:0')] [tensor(221.1932, device='cuda:0'), tensor
(216.4450, device='cuda:0')]
9 [tensor(0.0991, device='cuda:0'), tensor(0.0397, device='cuda:0')] [tensor(216.4450, device='cuda:0'), tensor
(209.8310, device='cuda:0')]
10 [tensor(0.0397, device='cuda:0'), tensor(0.0144, device='cuda:0')] [tensor(209.8310, device='cuda:0'), tensor
(198.0061, device='cuda:0')]
11 [tensor(0.0144, device='cuda:0'), tensor(0.0043, device='cuda:0')] [tensor(198.0061, device='cuda:0'), tensor
(181.1022, device='cuda:0')]
```

很明显了，在这种情况下，inital_step 进行的时第一步和第二步的更新，然后将第一步和第二步得到的t和noise进行存储，接下来，每次更新时，通过队列的形式pop一个，再push一个最新的，3阶可以类比。因此t_prev_list和model_prev_list其实记录的是当前t前order步的结果。

DPM-SOLVER-1:

代码在函数https://github.com/LuChengTHU/dpm-solver/blob/3d31b82a62799e62cbfbb37a109fdef6309f9cd3/dpm_solver_pytorch.py#L555 中，代码如下：

```
def dpm_solver_first_update(self, x, s, t, model_s=None, return_intermediate=False):
    ns = self.noise_schedule
    dims = x.dim()
    lambda_s, lambda_t = ns.marginal_lambda(s), ns.marginal_lambda(t)
    h = lambda_t - lambda_s
    log_alpha_s, log_alpha_t = ns.marginal_log_mean_coeff(s), ns.marginal_log_mean_coeff(t)
    sigma_s, sigma_t = ns.marginal_std(s), ns.marginal_std(t)
    alpha_t = torch.exp(log_alpha_t)
    phi_1 = torch.expm1(h)
    if model_s is None:
        model_s = self.model_fn(x, s)
    x_t = (
        torch.exp(log_alpha_t - log_alpha_s) * x
        - (sigma_t * phi_1) * model_s
    )
    return x_t
```

s来自于t_prev_list[-1],因此这里的s和t等同于论文中的$\lambda_{t_i-1}, \lambda_{t_i}$,h等同于论文中的$h_i = \lambda_{t_i} - \lambda_{t_i-1}$,log_alpha_s和log_alpha_t等同于论文的$\log(\alpha_{\lambda_{t_i-1}}), \log(\alpha_{\lambda_{t_i}})$,sigma_s和sigma_t等同于论文中的$\sigma_{\lambda_{t_i-1}}, \sigma_{\lambda_{t_i}}$,alpha_t等同于论文中的$\alpha_{\lambda_{t_i}}$,model_s等同于论文中的$\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1})$,x_t等同于论文中的$\tilde{x}_{t_i}$。因此这里的reverse process和论文完全一致，如下：

$$\tilde{x}_{t_i} = \frac{\alpha_{t_i}}{\alpha_{t_{i-1}}}\tilde{x}_{t_{i-1}} - \sigma_{t_i}(e^{h_i} - 1)\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}).$$

DPM-SOLVER-2:

代码在函数https://github.com/LuChengTHU/dpm-solver/blob/3d31b82a62799e62cbfbb37a109fdef6309f9cd3/dpm_solver_pytorch.py#L804 中，代码为：

```
def multistep_dpm_solver_second_update(self, x, model_prev_list, t_prev_list, t, solver_type="dpmsolver"):
    ns = self.noise_schedule
    model_prev_1, model_prev_0 = model_prev_list[-2], model_prev_list[-1]
    t_prev_1, t_prev_0 = t_prev_list[-2], t_prev_list[-1]
    lambda_prev_1, lambda_prev_0, lambda_t = ns.marginal_lambda(t_prev_1), ns.marginal_lambda(t_prev_0), ns.
```

```
marginal_lambda(t)
        log_alpha_prev_0, log_alpha_t = ns.marginal_log_mean_coeff(t_prev_0), ns.marginal_log_mean_coeff(t)
        sigma_prev_0, sigma_t = ns.marginal_std(t_prev_0), ns.marginal_std(t)
        alpha_t = torch.exp(log_alpha_t)

        h_0 = lambda_prev_0 - lambda_prev_1
        h = lambda_t - lambda_prev_0
        r0 = h_0 / h
        D1_0 = (1. / r0) * (model_prev_0 - model_prev_1)
        phi_1 = torch.expm1(h)
        if solver_type == 'dpmsolver':
            x_t = (
                (torch.exp(log_alpha_t - log_alpha_prev_0)) * x
                - (sigma_t * phi_1) * model_prev_0
                - 0.5 * (sigma_t * phi_1) * D1_0
            )
        elif solver_type == 'taylor':
            x_t = (
                (torch.exp(log_alpha_t - log_alpha_prev_0)) * x
                - (sigma_t * phi_1) * model_prev_0
                - (sigma_t * (phi_1 / h - 1.)) * D1_0
            )
        return x_t
```

仅查看分支 `if solver_type == 'dpmsolver'` 分支，首先仍然是一系列参数和论文的匹配，model_prev_1 = $\epsilon_\theta(\tilde{x}_{t_{i-2}}, t_{i-2})$，model_prev_0 = $\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1})$，t_prev_1 = t_{i-2},t_prev_0 = t_{i-1},lambda_prev_1 = $\lambda_{t_{i-2}}$, lambda_prev_0 = $\lambda_{t_{i-1}}$, lambda_t = λ_{t_i}，log_alpha_prev_0 = $\log(\alpha_{\lambda_{t_{i-1}}})$,log_alpha_t = $\log(\alpha_{\lambda_{t_i}})$,sigma_prev_0 = $\sigma_{t_{i-1}}$, sigma_t = σ_{t_i}, alpha_t = $\alpha_{\lambda_{t_i}}$, h_0 = $\lambda_{t_{i-1}} - \lambda_{t_{i-2}}$, h = $\lambda_{t_i} - \lambda_{t_{i-1}}$, r0 = $\dfrac{\lambda_{t_{i-1}} - \lambda_{t_{i-2}}}{\lambda_{t_i} - \lambda_{t_{i-1}}}$, D1_0 = $\dfrac{\lambda_{t_i} - \lambda_{t_{i-1}}}{\lambda_{t_{i-1}} - \lambda_{t_{i-2}}}(\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}) - \epsilon_\theta(\tilde{x}_{t_{i-2}}, t_{i-2}))$.

torch.expm1完成的是op：$y = e^x - 1$,因此phi_1 = $e^{\lambda_{t_i} - \lambda_{t_{i-1}}} - 1$.

最后x_t 相当于计算：

$$\tilde{x}_{t_i} = e^{\log(\alpha_{\lambda_{t_i}}) - \log(\alpha_{\lambda_{t_{i-1}}})}\tilde{x}_{t_{i-1}} - \sigma_{t_i}(e^{\lambda_{t_i} - \lambda_{t_{i-1}}} - 1)\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}) - \frac{1}{2}\sigma_{t_i}(e^{\lambda_{t_i} - \lambda_{t_{i-1}}} - 1)\left(\frac{\lambda_{t_i} - \lambda_{t_{i-1}}}{\lambda_{t_{i-1}} - \lambda_{t_{i-2}}}(\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}) - \epsilon_\theta(\tilde{x}_{t_{i-2}}, t_{i-2}))\right)$$

$$= \frac{\alpha_{\lambda_{t_i}}}{\alpha_{\lambda_{t_{i-1}}}}\tilde{x}_{t_{i-1}} - \sigma_{t_i}(e^{h_i} - 1)\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}) - \frac{1}{2}\sigma_{t_i}(e^{h_i} - 1)\left(\frac{h_i}{h_{i-1}}(\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}) - \epsilon_\theta(\tilde{x}_{t_{i-2}}, t_{i-2}))\right)$$

$$\approx \frac{\alpha_{\lambda_{t_i}}}{\alpha_{\lambda_{t_{i-1}}}}\tilde{x}_{t_{i-1}} - \left[\sigma_{t_i}(e^{h_i} - 1)\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}) + \frac{1}{2}\sigma_{t_i}(e^{h_i} - 1)\left(\frac{h_i}{h_{i-1}}(\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}) - \epsilon_\theta(\tilde{x}_{t_{i-2}}, t_{i-2}))\right) + \mathcal{O}(2)\right]$$

$$= \frac{\alpha_{\lambda_{t_i}}}{\alpha_{\lambda_{t_{i-1}}}}\tilde{x}_{t_{i-1}} - \sigma_{t_i}(e^{h_i} - 1)\left[\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}) + \frac{1}{2}\left(\frac{h_i}{h_{i-1}}(\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}) - \epsilon_\theta(\tilde{x}_{t_{i-2}}, t_{i-2}))\right) + \mathcal{O}(2)\right]$$

$$\approx \frac{\alpha_{\lambda_{t_i}}}{\alpha_{\lambda_{t_{i-1}}}}\tilde{x}_{t_{i-1}} - \sigma_{t_i}(e^{h_i} - 1)\left[\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}) + \frac{1}{2}h_i(\frac{d\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1})}{dt_{i-1}}) + \mathcal{O}(2)\right]$$

这个形式是不同于论文的算法流程图的结果，但反推回去是和论文公式（3.6）完全一致。思考为什么作者没有使用流程图的形式，我猜测是因为这样每一步要前传两次UNet，从而浪费了时间，事实上完全可以通过迭代完成。

DPM-SOLVER-3:

代码在函数https://github.com/LuChengTHU/dpm-solver/blob/3d31b82a62799e62cbfbb37a109fdef6309f9cd3/dpm_solver_pytorch.py#L862 中，代码为：

```
def multistep_dpm_solver_third_update(self, x, model_prev_list, t_prev_list, t, solver_type='dpmsolver'):
    ns = self.noise_schedule
    model_prev_2, model_prev_1, model_prev_0 = model_prev_list
    t_prev_2, t_prev_1, t_prev_0 = t_prev_list
    lambda_prev_2, lambda_prev_1, lambda_prev_0, lambda_t = ns.marginal_lambda(t_prev_2), ns.marginal_lambda
(t_prev_1), ns.marginal_lambda(t_prev_0), ns.marginal_lambda(t)
    log_alpha_prev_0, log_alpha_t = ns.marginal_log_mean_coeff(t_prev_0), ns.marginal_log_mean_coeff(t)
    sigma_prev_0, sigma_t = ns.marginal_std(t_prev_0), ns.marginal_std(t)
    alpha_t = torch.exp(log_alpha_t)
    h_1 = lambda_prev_1 - lambda_prev_2
    h_0 = lambda_prev_0 - lambda_prev_1
    h = lambda_t - lambda_prev_0
    r0, r1 = h_0 / h, h_1 / h
    D1_0 = (1. / r0) * (model_prev_0 - model_prev_1)
    D1_1 = (1. / r1) * (model_prev_1 - model_prev_2)
    D1 = D1_0 + (r0 / (r0 + r1)) * (D1_0 - D1_1)
    D2 = (1. / (r0 + r1)) * (D1_0 - D1_1)
    phi_1 = torch.expm1(h)
    phi_2 = phi_1 / h - 1.
    phi_3 = phi_2 / h - 0.5
    x_t = (
        (torch.exp(log_alpha_t - log_alpha_prev_0)) * x
        - (sigma_t * phi_1) * model_prev_0
        - (sigma_t * phi_2) * D1
        - (sigma_t * phi_3) * D2
    )
    return x_t
```

其实，根据二阶dpm solver的经验，已经能够猜到这也是一个泰勒展开至三阶的近似，相比于之前的二阶dpm solver，多的可能就是 `- (sigma_t * phi_3) * D2` 这一一行，同样首先是一系列参数匹配：

model_prev_2 = $\epsilon_\theta(\tilde{x}_{t_{i-3}}, t_{i-3})$, model_prev_1 = $\epsilon_\theta(\tilde{x}_{t_{i-2}}, t_{i-2})$, model_prev_0 = $\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1})$

t_prev_2 = t_{i-3},t_prev_1 = t_{i-2},t_prev_0 = t_{i-1}

lambda_prev_2 = $\lambda_{t_{i-3}}$, lambda_prev_1 = $\lambda_{t_{i-2}}$, lambda_prev_0 = $\lambda_{t_{i-1}}$, lambda_t = λ_{t_i}

log_alpha_prev_0 = $\log(\alpha_{\lambda_{t_{i-1}}})$,log_alpha_t = $\log(\alpha_{\lambda_{t_i}})$

sigma_prev_0 = $\sigma_{t_{i-1}}$, sigma_t = σ_{t_i}

alpha_t = $\alpha_{\lambda_{t_i}}$, h_0 = $\lambda_{t_{i-1}} - \lambda_{t_{i-2}}$,h_1 = $\lambda_{t_{i-2}} - \lambda_{t_{i-3}}$, h = $\lambda_{t_i} - \lambda_{t_{i-1}}$

r0 = $\frac{\lambda_{t_{i-1}} - \lambda_{t_{i-2}}}{\lambda_{t_i} - \lambda_{t_{i-1}}}$,r1 = $\frac{\lambda_{t_{i-2}} - \lambda_{t_{i-3}}}{\lambda_{t_i} - \lambda_{t_{i-1}}}$

D1_0 = $\frac{\lambda_{t_i} - \lambda_{t_{i-1}}}{\lambda_{t_{i-1}} - \lambda_{t_{i-2}}}(\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}) - \epsilon_\theta(\tilde{x}_{t_{i-2}}, t_{i-2}))$

D1_1 = $\frac{\lambda_{t_i} - \lambda_{t_{i-1}}}{\lambda_{t_{i-2}} - \lambda_{t_{i-3}}}(\epsilon_\theta(\tilde{x}_{t_{i-2}}, t_{i-2}) - \epsilon_\theta(\tilde{x}_{t_{i-3}}, t_{i-3}))$

D1 = $\frac{\lambda_{t_i}-\lambda_{t_{i-1}}}{\lambda_{t_{i-1}}-\lambda_{t_{i-2}}}(\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})-\epsilon_\theta(\tilde{x}_{t_{i-2}},t_{i-2}))+\frac{\lambda_{t_{i-1}}-\lambda_{t_{i-2}}}{\lambda_{t_{i-1}}-\lambda_{t_{i-3}}}\left[\frac{\lambda_{t_i}-\lambda_{t_{i-1}}}{\lambda_{t_{i-1}}-\lambda_{t_{i-2}}}(\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})-\epsilon_\theta(\tilde{x}_{t_{i-2}},t_{i-2}))-\frac{\lambda_{t_i}-\lambda_{t_{i-1}}}{\lambda_{t_{i-2}}-\lambda_{t_{i-3}}}(\epsilon_\theta(\tilde{x}_{t_{i-2}},t_{i-2})-\epsilon_\theta(\tilde{x}_{t_{i-3}},t_{i-3}))\right]$

D2 =

$$\frac{\lambda_{t_i}-\lambda_{t_{i-1}}}{\lambda_{t_{i-1}}-\lambda_{t_{i-3}}}\left[\frac{\lambda_{t_i}-\lambda_{t_{i-1}}}{\lambda_{t_{i-1}}-\lambda_{t_{i-2}}}(\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})-\epsilon_\theta(\tilde{x}_{t_{i-2}},t_{i-2}))-\frac{\lambda_{t_i}-\lambda_{t_{i-1}}}{\lambda_{t_{i-2}}-\lambda_{t_{i-3}}}(\epsilon_\theta(\tilde{x}_{t_{i-2}},t_{i-2})-\epsilon_\theta(\tilde{x}_{t_{i-3}},t_{i-3}))\right]$$

phi_1 = $e^{\lambda_{t_i}-\lambda_{t_{i-1}}}-1$，phi_2 = $\frac{e^{\lambda_{t_i}-\lambda_{t_{i-1}}}-1}{\lambda_{t_i}-\lambda_{t_{i-1}}}-1$，phi_3 = $\frac{\frac{e^{\lambda_{t_i}-\lambda_{t_{i-1}}}-1}{\lambda_{t_i}-\lambda_{t_{i-1}}}-1}{\lambda_{t_i}-\lambda_{t_{i-1}}}-\frac{1}{2}$

最后的结果是：

$$\tilde{x}_{t_i}=e^{\log(\alpha_{\lambda_{t_i}})-\log(\alpha_{\lambda_{t_{i-1}}})}\tilde{x}_{t_{i-1}}-\sigma_{t_i}(e^{\lambda_{t_i}-\lambda_{t_{i-1}}}-1)\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})-\sigma_{t_i}(\frac{e^{\lambda_{t_i}-\lambda_{t_{i-1}}}-1}{\lambda_{t_i}-\lambda_{t_{i-1}}}-1)D1-\sigma_{t_i}(\frac{\frac{e^{\lambda_{t_i}-\lambda_{t_{i-1}}}-1}{\lambda_{t_i}-\lambda_{t_{i-1}}}-1}{\lambda_{t_i}-\lambda_{t_{i-1}}}-\frac{1}{2})D2$$

这里的D1和D2相当于代码中的D1和D2，因为太长了不放上去，首先是化简为：

$$\tilde{x}_{t_i}=\frac{\alpha_{\lambda_{t_i}}}{\alpha_{\lambda_{t_{i-1}}}}\tilde{x}_{t_{i-1}}-\sigma_{t_i}(e^{h_i}-1)\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})-\sigma_{t_i}(\frac{e^{h_i}-1}{h_i}-1)D1-\sigma_{t_i}(\frac{e^{h_i}-1}{h_i^2}-\frac{1}{h_i}-\frac{1}{2})D2$$

这里的D1和D2和2阶dpm solver不同，不是简单的差分，而是对积分的近似所产生的中间变量。先化简D1和D2为：

D1 = $\frac{h_i}{h_{i-1}}(\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})-\epsilon_\theta(\tilde{x}_{t_{i-2}},t_{i-2}))+\frac{h_{i-1}}{h_{i-1}+h_{i-2}}\left[\frac{h_i}{h_{i-1}}(\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})-\epsilon_\theta(\tilde{x}_{t_{i-2}},t_{i-2}))-\frac{h_i}{h_{i-2}}(\epsilon_\theta(\tilde{x}_{t_{i-2}},t_{i-2})-\epsilon_\theta(\tilde{x}_{t_{i-3}},t_{i-3}))\right]$

D2 = $\frac{h_i}{h_{i-1}+h_{i-2}}\left[\frac{h_i}{h_{i-1}}(\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})-\epsilon_\theta(\tilde{x}_{t_{i-2}},t_{i-2}))-\frac{h_i}{h_{i-2}}(\epsilon_\theta(\tilde{x}_{t_{i-2}},t_{i-2})-\epsilon_\theta(\tilde{x}_{t_{i-3}},t_{i-3}))\right]$

这里三个有端点t_{i-3},t_{i-2},t_{i-1}，而可以通过近似继续化简：

D1 ≈ $h_i\frac{d\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})}{d\lambda_{t_{i-1}}}+\frac{h_{i-1}}{h_{i-1}+h_{i-2}}\left[h_i\frac{d\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})}{d\lambda_{t_{i-1}}}-h_i\frac{d\epsilon_\theta(\tilde{x}_{t_{i-2}},t_{i-2})}{d\lambda_{t_{i-2}}}\right]$

这个是对函数$\frac{d\epsilon_\theta(\tilde{x}_t,t)}{d\lambda_t}$在t = t_{i-1}处的展开，因此等价于：

D1≈ $h_i\frac{d\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})}{d\lambda_{t_{i-1}}}+h_ih_{i-1}\frac{d^2\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})}{d\lambda_{t_{i-1}}^2}+\mathcal{O}(3)$

D2 ≈ $\frac{h_i}{h_{i-1}+h_{i-2}}\left[h_i\frac{d\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})}{d\lambda_{t_{i-1}}}-h_i\frac{d\epsilon_\theta(\tilde{x}_{t_{i-2}},t_{i-2})}{d\lambda_{t_{i-2}}}\right]$

D2 $\approx h_i^2\frac{d^2\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})}{d\lambda_{t_{i-1}}^2}+\mathcal{O}(3)$

代入原式得：

$$\tilde{x}_{t_i}=\frac{\alpha_{\lambda_{t_i}}}{\alpha_{\lambda_{t_{i-1}}}}\tilde{x}_{t_{i-1}}-\sigma_{t_i}(e^{h_i}-1)\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})-\sigma_{t_i}(\frac{e^{h_i}-1}{h_i}-1)(h_i\frac{d\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})}{d\lambda_{t_{i-1}}}+h_ih_{i-1}\frac{d^2\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})}{d\lambda_{t_{i-1}}^2}+\mathcal{O}(3))-\sigma_{t_i}(\frac{e^{h_i}-1}{h_i^2}-\frac{1}{h_i}-\frac{1}{2})(h_i^2\frac{d^2\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})}{d\lambda_{t_{i-1}}^2}+\mathcal{O}(3))$$

$$\tilde{x}_{t_i}=\frac{\alpha_{\lambda_{t_i}}}{\alpha_{\lambda_{t_{i-1}}}}\tilde{x}_{t_{i-1}}-\sigma_{t_i}(e^{h_i}-1)\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})-\sigma_{t_i}(e^{h_i}-h_i-1)(\frac{d\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})}{d\lambda_{t_{i-1}}}+h_{i-1}\frac{d^2\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})}{d\lambda_{t_{i-1}}^2}+\mathcal{O}(3))-\sigma_{t_i}(e^{h_i}-h_i^2/2-h_i-1)(\frac{d^2\epsilon_\theta(\tilde{x}_{t_{i-1}},t_{i-1})}{d\lambda_{t_{i-1}}^2}+\mathcal{O}(3))$$

$$\tilde{x}_{t_i} \approx \frac{\alpha_{\lambda_{t_i}}}{\alpha_{\lambda_{t_{i-1}}}}\tilde{x}_{t_{i-1}} - \sigma_{t_i}(e^{h_i}-1)\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}) - \sigma_{t_i}(e^{h_i}-h_i-1)(\frac{d\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1})}{d\lambda_{t_{i-1}}} + h_{i-1}\frac{d^2\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1})}{d\lambda_{t_{i-1}}^2}) - \sigma_{t_i}(e^{h_i}-h_i^2/2-h_i-1)(\frac{d^2\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1})}{d\lambda_{t_{i-1}}^2}) + \mathcal{O}(3)$$

值得注意的一点是，3阶dpm solver中，论文中给出了$\phi_{k+1}(h)$的函数，如下：

$$\begin{aligned}
\varphi_1(h) &= \frac{e^h - 1}{h}, \\
\varphi_2(h) &= \frac{e^h - h - 1}{h^2}, \\
\varphi_3(h) &= \frac{e^h - h^2/2 - h - 1}{h^3}.
\end{aligned}$$

论文中给出的式子为：

$$\boldsymbol{x}_t = \frac{\alpha_t}{\alpha_s}\boldsymbol{x}_s - \sigma_t \sum_{k=0}^{n} h^{k+1}\varphi_{k+1}(h)\hat{\boldsymbol{\epsilon}}_\theta^{(k)}(\hat{\boldsymbol{x}}_{\lambda_s}, \lambda_s) + \mathcal{O}(h^{n+2}).$$

因此，h^{k+1}刚好和$\phi_{k+1}(h)$分母抵消，因此原式可以写为：

$$\tilde{x}_{t_i} = \frac{\alpha_{\lambda_{t_i}}}{\alpha_{\lambda_{t_{i-1}}}}\tilde{x}_{t_{i-1}} - \sigma_{t_i}(h\psi_1(h))\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1}) - \sigma_{t_i}(h^2\psi_2(h))(\frac{d\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1})}{d\lambda_{t_{i-1}}} + h_{i-1}\frac{d^2\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1})}{d\lambda_{t_{i-1}}^2}) - \sigma_{t_i}(h^3\psi_3(h))(\frac{d^2\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1})}{d\lambda_{t_{i-1}}^2}) + \mathcal{O}(3)$$

而对于剩余的近似，事实上就是数值分析的内容了，其中$(\frac{d\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1})}{d\lambda_{t_{i-1}}} + h_{i-1}\frac{d^2\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1})}{d\lambda_{t_{i-1}}^2})$近似于$\frac{d\epsilon_\theta(\tilde{x}_{t_s}, t_s)}{d\lambda_{t_s}}$，而$(\frac{d^2\epsilon_\theta(\tilde{x}_{t_{i-1}}, t_{i-1})}{d\lambda_{t_{i-1}}^2})$近似于$\frac{d^2\epsilon_\theta(\tilde{x}_{t_s}, t_s)}{d\lambda_{t_s}^2}$ $s.t. \lambda_{t_s} < \lambda_{t_i}$,之所以会出现看上去不太对齐的样子是因为第一个点是2/3点到1/3点的近似，而第二个则是1/3点到1/3点的近似。