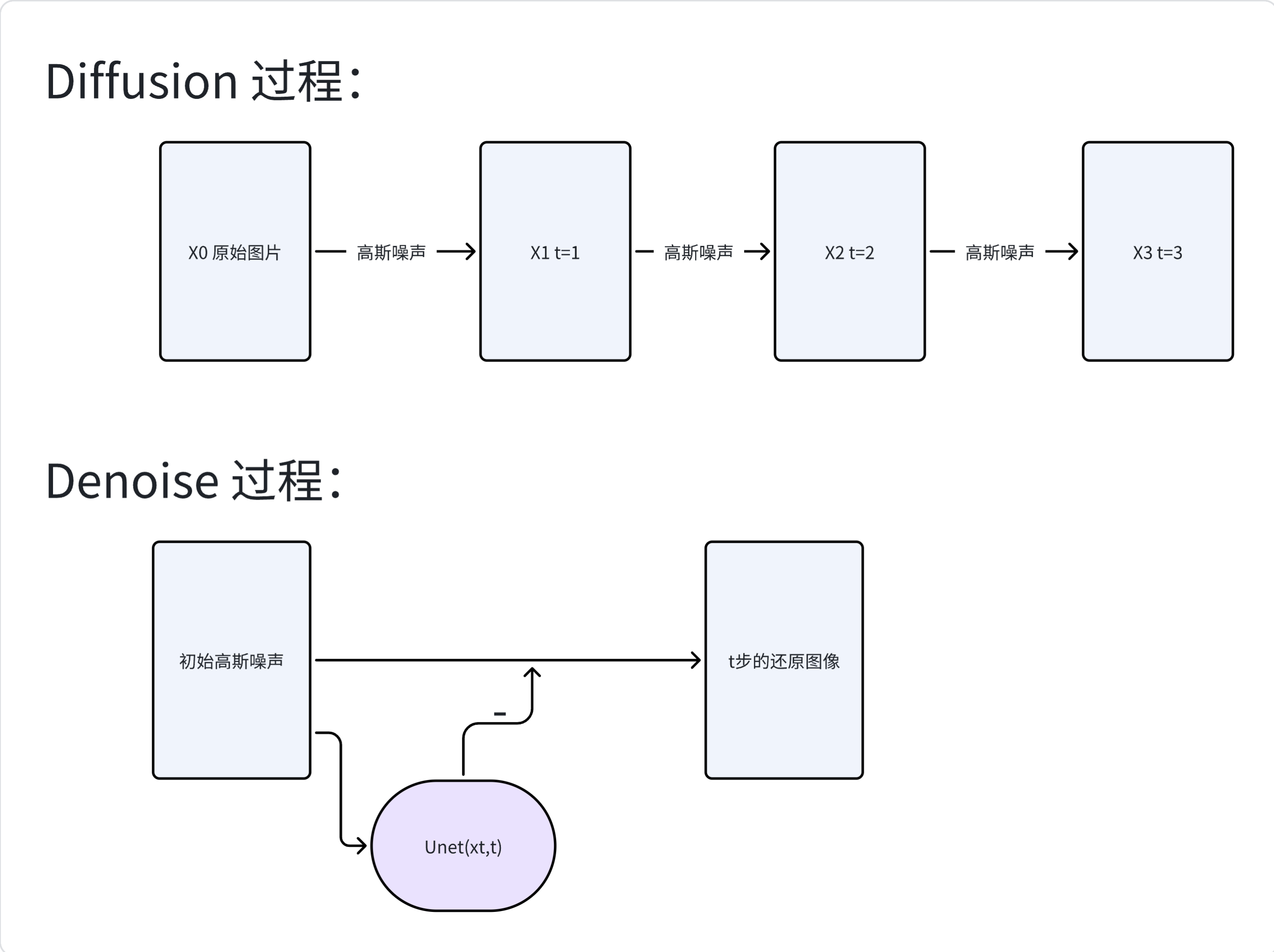
Diffusion&&Flow Matching 学生证准备一下

1.DDPM



- **Diffusion Process**：加噪过程。取一张干净的图片 x_0 ，逐步往上添加高斯噪声，执行T步后生成纯高斯噪声 x_T 。这个过程中遵从的分布，记为 $q(x_t|x_{t-1})$
- **Denoise Process**：去噪过程。这一步中我们训练一个UNet架构的去噪模型，它吃 x_t 和 t ，然后去预测噪声 ϵ_θ ，使得 ϵ_θ 逼近Diffusion Process对应步骤中采样的真值噪声 ϵ_t 。这个过程中遵从的分布，我们记为 $p_\theta(x_{t-1}|x_t)$ 。其中， θ 表示UNet参数。

1.1 数学原理

我们的目标是让模型产生的p分布接近于q分布，此即

$$argmin_\theta KL(P_{data}||P_\theta)$$

推导过程

1）转为P形式

$$argmax_{\theta}\prod_{i=1}^{m}P_{\theta}(x_i)$$

2) Evidence Lower Bound (ELBO)优化

$$logP_{\theta}(x) \geq E_{q_{\phi}(x_1:x_T|x_0)}log\frac{P_{\theta}(x_0:x_T)}{q_{\phi}(x_1:x_T|x_0)}$$

3）进一步拆解,优化目标转为最大化

$$\sum_{t=2}^{T}E_{q(x_t|x_0)}[D_{KL}(q(x_{t-1}|x_t,x_0)||p_{\theta}(x_{t-1}|x_t)]$$

4）推导出q的分布

$$\mu_q:\frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t x_0+\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})x_t}{1-\bar{\alpha}_t} \quad \longleftarrow \quad \mu_{\theta}$$

$$\sigma_q^2:\frac{(1-\alpha_t)(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} \quad \longleftarrow \quad \sigma_{\theta}^2$$

DDPM方差为常数，所以模型只需要去预测均值

5）然后我们再次重写均值Uq

$$
\begin{aligned}
\mu_q &= \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t x_0 + \sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})x_t}{1-\bar{\alpha}_t} \\
&= \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t \frac{x_t - \sqrt{1-\bar{\alpha}_t}\epsilon}{\sqrt{\bar{\alpha}_t}} + \sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})x_t}{1-\bar{\alpha}_t} \\
&= \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon\right)
\end{aligned}
$$

xt是已知值，alphat是参数，那么就只需要让模型去预测噪声了。

然后通过MSE_LOSS 来训练我们的Unet模型

1.2 源码解读

（1）Diffusion类

```python
from typing import Tuple, Optional
import torch
import torch.nn.functional as F
import torch.utils.data
from torch import nn
from labml_nn.diffusion.ddpm.utils import gather
class DenoiseDiffusion:
    """
    Denoise Diffusion
    """

    def __init__(self, eps_model: nn.Module, n_steps: int, device: torch.device):
        """
        Params:
            eps_model: UNet去噪模型。
            n_steps: 训练总步数T
            device: 训练所用硬件
        """
        super().__init__()
        # 定义UNet架构模型
```

```python
        self.eps_model = eps_model
        # 人为设置超参数beta，满足beta随着t的增大而增大，同时将beta搬运到训练硬件上
        self.beta = torch.linspace(0.0001, 0.02, n_steps).to(device)
        # 根据beta计算alpha
        self.alpha = 1. - self.beta
        # 根据alpha计算alpha_bar
        self.alpha_bar = torch.cumprod(self.alpha, dim=0)
        # 定义训练总步长
        self.n_steps = n_steps
        # sampling中的sigma_t
        self.sigma2 = self.beta

    def q_xt_x0(self, x0: torch.Tensor, t: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        """
        Diffusion Process的中间步骤，根据x0和t，推导出xt所服从的高斯分布的mean和var
        Params:
            x0：来自训练数据的干净的图片
            t：某一步time_step
        Return:
            mean: xt所服从的高斯分布的均值
            var：xt所服从的高斯分布的方差
        """

        # ----------------------------------------------------------------
        # gather：人为定义的函数，从一连串超参中取出当前t对应的超参alpha_bar
        # 由于xt = sqrt(alpha_bar_t) * x0 + sqrt(1-alpha_bar_t) * epsilon
        # 其中epsilon~N(0, I)
        # 因此根据高斯分布性质，xt~N(sqrt(alpha_bar_t) * x0, 1-alpha_bar_t)
        # 即为我们要求的mean和var
        # ----------------------------------------------------------------
        mean = gather(self.alpha_bar, t) ** 0.5 * x0
        var = 1 - gather(self.alpha_bar, t)

        return mean, var

    def q_sample(self, x0: torch.Tensor, t: torch.Tensor, eps: Optional[torch.Tensor] = None):
        """
        Diffusion Process，根据xt所服从的高斯分布的mean和var，求出xt
        Params:
            x0：来自训练数据的干净的图片
            t：某一步time_step
        Return:
            xt：第t时刻加完噪声的图片
        """

```

```
        # ----------------------------------------------------------------
        # xt = sqrt(alpha_bar_t) * x0 + sqrt(1-alpha_bar_t) * epsilon
        #    = mean + sqrt(var) * epsilon
        # 其中，epsilon~N(0, I)
        # ----------------------------------------------------------------
        if eps is None:
            eps = torch.randn_like(x0)

        mean, var = self.q_xt_x0(x0, t)
        return mean + (var ** 0.5) * eps

    def p_sample(self, xt: torch.Tensor, t: torch.Tensor):
        """
        Sampling, 当模型训练好之后，根据x_t和t，推出x_{t-1}
        Params:
            x_t: t时刻的图片
            t: 某一步time_step
        Return:
            x_{t-1}: 第t-1时刻的图片
        """

        # eps_model: 训练好的UNet去噪模型
        # eps_theta: 用训练好的UNet去噪模型，预测第t步的噪声
        eps_theta = self.eps_model(xt, t)

        # 根据Sampling提供的公式，推导出x_{t-1}
        alpha_bar = gather(self.alpha_bar, t)
        alpha = gather(self.alpha, t)
        eps_coef = (1 - alpha) / (1 - alpha_bar) ** .5
        mean = 1 / (alpha ** 0.5) * (xt - eps_coef * eps_theta)
        var = gather(self.sigma2, t)
        eps = torch.randn(xt.shape, device=xt.device)

        return mean + (var ** .5) * eps

    def loss(self, x0: torch.Tensor, noise: Optional[torch.Tensor] = None):
        """
        1. 随机抽取一个time_step t
        2. 执行diffusion process(q_sample)，随机生成噪声epsilon~N(0, I),
           然后根据x0, t和epsilon计算xt
        3. 使用UNet去噪模型（p_sample），根据xt和t得到预测噪声epsilon_theta
        4. 计算mse_loss(epsilon, epsilon_theta) 也可以是别的Loss


        Params:
            x0: 来自训练数据的干净的图片
            noise: diffusion process中随机抽样的噪声epsilon~N(0, I)
```

```
        Return:
            loss: 真实噪声和预测噪声之间的loss
        """

        batch_size = x0.shape[0]
        # 随机抽样t
        t = torch.randint(0, self.n_steps, (batch_size,), device=x0.device, dtype=torch.long)

        # 如果为传入噪声，则从N(0, I)中抽样噪声
        if noise is None:
            noise = torch.randn_like(x0)

        # 执行Diffusion process，计算xt
        xt = self.q_sample(x0, t, eps=noise)
        # 执行Denoise Process，得到预测的噪声epsilon_theta
        eps_theta = self.eps_model(xt, t)

        # 返回真实噪声和预测噪声之间的mse loss
        return F.mse_loss(noise, eps_theta)
```

（2）然后就是我们是需要预测噪声嘛，所以用Unet，这里感觉也是可以用别的，MiniUnet等

```


from typing import Tuple, Optional

import torch
import torch.nn.functional as F
import torch.utils.data
from torch import nn

from labml_nn.diffusion.ddpm.utils import gather

class DenoiseDiffusion:
    """
    Denoise Diffusion
    """

    def __init__(self, eps_model: nn.Module, n_steps: int, device: torch.device):
        """
        Params:
            eps_model: UNet去噪模型。
            n_steps: 训练总步数T
```

```
            device: 训练所用硬件
        """
        super().__init__()
        # 定义UNet架构模型
        self.eps_model = eps_model
        # 人为设置超参数beta，满足beta随着t的增大而增大，同时将beta搬运到训练硬件上
        self.beta = torch.linspace(0.0001, 0.02, n_steps).to(device)
        # 根据beta计算alpha
        self.alpha = 1. - self.beta
        # 根据alpha计算alpha_bar
        self.alpha_bar = torch.cumprod(self.alpha, dim=0)
        # 定义训练总步长
        self.n_steps = n_steps
        # sampling中的sigma_t
        self.sigma2 = self.beta

    def q_xt_x0(self, x0: torch.Tensor, t: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        """
        Diffusion Process的中间步骤，根据x0和t，推导出xt所服从的高斯分布的mean和var
        Params:
            x0: 来自训练数据的干净的图片
            t: 某一步time_step
        Return:
            mean: xt所服从的高斯分布的均值
            var: xt所服从的高斯分布的方差
        """

        # ----------------------------------------------------------------
        # gather: 人为定义的函数，从一连串超参中取出当前t对应的超参alpha_bar
        # 由于xt = sqrt(alpha_bar_t) * x0 + sqrt(1-alpha_bar_t) * epsilon
        # 其中epsilon~N(0, I)
        # 因此根据高斯分布性质，xt~N(sqrt(alpha_bar_t) * x0, 1-alpha_bar_t)
        # 即为我们要求的mean和var
        # ----------------------------------------------------------------
        mean = gather(self.alpha_bar, t) ** 0.5 * x0
        var = 1 - gather(self.alpha_bar, t)

        return mean, var

    def q_sample(self, x0: torch.Tensor, t: torch.Tensor, eps: Optional[torch.Tensor] = None):
        """
        Diffusion Process，根据xt所服从的高斯分布的mean和var，求出xt
        Params:
            x0: 来自训练数据的干净的图片
            t: 某一步time_step
```

```python
        Return:
            xt: 第t时刻加完噪声的图片
        """

        # ------------------------------------------------------------------
        # xt = sqrt(alpha_bar_t) * x0 + sqrt(1-alpha_bar_t) * epsilon
        #    = mean + sqrt(var) * epsilon
        # 其中, epsilon~N(0, I)
        # ------------------------------------------------------------------
        if eps is None:
            eps = torch.randn_like(x0)

        mean, var = self.q_xt_x0(x0, t)
        return mean + (var ** 0.5) * eps

    def p_sample(self, xt: torch.Tensor, t: torch.Tensor):
        """
        Sampling, 当模型训练好之后，根据x_t和t，推出x_{t-1}
        Params:
            x_t: t时刻的图片
            t: 某一步time_step
        Return:
            x_{t-1}: 第t-1时刻的图片
        """

        # eps_model: 训练好的UNet去噪模型
        # eps_theta: 用训练好的UNet去噪模型，预测第t步的噪声
        eps_theta = self.eps_model(xt, t)

        # 根据Sampling提供的公式，推导出x_{t-1}
        alpha_bar = gather(self.alpha_bar, t)
        alpha = gather(self.alpha, t)
        eps_coef = (1 - alpha) / (1 - alpha_bar) ** .5
        mean = 1 / (alpha ** 0.5) * (xt - eps_coef * eps_theta)
        var = gather(self.sigma2, t)
        eps = torch.randn(xt.shape, device=xt.device)

        return mean + (var ** .5) * eps

    def loss(self, x0: torch.Tensor, noise: Optional[torch.Tensor] = None):
        """
        1. 随机抽取一个time_step t
        2. 执行diffusion process(q_sample)，随机生成噪声epsilon~N(0, I)，
           然后根据x0, t和epsilon计算xt
        3. 使用UNet去噪模型（p_sample），根据xt和t得到预测噪声epsilon_theta
        4. 计算mse_loss(epsilon, epsilon_theta) 也可以是别的Loss

```

```

        Params:
            x0: 来自训练数据的干净的图片
            noise: diffusion process中随机抽样的噪声epsilon~N(0, I)
        Return:
            loss: 真实噪声和预测噪声之间的loss
        """

        batch_size = x0.shape[0]
        # 随机抽样t
        t = torch.randint(0, self.n_steps, (batch_size,), device=x0.device, dtype=torch.long)

        # 如果为传入噪声，则从N(0, I)中抽样噪声
        if noise is None:
            noise = torch.randn_like(x0)

        # 执行Diffusion process，计算xt
        xt = self.q_sample(x0, t, eps=noise)
        # 执行Denoise Process，得到预测的噪声epsilon_theta
        eps_theta = self.eps_model(xt, t)

        # 返回真实噪声和预测噪声之间的mse loss
        return F.mse_loss(noise, eps_theta)
```

（3）然后就可以写我们的主要的训练部分了

```
def train(self):
    """
    单epoch训练DDPM
    """

    # 遍历每一个batch（monit是自定义类，负责数据读取等）
    for data in monit.iterate('Train', self.data_loader):
        # step数+1（tracker是自定义类，记录日志等）
        tracker.add_global_step()
        # 将这个batch的数据移动到GPU上
        data = data.to(self.device)

        # 每个batch开始时，梯度清0
        self.optimizer.zero_grad()
        # self.diffusion即为DenoiseModel实例，执行forward，计算loss
        loss = self.diffusion.loss(data)
        loss.backward()
        self.optimizer.step()
```

```
        #tracker.save('loss', loss)

def sample(self):
    """
    利用当前模型，将一张随机高斯噪声(xt)逐步还原回x0,
    x0将用于评估模型效果（例如FID分数）
    """
    with torch.no_grad():
        # 随机抽取n_samples张纯高斯噪声
        x = torch.randn([self.n_samples, self.image_channels, self.image_size,
self.image_size],
                            device=self.device)

        # 对每一张噪声，按照sample公式，还原回x0
        for t_ in monit.iterate('Sample', self.n_steps):
            t = self.n_steps - t_ - 1
            x = self.diffusion.p_sample(x, x.new_full((self.n_samples,), t,
dtype=torch.long))

        # 保存x0
        tracker.save('sample', x)

def run(self):
    """
    train主函数
    """
    for _ in monit.loop(self.epochs):
        self.train()
        self.sample()
        tracker.new_line()
        # 保存模型（experiment是这个模块自定义类，为了方便存读，快速开启实验）
        experiment.save_checkpoint()
```

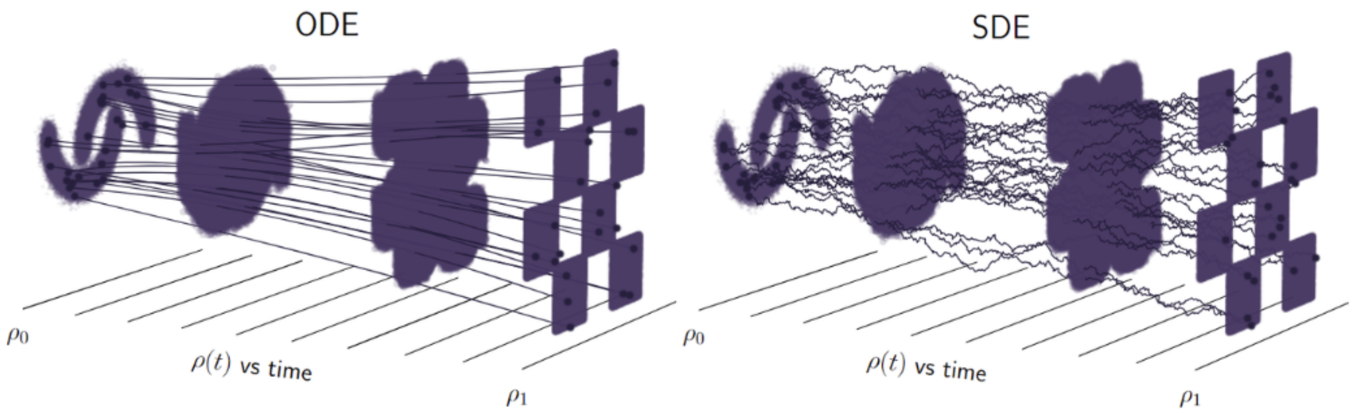
2.Flow Matching

我们可以从DDPM的过程知道，其每一步都是做了一个map映射（自身+高斯噪声，相当于一个线性映射）关键是这样子每一步每一步地是离散的，在预测和训练都需要考虑每一步t，t可能还会很大。Flow matching把这中间的过程看作是连续的，用ODE的思想来解决。最后发现非常方便快捷且结论很漂亮（向量场=x1-x0）

1.1 数学原理

我们知道生成式模型本质是从已知分布采样并通过某种变换得到目标分布，核心是在已知分布的情况下，如何建模得到目标分布的过程。比如DDPM，是从已知的高斯分布采样，通过逐渐的去噪得到目标分布，其建模过程本质上是在目标数据分布到先验分布的过程中不断加噪，并对噪声预测进行建模，从高观点下看的话实际上是在求解随机微分方程SDE

类似于下图，ODE是SDE的一种特殊情况



$$dx = u_t\left(x\right)dt$$

u为速度场，x为每个数据点

这里用 ut(x) 代替 u(t,x) ，包括后续内容，我们约定俗成的把时间项t放到脚标。常微分方程满足边界条件 x0~p0 和 x1~p1 ，其中 p0 和 p1 分别代表先验分布和目标分布，下标代表时间，0为起始，1为终止。直观的理解，我们要找到合适的 ut(x) ，使得对于满足初始先验分布的数据点，对这个进行从0到1的积分以后，能得到目标分布的数据点。按照原始论文，该式最终写成如下形式

$$\frac{d}{dt}\psi_t\left(x\right) = u_t\left(\psi_t\left(x\right)\right)$$

ψt(x) 称为flow map：将输入x理解为一个分布，那么 ψt(x) 可以理解为整个分布在时间t下的移动轨迹或者路径（所有采样点的移动轨迹集合）

p 本身是在 Rd 空间下的概率分布，但因为存在速度场 ut(x) ，所以概率分布会随着时间的变化而变化，原论文用一张图展现了出来：

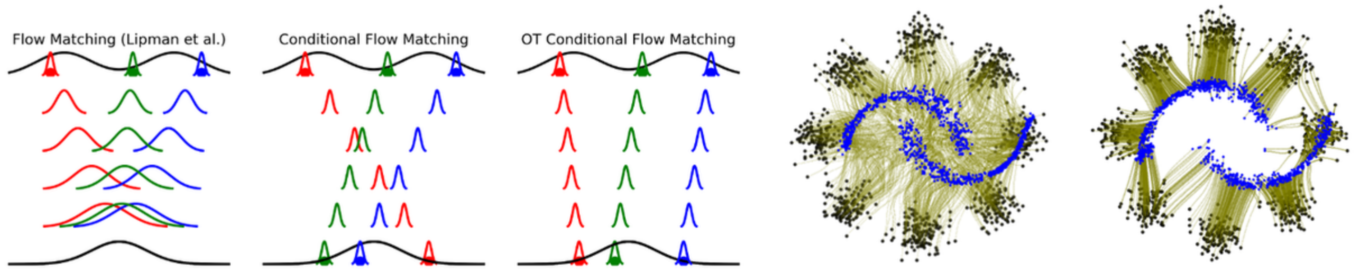


Figure 1: **Left:** Conditional flows from FM (Lipman et al., 2023), I-CFM (§3.2.2), and OT-CFM (§3.2.3). **Right:** Learned flows (green) from moons (blue) to 8gaussians (black) using I-CFM (centre-right) and OT-CFM (far right).

那么现在有一个具体的 ut(x) 的表达式，那么只要给出了初始的 x0 （比如DDPM任务里的一个高斯噪声采样），那么通过数值积分，你就可以得到满足目标分布的 x1 ，下标代表时间。

需要知道 ut(x) ，：知道 ut(x) 的具体表达式很难做到所以我们转而去知道 ut(x) 的一些采样分布，即在不同的t和x下，u的具体取值；

那么我们就去拟合ut

$$L_{FM}\left(\theta\right) := E_{t\sim Uniform(0,1),x\sim p_t(x)}\left\|v_\theta\left(t,x\right)-u_t\left(x\right)\right\|^2$$

但是该式不是那么好做的。所以需要一些推理

（1）

$$L_{CFM}\left(\theta\right) := E_{t\sim U(0,1),x\sim p_t(x|z),z\sim q(z)}\left\|v_\theta\left(t,x\right)-u_t\left(x|z\right)\right\|^2$$

原文证明了优化Lfm于Lcfm等价的（因为求导相同），条件分布的好处在于，可以让未知分布变成tractable的形式，比如ddpm，通过对样本持续加噪，那么中间分布都是高斯分布，就是可以追踪的！

（2）那么问题转到了如何求ut（x|z）的表达式了

注意到t=0,x0是噪声，其实就是可以设定的，比如就是高斯分布等，就是设定出边缘条件分布pt（x）

$$x_0 \sim N\left(\mu_0, \sigma_0^2\right)$$

那么Flow map就可以找出一条，因为Flow map不止一条嘛，我们找一个最直观的：（t=0取值为x0）

$$\psi_t(x_0) = \mu_t + \sigma_t\left(\frac{x_0 - \mu_0}{\sigma_0}\right)$$

(3)带入ODE方程，得到：

$$u_t(x) = \frac{\sigma_t^{'}}{\sigma_t}(x - \mu_t) + \mu_t^{'}$$

带条件的就是：

$$u_t(x|z) = \frac{\sigma_t(z)^{'}}{\sigma_t(z)}[x - \mu_t(z)] + \mu_t(z)^{'}$$

（4）到3就是一般性的结论了，然后我们可以考虑特殊的，因为flow map是可以设定的，只要满足初始条件即可。

（ODE解的唯一性与稳定性相关结论）

$$\begin{cases} p_t(x|z) = N\left(x|(1-t)x_0 + tx_1, \sigma^2\right) \\ \mu_t = (1-t)x_0 + tx_1 \\ \sigma_t = \sigma \end{cases}$$

那么就得到了非常漂亮的结论

$$u_t(x|z) = u_t(x|x_0, x_1) = x_1 - x_0$$

至此Ut就表示出来了，就是每个batch内的 x1-x0.

接下来就是让我们的模型Unet去拟合这个ut了。同样的方式我们去采样Vt，让LOSS=MSELOSS(UT,VT)即可训练

1.2 源码解读

CFM的源码也是依照其数学结果直接给出

（1）

```
class ConditionalFlowMatcher:
    """Base class for conditional flow matching methods. This class implements the independent
    conditional flow matching methods from [1] and serves as a parent class for all other flow
    matching methods.

    It implements:
    - Drawing data from gaussian probability path N(t * x1 + (1 - t) * x0, sigma) function
    - conditional flow matching ut(x1|x0) = x1 - x0
    - score function $\nabla log p_t(x|x0, x1)$
    """

    def __init__(self, sigma: Union[float, int] = 0.0):
        r"""Initialize the ConditionalFlowMatcher class. It requires the hyper-parameter $\sigma$.

        Parameters
        ----------
        sigma : Union[float, int]
        """
        self.sigma = sigma

    def compute_mu_t(self, x0, x1, t):
        """
        Compute the mean of the probability path N(t * x1 + (1 - t) * x0, sigma), see (Eq.14) [1].

        Parameters
        ----------
        x0 : Tensor, shape (bs, *dim)
            represents the source minibatch
        x1 : Tensor, shape (bs, *dim)
            represents the target minibatch
        t : FloatTensor, shape (bs)
```

```

        Returns
        -------
        mean mu_t: t * x1 + (1 - t) * x0

        References
        ----------
        [1] Improving and Generalizing Flow-Based Generative Models with minibatch optimal transport, Preprint, Tong et al.
        """
        t = pad_t_like_x(t, x0)
        return t * x1 + (1 - t) * x0

    def compute_sigma_t(self, t):
        """
        Compute the standard deviation of the probability path N(t * x1 + (1 - t) * x0, sigma), see (Eq.14) [1].

        Parameters
        ----------
        t : FloatTensor, shape (bs)

        Returns
        -------
        standard deviation sigma

        References
        ----------
        [1] Improving and Generalizing Flow-Based Generative Models with minibatch optimal transport, Preprint, Tong et al.
        """
        del t
        return self.sigma

    def sample_xt(self, x0, x1, t, epsilon):
        """
        Draw a sample from the probability path N(t * x1 + (1 - t) * x0, sigma), see (Eq.14) [1].

        Parameters
        ----------
        x0 : Tensor, shape (bs, *dim)
            represents the source minibatch
        x1 : Tensor, shape (bs, *dim)
            represents the target minibatch
        t : FloatTensor, shape (bs)
        epsilon : Tensor, shape (bs, *dim)
```

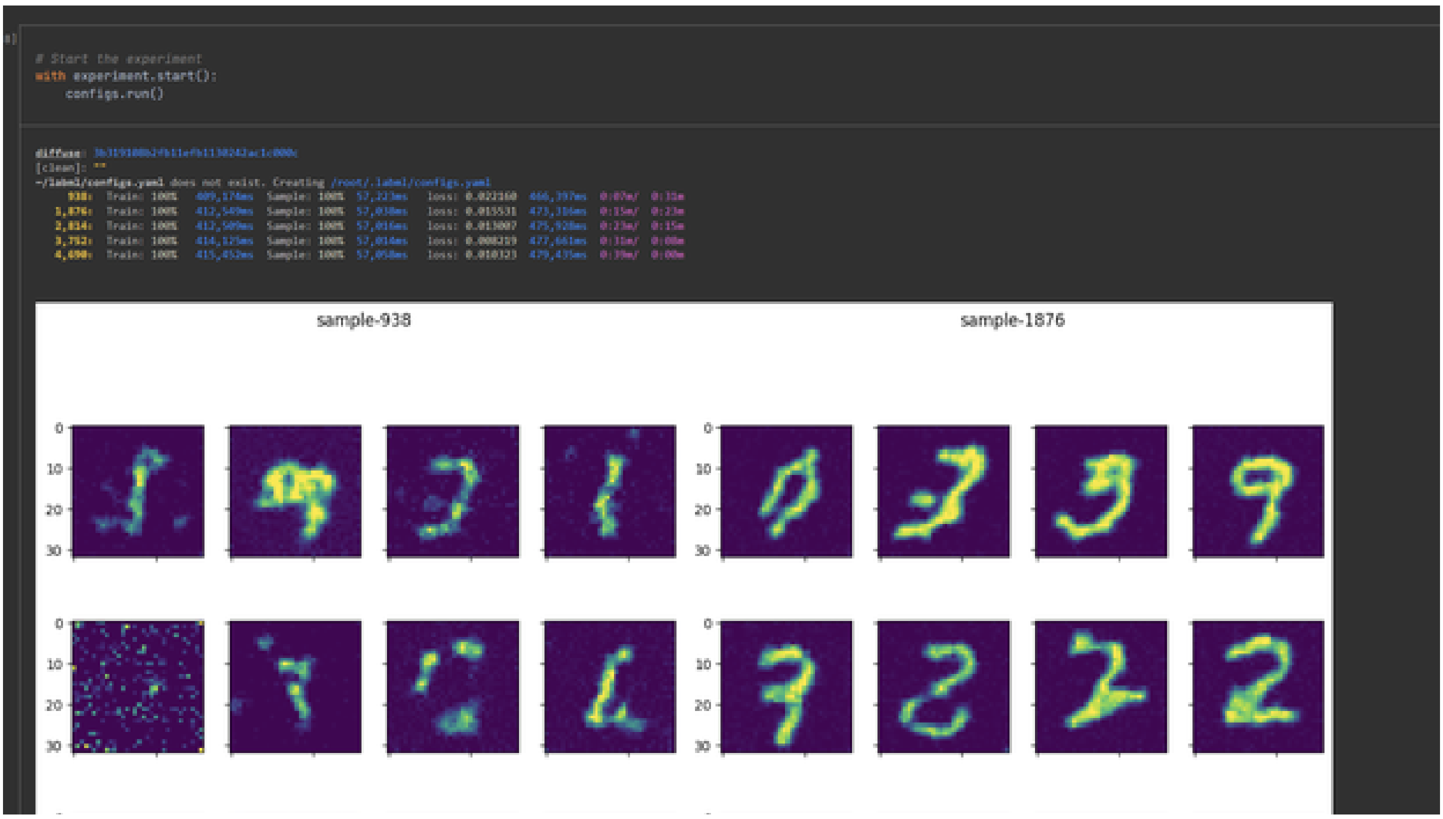
```
            noise sample from N(0, 1)

        Returns
        -------
        xt : Tensor, shape (bs, *dim)

        References
        ----------
        [1] Improving and Generalizing Flow-Based Generative Models with minibatch optimal transport, Preprint, Tong et al.
        """
        mu_t = self.compute_mu_t(x0, x1, t)
        sigma_t = self.compute_sigma_t(t)
        sigma_t = pad_t_like_x(sigma_t, x0)
        return mu_t + sigma_t * epsilon

    def compute_conditional_flow(self, x0, x1, t, xt):
        """
        Compute the conditional vector field ut(x1|x0) = x1 - x0, see Eq.(15) [1].

        Parameters
        ----------
        x0 : Tensor, shape (bs, *dim)
            represents the source minibatch
        x1 : Tensor, shape (bs, *dim)
            represents the target minibatch
        t : FloatTensor, shape (bs)
        xt : Tensor, shape (bs, *dim)
            represents the samples drawn from probability path pt

        Returns
        -------
        ut : conditional vector field ut(x1|x0) = x1 - x0

        References
        ----------
        [1] Improving and Generalizing Flow-Based Generative Models with minibatch optimal transport, Preprint, Tong et al.
        """
        del t, xt
        return x1 - x0

    def sample_noise_like(self, x):
        return torch.randn_like(x)

    #最终的 最核心的Function
```

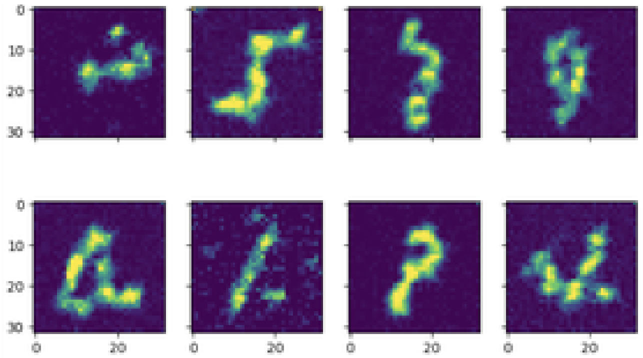
```python
    def sample_location_and_conditional_flow(self, x0, x1, t=None, return_noise=False):
        """
        计算采样与向量场
         xt (drawn from N(t * x1 + (1 - t) * x0, sigma))
        conditional vector field ut(x1|x0) = x1 - x0, see Eq.(15) [1].

        Parameters
        ----------
        x0 : Tensor, shape (bs, *dim)
            represents the source minibatch
        x1 : Tensor, shape (bs, *dim)
            represents the target minibatch
        (optionally) t : Tensor, shape (bs)
            represents the time levels
            if None, drawn from uniform [0,1]
        return_noise : bool
            return the noise sample epsilon


        Returns
        -------
        t : FloatTensor, shape (bs)
        xt : Tensor, shape (bs, *dim)
            represents the samples drawn from probability path pt
        ut : conditional vector field ut(x1|x0) = x1 - x0
        (optionally) eps: Tensor, shape (bs, *dim) such that xt = mu_t + sigma_t * epsilon

        References
        ----------
        [1] Improving and Generalizing Flow-Based Generative Models with minibatch optimal transport, Preprint, Tong et al.
        """
        if t is None:
            t = torch.rand(x0.shape[0]).type_as(x0)
        assert len(t) == x0.shape[0], "t has to have batch size dimension"

        eps = self.sample_noise_like(x0)
        xt = self.sample_xt(x0, x1, t, eps)
        ut = self.compute_conditional_flow(x0, x1, t, xt) #为什么ut就是x1-x0呢?见公式推导
        if return_noise:
            return t, xt, ut, eps
        else:
            return t, xt, ut
```

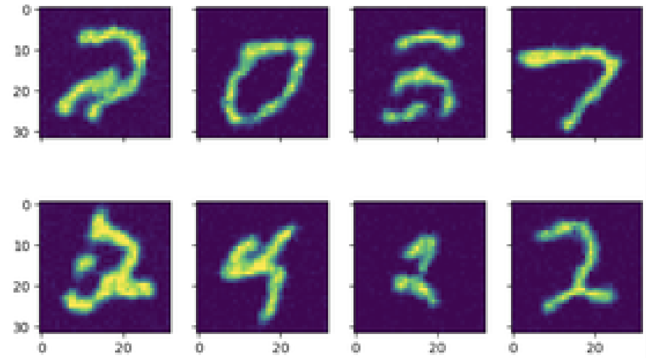
（2）训练循环 MINIST数据集

```python

for epoch in range(n_epochs):
    for i, data in enumerate(train_loader):
        optimizer.zero_grad()
        x1 = data[0].to(device)
        y = data[1].to(device)

        x0 = torch.randn_like(x1)

        t, xt, ut = FM.sample_location_and_conditional_flow(x0, x1) #计算t步的真实场

        vt = model(t, xt, y) #模型预测的向量场

        loss = torch.mean((vt - ut) ** 2)

        loss.backward()
        optimizer.step()
        #print(i,loss.item())
    print(f"epoch: {epoch}, loss: {loss.item():.4}")
```

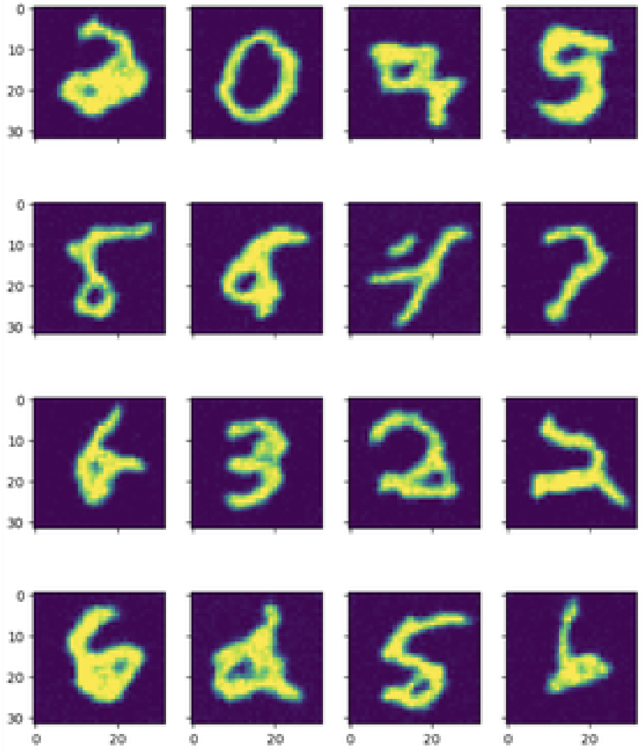
3.跑代码



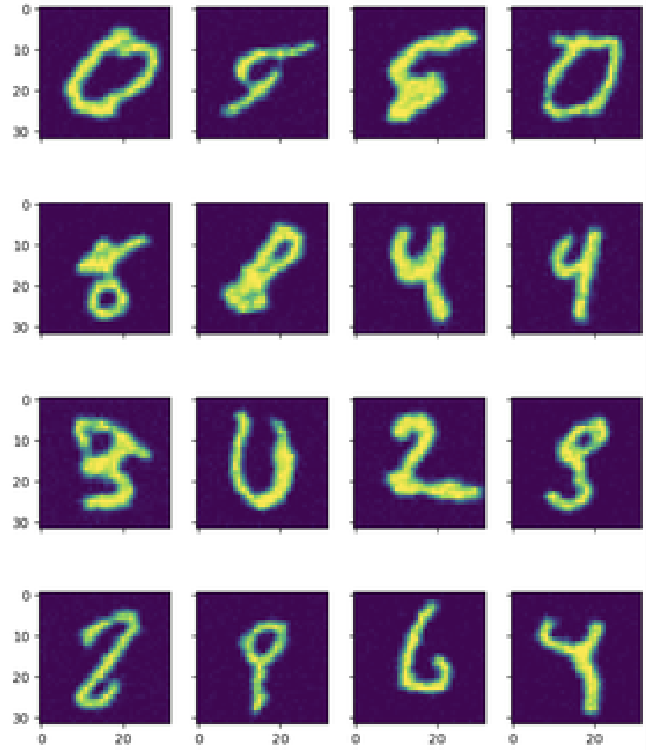




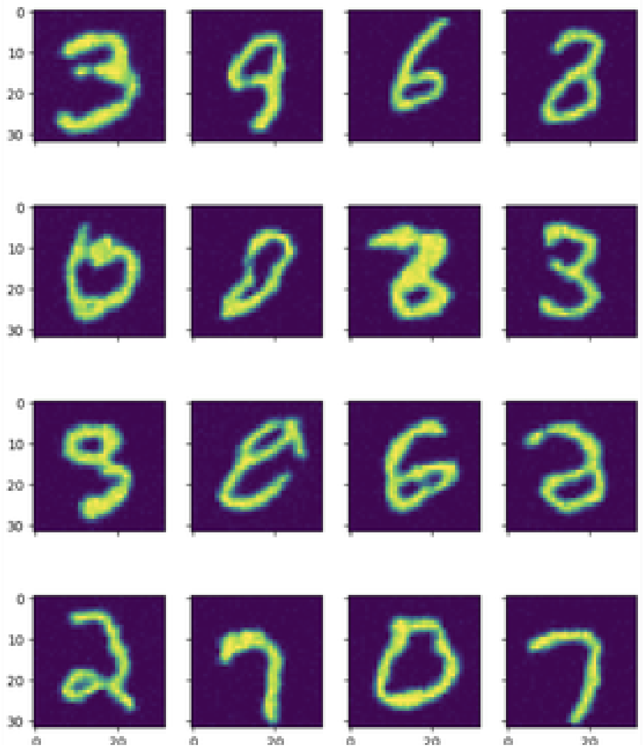
sample-2814



sample-3752



sample-4690



```
            atol=1e-4,
            rtol=1e-4,
            method="dopri5",
        )
    else:
        traj = node.trajectory(
            torch.randn(100, 1, 28, 28, device=device),
            t_span=torch.linspace(0, 1, 2, device=device),
        )
grid = make_grid(
    traj[-1, :100].view([-1, 1, 28, 28]).clip(-1, 1), value_ra
)
img = ToPILImage()(grid)
plt.imshow(img)
plt.show()
```