

# 10.15-10.24 Copy

## 一、上周总结与本周计划






### 1. 上周总结

上周主要是理论知识的拓展与丰富，在了解PI-0的过程中我发现了相当多的陌生领域和知识，感觉一下子了解到自己在VLA领域其实什么都不会；上周最先开始的是数据集信息的搜集以及分析（<https://github.com/jonyzhang2023/awesome-embodied-vla-vl-vln> 在持续更新相关的工作，昊林可以在survey section找1-2篇高质量papers，看看vla领域主要研究方向都是什么。），消耗了过多的时间，导致后续出现了没时间详细阅读新知识文献的情况；已经学习的理论中，我对flow matching部分的了解仍然严重欠缺，我认为还是没有找到好的例子导致的。

### 2. 本周计划

- ☐ 阅读下图中的文献（顺序见图，[加上PI-0.5](#)），加深对概念的理解，争取在读文献的过程中加强对理论的理解，并将上周周报中的理论知识系统地迁移到知识库中，避免后续寻找不便。
- ☐ coding方面开始学习，需要有类似于张教授给的周报范例中的逻辑图（[1. high-level逻辑框架构建清楚；2. 最核心逻辑的coding implementation看明白](#)），时间不够的话先看flow matching部分。
- ☐ 了解并学习“openai的diffusion model 高质量post”（上周欠缺）（[Lilian's Posts https://lilianweng.github.io/](#)）。

标题

- >  [π0: A Vision-Language-Action Flow Model for General Robot Control](#) 1
- >  [Diffusion Policy: Visuomotor Policy Learning via Action Diffusion](#) 2
- >  [Denoising Diffusion Probabilistic Models](#)
- >  [Flow Matching for Generative Modeling](#) 3
- >  [Transfusion: Predict the Next Token and Diffuse Images with One Multi-Modal Model](#) 4

### 3. 新任务：

#### 3.1 VLA领域High-Level Overview

- 阅读survey paper（重点关注第2-3章节的领域全景图/思维导图）

- 搜索并观看VLA/具身智能相关公开讲座（国内外）
- 建立领域road map认知

### 3.2 毕业设计进行中：基于attention 机制的VLA精度优化

- 背景调研
- 阅读ReconVLA

## 二、毕设相关：Recon VLA 论文核心解读

### 1. 研究动机与问题

当前VLA模型在预测动作时存在视觉注意力分散的问题，无法精确聚焦在目标物体上，这可能导致操作错误物体。作者通过可视化发现，传统VLA在长时序任务和复杂场景中表现不佳。

#### 现有方法局限

##### 显式定位（Explicit Grounding）：


- 依赖外部分割模型
- 引入视觉冗余

##### CoT定位：

- 输出边界框坐标
- 训练困难，效果有限

### 2. 核心创新点

#### 2.1 隐式定位范式（Implicit Grounding）

 引入辅助视觉重建模块，通过轻量级扩散变换器重建目标操作区域，促使VLA模型学习细粒度表示，从而精确分配视觉注意力。

- **重建式学习**：通过重建目标区域实现隐式定位
- 无需额外输入或输出
- 直接监督视觉输出
- 促进细粒度表示学习

##### 2.1.1 隐式定位 vs 其他范式

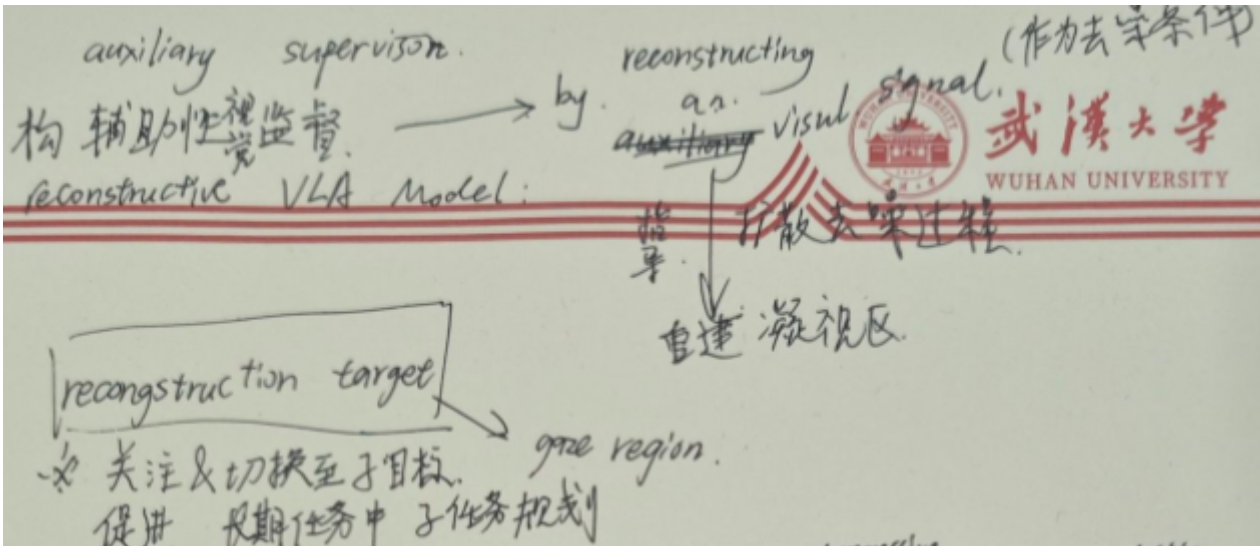
范式	方法	优点	缺点	性
显式定位	外部模型分割+裁剪图像	直观明确	视觉冗余、依赖外部模型	
CoT定位	输出边界框坐标	提供空间信息	训练困难、效果差	
隐式定位（本文）	重建凝视区域	直接监督视觉输出、无冗余	需要预训练	

## 2.2 凝视区域（Gaze Region）

👁️ **概念** 类比人眼的凝视行为，模型重建的目标是待操作的目标区域，而非整个图像，这引导模型聚焦正确目标。

类比人眼视觉机制：

- 人眼凝视：小范围清晰，周围模糊
- 目标：待操作的目标区域
- 效果：引导注意力聚焦正确目标
- 长时序任务：自适应切换不同子目标



## 2.3 大规模预训练

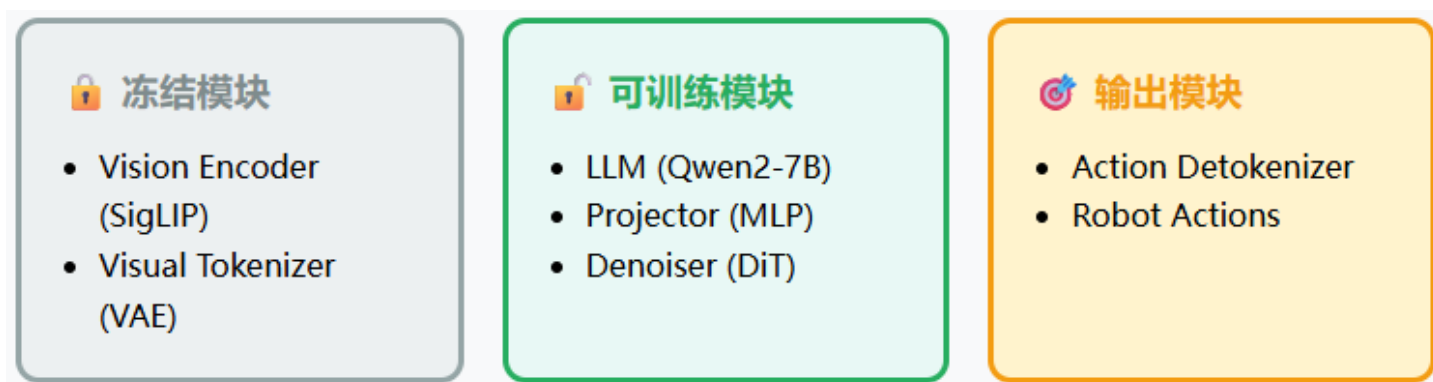
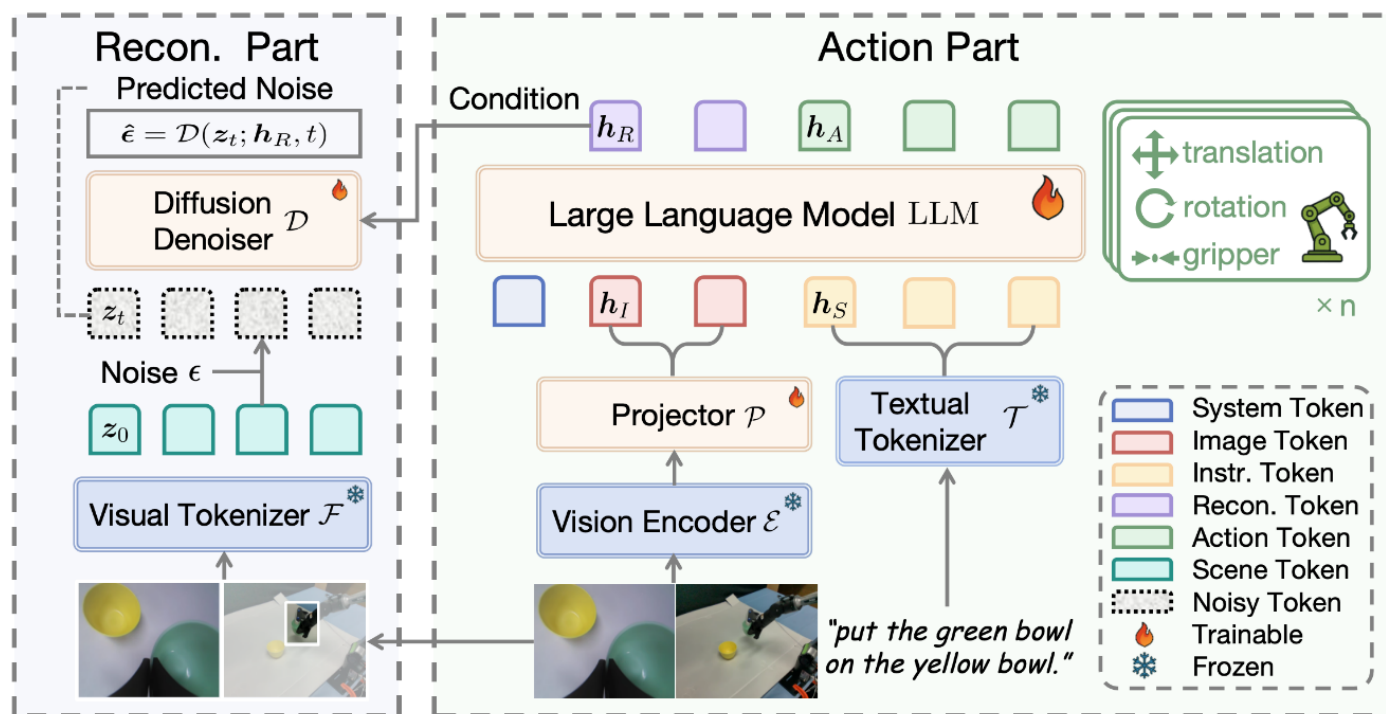
👁️ 构建了包含100k+轨迹和200万样本的预训练数据集，使用Grounding DINO自动标注凝视区域。

数据集规模：

- 100k+ 轨迹
- 200万+ 数据样本
- 来源：BridgeData V2, LIBERO, CALVIN

- 自动标注: Grounding DINO

### 3. 整体模块流程



#### 3.1 输入阶段

输入数据

- $\mathbf{I} = [\mathbf{I}_1, \mathbf{I}_2, \dots, \mathbf{I}_n]$  //  $n$ 个视角的图像
- $\mathbf{S} = \text{"put the blue block on the red one"}$  // 文本指令
- $\mathbf{I}' = \text{CropGazeRegion}(\mathbf{I}, \mathbf{S})$  // 凝视区域 (训练时)

维度信息:

$\mathbf{I}: [n, 3, H, W]$   $H=384, W=384$   $n$ : 视角数量

凝视区域获取:

- 训练时:** 使用Grounding DINO自动检测目标物体
- 推理时:** 不需要凝视区域 (只在训练时用于监督)

- **标注：**裁剪边界框区域，resize到固定大小

数据预处理

```
1 // 图像归一化
2  $I_{norm} = (I - \mu) / \sigma$ 
3 // 文本分词
4 tokens = Tokenizer(S)
5  $h_s = TextEmbed(tokens)$ 
```

处理后维度：

$I_{norm}$ : [n, 3, 384, 384]  $h_s$ : [L, d\_text] L: 文本长度 d\_text: 4096

 输入示例



## 3.2 视觉编码阶段

图像特征提取

```
1 // 使用SigLIP编码器 (冻结)
2  $h_i = VisionEncoder(I_{norm})$ 
3 // Patch embedding
4 patches = Patchify( $I_{norm}$ , patch_size=14)
5  $N_{patches} = (384 / 14)^2 = 27^2 = 729$ 
```

输出维度：

$h_i$ : [n,  $N_{patches}$ , d\_vision]  $N_{patches}$ : 729 d\_vision: 1152

关键点：

- Vision Encoder使用**预训练的SigLIP**，权重冻结
- 采用**Patch embedding**，将图像分成729个patches

- 每个patch对应一个1152维的特征向量

特征投影

```
1 // MLP投影到LLM空间
2 hi_proj = Projector(hi) hi_proj = GELU(Linear1(hi))
3 hi_proj = Linear2(hi_proj)
```

投影后维度：

$h_i\_proj: [n \times 729, d_{llm}]$   $d_{llm}: 4096$  (Qwen2-7B)

层	输入维度	输出维度
Vision Encoder	$[n, 3, 384, 384]$	$[n, 729, 1152]$
Linear <sub>1</sub>	$[n \times 729, 1152]$	$[n \times 729, 4096]$
GELU	$[n \times 729, 4096]$	$[n \times 729, 4096]$
Linear <sub>2</sub>	$[n \times 729, 4096]$	$[n \times 729, 4096]$

3.3 LLM处理阶段

Token序列构建

```
1 // 关键：指令前置！
2 h_input = [h_system, hs, hi_proj]
3 // 详细序列
4 h_input = [
5     h_system, // 系统提示 [1, dllm]
6     hs, // 指令tokens [L, dllm]
7     hi_proj // 图像tokens [n×729, dllm]
8 ]
```

总序列长度：

$T_{total} = 1 + L + n \times 729$  例：  $1 + 20 + 2 \times 729 \approx 1479$  tokens

为什么指令前置？

- 因果注意力：图像tokens能attend到指令tokens
- 任务相关：图像特征与指令语义融合
- 实验验证：这种顺序效果最好

LLM自回归推理

```
1 // Qwen2-7B处理
```

```

2  h_output = LLM(h_input)
3  // 自回归生成
4  h_output = { h1, h2, ..., ht }
5  // 每个token依赖之前所有tokens
6  ht = LLM(h1, h2, ..., ht-1)

```

输出维度：

**h\_output: [T\_total, d\_llm]**

LLM的作用：

- **多模态融合**：整合视觉、语言、动作信息
- **上下文建模**：理解长期依赖关系
- **任务理解**：根据指令调制视觉特征
- **双路输出**：生成动作token和重建token

### 3.4 双路输出阶段

Action Part（动作部分）：

- 输入：图像 + 文本指令
- 输出：离散动作token
- 自回归生成

路径A：动作预测

```

1  // 从LLM输出中提取动作tokens
2  h_action = h_output[-N_action:]
3  // 自回归生成动作token
4  p(a) = ∏i=1N p_LLM(ai | a1:i-1; h_input)
5  // Detokenizer: 离散→连续
6  A = Detokenizer(a)

```

动作空间（CALVIN）：

**A: [7] [x, y, z, roll, pitch, yaw, gripper]**

动作损失函数：

$$L_{VLA}^{action} = - \sum_{i=1}^N \log p_{LLM}(a_i | a_{1:i-1}; h_i; h_s)$$

Reconstruction Part（重建部分）：

- 输出：重建token ( $h_R$ )
- 作为扩散过程的条件
- 重建凝视区域的场景token

路径B：重建Token

```
1 // 从LLM输出中提取重建tokens
2
3  $h_R = h\_output[N\_recon\_start : N\_recon\_end]$ 
4 // 重建token作为扩散条件
5  $h_R: [N_R, d\_llm]$ 
6 //  $N_R$ : 重建token数量
```

重建Token特性：

**$N_R$ : 通常设为256 包含任务相关视觉信息**

**$h_R$ 的关键作用：**

- 条件信号：引导扩散模型重建正确区域
- 注意力媒介：必须关注目标才能重建
- 隐式监督：反向传播改善视觉特征
- 可解释性：可视化 $h_R$ 的注意力图

## 3.5 扩散重建阶段

凝视区域编码

```
1 // 使用预训练VAE编码（冻结）
2  $z_0 = VAE.encode(I')$ 
3 // VAE编码：图像→潜在空间
4  $I': [3, H', W'] \rightarrow z_0: [4, H'/8, W'/8]$ 
5
6 // 例：256×256图像  $I': [3, 256, 256] \rightarrow z_0: [4, 32, 32]$ 
```

潜在表示：

**$z_0: [B, 4, 32, 32]$  潜在维度：4096 ( $32 \times 32 \times 4$ )**

为什么在潜在空间重建？

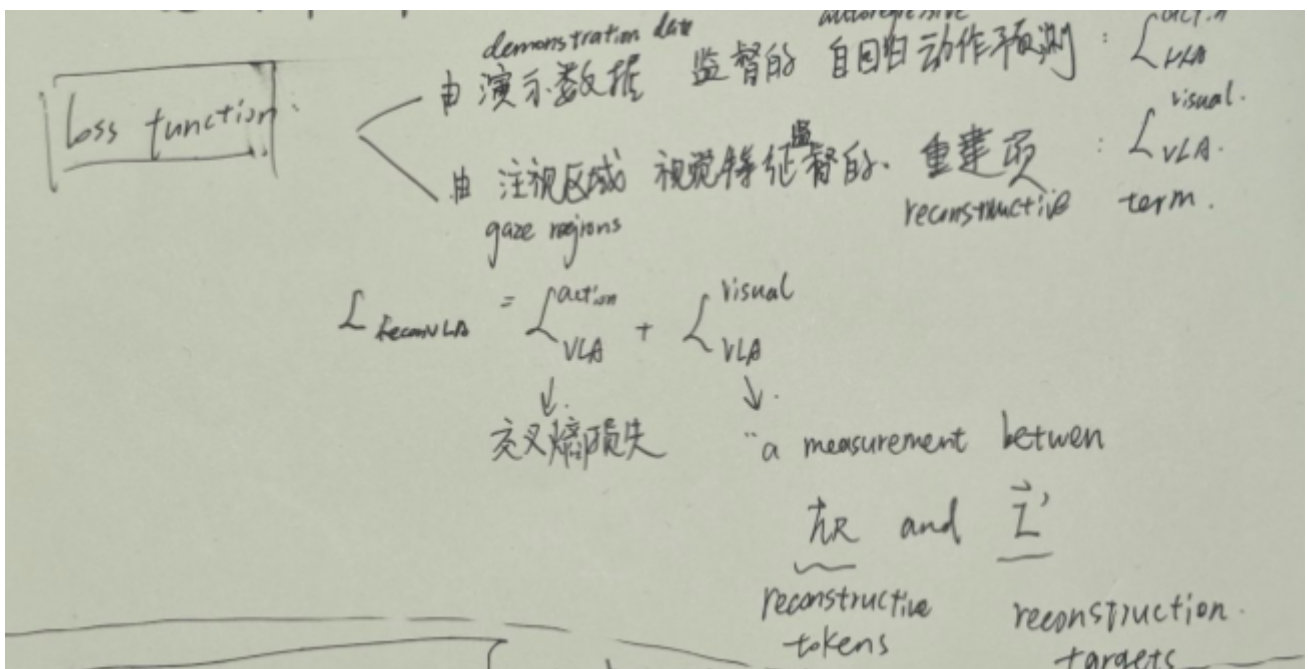
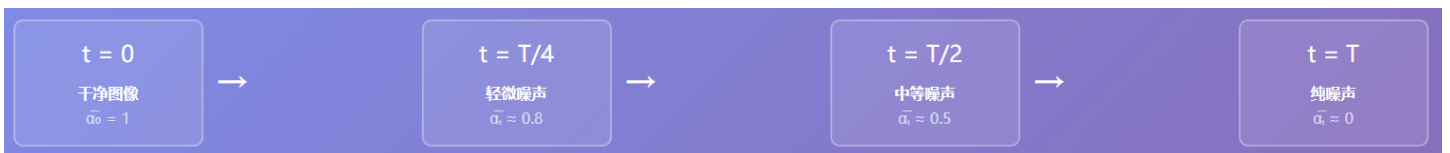
- 计算效率：潜在空间维度小64倍 ( $8 \times 8$ 压缩)
- 语义丰富：VAE学到的是语义特征，不是像素
- 视觉质量：Stable Diffusion的VAE质量很高



### 3.5.1 扩散过程数学详解

前向扩散（加噪）过程

```
1 // 逐步添加高斯噪声
2
3  $q(\mathbf{z}_t | \mathbf{z}_0) = \mathcal{N}(\mathbf{z}_t; \sqrt{\alpha_t} \cdot \mathbf{z}_0, (1 - \alpha_t) \cdot \mathbf{I})$ 
4 // 重参数化技巧
5  $\mathbf{z}_t = \sqrt{\alpha_t} \cdot \mathbf{z}_0 + \sqrt{(1 - \alpha_t)} \cdot \epsilon$ 
6 //  $\epsilon \sim \mathcal{N}(0, \mathbf{I})$  是标准高斯噪声
7 //  $\alpha_t$  是累积噪声调度系数
```



#### 总体损失函数

$$L_{ReconVLA} = L_{VLA}^{action} + L_{VLA}^{visual}$$

- $L_{VLA}^{action}$ : 交叉熵损失（动作预测）
- $L_{VLA}^{visual}$ : 扩散重建损失（视觉重建）

#### 扩散损失

$$L_{VLA}^{visual} = E_{t,\epsilon} [\|D(z_t; h_R, t) - \epsilon\|^2]$$

逆向去噪（重建）过程

```
1 // 训练目标: 预测噪声
2  $L(\text{visual})(VLA)(h_R, \mathbf{I}') = \mathbb{E}(t, \epsilon) [\|D(\mathbf{z}_t; h_R, t) - \epsilon\|^2]$ 
3 // 去噪器D预测噪声
```

```

4   $\epsilon$  = D(z_t; h_R, t)
5  // 从预测的噪声恢复z_0
6   $\hat{z}_0 = (z_t - \sqrt{(1-\alpha_t)} \cdot \epsilon) / \sqrt{\alpha_t}$ 
7  // 迭代去噪 (DDIM采样)
8   $z_{t-1} = \sqrt{\beta_{t-1}} \cdot \hat{z}_0 + \sqrt{(1-\beta_{t-1})} \cdot \epsilon$ 

```

### 关键参数：

- **t**: 扩散时间步，训练时随机采样  $t \sim \text{Uniform}(1, T)$
- **T**: 总扩散步数，训练时T=1000
- **$\alpha_t$** : 累积噪声系数，使用cosine schedule
- **$\epsilon$** : 标准高斯噪声， $\epsilon \sim \mathcal{N}(0, I)$

## 3.6 去噪器结构

输入处理

```

1  // 展平潜在token
2  z_t_flat = Flatten(z_t)           z_t_flat: [B, 32×32, 4] = [B, 1024, 4]
3  // 时间步嵌入 (Sinusoidal)
4  t_emb = TimestepEmbed(t)          t_emb: [B, d_time] // d_time = 256
5  // 拼接噪声token和条件token
6  x = Concat([z_t_flat, h_R], dim=1) x: [B, 1024+256, d]

```

时间步嵌入：

$$PE(t, 2i) = \sin(t / 10000^{(2i/d_{time})})$$

$$PE(t, 2i + 1) = \cos(t / 10000^{(2i/d_{time})})$$

Transformer处理

```

1  // Diffusion Transformer Block
2  for layer in self.layers:
3  // 1. 自注意力
4      x = x + SelfAttention(x)
5  // 2. 时间条件 (AdaLN)
6      scale, shift = AdaLN(t_emb) x = x * scale + shift
7  // 3. FFN
8      x = x + FFN(x)
9  // 提取噪声预测
10      $\epsilon$  = x[:, :1024, :] // 只取前1024个token

```

模块	参数	说明
Transformer层数	12	平衡性能与效率
注意力头数	16	Multi-head attention
隐藏维度	768	d_model
FFN维度	3072	4×d_model

### 3.7 联合训练与优化（Joint Training）

总体损失函数

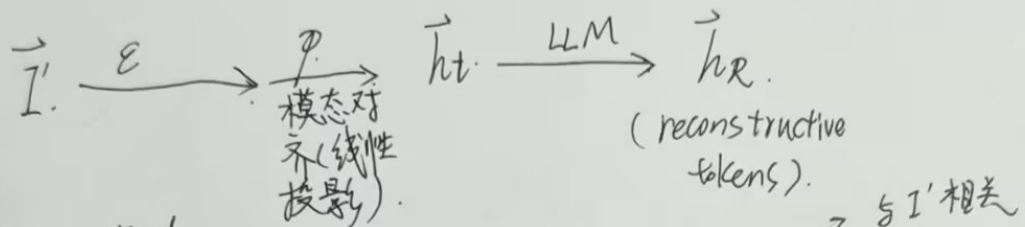
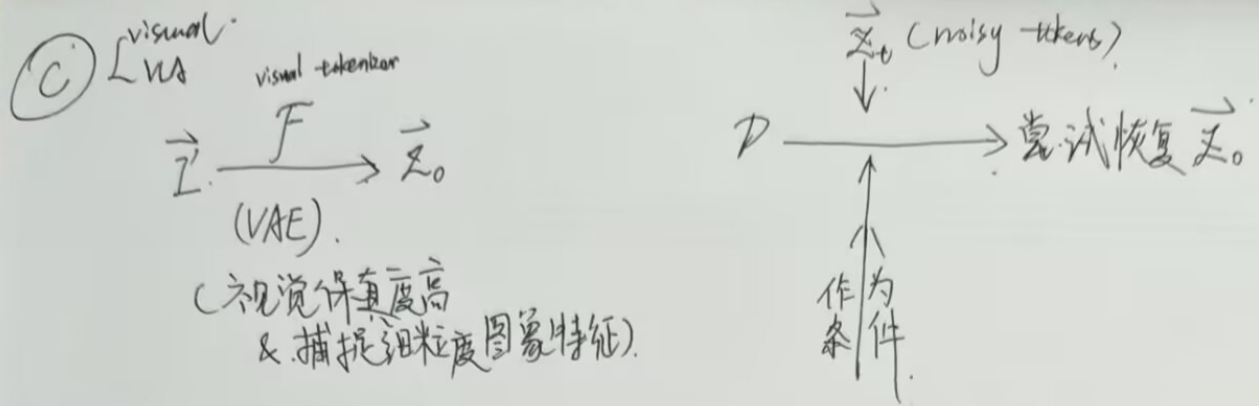
```
1 // ReconVLA的完整训练目标
2 L(ReconVLA) = L(action)(VLA) + λ · L(visual)(VLA)
3 // 展开形式
4 L(ReconVLA) = - ∑i log p(ai|a1:i-1; hi; hs) + λ · E(t,ε) [ || D(z_t; h_R, t) - ε ||2 ]
```

#### 动作损失 L<sup>action</sup>

- 交叉熵损失
- 监督动作预测
- 确保操作正确性

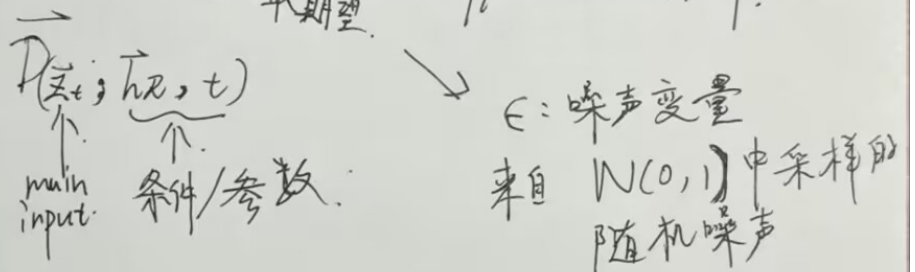
#### 视觉损失 L<sup>visual</sup>

- MSE（噪声预测）
- 监督视觉重建
- 引导注意力分配



$\mathcal{L}_{VA}^{visual}(\vec{h}_R, \vec{I}') = E_{t, \epsilon} [\| \mathcal{D}(\vec{z}_t; \vec{h}_R, t) - \epsilon \|^2]$

对  $t, \epsilon$  取期望  $t$ : diffusion timestep



\* ReconULA

使用因果注意力 causal

1. 信息流动方向正确

指令 token  $\rightarrow$  视觉 token 处理

2. 训练时与推理时逻辑相同, 都是先(只)看得到指令 token, 后处理视觉 token, 推理动作 token

步骤: every iteration:

(DPPM 标准范式)

1. 随机采样时间步  $t \sim \text{Uniform}(0, 1)$
2. 随机采样噪声:  $\epsilon \sim N(0, 1)$
3. 生成加噪 tokens:  $\vec{z}_t = \text{加噪}(\vec{z}_0, \epsilon, t)$  "noisy"
4. 预测噪声:  $\hat{\epsilon} = \mathcal{D}(\vec{z}_t; \vec{h}_R, t)$
5. 计算损失  ~~$\|\hat{\epsilon} - \epsilon\|^2$~~   $\|\hat{\epsilon} - \epsilon\|^2$

### 🔑 $\lambda$ 的选择 (超参数):

- 预训练:  $\lambda = 1.0$  (两个任务同等重要)
- 微调:  $\lambda = 0.5$  (更侧重动作)

- 调参原则：确保两个loss在同一数量级

梯度反向传播

```
1 // 计算梯度
2  $\nabla_{\theta} L(\text{ReconVLA}) = \nabla_{\theta} L(\text{action}) + \lambda \cdot \nabla_{\theta} L(\text{visual})$ 
3 // 两个损失都对LLM参数求导
4  $\nabla(\text{LLM}) L(\text{ReconVLA}) = \nabla(\text{LLM}) L(\text{action}) + \lambda \cdot \nabla(\text{LLM}) L(\text{visual})$ 
5 // 视觉损失通过h_R反向传播到LLM
6  $\nabla(\text{LLM}) L(\text{visual}) = \nabla(h_R) L(\text{visual}) \cdot \nabla(\text{LLM}) h_R$ 
```

### 关键机制：

- 视觉重建损失通过 `h_R` 反向传播到LLM
- 如果 `h_R` 不包含目标信息，重建失败，损失大
- 梯度迫使LLM在生成 `h_R` 时关注目标区域
- 注意力权重被优化，自然聚焦到凝视区域

优化算法

```
1 // AdamW优化器
2  $m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot \nabla_{\theta}$ 
3  $v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot (\nabla_{\theta} L)^2$ 
4 // 参数更新 (含权重衰减)
5  $\theta = \theta - \eta \cdot (m_t / \sqrt{v_t} + \lambda_{wd} \cdot \theta)$ 
6 // 学习率warmup + cosine decay
7  $\eta(\text{step}) = \{ \eta_{\text{base}} \cdot \text{step} / \text{warmup\_steps}, \text{step} \leq \text{warmup\_steps} \}$   
 $\eta(\text{step}) = \{ 0.5 \cdot \eta_{\text{base}} \cdot (1 + \cos(\pi \cdot \text{progress})), \text{step} > \text{warmup\_steps} \}$ 
```

超参数	预训练	微调
学习率 $\eta_{\text{base}}$	1.00E-04	5.00E-05
Batch size	256	64
Warmup steps	2000	500
Weight decay $\lambda_{\text{wd}}$	0.01	0.01
$\beta_1, \beta_2$	0.9, 0.999	0.9, 0.999
梯度裁剪	1	1

## 完整流程

### 前向传播

1. 输入图像I和指令S
2. Vision Encoder提取特征 $h_I$
3. Projector投影到LLM空间
4. LLM融合多模态信息
5. 双路输出：动作a和重建 $h_R$
6. VAE编码凝视区域 $I' \rightarrow z_0$
7. 扩散加噪 $z_0 \rightarrow z_t$
8. Denoiser预测噪声 $\epsilon$

### 反向传播

1. 计算动作损失 $L^{action}$
2. 计算重建损失 $L^{visual}$
3. 联合损失 $L = L^{action} + \lambda \cdot L^{visual}$
4. 反向传播到Denoiser
5. 梯度通过 $h_R$ 传到LLM
6. 更新LLM参数（注意力层）
7. 注意力自然聚焦目标
8. 实现精确操作

### 3.7.1 训练策略

两阶段训练：

- 预训练阶段：
  - 大规模数据集
  - 学习通用重建能力
  - 同时训练动作和重建
- 微调阶段：
  - 特定任务数据
  - 对齐操作能力
  - 适配动作空间

## 4. Reconstruction部分详解

👁 Latent Visual Reconstruction = VAE编码器 + 扩散去噪器 + 重建Token( $h_R$ )

#### 1 Visual Tokenizer

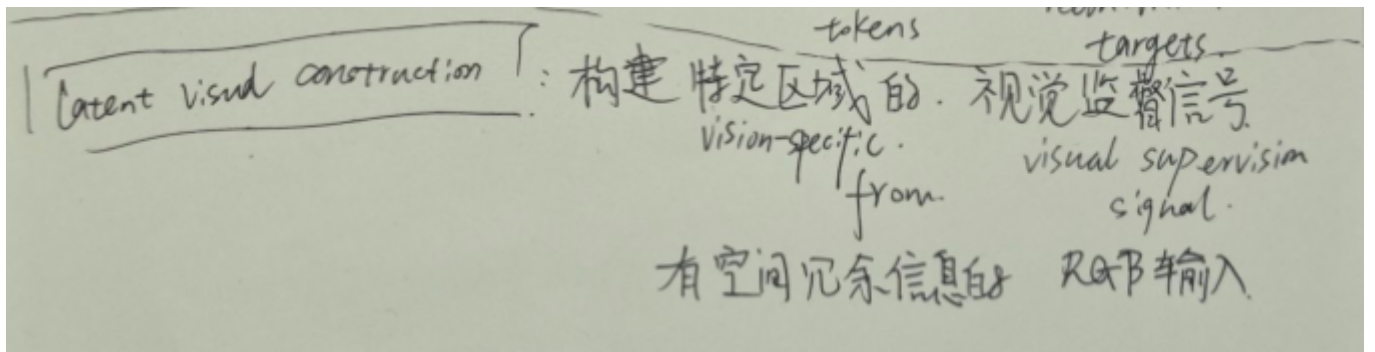
- 模型：VAE
- 来源：Stable Diffusion
- 作用：图像 $\leftrightarrow$ 潜在表示
- 状态：冻结权重

#### 2 Denoiser

- 架构：DiT
- 输入：噪声 $z_t + h_R$
- 输出：噪声预测 $\epsilon$
- 状态：可训练

#### 3 重建Token $h_R$

- 来源：LLM输出
- 维度：[256, 4096]
- 作用：扩散条件
- 关键：注意力媒介



完整重建流程：

#### 重建流程

```

1  // Step 1: 编码凝视区域到潜在空间
2   $\mathbf{z}_0 = \mathcal{F}(\mathbf{I}')$  //  $\mathbf{I}': [3, 256, 256] \rightarrow \mathbf{z}_0: [4, 32, 32]$ 
3
4  // Step 2: 前向扩散 (加噪)
5   $\mathbf{z}_t = \sqrt{\alpha_t} \cdot \mathbf{z}_0 + \sqrt{(1 - \alpha_t)} \cdot \boldsymbol{\epsilon}$  //  $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), t \sim \text{Uniform}(1, T)$ 
6
7  // Step 3: 去噪器预测噪声 (条件:  $\mathbf{h}_R$ )
8   $\boldsymbol{\epsilon} = \mathcal{D}(\mathbf{z}_t; \mathbf{h}_R, t)$ 
9
10 // Step 4: 计算重建损失
11  $\mathcal{L}(\text{visual})(\text{VLA})(\mathbf{h}_R, \mathbf{I}') = \mathbb{E}(t, \boldsymbol{\epsilon}) [\| \mathcal{D}(\mathbf{z}_t; \mathbf{h}_R, t) - \boldsymbol{\epsilon} \|^2]$ 
12
13
14 总结:
15
16 凝视区域  $\mathbf{I}' [3, 256, 256]$ 
17     ↓ VAE Encoder (冻结)
18 潜在表示  $\mathbf{z}_0 [4, 32, 32] \leftarrow 48\times$ 压缩!
19     ↓ 加噪声
20 噪声版本  $\mathbf{z}_t [4, 32, 32]$ 
21     ↓ Denoiser (条件:  $\mathbf{h}_R$ )
22 预测噪声  $\boldsymbol{\epsilon} [4, 32, 32]$ 
23     ↓ 计算损失
24  $\mathcal{L}^{\text{visual}} = \| \quad - \boldsymbol{\epsilon} \|^2$ 
25     ↓ 反向传播
26 更新LLM → 优化注意力
  
```

其中：

符号	含义	维度
$\mathcal{F}$	Visual Tokenizer (VAE Encoder)	$[3, H, W] \rightarrow [4, H/8, W/8]$
$I'$	凝视区域图像	$[3, 256, 256]$
$z_0$	潜在表示 (场景token)	$[4, 32, 32]$
$z_t$	加噪后的潜在表示	$[4, 32, 32]$
$h_R$	重建token (LLM输出)	$[256, 4096]$
$\mathcal{D}$	Denoiser (Diffusion Transformer)	-
$\epsilon$	高斯噪声	$[4, 32, 32]$
$\alpha_{\boxtimes}$	累积噪声系数	标量

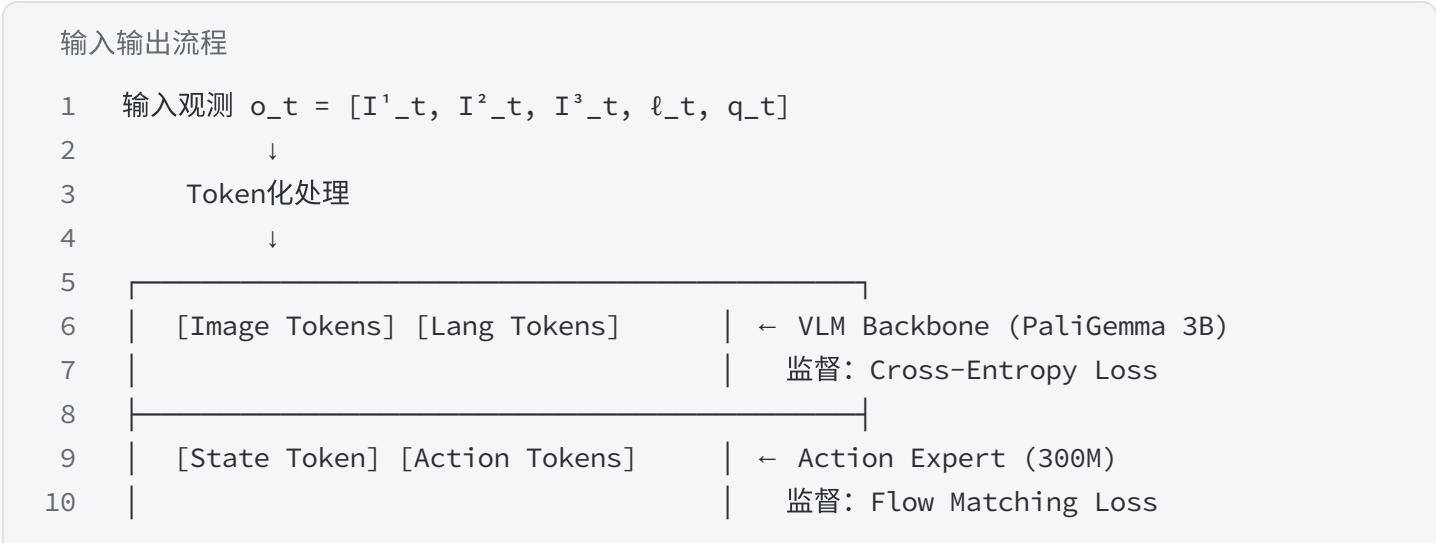
总体上来看：

维度	像素空间	潜在空间
分辨率	256×256	32×32
通道数	3 (RGB)	4
总维度	196,608	4,096
压缩比	1×	48×

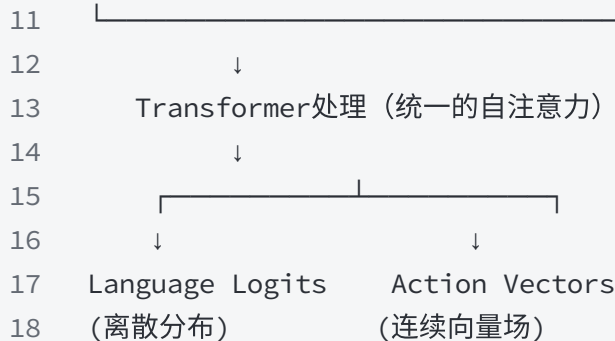
### 三、PI-0/0.5 深入学习：π0的混合监督机制详解

#### 1. 整体架构概览

##### 1.1 输入输出流程







## 2. Flow Matching Loss (连续动作监督)

### 2.1 核心数学框架

#### 2.1.1 目标分布建模

目标：学习条件分布  $p(\mathbf{A}_t | \mathbf{o}_t)$

其中：

- $\mathbf{A}_t = [\mathbf{a}_t, \mathbf{a}_{t+1}, \dots, \mathbf{a}_{t+H-1}]$ : 动作序列 (action chunk) ,  $H=50$
- $\mathbf{o}_t = [l^1_t, \dots, l^n_t, \ell_t, q_t]$ : 观测
  - $l^i_t$ : 第*i*个摄像头图像
  - $\ell_t$ : 语言指令 (token序列)
  - $q_t$ : 机器人关节状态 (维度随机器人而定)

#### 2.1.2 Flow Matching损失函数

论文原文第5页给出了完整公式：

$$L^\tau(\theta) = \mathbb{E}_{p(\mathbf{A}_t | \mathbf{o}_t), q(\mathbf{A}_t^\tau | \mathbf{A}_t)} \|\mathbf{v}_\theta(\mathbf{A}_t^\tau, \mathbf{o}_t) - \mathbf{u}(\mathbf{A}_t^\tau | \mathbf{A}_t)\|^2$$

参数解释：

- $\tau \in [0, 1]$ : flow matching时间步 (不是真实时间)
- $\mathbf{A}^\tau_t$ :  $\tau$ 时刻的"噪声动作"
- $\mathbf{v}_\theta(\mathbf{A}^\tau_t, \mathbf{o}_t)$ : 神经网络预测的向量场
- $\mathbf{u}(\mathbf{A}^\tau_t | \mathbf{A}_t)$ : 真实的去噪向量场

### 2.2 概率路径设计

#### 2.2.1 Linear-Gaussian路径 (Optimal Transport)

$$q(\mathbf{A}_t^\tau | \mathbf{A}_t) = \mathcal{N}(\tau \mathbf{A}_t, (1 - \tau)^2 \mathbf{I})$$

实际采样过程：

$$\mathbf{A}_t^\tau = \tau \mathbf{A}_t + (1 - \tau) \epsilon, \quad \text{其中 } \epsilon \sim \mathcal{N}(0, \mathbf{I})$$

向量场目标：

$$\mathbf{u}(\mathbf{A}_t^\tau | \mathbf{A}_t) = \epsilon - \mathbf{A}_t$$

物理意义：

- $\tau=0$ ：纯噪声  $\mathbf{A}^0_t = \epsilon$
- $\tau=1$ ：真实动作  $\mathbf{A}^1_t = \mathbf{A}_t$
- 中间过程：从噪声平滑过渡到真实动作

## 2.3 训练实现细节

### 2.3.1 伪代码实现

代码块

```

1  def flow_matching_loss(model, observations, actions, timestep_sampler):
2      """
3      Args:
4          observations: dict with keys ['images', 'language', 'state']
5          actions: [B, H, action_dim] # H=50, action_dim取决于机器人
6          timestep_sampler: Beta分布采样器
7      """
8      batch_size = actions.shape[0]
9
10     # 1. 采样flow matching时间步  $\tau \sim \text{Beta}(1.5, 1)$ , 截断到[0, 0.999]
11     tau = timestep_sampler.sample(batch_size) # [B]
12
13     # 2. 采样高斯噪声
14     epsilon = torch.randn_like(actions) # [B, H, action_dim]
15
16     # 3. 构造噪声动作  $\mathbf{A}^{\tau}_t = \tau \mathbf{A}_t + (1-\tau) \epsilon$ 
17     tau_expanded = tau.view(-1, 1, 1) # [B, 1, 1]
18     noisy_actions = tau_expanded * actions + (1 - tau_expanded) * epsilon
19
20     # 4. 预测向量场  $v\theta(\mathbf{A}^{\tau}_t, o_t)$ 
21     predicted_velocity = model(
22         images=observations['images'],
23         language=observations['language'],
24         state=observations['state'],
25         noisy_actions=noisy_actions,
26         tau=tau
27     )
28
29     # 5. 计算目标向量场  $\mathbf{u}(\mathbf{A}^{\tau}_t | \mathbf{A}_t) = \epsilon - \mathbf{A}_t$ 
30     target_velocity = epsilon - actions

```

```

31
32     # 6. MSE损失
33     loss = F.mse_loss(predicted_velocity, target_velocity)
34
35     return loss

```

### 2.3.2 Flow Matching时间步采样策略

论文原文 Appendix B（第15-16页）强调了特殊的采样分布：

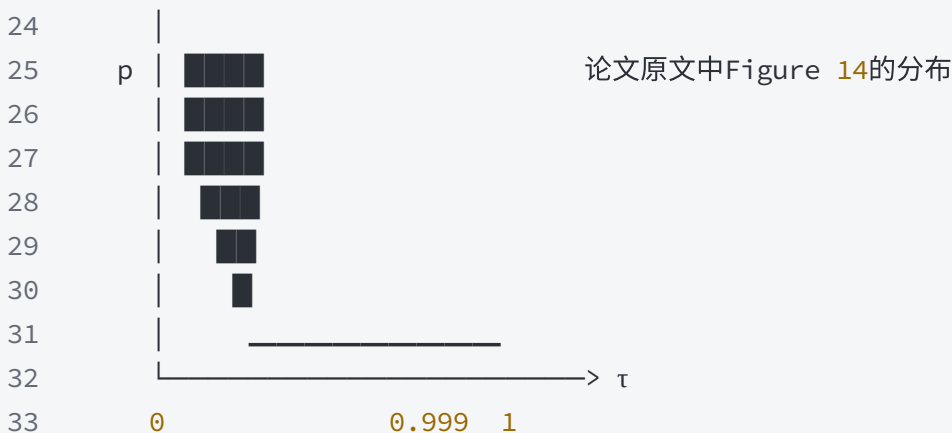
代码块

```

1  class FlowMatchingTimestepSampler:
2      def __init__(self, s=0.999, alpha=1.5, beta=1.0):
3          """
4          Args:
5              s: 截断阈值（不采样  $\tau > s$  的时间步）
6              alpha, beta: Beta分布参数
7          """
8          self.s = s
9          self.beta_dist = torch.distributions.Beta(alpha, beta)
10
11     def sample(self, batch_size):
12         # 采样  $u \sim \text{Beta}(1.5, 1)$ 
13         u = self.beta_dist.sample((batch_size,))
14
15         # 映射到  $[0, s]$ :  $\tau = s - s*u$ 
16         tau = self.s - self.s * u
17
18         return tau.to(device)
19     ...
20
21     **为什么这样设计？**（论文原文第16页解释）
22     ...

```

23  $p(\tau)$  的形状（强调低 $\tau$ ，即高噪声）：



35 原因：

```
36 1. 低 $\tau$ （高噪声）：学习数据的整体分布结构（最难）
37 2. 中 $\tau$ ：学习细节调整
38 3. 高 $\tau$ （低噪声）：学习最终精修（相对简单）
39
40 截断 $s=0.999$ :
41 - 允许 $\delta > 0.001$ 的积分步长
42 - 最后一点噪声可以通过积分自然消除
```

## 2.4 推理过程 (Sampling)

### 2.4.1 前向欧拉积分

代码块

```
1  def sample_actions(model, observations, num_steps=10):
2      """
3      通过积分向量场生成动作
4
5      Args:
6          observations: 当前观测
7          num_steps: 积分步数（论文使用10步）
8      """
9      delta = 1.0 / num_steps  #  $\delta = 0.1$ 
10
11     # 1. 从纯噪声开始  $A^0 \sim N(0, I)$ 
12     actions = torch.randn(1, 50, action_dim).to(device)
13
14     # 2. 迭代积分:  $A^{\tau+\delta} = A^\tau + \delta * v_\theta(A^\tau, o)$ 
15     for step in range(num_steps):
16         tau = step * delta
17
18         # 预测向量场（可以缓存观测的keys/values）
19         with torch.no_grad():
20             velocity = model(
21                 images=observations['images'],
22                 language=observations['language'],
23                 state=observations['state'],
24                 noisy_actions=actions,
25                 tau=torch.tensor([tau]).to(device)
26             )
27
28         # 欧拉更新
29         actions = actions + delta * velocity
30
31     return actions  # [1, 50, action_dim]
```

## 计算效率优化（论文原文 Appendix D）：

代码块

```
1  class CachedFlowMatching:
2      def __init__(self, model):
3          self.model = model
4          self.cached_kv = None # 缓存观测的K/V矩阵
5
6      def sample(self, observations, num_steps=10):
7          # 只在第一步编码观测
8          if self.cached_kv is None:
9              self.cached_kv = self.model.encode_observations(observations)
10
11         actions = torch.randn(1, 50, action_dim)
12
13         for step in range(num_steps):
14             tau = step / num_steps
15             # 只需前向传播action tokens
16             velocity = self.model.predict_velocity_fast(
17                 actions, tau, cached_kv=self.cached_kv
18             )
19             actions = actions + velocity / num_steps
20
21         return actions
22     ...
23
24     **推理时间分析**（论文原文 Table I）：
25     ...
26     Total on-board: 73ms
27     |— Image encoders: 14ms
28     |— Observation forward: 32ms ← 缓存后只需一次
29     |— 10x Action forward: 27ms ← 每次2.7ms
```

## 3. Cross-Entropy Loss（离散文本监督）

### 3.1 语言Token的处理

虽然论文原文没有明确展开cross-entropy loss的公式，但从架构描述可以推断：

#### 3.1.1 标准语言建模目标

$$\mathcal{L}_{\text{language}} = - \sum_{i=1}^{|\ell|} \log p(\ell_i \mid \ell_{<i}, I_t, q_t)$$

其中：

- $\ell_i$ : 第 $i$ 个语言token
- $\ell_{\{<i\}}$ : 之前的语言上下文
- $l_t, q_t$ : 视觉和状态条件

### 3.1.2 实现代码

代码块

```
1  def cross_entropy_loss(model, observations, language_targets):
2      """
3      Args:
4          language_targets: [B, seq_len] 语言token ID
5      """
6      # 1. 前向传播获得语言logits
7      outputs = model(
8          images=observations['images'],
9          language=observations['language'], # 输入的语言prompt
10         state=observations['state']
11     )
12
13     # 2. 提取语言token位置的输出
14     language_logits = outputs['language_logits'] # [B, seq_len, vocab_size]
15
16     # 3. 标准交叉熵损失
17     loss = F.cross_entropy(
18         language_logits.view(-1, vocab_size),
19         language_targets.view(-1),
20         ignore_index=PAD_TOKEN_ID
21     )
22
23     return loss
```

## 3.2 在 $\pi_0$ 中的角色

基于论文原文第3-4页的描述：

代码块

```
1  class Pi0Model(nn.Module):
2      def __init__(self):
3          # 1. VLM backbone (处理图像+语言)
4          self.vlm_backbone = PaliGemma3B() # 预训练的VLM
5
6          # 2. Action expert (处理状态+动作)
```

```

7         self.action_expert = ActionExpert(dim=1024, mlp_dim=4096)
8
9         # 3. 输出头
10        self.language_head = nn.Linear(2048, vocab_size) # 离散
11        self.action_head = nn.Linear(1024, action_dim) # 连续
12
13    def forward(self, images, language, state, noisy_actions=None, tau=None):
14        # Tokenize inputs
15        img_tokens = self.image_encoder(images)
16        lang_tokens = self.lang_embedding(language)
17        state_tokens = self.state_projection(state)
18
19        if noisy_actions is not None:
20            # 训练模式: 包含action tokens
21            action_tokens = self.action_embedding(noisy_actions, tau)
22            all_tokens = [img_tokens, lang_tokens, state_tokens, action_tokens]
23        else:
24            # 推理模式: 只编码观测
25            all_tokens = [img_tokens, lang_tokens, state_tokens]
26
27        # 路由到不同expert
28        vlm_tokens = img_tokens + lang_tokens
29        action_tokens_input = state_tokens + action_tokens
30
31        # Transformer with mixture of experts
32        hidden = self.transformer_with_experts(all_tokens)
33
34        # 双输出
35        lang_logits = self.language_head(hidden['language_positions'])
36        action_velocity = self.action_head(hidden['action_positions'])
37
38        return {
39            'language_logits': lang_logits,
40            'action_velocity': action_velocity
41        }

```

## 4. 混合训练流程

### 4.1 完整训练步骤

代码块

```

1  def train_step(model, batch, optimizer):
2      """
3      单步训练包含两个损失
4      """

```

```

5      # 解包batch
6      observations = {
7          'images': batch['images'],      # [B, num_cams, H, W, 3]
8          'language': batch['language'],  # [B, seq_len]
9          'state': batch['state']         # [B, state_dim]
10     }
11     actions = batch['actions']           # [B, 50, action_dim]
12     language_targets = batch['language_targets'] # [B, target_len]
13
14     # ===== 1. Flow Matching Loss =====
15     loss_action = flow_matching_loss(
16         model=model,
17         observations=observations,
18         actions=actions,
19         timestep_sampler=beta_sampler
20     )
21
22     # ===== 2. Cross-Entropy Loss =====
23     outputs = model(
24         images=observations['images'],
25         language=observations['language'],
26         state=observations['state']
27     )
28     loss_language = F.cross_entropy(
29         outputs['language_logits'].view(-1, vocab_size),
30         language_targets.view(-1)
31     )
32
33     # ===== 3. 联合优化 =====
34     total_loss = loss_language + loss_action
35
36     optimizer.zero_grad()
37     total_loss.backward()
38     optimizer.step()
39
40     return {
41         'total_loss': total_loss.item(),
42         'language_loss': loss_language.item(),
43         'action_loss': loss_action.item()
44     }

```

## 4.2 Attention Mask设计

论文原文第5页和Appendix B描述了blockwise causal mask:



```

1  def create_attention_mask(img_len, lang_len, state_len, action_len):
2      """
3      三个block的注意力mask:
4      Block 1: [Images, Language] - 预训练的VLM部分
5      Block 2: [State] - 可缓存的机器人状态
6      Block 3: [Actions] - Flow matching的动作tokens
7      """
8      total_len = img_len + lang_len + state_len + action_len
9      mask = torch.zeros(total_len, total_len)
10
11     # Block 1: 图像和语言内部全连接
12     block1_end = img_len + lang_len
13     mask[:block1_end, :block1_end] = 1
14
15     # Block 2: 状态可以看到Block 1
16     block2_start = block1_end
17     block2_end = block2_start + state_len
18     mask[block2_start:block2_end, :block2_end] = 1
19
20     # Block 3: 动作可以看到所有
21     block3_start = block2_end
22     mask[block3_start:, :] = 1
23
24     return mask.bool()

```

设计原因（论文原文第15页）：

1. **Block 1（VLM部分）不能看未来**：保持与VLM预训练的分布一致
2. **Block 2（状态）独立**：推理时可以缓存K/V，加速flow matching循环
3. **Block 3（动作）全连接**：需要整合所有信息生成动作

## 5. 关键技术细节

### 5.1 Action Expert的嵌入设计

论文原文 Appendix B（第15页）给出了详细设计：

代码块

```

1  class ActionTokenEmbedding(nn.Module):
2      def __init__(self, action_dim, width=1024):
3          super().__init__()
4          self.W1 = nn.Linear(action_dim, width)
5          self.W2 = nn.Linear(2 * width, 2 * width) # 包含τ的编码
6          self.W3 = nn.Linear(2 * width, width)
7

```

```

8     def sinusoidal_encoding(self, tau, dim=1024):
9         """
10         $\phi(\tau)$ : 正弦位置编码
11        类似Transformer的位置编码, 但用于flow matching时间步
12        """
13        position = tau.unsqueeze(-1) # [B, 1]
14        div_term = torch.exp(
15            torch.arange(0, dim, 2) * -(math.log(10000.0) / dim)
16        ).to(tau.device)
17
18        encoding = torch.zeros(tau.shape[0], dim).to(tau.device)
19        encoding[:, 0::2] = torch.sin(position * div_term)
20        encoding[:, 1::2] = torch.cos(position * div_term)
21        return encoding
22
23     def forward(self, noisy_actions, tau):
24         """
25        公式:  $W3 \cdot \text{swish}(W2 \cdot \text{concat}(W1 \cdot a^\tau, \phi(\tau)))$ 
26        """
27        # 1. 动作的线性投影
28        action_proj = self.W1(noisy_actions) # [B, H, width]
29
30        # 2.  $\tau$ 的正弦编码
31        tau_encoding = self.sinusoidal_encoding(tau) # [B, width]
32        tau_encoding = tau_encoding.unsqueeze(1).expand(-1,
noisy_actions.shape[1], -1)
33
34        # 3. 拼接
35        combined = torch.cat([action_proj, tau_encoding], dim=-1) # [B, H,
2*width]
36
37        # 4. MLP with swish activation
38        hidden = F.silu(self.W2(combined)) # swish = SiLU
39        output = self.W3(hidden)
40
41        return output

```

## 5.2 Zero-Padding for Cross-Embodiment

论文原文第6页描述的跨embodiment处理:

代码块

```

1     def pad_to_max_dimensions(state, action, max_dim=18):
2         """
3         统一不同机器人的动作空间
4

```

```

5     Examples:
6         UR5e (7-DoF): pad to 18
7         Bi-UR5e (14-DoF): pad to 18
8         Mobile Fibocom (17-DoF): pad to 18
9     """
10    state_padded = F.pad(state, (0, max_dim - state.shape[-1]))
11    action_padded = F.pad(action, (0, max_dim - action.shape[-1]))
12
13    # 创建mask标记有效维度
14    valid_mask = torch.zeros(max_dim, dtype=torch.bool)
15    valid_mask[:state.shape[-1]] = True
16
17    return state_padded, action_padded, valid_mask

```

## 5.3 数据增强策略

根据论文原文推断标准的做法：

代码块

```

1  class RobotDataAugmentation:
2      def __init__(self):
3          self.img_aug = torchvision.transforms.Compose([
4              torchvision.transforms.RandomCrop(224),
5              torchvision.transforms.ColorJitter(0.2, 0.2, 0.2),
6          ])
7
8      def augment(self, images, actions, state):
9          # 图像增强
10         images_aug = torch.stack([self.img_aug(img) for img in images])
11
12         # 动作不增强（保持物理一致性）
13         # 状态不增强
14
15         return images_aug, actions, state

```

## 6. 训练配置与超参数

根据论文原文的描述和常见实践：

代码块

```

1  # 训练配置
2  config = {
3      # Model
4      'vlm_backbone': 'paligemma-3b',

```

```

5     'action_expert_dim': 1024,
6     'action_expert_depth': 18,
7
8     # Flow Matching
9     'action_horizon': 50,
10    'num_flow_steps': 10,
11    'tau_sampler': 'beta(1.5, 1.0)',
12    'tau_cutoff': 0.999,
13
14    # Training
15    'batch_size': 256,
16    'learning_rate': 1e-4,
17    'weight_decay': 0.01,
18    'warmup_steps': 10000,
19    'total_steps': 700000, # 全量训练
20
21    # Data
22    'image_size': 224,
23    'num_cameras': 2, # or 3
24    'max_action_dim': 18,
25
26    # Loss weights
27    'lambda_language': 1.0,
28    'lambda_action': 1.0,
29 }

```

## 7. 参考实现（基于GitHub生态）

### 7.1 Flow Matching基础

代码块

```

1  # 参考: https://github.com/atong01/conditional-flow-matching
2  from torchdyn.core import NeuralODE
3
4  class FlowMatchingPolicy:
5      def __init__(self, model):
6          self.model = model
7          self.ode_solver = NeuralODE(
8              model, solver='euler', sensitivity='adjoint'
9          )
10
11     def sample(self, observations):
12         # 初始噪声
13         z0 = torch.randn(1, 50, action_dim)
14

```

```

15         # ODE积分 [0, 1]
16         trajectory = self.ode_solver(z0, torch.linspace(0, 1, 11))
17
18         # 返回最终状态
19         return trajectory[-1]

```

## 7.2 Transfusion-style混合训练

代码块

```

1  # 参考Transfusion架构
2  class TransfusionBlock(nn.Module):
3      def __init__(self, dim):
4          super().__init__()
5          self.attn = MultiHeadAttention(dim)
6          self.ff_discrete = FeedForward(dim) # 用于语言
7          self.ff_continuous = FeedForward(dim) # 用于动作
8
9      def forward(self, x, token_type_mask):
10         # 共享的注意力
11         x = x + self.attn(x)
12
13         # 分离的FFN
14         discrete_mask = token_type_mask == 'discrete'
15         continuous_mask = token_type_mask == 'continuous'
16
17         x[discrete_mask] = x[discrete_mask] +
self.ff_discrete(x[discrete_mask])
18         x[continuous_mask] = x[continuous_mask] +
self.ff_continuous(x[continuous_mask])
19
20         return x

```

## 8. 与其他方法的对比

### 8.1 vs. 传统VLA (RT-2, OpenVLA)

维度	传统VLA	$\pi 0$
动作表示	自回归离散化	Flow matching连续
控制频率	2-10Hz	20-50Hz
动作精度	受词表限制	理论无限精度
训练目标	全部Cross-entropy	混合损失
推理速度	自回归慢	可缓存KV加速

代码块

```
1  # RT-2风格 (全离散)
2  actions_discrete = model.generate(
3      observations, max_length=50
4  ) # 自回归生成50步
5
6  #  $\pi 0$ 风格 (flow matching)
7  actions_continuous = flow_matching_sample(
8      model, observations, num_steps=10
9  ) # 10次积分生成50步
```

8.2 vs. Diffusion Policy

维度	Diffusion Policy	$\pi 0$
骨干网络	从头训练	VLM预训练
扩散类型	DDPM/DDIM	Flow matching
语言能力	有限	强 (继承VLM)
跨embodiment	单个机器人	7种机器人统一

9. 实验验证

9.1 消融实验 (隐含在论文原文中)

代码块

```
1  # 从论文原文的对比可以推断
2  experiments = {
3      ' $\pi 0$  (full)': {
4          'vlm_pretrain': True,
5          'flow_matching': True,
6          'cross_embodiment': True,
7          'performance': 0.9 # 归一化
8      },
```

```

9      'π0 (scratch)': {
10          'vlm_pretrain': False,
11          'flow_matching': True,
12          'cross_embodiment': True,
13          'performance': 0.65 # 下降明显
14      },
15      'π0-small': {
16          'vlm_pretrain': False,
17          'flow_matching': True,
18          'cross_embodiment': True,
19          'performance': 0.7
20      },
21      'OpenVLA': {
22          'vlm_pretrain': True,
23          'flow_matching': False, # 自回归
24          'cross_embodiment': True,
25          'performance': 0.3 # 难以处理高频
26      }
27  }

```

## 10. 实验验证

### 10.1 混合训练的核心逻辑：为什么要两种Loss？

代码块

```

1  Cross-Entropy Loss:
2  |— 处理离散的语言token
3  |— 保持VLM预训练的语言理解能力
4  |— 支持语言输入/输出
5  |— 数学上自然适配分类任务
6
7  Flow Matching Loss:
8  |— 处理连续的动作向量
9  |— 精确建模复杂的动作分布
10 |— 支持高频、多模态控制
11 |— 比离散化保留更多信息
12
13 统一Transformer:
14 |— 自注意力机制实现跨模态交互

```

### 10.2 实现关键点

1.  $\tau$ 的采样策略：Beta(1.5, 1)截断到[0, 0.999]

- 2. 向量场的形式： $u = \epsilon - At$ （简单但有效）
- 3. 推理加速：缓存观测的K/V矩阵
- 4. Action Expert设计：集成 $\tau$ 编码的MLP
- 5. Blockwise Mask：平衡预训练兼容性和推理效率

## 四、Diffusion Policy 深入学习

### 1. 核心重点

#### 1.1 基本思想

👁️ Diffusion Policy将机器人的视觉运动策略表示为**条件去噪扩散过程（Conditional Denoising Diffusion Process）**，而不是传统的直接从观察到动作的映射。

#### 1.2 三大关键优势

- 多模态动作分布表达：通过学习动作得分函数的梯度场，可以表达任意可归一化的分布
- 高维输出空间：可以预测动作序列而非单步动作，提高时序一致性
- 训练稳定性：避免了能量模型中的负采样问题

### 2. 核心创新点

#### 2.1 策略表示范式转变

传统方法对比：

方法类型	表示方式	问题
Explicit Policy	$F_{\theta}(o) \rightarrow a$	难以处理多模态
Implicit Policy (IBC)	$E_{\theta}(o,a)$	训练不稳定（需要负采样）
<b>Diffusion Policy</b>	$\nabla_a \log p(a \parallel o)$	<b>稳定且表达力强</b>

#### 2.2 位置控制优于速度控制

论文发现Diffusion Policy与位置控制有协同效应：

- 位置控制的多模态更显著，恰好发挥Diffusion的优势
- 动作序列预测减少了位置控制的累积误差
- 与主流方法使用速度控制形成对比



## 2.3 动作序列预测 (Action Chunking)

关键参数：

- T\_o: 观察窗口 (Observation Horizon)
- T\_p: 预测窗口 (Prediction Horizon)
- T\_a: 执行窗口 (Action Horizon)

优势：

- 时序一致性**：避免单步预测的抖动
- 鲁棒性**：对idle actions（暂停动作）不敏感
- 延迟容忍**：通过预测未来补偿系统延迟

## 3. 数学公式详解

### 3.1 DDPM去噪迭代过程

$$x^{k-1} = \alpha(x^k - \gamma \varepsilon_{\theta}(x^k, k) + \mathcal{N}(0, \sigma^2 I))$$

其中：

- $x^k$ : 第k步的带噪动作
- $\varepsilon_{\theta}(x^k, k)$ : 噪声预测网络
- $\alpha, \gamma, \sigma$ : 噪声调度参数 (noise schedule)
- k: 去噪迭代步数 ( $k=K \rightarrow 0$ )

物理意义：从高斯噪声  $x^K \sim \mathcal{N}(0, I)$  开始，逐步去噪得到动作  $x^0$ 。

### 3.2 梯度下降视角

$$x' = x - \gamma \nabla E(x)$$

去噪过程等价于：

- $\varepsilon_{\theta}(x, k)$  近似能量函数梯度  $\nabla E(x)$
- 每次迭代是带噪声的梯度下降步
- $\gamma$  对应学习率

### 3.3 训练目标 (噪声预测)

$$\mathcal{L} = \text{MSE}(\varepsilon^k, \varepsilon_{\theta}(x^0 + \varepsilon^k, k))$$

训练流程：

1. 从数据集采样干净样本  $x^0$
2. 随机选择去噪步数  $k \in \{1, \dots, K\}$
3. 采样对应方差的噪声  $\epsilon^k$
4. 预测加噪样本  $x^0 + \epsilon^k$  的噪声

### 3.4 视觉条件化扩散

**关键修改：** 将观察  $O_t$  作为条件而非联合分布的一部分

$$A_t^{k-1} = \alpha(A_t^k - \gamma \epsilon_\theta(O_t, A_t^k, k) + \mathcal{N}(0, \sigma^2 I))$$

对应的训练损失：

$$\mathcal{L} = \text{MSE}(\epsilon^k, \epsilon_\theta(O_t, A_t^0 + \epsilon^k, k))$$

**优势：**

- 视觉编码器只需运行一次（不随去噪迭代重复）
- 大幅降低计算量，实现实时控制
- 便于端到端训练

### 3.5 与能量模型的联系

隐式策略的能量表示：

$$p_\theta(a|o) = \frac{e^{-E_\theta(o,a)}}{Z(o, \theta)}$$

Diffusion Policy 通过学习 **得分函数 (Score Function)** 绕过归一化常数  $Z$ ：

$$\nabla_a \log p(a|o) = -\nabla_a E_\theta(a, o) - \underbrace{\nabla_a \log Z(o, \theta)}_{=0} \approx -\epsilon_\theta(a, o)$$

**关键insight：**

- IBC需要负采样估计  $Z \rightarrow$  训练不稳定
- Diffusion直接学习梯度  $\rightarrow$  无需负采样  $\rightarrow$  稳定训练

## 4. 网络架构设计

### 4.1 CNN-based架构

$$\epsilon_\theta(O_t, A_t^k, k) = \text{Conv1D}_{\text{FiLM}}(A_t^k | \text{ResNet}(O_t), k)$$

- 使用FiLM (Feature-wise Linear Modulation) 融合观察特征
- 1D卷积处理时序动作
- 鲁棒但对高频信号过度平滑

## 4.2 Transformer-based架构

$$\varepsilon_{\theta}(O_t, A_t^k, k) = \text{Transformer}_{\text{causal}}(A_t^k | O_t, k)$$

- 因果注意力机制 (Causal Attention)
- 交叉注意力 (Cross Attention) 融合观察
- 适合高频动作变化和速度控制

## 4.3 推荐使用策略

- 首选CNN (鲁棒易调)
- 任务复杂或高频时用Transformer (需更多调参)

## 5. 与控制理论的联系

### 5.1 线性系统的极限行为

对于线性系统  $s_{t+1} = As_t + Ba_t + w_t$  , 线性反馈策略  $a_t = -Ks_t$

最优去噪器:

$$\varepsilon_{\theta}(s, a, k) = \frac{1}{\sigma_k} [a + Ks]$$

DDIM采样收敛到:

$$a = -Ks$$

意义: 对于简单任务, Diffusion Policy自动退化为线性控制器。

## 6. 总结

Diffusion Policy的核心贡献是:

1. **理论**: 将策略学习转化为得分函数学习, 避免归一化难题
2. **工程**: 闭环动作序列+位置控制+视觉条件化的系统设计
3. **实证**: 在15个任务上证明稳定性和泛化能力

## 五、Flow Matching 深入学习

### 1. Flow Matching核心思想

**目标**: 通过学习一个时间依赖的速度场  $v_{\theta}$ , 将简单分布 (如高斯分布)  $P_0$  推向复杂的数据分布  $P_1$ 。

**关键优势**:

- 直接求解确定性ODE, 避免扩散模型中的噪声反演

- 可以看作是"速度 × 时间"的路径积分过程

## 2. 理论基础：归一化流

### 2.1 变量替换公式

对于可逆变换  $\phi: \mathbb{R}^d \rightarrow \mathbb{R}^d$ ，概率密度的变化遵循：

$$P_1(y) = P_0(x) \frac{1}{|\det J_\phi(x)|}$$

其中  $J_\phi(x) = \frac{\partial \phi}{\partial x}$  是Jacobian矩阵。

**关键点：**雅可比行列式反映了空间体积的伸缩比例，这是从输入分布到输出分布的核心。

### 2.2 最大似然训练

优化目标：

$$\max_{\theta} \mathbb{E}_{x \sim D} [\log p_{\theta}(x)]$$

等价于最小化KL散度：

$$\min_{\theta} \text{KL}(p_{\text{data}} \| p_{\theta}) \Leftrightarrow \max_{\theta} \mathbb{E}_{x \sim p_{\text{data}}} [\log p_{\theta}(x)]$$

## 3. 从残差流到连续归一化流

### 3.1 残差流的局限

残差流定义为：

$$\phi_k(x) = x + \delta u_k(x)$$

**优点：**可以利用矩阵的迹来简化计算：

$$\log \det(I + A) = \text{Tr}(\log(I + A))$$

**缺点：**

- 需要堆叠多层
- 步长  $\delta$  需要人为设计

### 3.2 连续归一化流（CNF）

**关键转变：**当步长  $\delta \rightarrow 0$  时，定义时间依赖的速度场：

$$u_t(x) = \frac{dx}{dt}$$

轨迹的演化方程：

$$\frac{d\phi_t(x)}{dt} = u_t(\phi_t(x)), \quad \phi_0(x) = x_0$$

积分形式：

$$X_t = X_0 + \int_0^t u_s(X_s) ds$$

### 3.3 传输方程 (Transport Equation)

概率密度的演化遵循：

$$\frac{\partial p_t(x)}{\partial t} = -\nabla \cdot (u_t p_t)(x)$$

**log空间推导 (重点)：**

沿着轨迹  $X_t$ ，log密度的变化为：

$$\frac{d}{dt} \log p_t(X_t) = -(\nabla \cdot u_t)(X_t)$$

积分得到：

$$\log p_t(X_t) = \log p_0(X_0) - \int_0^t (\nabla \cdot u_s)(X_s) ds$$

**意义：**将Jacobian行列式的计算转化为速度场散度的积分。

## 4. Flow Matching的创新

### 4.1 核心理想

**问题：**CNF需要在每次训练时数值求解ODE，计算代价高。

**解决方案：**将求解ODE转化为**回归问题**，直接拟合速度场：

$$\mathcal{L}_{\text{FM}} = \mathbb{E}_{t \sim U(0,1), x_t \sim p_t} [\|u_\theta(t, x_t) - u_t(x_t)\|^2]$$

### 4.2 Conditional Flow Matching (CFM)

**挑战：**从  $p_0 \rightarrow p_1$  的路径不唯一，速度场  $u_t(x)$  难以直接确定。

**关键创新：**引入条件路径，以数据点  $x_1$  为条件：

$$p_t(x) = \int p_t(x|x_1) q_1(x_1) dx_1$$

其中  $q_1(x_1)$  是数据分布。

**条件速度场**与**边际速度场**的关系：

$$u_t(x) = \mathbb{E}_{x_1 \sim p_1|t} [u_t(x|x_1)] = \int u_t(x|x_1) \frac{p_t(x|x_1) q_1(x_1)}{p_t(x)} dx_1$$

### 4.3 边界条件

为保证正确性，需要满足：

## 1. 起点条件 (t=0) :

$$p_0(x|x_1) = p_0(x)$$

## 2. 终点条件 (t=1) :

$$p_1(x|x_1) = \mathcal{N}(x; x_1, \sigma_{\min}^2 I), \quad \sigma_{\min} \rightarrow 0$$

意义：每条条件路径最终收敛到对应的目标点  $x_1$ 。

## 4.4 关键定理：Loss等价性

CFM的损失函数：

$$\mathcal{L}_{\text{CFM}}(\theta) = \mathbb{E}_{t \sim U(0,1), x_1 \sim q_1, x_t \sim p_t(x|x_1)} [\|u_\theta(t, x_t) - u_t(x_t|x_1)\|^2]$$

核心结论：

$$\nabla_\theta \mathcal{L}_{\text{FM}} = \nabla_\theta \mathcal{L}_{\text{CFM}}$$

证明要点：展开平方项，关键在于交叉项：

$$\mathbb{E} \langle u_\theta, u_t(x) \rangle = \iint \langle u_\theta(t, x), u_t(x|x_1) \rangle p_t(x|x_1) q_1(x_1) dx_1 dx$$

这正好等于CFM的交叉项，因此两者梯度相同。

## 5. 总结：为什么CFM有效？

- a. 计算效率：不需要求解ODE，只需要采样和回归
- b. 路径明确：条件路径  $p_t(x|x_1)$  容易设计（如线性插值）
- c. 训练等价：CFM与FM在优化意义上完全等价
- d. 避免散度计算：不需要显式计算  $\nabla \cdot u_t$

实践流程：

- a. 设计条件路径（如  $x_t = (1-t)x_0 + tx_1$ ）
- b. 计算条件速度场  $u_t(x|x_1) = x_1 - x_0$
- c. 最小化CFM损失训练模型
- d. 推理时求解ODE生成样本

# 六、本周总结与下周计划

## 1. 本周总结

本周首先推进了毕业设计的背景调研与ReconVLA的学习，了解了“gaze zone”这种基于attention layer的vla action生成精度提升的策略；接下来借助我找到的论文《Diffusion Policy:

Visuomotor Policy Learning via Action Diffusion》以及张教授分享的“Lilian's Posts”学习了diffusion policy的核心数学知识；同时学习了flow matching的数学原理；在前两项工作进行的过程中，结合PI的代码和论文进行了PI-0/0.5 混合监督机制的深入学习，这周的工作量与上周相比大了不少并且主要是公式的表达和代码的提炼（周报中多次使用伪代码表示）。

遗憾的是张教授提供的“Lilian's Posts”我忘记去看了，然后与vla相关的演讲之类也没开始看，下周补上；flow matching 和diffusion model应该多见点代码；






在论文阅读的过程中我最先使用纸张做笔记，比较直观但是转换到电子周报时出现了较大的麻烦，从下周开始可以多使用draw.io等流程图绘制工具；虽然可能耗时较长，但正如本科期间跟过的高智教授（毕设的校内挂名导师）说过“平时学习工作过程中多将自己的方法、思路记录下来，并绘制成流程图，将来在自己的论文中都得上，（流程图）要作为知识财产保留下来、保护起来”。

## 2. 想与张教授交流的问题

1. 请问我的毕业指标是什么（比如说多少篇CCFA之类的）？之前和同级的朋友聊天时问过，发现回答不上来
2. 我跟着您学习的领域，是否会将CV三大会作为target？Embodied AI有哪些可能发的顶会？
3. 目前我学习的主要是模型层面的知识，请问后续是否会涉及到cv、3dgs等方面的学习？
4. 关于diffusion model，我找了几篇文章，但还没深入了解，请张教授把控一下，看是否有需要重点关注的或是没有必要阅读的：
  - a. <https://yang-song.net/blog/2021/score/>
  - b. <https://huggingface.co/blog/annotated-diffusion>
  - c. <https://arxiv.org/abs/2105.05233>
  - d. <https://bair.berkeley.edu/blog/2023/07/14/ddpo/>
5. flow matching 和diffusion model您有没有推荐的项目？我想看看他们的代码。
6. 这周因病耽误了三天，下次周报提交应该就是10.31或者11.1了；我预计11.2或11.3动身前往苏州，想了解一下租房相关的信息，上次交流中您提到组里已经有位同学前去学习并就近租到了房间，请问能否加一下他的联系方式？我想跟他交流一下，提前着手解决这个问题。

## 3. 下周计划

- ☒ **继续进行：** 阅读下图中的文献（顺序见图），加深对概念的理解，争取在读文献的过程中加强对理论的理解，并将本周周报中的理论知识系统地迁移到知识库中，避免后续寻找不便。
- ☐ 了解并学习“Lilian's Posts”以及张教授今天中午推荐的“**Federico Sarrocco post**”，以及其他相关post。
- ☐ 深入了解ReconVLA的背景，思考其idea的来源，并调研ReconVLA之后的相关解决方案，比较他们并梳理出vla精度提升的解决方案的timeline，思考毕设大致方向
- ☐ 开始观看相关演讲，阅读相关paper，了解VLA领域High-Level Overview

- >   $\pi_0$ : A Vision-Language-Action Flow Model for General Robot Control 1
- >  Diffusion Policy: Visuomotor Policy Learning via Action Diffusion 2
- >  Denoising Diffusion Probabilistic Models
- >  Flow Matching for Generative Modeling 3
- >  Transfusion: Predict the Next Token and Diffuse Images with One Multi-Modal Model 4