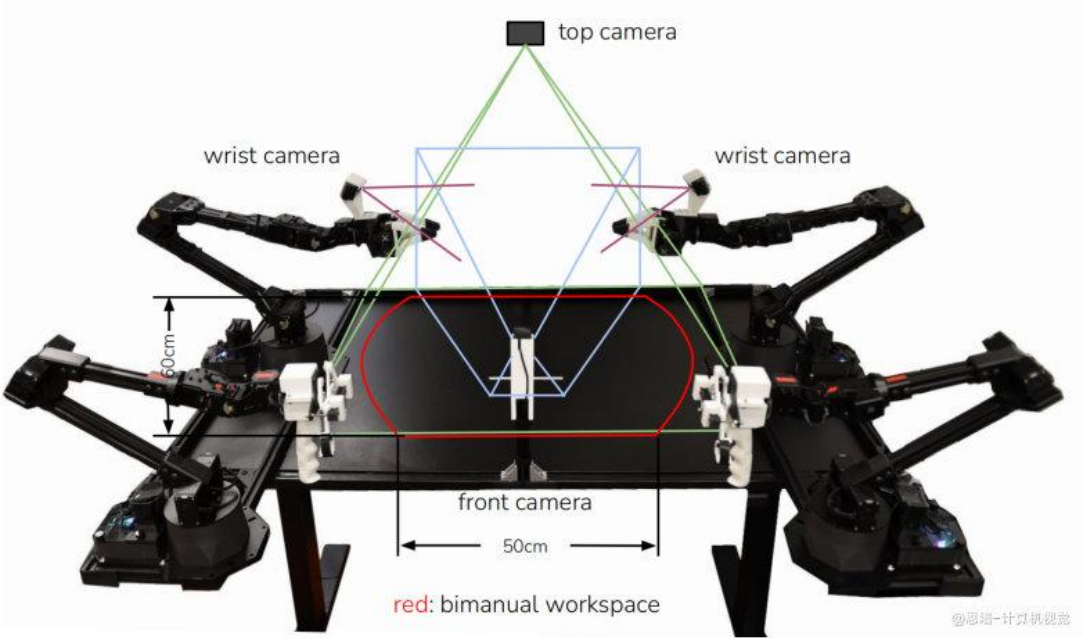
一、ACT 简介

下面的原理介绍，基于 ACT 论文，知识点较多，如果有谬误，欢迎反馈

- 论文地址：https://arxiv.org/pdf/2304.13705
- 项目官网：https://tonyzhaozh.github.io/aloha/

1.1.1 ALOHA 硬件系统

- **低成本双臂遥操作平台**：整套系统成本不到 2 万美元，使用现成机器人和 3D 打印组件构建
- **设计原则**：低成本、多功能、用户友好、易维修、易组装
- **技术特点**：
  - 使用关节空间映射进行遥操作（操作员通过反驱动较小的"领导"机器人来控制较大的"跟随"机器人）
  - 配备 4 个摄像头提供多视角观察
  - 50Hz 高频控制，支持精确操作



Aloha 实物图

（演示视频）

1.1.2 ACT 学习算法

Action Chunking with Transformers，是一种新的模仿学习算法：

核心创新：

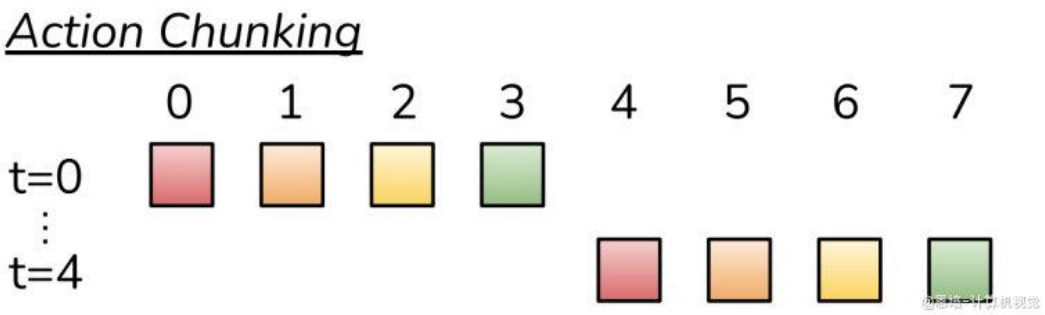
- **动作分块**：预测未来 k 个时间步的动作序列，而非单步动作，有效减少复合误差
- **时序集成**：通过重叠动作块的加权平均实现平滑执行
- **条件变分自编码器(CVAE)**：处理人类演示数据的随机性和多模态性
- **Transformer 架构**：用于序列建模和多模态信息融合

实验成果

- **任务能力**：能执行开调料杯盖、插电池、穿束带等 6 种精细操作任务
- **性能表现**：80-90% 成功率，仅需约 10 分钟演示数据（50 个轨迹）
- **显著优势**：相比现有方法（BeT、RT-1、VINN 等）有大幅性能提升

（演示视频）

二、核心创新：动作分块 (Action Chunking)



2.1 基本概念：什么是 Action Chunking？

- **传统方法**：每个时间步预测一个动作 $\pi_\theta(a_t|s_t)$
  - $\pi_\theta(a_t|s_t)$ =给定当前状态 s_t ，预测下一个动作 a_t
  - 状态s_t比如：机械臂当前各关节角度、摄像头拍摄的画面
  - 动作a_t比如：机械臂各关节目标角度
- **Action Chunking**：每 k 个时间步预测 k 个连续动作 $\pi_\theta(a_{t:t+k}|s_t)$
  - $\pi_\theta(a_{t:t+k}|s_t)$ =给定当前状态 s_t ，预测接下来 k 步的动作序列
- **核心思想**：将动作序列"打包"成块（chunk），作为一个整体单元执行

- 举例说明：
  - 任务场景：抓取桌上的苹果
  - 机械臂状态
    - 6 个关节
    - 1 个夹爪：开合程度 (0=完全闭合，1=完全张开)
    - 动作维度：每个$a_t = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6, gripper]$(7 维向量)

- 传统方法：

```
t=1: 观察→苹果在右前方
     预测→ a_1 = [10°, 5°, 0°, 0°, 0°, 0°, 0.8] (向右转，夹爪张开)

t=2: 观察→机械臂开始转动
     预测→ a_2 = [8°, 10°, 5°, 0°, 0°, 0°, 0.8] (继续调整)

t=3: 观察→接近苹果
     预测→ a_3 = [2°, 15°, 10°, 5°, 0°, 0°, 0.8] (伸向苹果)

... (每步都要重新观察和决策)
```

- Action Chunking （假设 k=10）

```
t=1: 观察→苹果在右前方
     一次性预测→
     a_1:11 = [
       [10°, 5°, 0°, 0°, 0°, 0°, 0.8],    # 第1步：向右转
       [8°, 10°, 5°, 0°, 0°, 0°, 0.8],     # 第2步：调整角度
       [5°, 15°, 10°, 5°, 0°, 0°, 0.8],    # 第3步：伸向苹果
       [2°, 18°, 15°, 8°, 2°, 0°, 0.8],    # 第4步：继续接近
       [0°, 20°, 18°, 10°, 5°, 2°, 0.8],   # 第5步：精确定位
       [0°, 20°, 18°, 10°, 5°, 2°, 0.6],   # 第6步：准备抓取
       [0°, 20°, 18°, 10°, 5°, 2°, 0.4],   # 第7步：夹爪收缩
       [0°, 20°, 18°, 10°, 5°, 2°, 0.2],   # 第8步：继续收缩
       [0°, 20°, 18°, 10°, 5°, 2°, 0.0],   # 第9步：夹紧
       [0°, 15°, 12°, 5°, 2°, 0°, 0.0]     # 第10步：轻微提起
     ]

连续执行这10步，然后在t=11时重新观察环境
```

- 效果

方面	传统方法	**Action Chunking (k=10)**
观察频率	每步观察	每 10 步观察
预测内容	(1,7)	(10,7)
决策次数	10 次	1 次
误差累积	高（10 次机会出错）	低（1 次机会出错）

2.2 解决的核心问题

复合误差问题（Compounding Error）

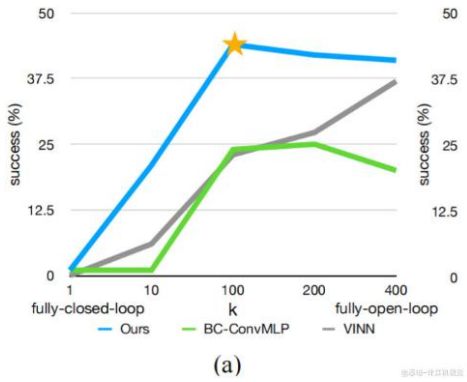
```
时间步1：小误差 ε₁
时间步2：误差 ε₁ + ε₂
时间步3：误差 ε₁ + ε₂ + ε₃
...
最终误差：Σ εᵢ → 任务失败
```

Action Chunking 如何缓解？

- **有效视野缩减**：k=100 时，400 步任务变成 4 步决策
- **误差累积减少**：从 400 次累积变成 4 次累积
- **决策点减少**：减少了 96% 的关键决策点

2.3 Chunk Size (k) 的影响

实验结果分析（在作者自己的实验上测量）



(a)

Chunk Size (k)	成功率表现	特点
k=1	1%	传统单步预测，复合误差严重
k=10	12%	轻微改善
k=50	35%	显著提升
k=100	44%	**最佳性能**
k=200	40%	性能轻微下降
k=400	24%	过度开环，缺乏反应性

选择原则

- 太小 **(k<50)**：复合误差仍然明显
- 适中 **(k=100)**：平衡了误差缓解和反应性
- 太大 **(k>200)**：接近开环控制，失去闭环反馈能力

2.4 与传统方法对比

单步预测的局限

```
# 传统BC方法的问题
t=1: predict(obs₁) → action₁ (小误差)
t=2: predict(obs₂) → action₂ (obs₂已经偏离训练分布)
t=3: predict(obs₃) → action₃ (误差进一步累积)
...
结果：机器人"迷失"在未见过的状态空间
```

Action Chunking 的优势

```
# ACT方法
t=1: predict(obs₁) → [action₁, action₂, ..., action₁₀₀]
# 执行100步后重新观测
t=101: predict(obs₁₀₁) → [action₁₀₁, action₁₀₂, ..., action₂₀₀]
# 大幅减少累积误差的机会
```

2.5 处理非马尔可夫行为

人类演示中的问题

- **暂停行为**：人类在演示中会暂停思考
- **时序相关干扰**：行为不仅依赖状态，还依赖时间步
- **单步策略难以建模**：$\pi(a_t|s_t)$无法捕捉这种时序依赖

Action Chunking 的解决方案

- **时序上下文**：k 步序列包含了时序信息
- **行为连贯性**：一个 chunk 内的动作保持连贯
- **减少暂停影响**：暂停行为被"稀释"在 chunk 中

三、时序集成(Temporal Ensembling)

3.1 为什么需要？

3.1.1 Action Chunking 的朴素实现问题

- 离散执行策略

```python
# 朴素的Action Chunking实现
def naive_chunking():
    for t in range(0, episode_length, k):  # 每k步执行一次
        obs = get_observation(t)           # 获取观测
        action_chunk = policy(obs)         # 预测k步动作
        for i in range(k):
            execute(action_chunk[i])       # 逐步执行
        # 问题：下一个chunk与当前chunk完全断开！
```

- 核心问题分析
  - **突然的信息更新**：每 k 步才融入新的环境观测
  - **动作不连续**：相邻 chunk 之间可能存在跳跃
  - **响应延迟**：环境变化后需要等待 k 步才能响应

3.1.2 实际任务中的问题表现

- 开调料杯任务示例

```
时刻0-99:   基于初始观测的动作序列
时刻100:    突然基于新观测重新规划
结果:       机器人动作出现"跳跃"，杯子可能掉落
```

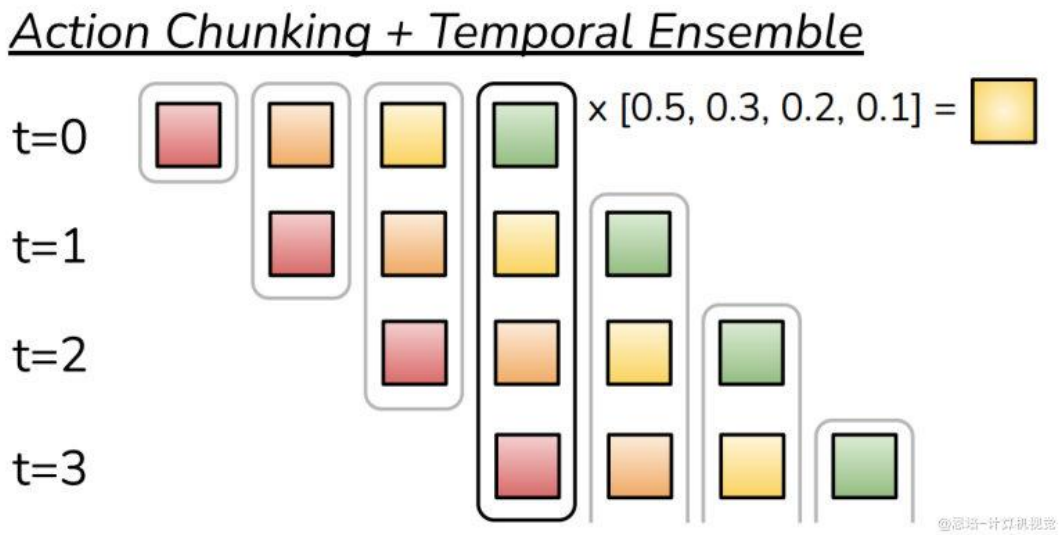
- 插电池任务示例

```
Chunk 1: [抓取电池] - 基于t=0时的观测
Chunk 2: [插入电池] - 基于t=100时的观测
问题: 如果电池在执行过程中位置发生微调，
      新chunk可能基于过时信息规划动作
```

3.1.3 对精细操作的影响

- **精度要求高**：毫米级误差导致任务失败
- **实时反馈需要**：需要持续根据视觉反馈调整
- **平滑性要求**：突然的动作变化会产生冲击力

3.2 如何实现？



3.2.1 核心思想：重叠预测

从离散切换到连续融合

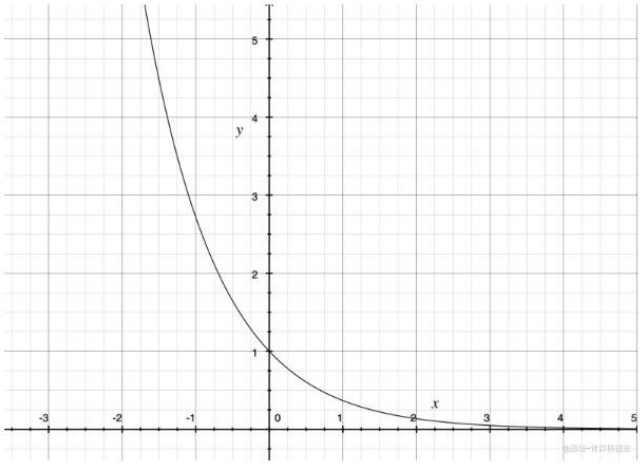
```
# 原始方法：离散切换
t=0:   预测 [a_0, a_1, a_2, ..., a_99]
t=100: 预测 [a_100, a_101, a_102, ..., a_199]

# Temporal Ensembling：重叠预测
t=0:   预测 [a_0, a_1, a_2, ..., a_99]
t=1:   预测 [a_1, a_2, a_3, ..., a_100]
t=2:   预测 [a_2, a_3, a_4, ..., a_101]
...
```

3.2.2 具体实现算法——加权平均

> *ensemble* to combine these predictions. Our *temporal ensemble* performs a weighted average over these predictions with an exponential weighting scheme $w_i = \exp(-m * i)$, where w_0 is the weight for the oldest action. The speed for incorporating new observation is governed by m, where a smaller m means faster incorporation. We note that unlike typical smoothing,
>
> 我们的时间集成使用指数加权方案对这些预测执行加权平均：$w_i = e^{(-m*i)}$，其中w_0是第一个动作的权重。

论文中权重计算公式：



$w_i = e^{(-m \times i)}$

- w_i：第i个预测的权重值w
- $e \approx 2.718$
- m：衰减速率参数
- i：预测的"时间点"（0=最新，1=1 步前，2=2 步前...）

权重的含义：

- **最新预测**（i=0）：权重最大，影响最大
- **较旧预测**（i=1,2,3...）：权重递减，影响逐渐减小
- **很旧预测**：权重接近 0，几乎不影响最终结果

实际原作代码中的具体实现如下

```
# 模型根据当前观测，推理得到一批actions
all_actions = policy(qpos, curr_image)

# 将当前时间步的动作预测存储到all_time_actions张量中
all_time_actions[[t], t:t+num_queries] = all_actions

# 获取当前时间步t的所有动作预测
actions_for_curr_step = all_time_actions[:, t]

# 筛选出有效的动作预测（非零值）
actions_populated = torch.all(actions_for_curr_step != 0, axis=1)
actions_for_curr_step = actions_for_curr_step[actions_populated]

# 设置指数衰减的系数k
k = 0.01

# 计算指数衰减权重：较早的预测权重较小，较新的预测权重较大
exp_weights = np.exp(-k * np.arange(len(actions_for_curr_step)))

# 归一化权重，使所有权重之和为1
exp_weights = exp_weights / exp_weights.sum()

# 将numpy数组转换为cuda张量并增加维度
exp_weights = torch.from_numpy(exp_weights).cuda().unsqueeze(dim=1)

# 计算加权平均的最终动作
raw_action = (actions_for_curr_step * exp_weights).sum(dim=0, keepdim=True)
```

这个算法的核心思想是：

1. 收集不同时间步对当前时间步 t 的动作预测
2. 使用指数衰减函数给这些预测分配权重，使得较新的预测有更高的权重
3. 归一化这些权重（确保它们的总和为 1）
4. 计算加权平均得到最终的动作

用数学公式表示为：

$$
\begin{aligned}
a_t &= \sum_{i=1}^{n} w_i \cdot p_i \\
&= w_1 \cdot p_1 + w_2 \cdot p_2 + w_3 \cdot p_3 + \ldots + w_n \cdot p_n
\end{aligned}
$$

其中：

- a_t 是时间步 t 的最终动作
- p_i 是第 i 个动作预测
- $w_i = \frac{e^{-k(i-1)}}{\sum_{j=1}^{n} e^{-k(j-1)}}$ 是归一化的权重

  > `np.arange(len(actions_for_curr_step))` 生成的是从 0 开始的序列$[0, 1, 2, \ldots, n-1]$，所以这里是$j-1$
- $k = 0.01$ 是衰减系数

这种方法通过综合考虑多个时间步的预测，提高了动作预测的稳定性和准确性。

四、网络架构

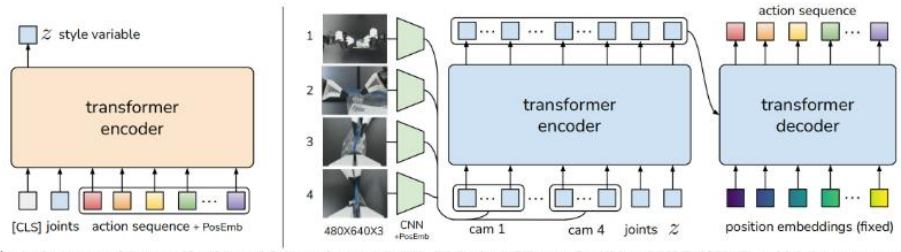


Fig. 4: *Architecture of Action Chunking with Transformers (ACT).* We train ACT as a Conditional VAE (CVAE), which has an encoder and a decoder. *Left:* The encoder of the CVAE compresses action sequence and joint observation into z, the style variable. The encoder is discarded at test time. *Right:* The decoder or policy of ACT synthesizes images from multiple viewpoints, joint positions, and z with a transformer encoder, and predicts a sequence of actions with a transformer decoder. z is simply set to the mean of the prior (i.e. zero) at test time.

基于 Transformer 的动作分块(ACT)架构，将 ACT 训练为条件变分自编码器(CVAE)，它包含一个编码器和一个解码器。

左侧：CVAE 的编码器将动作序列和关节观测压缩为风格变量 z。编码器在测试时被丢弃。

右侧：ACT 的解码器或策略使用 Transformer 编码器融合来自多个视角的图像、关节位置和 z，并通过 Transformer 解码器预测动作序列。在测试时，z 简单地设置为先验分布的均值(即零)。

4.1 整体架构概览

ACT 采用**条件变分自编码器(Conditional VAE)**架构：

- **编码器(Encoder)**：仅在训练时使用，用于推断风格变量 z
- **解码器(Decoder)**：实际的策略网络，用于预测动作序列

设计理念

- **编码器**：学习如何从人类演示中提取"风格信息"
- **解码器**：学习如何根据观测和风格生成动作序列
- **测试时**：丢弃编码器，解码器独立工作

4.2 网络架构详细设计

4.2.1 CVAE 编码器架构

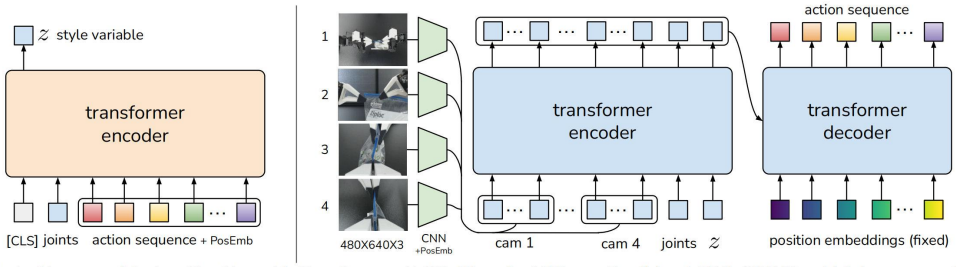


Fig. 4: *Architecture of Action Chunking with Transformers (ACT).* We train ACT as a Conditional VAE (CVAE), which has an encoder and a decoder. *Left:* The encoder of the CVAE compresses action sequence and joint observation into z, the style variable. The encoder is discarded at test time. *Right:* The decoder or policy of ACT synthesizes images from multiple viewpoints, joint positions, and z with a transformer encoder, and predicts a sequence of actions with a transformer decoder. z is simply set to the mean of the prior (i.e. zero) at test time.

4.2.1.1 输入组成

编码器接收三个输入：

- **[CLS] token**：可学习的特殊标记（类似 BERT）
- **当前关节位置**：14 维向量（双臂各 7 个关节）
- **目标动作序列**：k×14 维张量（k 步的关节位置目标）

4.2.1.2 处理组成

1. **嵌入投影**：
    - 关节位置通过线性层投影到 512 维
    - 动作序列通过另一线性层投影到 512 维
    - [CLS] token 是直接学习的 512 维向量
2. **序列构建**：
    - 形成长度为(k+2)的输入序列
    - 每个元素都是 512 维向量
3. **Transformer 编码**：
    - 使用 4 层 Transformer 编码器
    - 自注意力机制融合所有信息
4. **风格变量推断**：
    - 取[CLS] token 对应的输出特征
    - 通过线性层预测 z 的均值和方差
    - z 是 32 维的对角高斯分布

4.2.2 CVAE 解码器架构（策略网络）

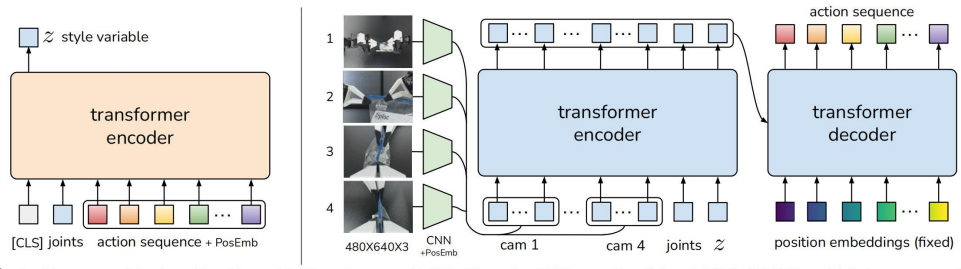


Fig. 4: *Architecture of Action Chunking with Transformers (ACT).* We train ACT as a Conditional VAE (CVAE), which has an encoder and a decoder. *Left:* The encoder of the CVAE compresses action sequence and joint observation into z, the style variable. The encoder is discarded at test time. *Right:* The decoder or policy of ACT synthesizes images from multiple viewpoints, joint positions, and z with a transformer encoder, and predicts a sequence of actions with a transformer decoder. z is simply set to the mean of the prior (i.e. zero) at test time.

4.2.2.1 输入处理

解码器处理多模态输入：

图像处理流程：

1. **ResNet18** 骨干网络：
    - 4 个 480×640×3 的 RGB 图像
    - 每个图像 →15×20×512 特征图
    - 展平为 300×512 的特征序列

2. **空间位置编码：**

- 添加 2D 正弦位置编码
- 保持空间结构信息

3. **多视角融合：**

- 4 个摄像头的特征序列拼接
- 总计 1200×512 的图像特征序列

其他输入处理：

- **当前关节位置**：线性投影到 512 维
- **风格变量 z**：线性投影到 512 维

4.2.2.2 Transformer 架构

1. **编码器部分：**

- 4 层 Transformer 编码器
- 输入：1202×512（图像 1200 + 关节 1 + 风格 1）
- 功能：多模态信息融合

2. **解码器部分：**

- 7 层 Transformer 解码器
- Query：固定的正弦位置编码（k×512）
- Key/Value：来自编码器输出
- 交叉注意力机制生成动作序列

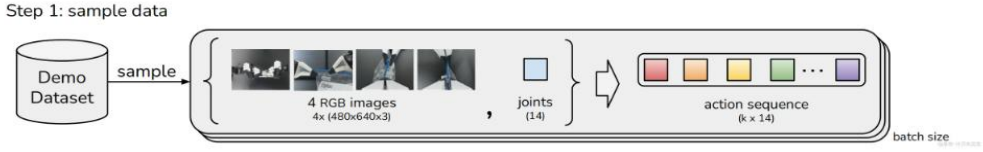
4.2.2.3 输出生成

- **解码器输出**：k×512 维特征
- **动作预测**：通过 MLP 投影到 k×14 维
- **最终输出**：k 步的关节位置目标

4.3 训练流程（Training Pipeline）

4.3.1 整体流程概览

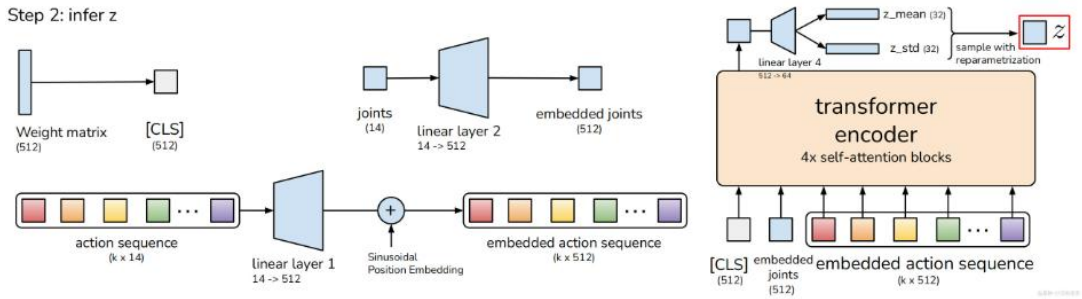
Step 1: 数据采样 (Sample Data)



从演示数据集中采样训练批次：

- **输入观测**：4 个 RGB 图像 (4×480×640×3) + 关节位置 (14 维)
- **目标动作序列**：k×14 维的关节位置序列
- **批次组织**：按 batch size 组织数据

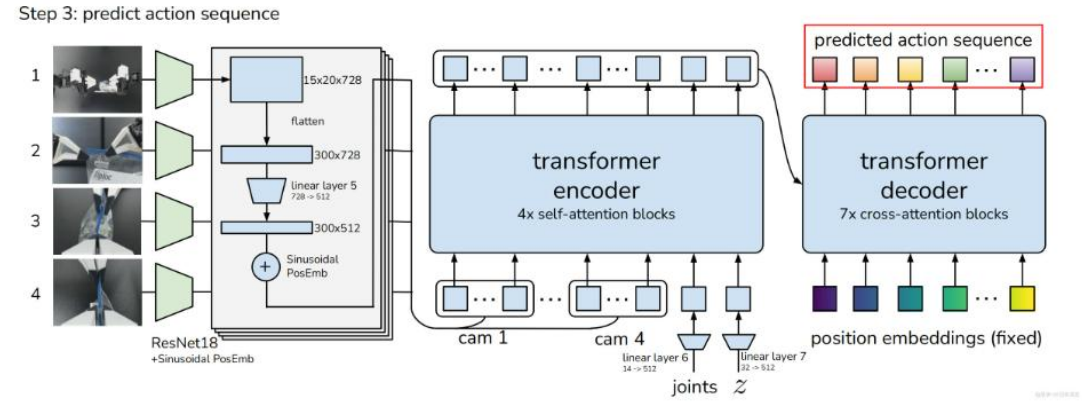
Step 2: 风格变量推断 (Infer z)



通过 CVAE 编码器推断风格变量：

- **编码器输入：**
  - [CLS] token (512 维)
  - 当前关节位置 (14→512 维投影)
  - 动作序列 (k×14→k×512 维投影)
- **Transformer 编码：**4 层自注意力块处理
- **输出分布：**z 的均值和标准差 (各 32 维)
- **采样：**使用重参数化技巧采样 z

Step 3: 动作序列预测 (Predict Action Sequence)



通过 CVAE 解码器生成动作预测：

- **图像处理：**
  - 4 个摄像头图像通过 ResNet18
  - 输出 4×300×512 的特征序列
  - 添加正弦位置编码
- **多模态融合：**
  - Transformer 编码器融合图像、关节、风格信息
  - 输入序列长度：1202 (图像 1200 + 关节 1 + 风格 1)
- **序列生成：**
  - Transformer 解码器通过交叉注意力生成
  - 固定位置编码作为查询 (k×512)
  - 输出预测动作序列 (k×14)

4.3.2 详细算法流程

Algorithm 1 ACT Training

Given: Demo dataset $\mathcal{D}$, chunk size k, weight β.
Let a_t, o_t represent action and observation at timestep t, $\bar{o}_t$ represent o_t without image observations.
Initialize encoder $q_\phi(z|a_{t:t+k}, \bar{o}_t)$
Initialize decoder $\pi_\theta(\hat{a}_{t:t+k}|o_t, z)$
for iteration $n = 1, 2, ...$ **do**
    Sample o_t, $a_{t:t+k}$ from $\mathcal{D}$
    Sample z from $q_\phi(z|a_{t:t+k}, \bar{o}_t)$
    Predict $\hat{a}_{t:t+k}$ from $\pi_\theta(\hat{a}_{t:t+k}|o_t, z)$
    $\mathcal{L}_{reconst} = MSE(\hat{a}_{t:t+k}, a_{t:t+k})$
    $\mathcal{L}_{reg} = D_{KL}(q_\phi(z|a_{t:t+k}, \bar{o}_t) \parallel \mathcal{N}(0, I))$
    Update θ, ϕ with ADAM and $\mathcal{L} = \mathcal{L}_{reconst} + \beta\mathcal{L}_{reg}$

参数定义与初始化：

第 1 步：给定参数

- **演示数据集 $\mathcal{D}$**：包含人类演示的观测-动作对

  > 观测（Observation）o_t：
  >
  > - 图像：4 个摄像头的 RGB 图像
  > - 关节角度：机械臂当前关节位置（14 维）
  >
  > 动作对（Action）$a_{t:t+k}$：
  >
  > - Ground Truth：未来 k 步的关节角度命令（来自人类演示）
  >
  > 数据关系：当前观测 → 预测未来 k 步动作序列

- **块大小 k**：每次预测的动作序列长度
- **权重 β**：正则化项的平衡权重

第 2 步：符号定义

- a_t：时刻 t 的动作
- o_t：时刻 t 的完整观测（包含图像 + 关节位置）
- $\bar{o}_t$：时刻 t 的简化观测（仅关节位置，不含图像）

第 3 步：初始化编码器

- $q_\phi(z|a_{t:t+k}, \bar{o}_t)$：编码器网络

  > 编码器不使用图像的设计体现了 ACT 的核心思想：
  >
  > - **风格与内容分离**：风格存在于动作模式中，内容存在于环境状态中
  > - **效率优先**：避免不必要的计算复杂度
  > - **功能专业化**：每个组件专注于特定类型的信息处理

- **功能**：根据动作序列和简化观测推断风格变量 z 的分布
- **参数ϕ**：编码器的可学习参数

第 4 步：初始化解码器

- $\pi_\theta(\hat{a}_{t:t+k}|o_t, z)$：解码器网络（策略网络）
- **功能**：根据完整观测和风格变量预测动作序列
- **参数**θ：解码器的可学习参数

训练循环：

第 5 步：开始训练迭代

- $n = 1, 2, \ldots$：训练轮次计数

第 6 步：数据采样

- **从数据集**$\mathcal{D}$**中采样**：$(o_t, a_{t:t+k})$
- o_t：当前时刻的观测（4 个 RGB 图像 + 14 维关节位置）
- $a_{t:t+k}$：从时刻 t 开始的 k 步动作序列

第 7 步：风格变量采样

- **编码器推断**：$z \sim q_\phi(z|a_{t:t+k}, \bar{o}_t)$
- **输入**：真实动作序列 + 简化观测
- **输出**：风格变量 z 的采样值
- **技术**：使用重参数化技巧进行可微分采样

第 8 步：动作序列预测

- **解码器预测**：$\hat{a}_{t:t+k} = \pi_\theta(\hat{a}_{t:t+k}|o_t, z)$
- **输入**：完整观测 + 采样的风格变量
- **输出**：预测的 k 步动作序列

第 9 步：计算重建损失

- $\mathcal{L}_{reconst} = MSE(\hat{a}_{t:t+k}, a_{t:t+k})$
- **功能**：衡量预测动作与真实动作的差异
- **损失类型**：均方误差（论文中实际使用 L1 损失）

第 10 步：计算正则化损失

- $\mathcal{L}_{reg} = D_{KL}(q_\phi(z|a_{t:t+k}, \bar{o}_t) \| \mathcal{N}(0, I))$
- **功能**：KL 散度，使编码器输出的 z 分布接近标准正态分布
- $\mathcal{N}(0, I)$：标准多元正态分布（均值 0，协方差矩阵为单位矩阵）

第 11 步：参数更新

- **优化器**：ADAM
- **总损失**：$\mathcal{L} = \mathcal{L}_{reconst} + \beta\mathcal{L}_{reg}$
- **更新参数**：θ（解码器）和ϕ（编码器）
- β**作用**：平衡重建精度和风格变量的正则化

4.3.3 训练流程总结

数据流向

```
演示数据 → 编码器推断风格 → 解码器预测动作 → 计算损失 → 更新参数
```

关键特点

1. **联合训练**：编码器和解码器同时训练
2. **CVAE 结构**：条件变分自编码器框架
3. **重参数化**：保证采样过程可微分
4. **双重损失**：重建损失确保预测准确，正则化损失防止过拟合

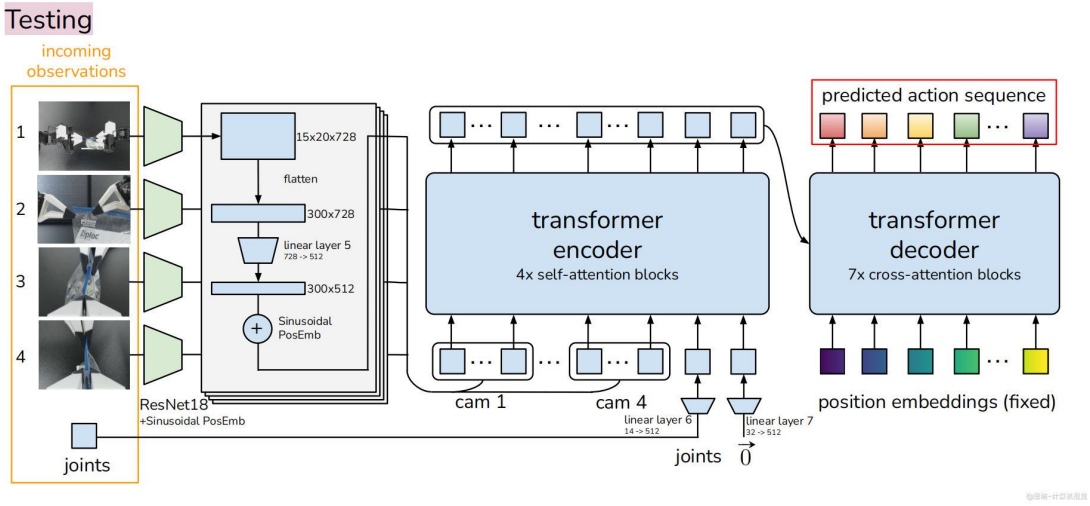
训练目标

- **编码器学习**：如何从演示中提取行为风格
- **解码器学习**：如何根据观测和风格生成合适的动作序列
- **整体目标**：学会在测试时仅用观测生成高质量的动作序列

数学表达式汇总

- **编码器分布**：$q_\phi(z|a_{t:t+k}, \bar{o}_t)$
- **解码器策略**：$\pi_\theta(\hat{a}_{t:t+k}|o_t, z)$
- **重建损失**：$\mathcal{L}_{reconst} = MSE(\hat{a}_{t:t+k}, a_{t:t+k})$
- **正则化损失**：$\mathcal{L}_{reg} = D_{KL}(q_\phi(z|a_{t:t+k}, \bar{o}_t)||\mathcal{N}(0, I))$
- **总损失**：$\mathcal{L} = \mathcal{L}_{reconst} + \beta\mathcal{L}_{reg}$

4.4 推理流程 (Inference Pipeline)



4.4.1 整体流程概览

网络简化

- 移除编码器：测试时不需要风格推断
- 固定风格变量：z=0 (使用先验分布均值)
- 仅保留解码器：作为确定性策略网络

推理步骤

- 输入观测：当前 4 个 RGB 图像 + 关节位置
- 特征提取：与训练时相同的图像和关节处理
- 序列生成：解码器直接输出 k 步动作预测
- 时序集成：对重叠预测进行加权平均

4.4.2 详细算法流程

Algorithm 2 ACT Inference

1: Given: trained π_θ, episode length T, weight m.
2: Initialize FIFO buffers $\mathcal{B}[0:T]$, where $\mathcal{B}[t]$ stores actions predicted *for* timestep t.
3: **for** timestep $t = 1, 2, ...T$ **do**
4: Predict $\hat{a}_{t:t+k}$ with $\pi_\theta(\hat{a}_{t:t+k}|o_t, z)$ where $z = 0$
5: Add $\hat{a}_{t:t+k}$ to buffers $\mathcal{B}[t:t+k]$ respectively
6: Obtain current step actions $A_t = \mathcal{B}[t]$
7: Apply $a_t = \sum_i w_i A_t[i] / \sum_i w_i$, with $w_i = \exp(-m * i)$

参数定义与初始化

第 1 步：给定参数

- **训练好的策略** π_θ：已完成训练的解码器网络（编码器已移除）
- **任务时长** T：完整任务执行的总时间步数
- **权重** m：时序集成中的衰减参数，控制新旧预测的相对权重

第 2 步：初始化缓冲区系统

- **FIFO 缓冲区** $\mathcal{B}[0:T]$：先进先出的环形缓冲区数组
- $\mathcal{B}[t]$：存储所有"预测在时刻t执行的动作"
- **功能**：管理重叠的动作预测，实现时序集成

推理循环

第 3 步：开始时序推理

- $t = 1, 2, \ldots T$：按时间步顺序执行任务

第 4 步：策略预测

- $\hat{a}_{t:t+k} = \pi_\theta(\hat{a}_{t:t+k}|o_t, z)$ where $z = 0$
- **输入观测** o_t：当前时刻的完整观测（4 个 RGB 图像 + 14 维关节位置）
- **固定风格变量**：$z = 0$（使用先验分布的均值，移除随机性）
- **输出**：从时刻t开始的k步动作序列预测

第 5 步：缓冲区更新

- **将 $\hat{a}_{t:t+k}$ 分别添加到缓冲区 $\mathcal{B}[t:t+k]$**
- **分配预测**：将k步预测分别添加到对应时刻的缓冲区
- $\mathcal{B}[t] \leftarrow \mathcal{B}[t] \cup \{\hat{a}_t\}$，$\mathcal{B}[t+1] \leftarrow \mathcal{B}[t+1] \cup \{\hat{a}_{t+1}\}$，....，$\mathcal{B}[t+k-1] \leftarrow \mathcal{B}[t+k-1] \cup \{\hat{a}_{t+k-1}\}$
- **重叠积累**：多个预测可能对同一时刻产生不同的动作建议

第 6 步：获取当前时刻动作集合

- $A_t = \mathcal{B}[t]$
- **功能**：提取所有针对当前时刻t的动作预测
- **内容**：可能包含来自不同历史时刻预测的多个动作值

第 7 步：时序集成与执行

- $a_t = \frac{\sum_i w_i A_t[i]}{\sum_i w_i}$
- **权重计算**：$w_i = \exp(-m \times i)$
- **i含义**：预测的"年龄"（0 表示当前预测，1 表示 1 步前的预测，依此类推）
- **加权平均**：越新的预测权重越高，越旧的预测权重越低
- **最终执行**：a_t作为当前时刻的实际执行动作

原著代码实现：

```
if config['policy_class'] == "ACT":
  if t % query_frequency == 0:
    all_actions = policy(qpos, curr_image)
    if temporal_agg:
      all_time_actions[[t], t:t+num_queries] = all_actions
      actions_for_curr_step = all_time_actions[:, t]
      actions_populated = torch.all(actions_for_curr_step != 0, axis=1)
      actions_for_curr_step = actions_for_curr_step[actions_populated]
      k = 0.01
      exp_weights = np.exp(-k * np.arange(len(actions_for_curr_step)))
      exp_weights = exp_weights / exp_weights.sum()
      exp_weights = torch.from_numpy(exp_weights).cuda().unsqueeze(dim=1)
      raw_action = (actions_for_curr_step * exp_weights).sum(dim=0, keepdim=True)
    else:
      raw_action = all_actions[:, t % query_frequency]
```

4.4.3 推理流程总结

数据流向

```
当前观测 → 策略预测k步动作 → 更新缓冲区 → 时序集成 → 执行动作
```

FIFO 缓冲区管理

- **多预测存储**：每个时刻可能有多个来源的动作预测
- **自动更新**：新预测自动添加到相应时刻
- **内存高效**：固定大小的环形缓冲区

时序集成策略

- **指数衰减权重**：$w_i = \exp(-m \times i)$
  - m**值越大**：更偏重新预测
  - m**值越小**：新旧预测权重更平衡
- **加权平均**：所有预测按权重融合
- **平滑执行**：避免动作突变，提高执行稳定性

推理特点

1. **确定性执行**：$z = 0$移除随机性，保证可重复性
2. **实时预测**：每个时刻都生成新的k步预测
3. **重叠融合**：多个预测为同一时刻提供不同建议
4. **自适应调整**：新观测信息不断融入决策过程

与训练的差异

方面	训练阶段	推理阶段
网络结构	编码器 + 解码器	仅解码器
风格变量	从数据推断	固定为 0
输入信息	观测 + 真实动作（主臂角度）	仅当前观测
输出处理	直接计算损失	时序集成后执行
计算模式	批处理训练	实时单步推理

附录一、策略（Policy）

策略就是"智能体的行为准则"：告诉机器人/AI 系统在特定情况下应该做什么的决策函数。

1. 核心定义

- 策略 = 在给定情况下，应该采取什么行动的决策规则
- 数学表示：$\pi(a|s)$ = "在状态 s 下选择动作 a 的概率"

2. 历史发展脉络

时期	领域	策略含义	特点
1940s	博弈论	完整行动计划	静态、预设
1950s	控制论	控制法则	动态反馈、数学化
1980s	强化学习	状态 → 动作映射	概率化、可学习
2000s	机器人学习	行为决策模型	端到端、神经网络

3. 通用模式

策略的本质

```
策略 = 情境 → 行动 的映射关系
```

工作流程

```
1. 感知环境（观察状态s）
   ↓
2. 查询策略（π函数）
   ↓
3. 选择动作（输出a）
   ↓
4. 执行行动
   ↓
5. 观察结果（可能更新策略）
```

4. 在机器人学习中的具体含义

比如在 ACT 中的策略：

$\pi_\theta(a_{t:t+k}|s_t)$ = 机器人的"决策大脑"

```
输入：当前看到的环境情况
输出：接下来k步要执行的动作序列
功能：告诉机器人"在这种情况下应该怎么做"
```

5. 为什么策略概念重要？

- **统一框架**：为不同领域的决策问题提供通用语言
- **数学化**：可以用严格数学描述和分析
- **可优化**：可以通过学习算法不断改进
- **可实现**：可以用程序代码具体实现
- **可评估**：可以量化策略的好坏

应用广泛性

```
博弈 → 控制 → 机器学习 → 机器人 → 自动驾驶 → 推荐系统 → ...
```

附录二、马尔可夫 vs 非马尔可夫行为详解

2.1. 基本概念

马尔可夫性质

- 定义：未来状态只依赖于当前状态，与历史无关
- 数学表达：$P(s_{t+1}|s_t, s_{t-1}, \ldots, s_0) = P(s_{t+1}|s_t)$
- 例子——**下棋游戏**：
  - 当前棋盘状态包含了所有决策所需信息
  - 不需要知道具体是怎么走到这个棋盘状态的
  - 下一步最优走法只看当前棋盘

非马尔可夫行为

- 定义：未来状态/动作依赖于历史信息，不仅仅是当前状态
- 数学表达：$P(s_{t+1}|s_t, s_{t-1}, \ldots, s_0) \neq P(s_{t+1}|s_t)$
- 例子——**股票投资**：
  - 股票当前价格：100 元
  - 但投资决策还需要知道：
    - 是从 90 元涨到 100 元？（上升趋势，可能继续买入）
    - 还是从 110 元跌到 100 元？（下降趋势，可能卖出）
  - 相同的当前状态，不同的历史导致不同的决策

2.2. 机器人学习中的例子（ACT 论文）

2.2.1 马尔可夫任务示例

抓取固定物体

```
状态：机器人手臂位置 + 物体位置 + 夹爪状态
动作：只需根据当前状态决定下一步动作
历史：不重要，当前几何关系决定一切
```

为什么是马尔可夫的？

- 物体位置固定，环境静态
- 最优动作只依赖当前的空间几何关系
- 历史轨迹不影响决策

2.2.2 非马尔可夫任务示例

例子 1：人类演示中的"停顿"行为

场景：开调料杯任务

```
# 时间序列演示数据
t=10: 右手接近杯子 (正常速度)
t=11: 右手接近杯子 (正常速度)
t=12: 右手停顿     (速度=0)    # 人类在思考/观察
t=13: 右手停顿     (速度=0)    # 人类在思考/观察
t=14: 右手推杯子   (恢复动作)
```

马尔可夫 policy 的困惑：

- 在 t=12 时刻：`state = [手臂位置A, 杯子位置B]`
- 在 t=14 时刻：`state = [手臂位置A, 杯子位置B]` (相同状态！)
- 但应该采取的动作不同：
  - t=12: 继续停顿
  - t=14: 开始推杯子

非马尔可夫性：

- 相同的空间状态，不同的时间上下文
- 需要知道"已经停顿了多久"才能决定下一步

例子 2：渐进式调整行为

场景：插电池任务


```
# 人类演示轨迹
阶段1: 粗略对准电池和槽口
阶段2: 精细调整角度 (小幅震荡)
阶段3: 最终插入
```

状态看起来相似，但行为不同：

```
# 两个时刻的相似状态
时刻A: [电池位置=(10,20), 角度=45°, 接触力=0.1N]
时刻B: [电池位置=(10,20), 角度=45°, 接触力=0.1N]

# 但处于不同阶段
时刻A: 刚开始精调 → 应该继续小幅调整
时刻B: 调整了30秒 → 应该尝试插入
```

2.3. 传统单步 Policy 的问题

单步预测的局限

```
# 传统BC方法
def policy(current_state):
    return predict_action(current_state)  # 只看当前帧

# 面临的问题
相同的current_state → 可能需要不同的action
```

具体失败模式

- 无限停顿问题

```
# 在演示数据中，人类在某状态停顿了3秒
state_pause = [hand_pos=(5,10), object_pos=(20,30)]

# 单步policy学到：在这个状态 → 动作=停顿
# 测试时：机器人可能在这个状态无限停顿
```

- 时序混乱问题

```
# 演示中的正确序列
t1: 接近物体（慢速）
t2: 接近物体（慢速）
t3: 接近物体（快速）# 人类决定加速
t4: 抓取物体

# 单步policy可能学到：
在"接近物体"状态 → 随机选择慢速/快速
导致运动不连贯
```

2.4. Action Chunking 如何解决

- 时间窗口建模

```
# 传统方法
action_t = policy(state_t)  # 单点决策

# Action Chunking
action_sequence = policy(state_t)  # 预测未来k步
# action_sequence = [a_t, a_t+1, ..., a_t+k-1]
```

- 解决停顿问题

原理：在一个 chunk 内部处理时间依赖

```
# Chunk包含停顿→行动的完整序列
chunk_example = [
    action_t=0:   "接近",     # 开始接近
    action_t+1:   "停顿",     # 观察思考
    action_t+2:   "停顿",     # 继续观察
    action_t+3:   "推杯子"    # 执行动作
]

# 这样policy学会了"停顿多久后该行动"
```

- 解决协调问题

原理：chunk 内部保持动作的时序一致性

```
# 双手协调的chunk
bimanual_chunk = [
    t: [left_hand="hold_steady", right_hand="approach"],
    t+1: [left_hand="hold_steady", right_hand="fine_adjust"],
    t+2: [left_hand="hold_steady", right_hand="grasp"],
    t+3: [left_hand="coordinate_pull", right_hand="coordinate_pull"]
]

# 确保双手动作的时序匹配
```

2.5. 总结

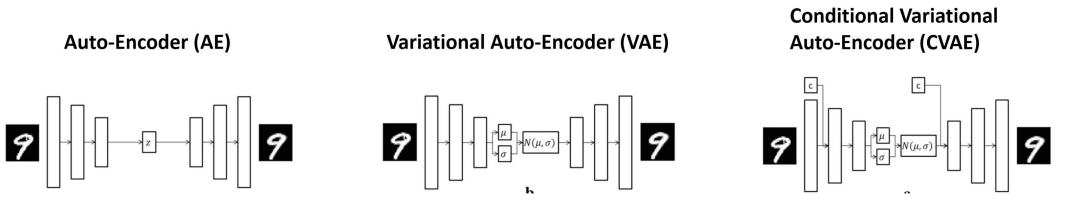
- 马尔可夫 vs 非马尔可夫对比表

特征	马尔可夫行为	非马尔可夫行为
决策依赖	仅当前状态	当前状态 + 历史
时序重要性	不重要	很重要
典型场景	静态环境抓取	人类演示、协调任务
Policy 复杂度	简单	复杂
Chunking 收益	小	大

- Action Chunking 的核心价值
  - 将非马尔可夫问题转化为马尔可夫问题
  - 在 chunk 内部处理时间依赖关系
  - 保持动作序列的时序一致性
  - 特别适合处理人类演示数据中的复杂时序模式

这就是为什么 ACT 在处理精细操作任务时，相比传统方法有如此显著的性能提升。

附录三、条件变分自编码器



3.1 AE vs VAE vs CVAE 对比

生活例子，想象你是一个画家：

- **普通自编码器**（AE）：就像照着原画临摹，能画出很像的作品，但不能创作新的
- **变分自编码器**（VAE）：不仅能临摹，还能理解绘画的"精髓"，创作出全新但合理的作品
- **条件变分自编码器**（CVAE）：就像一位"私人定制画家"，不仅能创作新作品，还能按客户需求画画：
  - 客户说"我要一朵玫瑰"→ 画出各种风格的玫瑰（写实的、印象派的、素描的...）

3.2. 标准自编码器 (AE)

目标: 学会重构输入数据

```
输入图像 → [编码器] → 潜在向量 → [解码器] → 重构图像
```

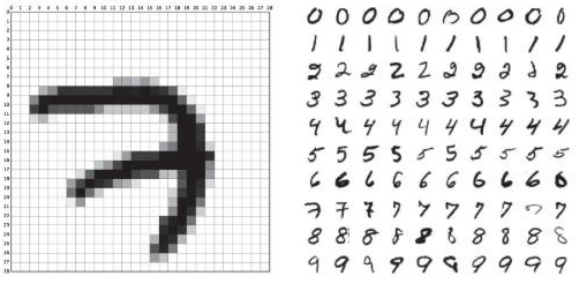
特点:

- 能很好地重构训练数据
- **无法生成新样本** - 这是关键问题！
- 原因：如果我们随机选择一个潜在向量输入解码器，很可能得到无意义的输出

如果你不了解编码器、潜在向量、解码器，请继续看：

3.2.1. [编码器] - 数据压缩过程

技术实现: 通常是多层神经网络，逐步降维



(a) MNIST sample belonging to the digit '7'. (b) 100 samples from the MNIST training set.

```
# 以MNIST为例（28×28像素图像）
输入：784维向量（28×28展平）
↓ 全连接层1：784 → 512
↓ 激活函数：ReLU
↓ 全连接层2：512 → 256
↓ 激活函数：ReLU
↓ 全连接层3：256 → 64
输出：64维潜在向量
```

3.2.2. 潜在向量 - 压缩表示

> 英文表示： Latent Vector、 Latent Space 等

本质: 原始数据的低维度抽象表示

- **维度:** 远小于原始数据 (64 维 vs 784 维)
- **含义:** 包含重构原数据的关键信息
- **类比:** 就像用几个关键词概括一篇文章

```
# 具体例子
原图像：[0.1, 0.8, 0.3, ..., 0.9]  # 784个数字
潜在向量：[2.1, -0.5, 1.8, 0.3]    # 只有4个数字(简化例子)
```

3.2.3. [解码器] - 数据重构过程

技术实现: 编码器的"反向"过程，逐步升维

```
# 解码器结构
输入：64维潜在向量
↓ 全连接层1：64 → 256
↓ 激活函数：ReLU
↓ 全连接层2：256 → 512
↓ 激活函数：ReLU
↓ 全连接层3：512 → 784
↓ 激活函数：Sigmoid（输出0-1像素值）
输出：784维重构图像
```

压缩迫使模型学习数据的核心特征而非死记硬背，获得去噪、泛化能力，且压缩后的潜在空间可用于生成新样本。

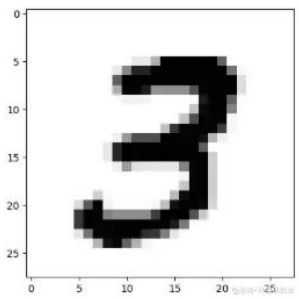
简单说：**压缩=理解本质，解码=验证理解，最终目标是生成新内容。**

想象你在传纸条：

- **编码器:** 把长句子压缩成几个关键词
- **潜在向量:** 这几个关键词
- **解码器:** 根据关键词还原出完整句子

3.2.4 具体数字示例

假设处理一个手写数字"3"：




```
1. 输入图像：28×28 = 784个像素值
   [0.0, 0.1, 0.8, 0.9, 0.7, ..., 0.0]

2. 编码器处理：
   784 → 512 → 256 → 64
```

```
3. 潜在向量（64维）：
   [1.2, -0.8, 2.1, 0.5, ..., -1.1] # 这64个数字包含了"3"的核心特征

4. 解码器处理：
   64 → 256 → 512 → 784

5. 重构图像：784个像素值
   [0.02, 0.09, 0.79, 0.91, 0.68, ..., 0.01] # 尽量接近原图
```

3.3. 变分自编码器 (VAE)

目标: 既能重构，又能生成新样本

```
输入图像 → [编码器] → 分布参数(μ, σ) → 采样 → [解码器] → 重构图像
```

关键改进:

- **编码器输出概率分布**（均值和方差），而不是固定值
- **可以生成新样本** - 从标准正态分布采样点输入解码器
- 使用 KL 散度确保潜在分布接近标准正态分布

比喻: 想象 AE 是"死记硬背"，VAE 是"理解规律"

3.4. 条件变分自编码器 (CVAE)

目标: 根据指定条件生成特定类型的样本

```
输入图像+标签 → [编码器] → 分布参数 → 采样+标签 → [解码器] → 重构图像
```

核心优势:

- **可控生成** - 想要数字"3"就能生成"3"
- 潜在空间编码风格信息（粗细、角度等），标签编码内容信息

> CVAE 作为早期的生成建模框架，虽然在 ACT 中表现良好，但其生成能力相比近年来兴起的 diffusion 模型仍有显著差距。随着技术演进，后续的视觉-语言-动作(VLA)算法开始广泛采用 diffusion 机制，典型代表包括 Diffusion Policy 和 Pi0 等方法。