華南師範大學
SOUTH CHINA NORMAL UNIVERSITY

Flow-Matching

FancyAI-Tutorial系列

华师人工智能学院

FancyAI Team

范晨悠，洪铭乐，吴瑞涵

Aug. 2025

Flow Matching Guide and Code

Yaron Lipman[1], **Marton Havasi**[1], **Peter Holderrieth**[2], **Neta Shaul**[3], **Matt Le**[1], **Brian Karrer**[1], **Ricky T. Q. Chen**[1], **David Lopez-Paz**[1], **Heli Ben-Hamu**[3], **Itai Gat**[1]

[1]FAIR at Meta, [2]MIT CSAIL, [3]Weizmann Institute of Science

Flow Matching (FM) is a recent framework for generative modeling that has achieved state-of-the-art performance across various domains, including image, video, audio, speech, and biological structures. This guide offers a comprehensive and self-contained review of FM, covering its mathematical foundations, design choices, and extensions. By also providing a PyTorch package featuring relevant examples (*e.g.*, image and text generation), this work aims to serve as a resource for both novice and experienced researchers interested in understanding, applying and further developing FM.

Date: December 10, 2024

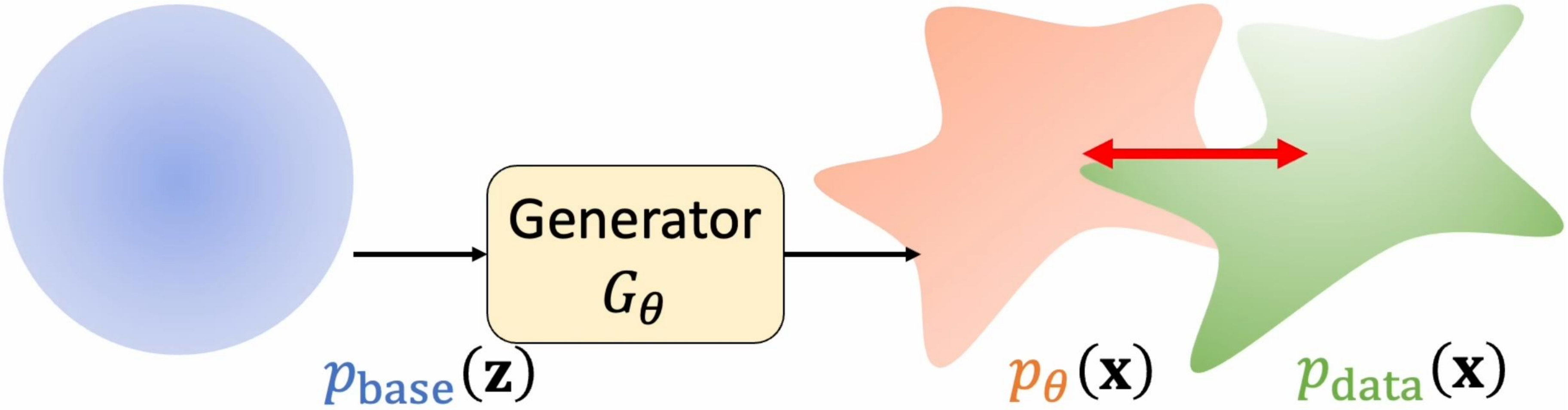
Code: `flow_matching` library at https://github.com/facebookresearch/flow_matching

Meta

生成模型

生成模型的任务目标通常是完成从某种潜在分布$p_{base}(z)$到真实数据分布$p_{data}(x)$的转换过程，通过最小化模型实际转换结果的分布$p_\theta(x)$与真实数据分布$p_{data}(x)$的 KL散度来进行拟合。一种最直接的办法便是让模型接受潜在样本z然后直接给出生成结果x，但这种方法效果往往并不佳

$$L(\theta) = \frac{1}{m}\sum_{i=1}^{m}\log p_\theta(\mathbf{x}^i)$$
$$= -D_{\mathrm{KL}}[p_{\mathrm{data}}(\mathbf{x})||p_\theta(\mathbf{x})] + C$$



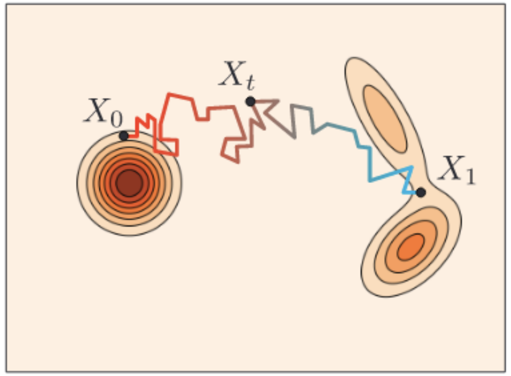
Introductio
n

扩散模型

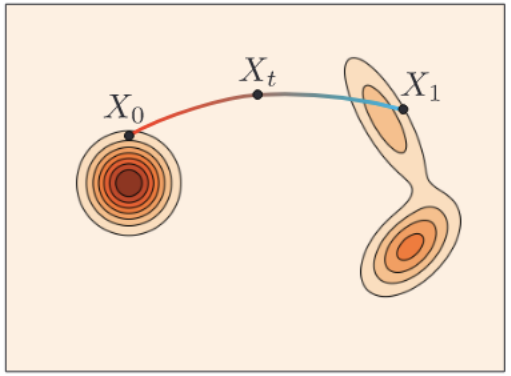
- 对原始数据进行逐步加噪，再通过模型预测噪声逐步去噪
- 适用于生成复杂数据分布，如图像和音频。

流匹配

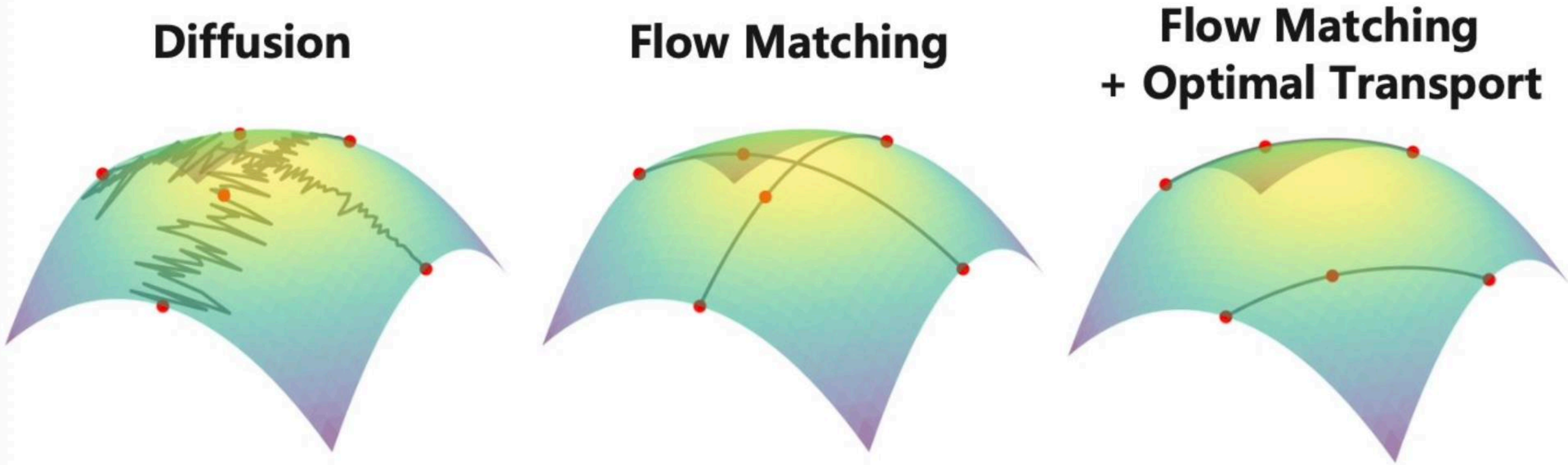
- 构建一个从噪声分布到真实数据分布的变换过程（即概率流）
- 流模型： 强调可逆性和平滑性，适用于需要精确控制数据分布转换的场景。



(b) Diffusion



(a) Flow

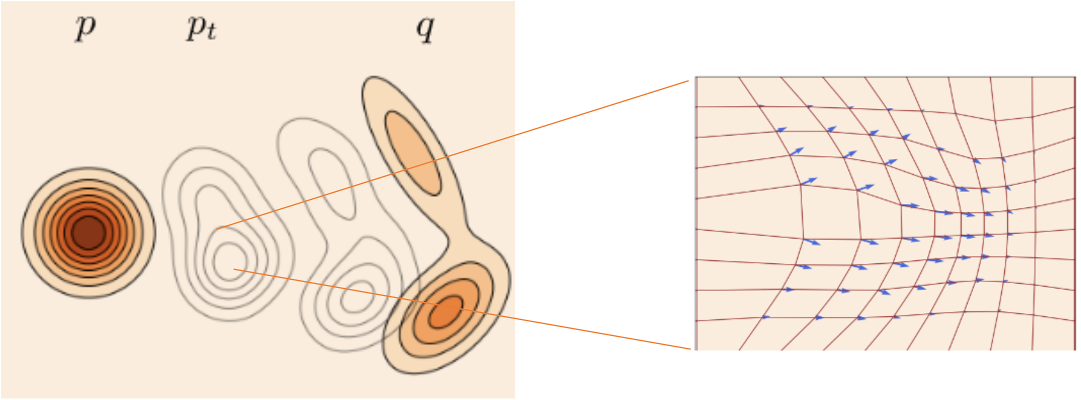

Diffusion
Flow Matching
Flow Matching
+ Optimal Transport

基本概念

向量场驱动流 → 流生成概率路径 → 实现分布变换

1. 向量场（速度场） $u_t : \mathbb{R}^d \to \mathbb{R}^d$

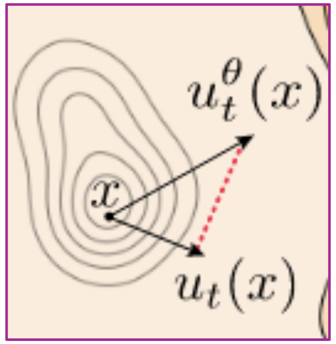
- 为每个时刻t和空间点x指定运动方向和速度
- 控制样本的演化方向（如$\frac{dx}{dt} = u_t(x)$）



2. 流（Flow） $\psi_t : \mathbb{R}^d \to \mathbb{R}^d$

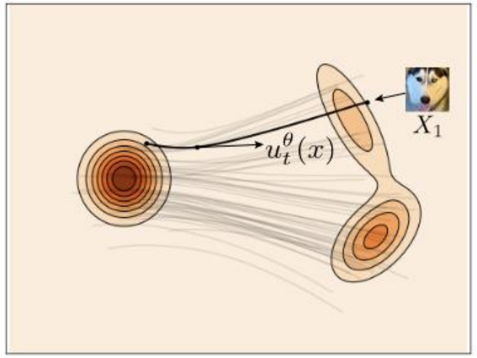
- 通过对向量场积分生成：$\psi_t(x) = x + \int_0^t u_s(\psi_s(x))ds$

- 流是速度场在时间上的积分，它描述了物体在时间 t 时的位置。

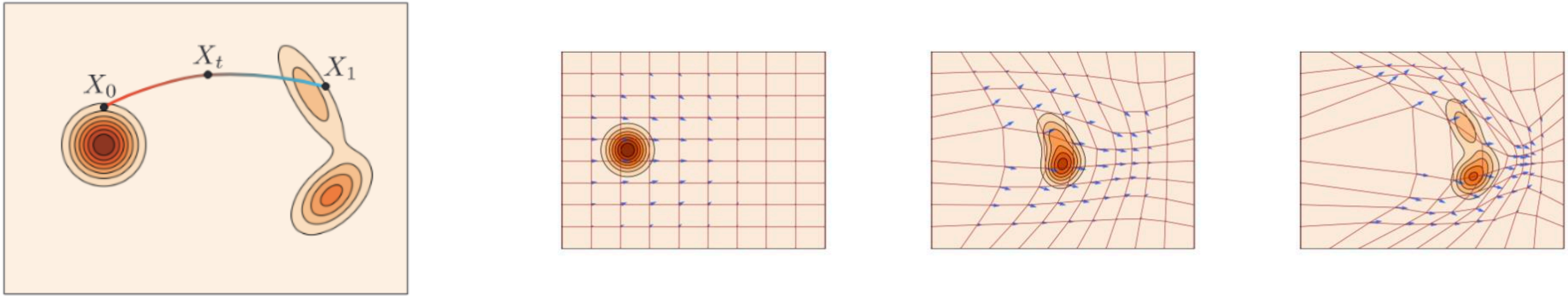


3. 概率路径 p_t（$\mathbb{R}^d$上的概率分布）

- 描述$X_t = \psi_t(X_0)$的分布（$X_0 \sim p_0$）
- 连接源分布p_0和目标分布p_1：



- Flow Matching 基于学习一个速度场 velocity field（也称为向量场）。
- 每个速度场通过求解常微分方程（ODE）来定义一个流 ψ_t。
- 流是 d 维欧几里得空间 $\mathbb{R}^d$ 的一个确定性、时间连续的双射变换。
- Flow Matching 的目标是构建一个流，将来自源分布 p 的样本 $X_0 \sim p$ 转换为目标样本 $X_1 := \psi_1(X_0)$，使得 $X_1 \sim q$ 具有所需的分布 q。



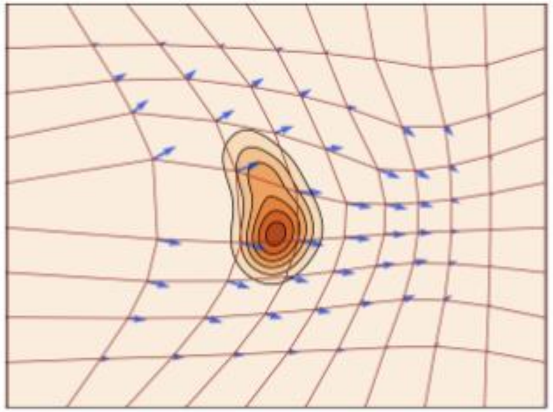
(a) Flow

Flow Matching（FM）的ODE定义

- ODE通过时间依赖的向量场 $u:[0,1]\times\mathbb{R}^d\rightarrow\mathbb{R}^d$ 定义
- 速度场 u_t 由神经网络建模　　学习参数化的模型
- 时间依赖流 $\psi:[0,1]\times\mathbb{R}^d\rightarrow\mathbb{R}^d$ 定义为：

Time-dependent Flow

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



- 初始条件：$\psi_t:=\psi(t,x)$ 和 $\psi_0(x)=x$

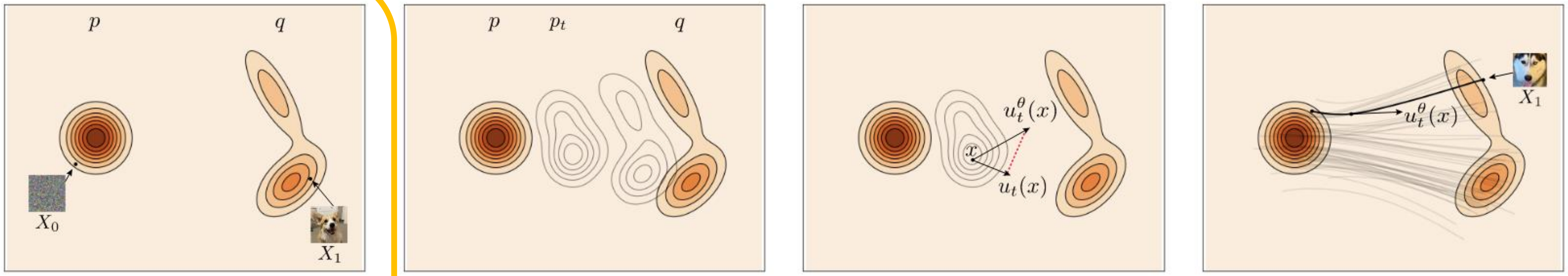
FM场中的概率路径生成

- 速度场 u_t 生成概率路径 p_t，如果其流 ψ_t 满足：

$$X_t := \psi_t(X_0) \sim p_t \text{ for } X_0 \sim p_0.$$

- 速度场 u_t 是从 p_t 采样的唯一工具
- 解ODE直到 $t = 1$ 提供样本 $X_1 = \psi_1(X_0)$，类似于目标分布 q
- 流匹配的目标是学习向量场 u_t^θ，使其流 ψ_t 生成概率路径 p_t，满足 $p_0 = p$ 和 $p_1 = q$

从数据中学习向量场 -- 1. 数据

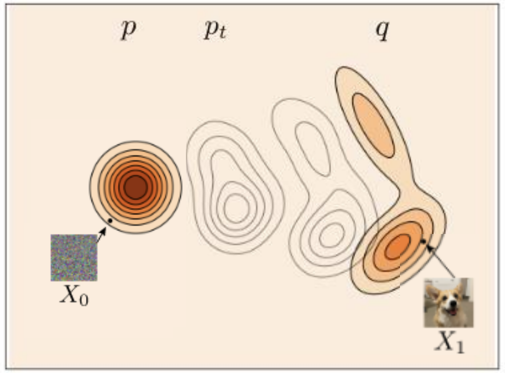


(a) Data. **(b)** Path design. **(c)** Training. **(d)** Sampling.

1. **数据准备（Data）**

- 目标是找到一种流，将已知的源分布 p（或噪声分布 q）中的样本 X_0 映射到未知的目标分布 q（或数据分布 q）中的样本 X_1。

从数据中学习向量场 -- 2. 路径构建



(b) Path design.

2. 路径设计（Path-Design）

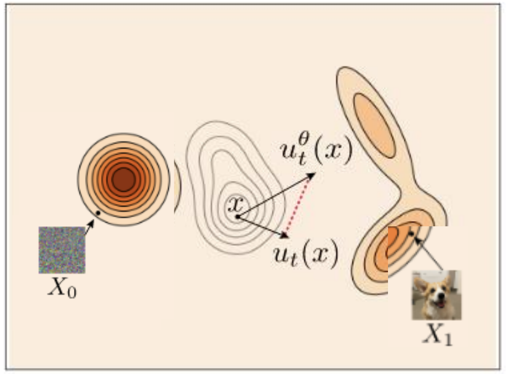
- 设计一个时间连续的概率路径 p_t（$0 \le t \le 1$），在 $p := p_0$ 和 $q := p_1$ 之间进行插值。
- 例如，源分布 $p := p_0 = N(x|0, I)$，构建概率路径 p_t 作为条件概率路径 $p_{t|1}(x|x_1)$ 的聚合，每个路径都基于训练数据集中的一个数据样本 $X_1 = x_1$。
- 概率路径 p_t 的表达式为：

$$p_t(x) = \int p_{t|1}(x|x_1)q(x_1)\mathrm{d}x_1,$$

$$p_{t|1}(x|x_1) = \mathcal{N}(x|tx_1, (1-t)^2 I).$$

$$X_t = tX_1 + (1-t)X_0 \sim p_t.$$

从数据中学习向量场 -- 3. 训练场模型

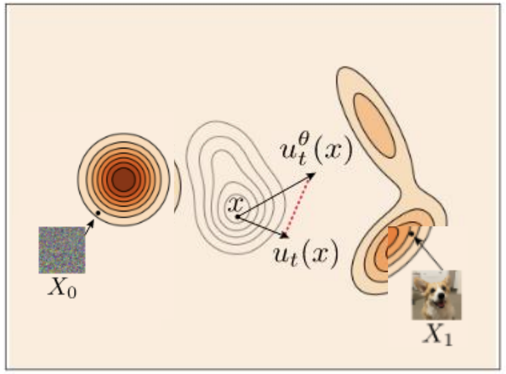


(c) Training.

3. **训练（Training）**

- 使用回归方法估计已知生成 p_t 的速度场 u_t。
- 流匹配损失函数为：$L_{FM}(\theta) = \mathbb{E}_{t,X_t}\|u_t^\theta(X_t) - u_t(X_t)\|^2$，其中 $t \sim U[0,1]$ 且 $X_t \sim p_t$。
- 实际上，由于 u_t 是一个复杂的对象，控制着两个高维分布之间的联合变换，通常很难实现上述目标。幸运的是，通过从训练集中随机选择一个目标样本 $X_1 = x_1$ 来简化损失函数，

从数据中学习向量场 -- 3. 训练场模型



(c) Training.

3. **训练（Training）** continue'd

随机选取一个目标样本 $X_1 = x_1$

在目标分布变成一个样本的情况下，那么中间时刻t的条件分布

$$X_{t|1} = tx_1 + (1-t)X_0 \quad \sim \quad p_{t|1}(\cdot|x_1) = \mathcal{N}(\cdot \mid tx_1, (1-t)^2 I).$$

接下来，我们利用流方程 $\frac{d}{dt}\psi_t(x) = u_t(\psi_t(x))$ 来推导速度场 $u_t(x|x_1)$。

利用流方程 $\frac{d}{dt}\psi_t(x) = u_t(\psi_t(x))$ 来推导速度场 $u_t(x|x_1)$

给定 $\psi_t(x)$ 是从 x 到 x_t 的映射，我们有 $x_t = \psi_t(x)$。

求解微分方程

$$\frac{d}{dt}X_{t|1} = u_t(X_{t|1}|x_1).$$

从定义 $X_{t|1} = tx_1 + (1-t)X_0$，对时间 t 求导：

$$\frac{d}{dt}X_{t|1} = x_1 - X_0.$$

$$X_0 = \frac{X_{t|1} - tx_1}{1-t}.$$

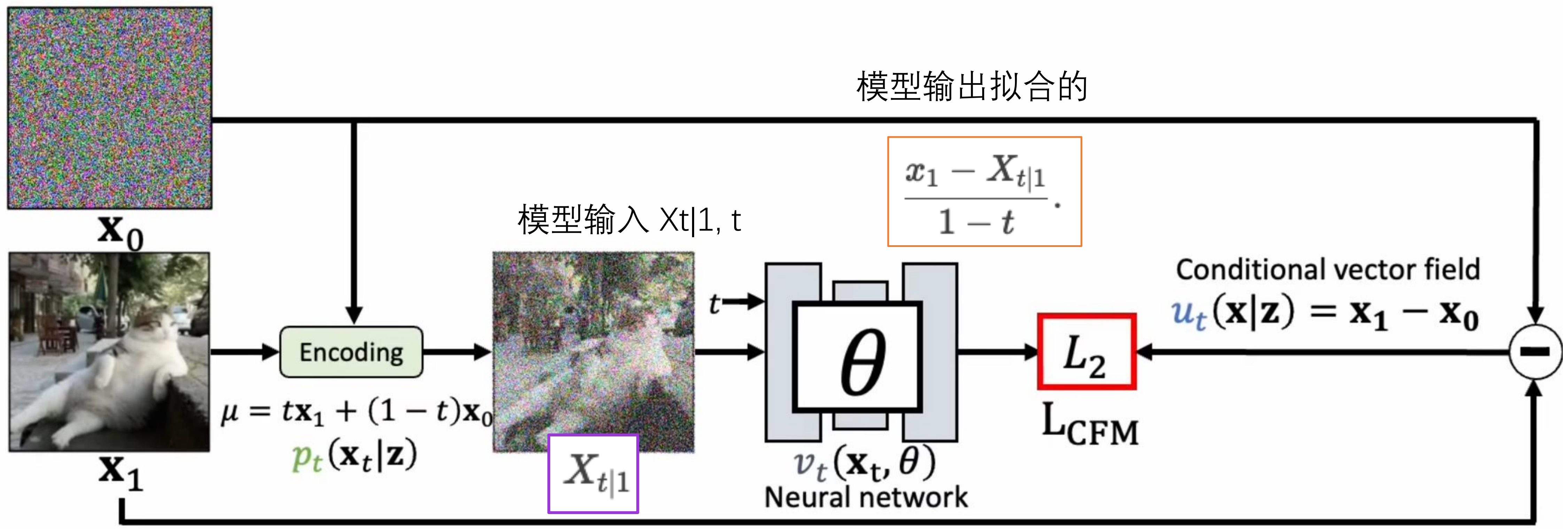
$$\frac{d}{dt}X_{t|1} = x_1 - \frac{X_{t|1} - tx_1}{1-t} = \frac{x_1 - X_{t|1}}{1-t}.$$

微分方程的解：

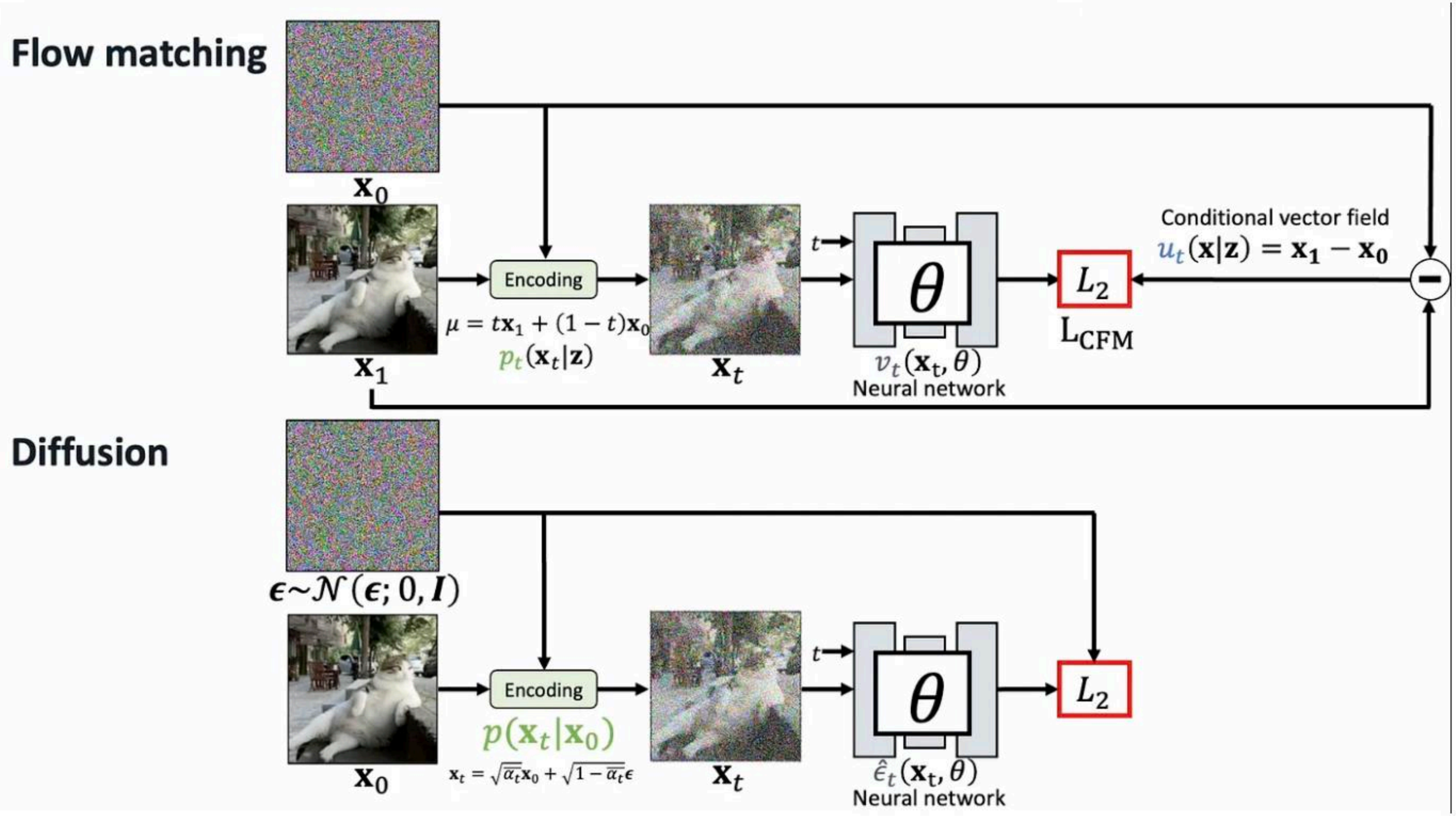
$$u_t(x|x_1) = \frac{x_1 - x}{1-t},$$

其中 $x = X_{t|1}$。

构建条件分布（2）



模型输出拟合的
$\frac{x_1 - X_{t|1}}{1-t}.$
模型输入 Xt|1, t
$\mathbf{x}_0$
$\mathbf{x}_1$
Encoding
$\mu = t\mathbf{x}_1 + (1-t)\mathbf{x}_0$
$p_t(\mathbf{x}_t|\mathbf{z})$
$X_{t|1}$
t
θ
$v_t(\mathbf{x}_t, \theta)$
Neural network
L_2
L_{CFM}
Conditional vector field
$u_t(\mathbf{x}|\mathbf{z}) = \mathbf{x}_1 - \mathbf{x}_0$



Flow matching
$\mathbf{x}_0$
$\mathbf{x}_1$
Encoding
$\mu = t\mathbf{x}_1 + (1-t)\mathbf{x}_0$
$p_t(\mathbf{x}_t|\mathbf{z})$
$\mathbf{x}_t$
t
θ
$v_t(\mathbf{x}_t, \theta)$
Neural network
L_2
L_{CFM}
Conditional vector field
$u_t(\mathbf{x}|\mathbf{z}) = \mathbf{x}_1 - \mathbf{x}_0$
Diffusion
$\epsilon \sim \mathcal{N}(\epsilon; 0, \boldsymbol{I})$
$\mathbf{x}_0$
Encoding
$p(\mathbf{x}_t|\mathbf{x}_0)$
$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$
$\mathbf{x}_t$
t
θ
$\hat{\epsilon}_t(\mathbf{x}_t, \theta)$
Neural network
L_2

推理过程（ODE ）

训练过程

Code 1: Standalone Flow Matching code

flow_matching/examples/standalone_flow_matching.ipynb

```
import torch
from torch import nn, Tensor
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons

class Flow(nn.Module):
  def __init__(self, dim: int = 2, h: int = 64):
    super().__init__()
    self.net = nn.Sequential(
      nn.Linear(dim + 1, h), nn.ELU(),
      nn.Linear(h, h), nn.ELU(),
      nn.Linear(h, h), nn.ELU(),
      nn.Linear(h, dim))

  def forward(self, x_t: Tensor, t: Tensor) -> Tensor:
    return self.net(torch.cat((t, x_t), -1))

  def step(self, x_t: Tensor, t_start: Tensor, t_end: Tensor) -> Tensor:
    t_start = t_start.view(1, 1).expand(x_t.shape[0], 1)
    # For simplicity, using midpoint ODE solver in this example
    return x_t + (t_end - t_start) * self(x_t + self(x_t, t_start) * (t_end - t_start) / 2,
                                         t_start + (t_end - t_start) / 2)

# training
flow = Flow()
optimizer = torch.optim.Adam(flow.parameters(), 1e-2)
loss_fn = nn.MSELoss()

for _ in range(10000):
  x_1   = Tensor(make_moons(256, noise=0.05)[0])
  x_0   = torch.randn_like(x_1)
  t     = torch.rand(len(x_1), 1)
  x_t   = (1 - t) * x_0 + t * x_1
  dx_t = x_1 - x_0
  optimizer.zero_grad()
  loss_fn(flow(x_t, t), dx_t).backward()
  optimizer.step()
```

训练过程

$X_{t|1} = tx_1 + (1-t)X_0$，对时间 t 求导：

```python
class Flow(nn.Module):
    def __init__(self, dim: int = 2, h: int = 64):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim + 1, h), nn.ELU(),
            nn.Linear(h, h), nn.ELU(),
            nn.Linear(h, h), nn.ELU(),
            nn.Linear(h, dim))

    def forward(self, x_t: Tensor, t: Tensor) -> Tensor:
        return self.net(torch.cat((t, x_t), -1))
```

```python
# training
flow = Flow()
optimizer = torch.optim.Adam(flow.parameters(), 1e-2)
loss_fn = nn.MSELoss()

for _ in range(10000):
    x_1  = Tensor(make_moons(256, noise=0.05)[0])
    x_0  = torch.randn_like(x_1)
    t    = torch.rand(len(x_1), 1)
    x_t  = (1 - t) * x_0 + t * x_1
    dx_t = x_1 - x_0
    optimizer.zero_grad()
    loss_fn(flow(x_t, t), dx_t).backward()
    optimizer.step()
```

用模型参数拟合

$$\frac{d}{dt}X_{t|1} = x_1 - X_0.$$

$$\frac{x_1 - X_{t|1}}{1-t}.$$

$$\mathcal{L}_{\mathrm{CFM}}^{\mathrm{OT,Gauss}}(\theta) = \mathbb{E}_{t,X_0,X_1}\|u_t^\theta(X_t) - (X_1 - X_0)\|^2, \text{ where } t \sim U[0,1],\ X_0 \sim \mathcal{N}(0,I),\ X_1 \sim q. \tag{2.9}$$

推理过程（ODE）

Compact Form (Analogous to Euler's $X_{t+h} = X_t + hu_s(X_t)$):

$$X_{t+h} = X_t + h \cdot u_s\left(X_t + \frac{h}{2} \cdot u_s(X_t)\right)$$

```python
def step(self, x_t: Tensor, t_start: Tensor, t_end: Tensor) -> Tensor:
  t_start = t_start.view(1, 1).expand(x_t.shape[0], 1)
  # For simplicity, using midpoint ODE solver in this example
  return x_t + (t_end - t_start) * self(x_t + self(x_t, t_start) * (t_end - t_start) / 2,
                                        t_start + (t_end - t_start) / 2)
```

```python

solver = ODESolver(velocity_model=velocity_model)
num_steps = 100
x_1 = solver.sample(x_init=x_0, method='midpoint', step_size=1.0 / num_steps)
```