Diffusion Policy 学习笔记

INTRODUCTION

Policy learning from demonstration

一种机器学习方法，主要用于训练机器人或智能体（agent）来完成特定任务。它的核心思想是通过观察和学习人类或专家的示范动作，来自动生成策略，使机器人能够自主执行类似的任务。

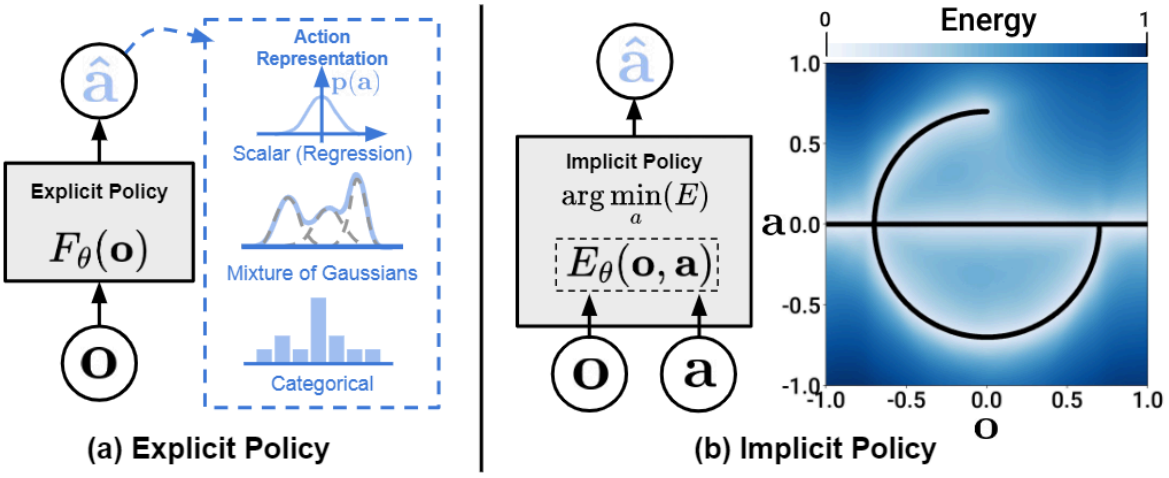
Explicit Policy： 一个直白自然的想法，把这视为一个**supervised regression task**，学习如何从输入的观察数据（如视觉输入或传感器数据）映射到输出的动作（如机器人手臂的移动）

然而，这样做的效果非常糟糕：对比其它的监督学习任务，predicting robot actions具备一些难点：

1. multimodal distributions过于复杂；
2. sequential correlation序列操作之间存在前后关联
3. requirement of high precision

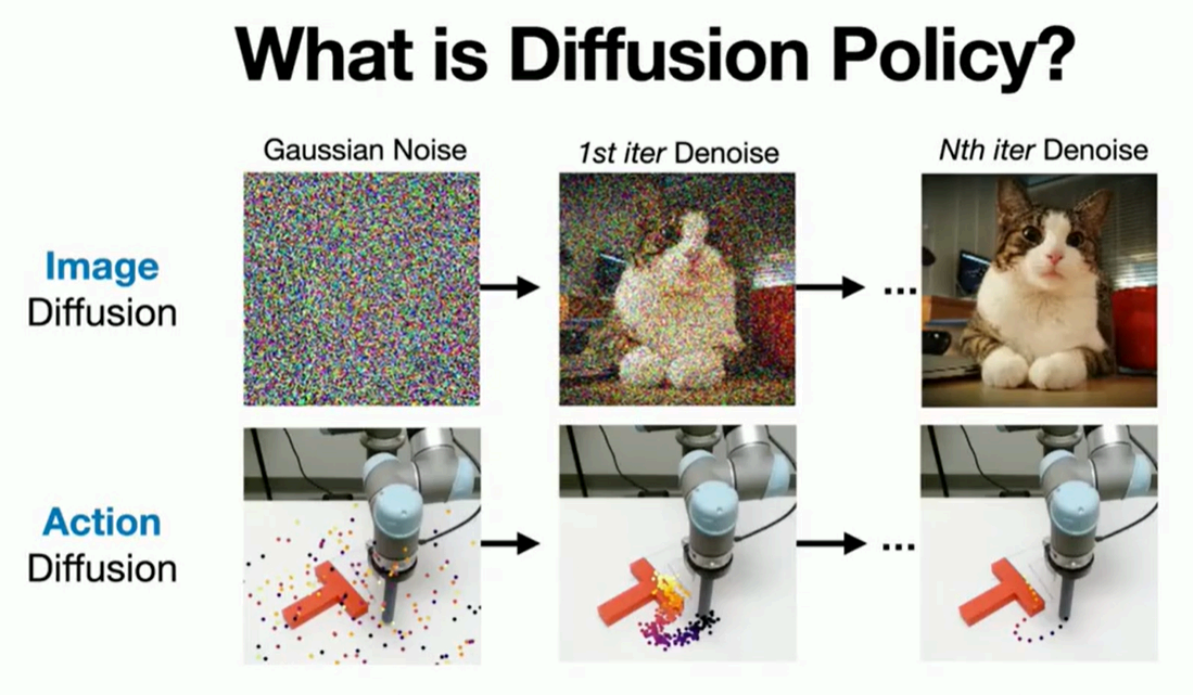
为了解决这些问题，前人做出的尝试包括：

1. 尝试各种action representations：比如Gaussians的混合、将action离散化后分类
2. 转向**Implicit Policy**，通过优化能量函数或其他间接方法来推导动作

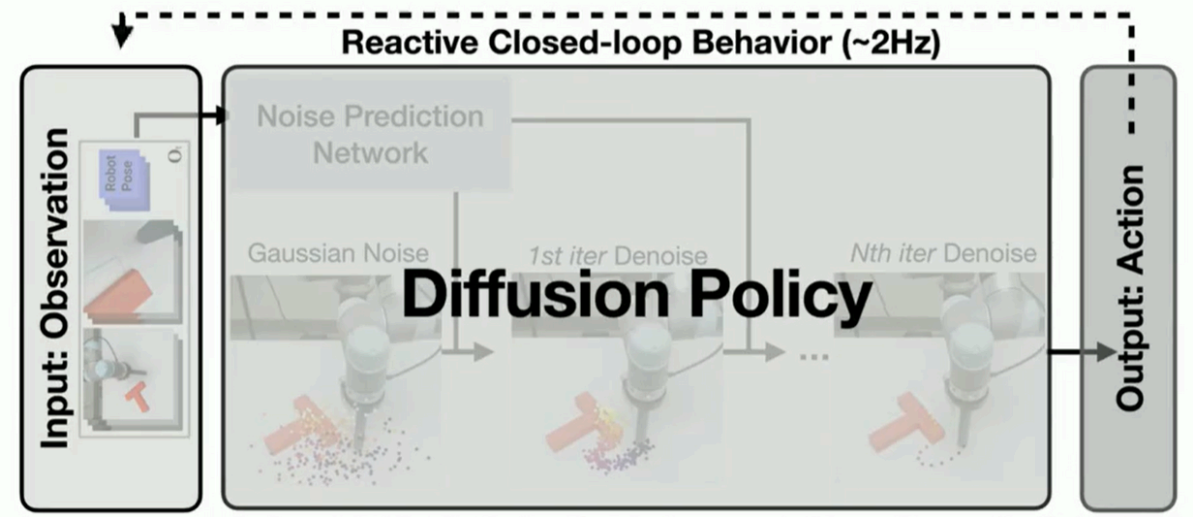


Diffusion Policy的初心是非常自然的,也就是用diffusion的方法来替换能量函数这个东西，生成出动作的序列

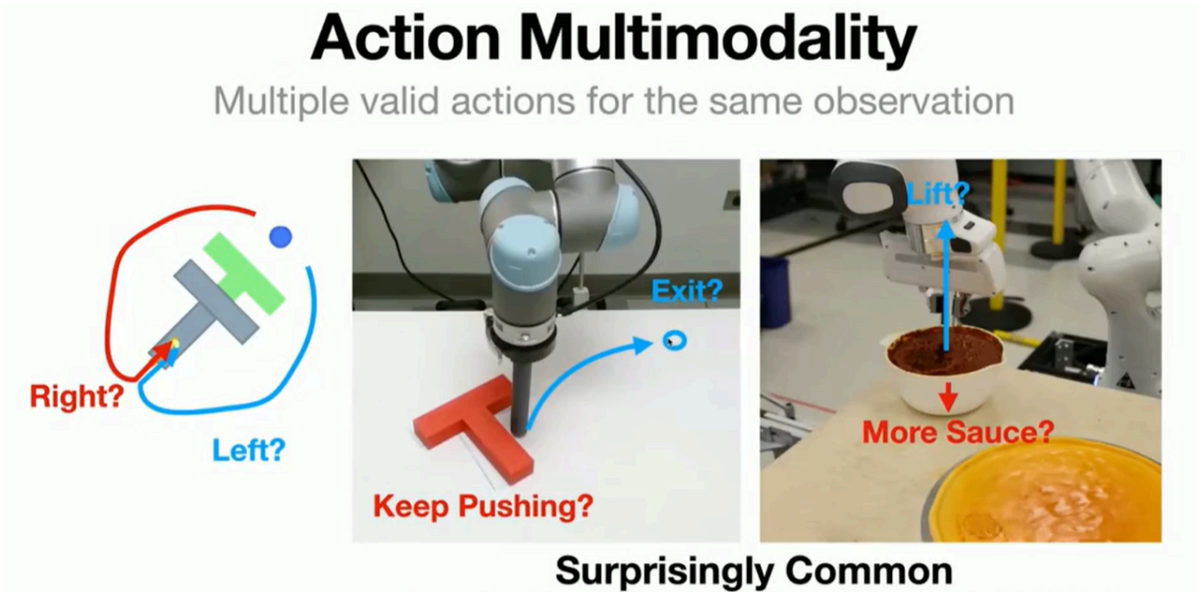
具体来说，就像生成图像的扩散模型通过去噪生成图片一样，Diffusion Policy希望对action的“点集”进行去噪，得到一条简单清晰的轨迹



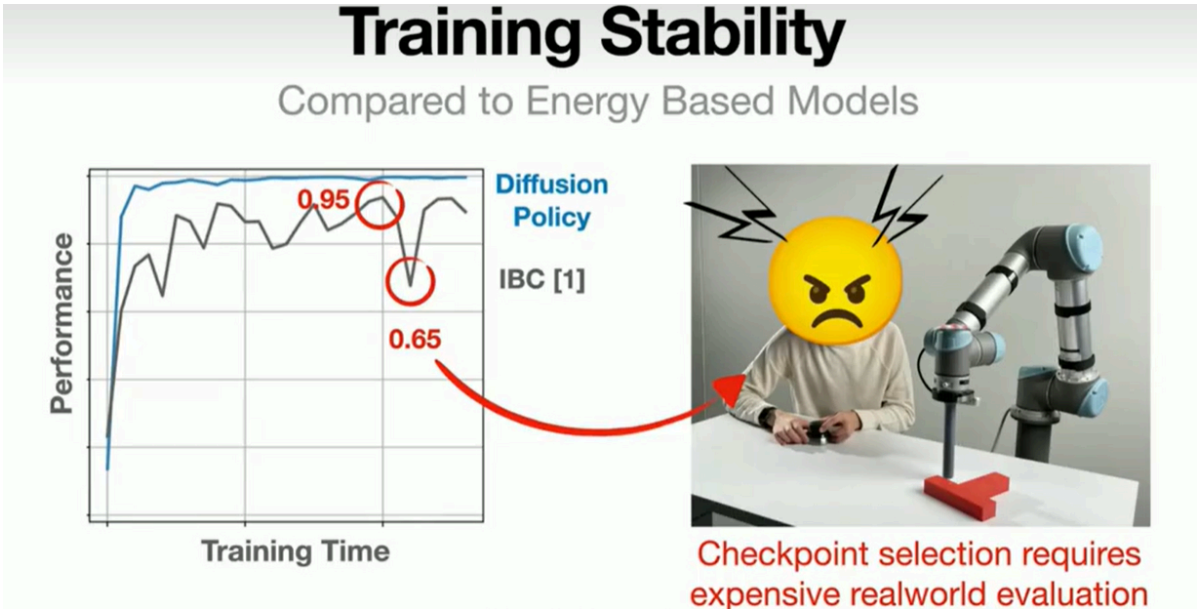
Diffusion Policy的流程如下，把observation作为输入，通过预测noise来对去噪，最终得到action，而这整个过程是Closed-loop的，这个想法非常自然，事实上两年前Microsoft和德国的KIT也在同时间完成了类似的工作



这个做法的好处在于，它非常完美地适配了真实action的多模态性（multimodality），以下图为例，想把T推进绿框，可以从右边推也可以从左边推，如果要通过一个函数来预测这样的行为，这会非常的纠结。但diffusion的方法就能很好地从示范中捕捉到这样的性质，自然地生成出多种可能的操作。同样，对于推这个动作，随时我们可能都会决定"嗯，我已经推好了"然后选择一个地方，把手缩回去，这个缩回去的地方，以及停手的时间，都是具有multimodality的，而diffusion能很好地捕捉这些信息。



Cheng Chi还强调了该方法在训练上的稳定性，考虑到真实测试中的硬件开销问题，这样的稳定性是很重要的，而Diffusion Policy做得很好



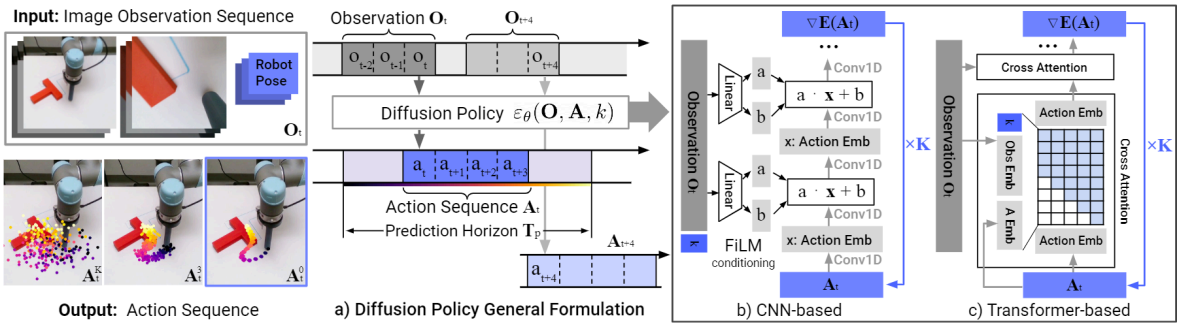
对比同期的工作，Cheng Chi在近日的演讲前做了个总结：他认为自己的方法能做得更好，很大程度上是因为自己预测的是未来的动作序列，而不是一个瞬间的动作。并且认为，**只要是通过预测动作序列而不是一个函数来生成一个动作,加上一个能捕捉multimodality的模型,就很可能能取得好的成果**

DIFFUSION POLICY FORMULATION

如前文所述，该方法的优势在于扩散策略能够表达复杂的多模态动作分布，并且具有稳定的训练行为——几乎不需要特定任务的超参数调整

在该章节的Denoising Diffusion Probabilistic Models和DDPM Training里，作者介绍了裸的DDPM推导和训练方法，这里省略

而在Diffusion for Visuomotor Policy Learning这一板块，考虑使用 DDPM 学习机器人视觉运动策略。这需要在公式上进行两项主要修改：1. 将输出 x 更改为表示机器人动作。2. 使去噪过程依赖于输入观测 O_t



在这里,作者提出了几个重要的概念:

Closed-loop action-sequence prediction：一个有效的动作公式应鼓励长时间规划中的时间一致性和平滑性，同时允许快速响应意外观测。

1. **闭环动作序列预测**：闭环系统是指系统的输出会反馈到输入，从而形成一个反馈回路。在机器人控制中，闭环动作序列预测意味着机器人在执行动作的过程中，会不断地利用最新的观测数据来调整和优化接下来的动作序列。
2. **时间一致性和平滑性**：
    - **时间一致性**：指的是机器人在长时间执行任务时，其动作应该具有连贯性和一致性，避免突然的、不连续的动作变化。这对于确保任务的顺利执行和避免机器人在长时间工作中出现错误非常重要。
    - **平滑性**：指的是机器人动作的变化应该是平滑的，即动作之间的过渡应该尽可能地平顺。这可以减少机器人在执行任务时的抖动和不稳定性，提高动作的可靠性和精确性。
3. **快速响应意外观测**：
    - **快速响应**：意味着机器人能够在遇到意外情况或新的观测数据时，迅速调整其动作计划。例如，如果机器人在行进过程中突然发现障碍物，它需要能够快速反应，改变路径以避免碰撞。
    - **意外观测**：指的是在执行任务过程中，机器人可能会遇到一些未预料到的情况或新的环境信息。这些信息需要被迅速处理，以便机器人能够及时做出调整。

为了实现这一目标，将扩散模型生成的动作序列预测与退缩域控制相结合，具体而言，在时间步 t，策略将最新的 T_o 步观测数据 O_t 作为输入，并预测 T_p 步动作，其中 T_a 步动作在不重新规划的情况下在机器人上执行。这里，我们定义 T_o 为观测视界，T_p 为动作预测视界，T_a 为动作执行视界。

退缩域控制（Receding Horizon Control, RHC）：

- **退缩域控制**是一种常用的控制策略，也称为移动时间窗口控制（Model Predictive Control, MPC）。它通过在每个时间步上优化一个有限的未来时间窗口内的控制动作，然后只执行优化结果中的第一个动作，并在下一个时间步重复这一过程。
- 这种方法允许系统在不断更新的观测数据基础上进行实时优化，从而能够更好地应对动态变化的环境。

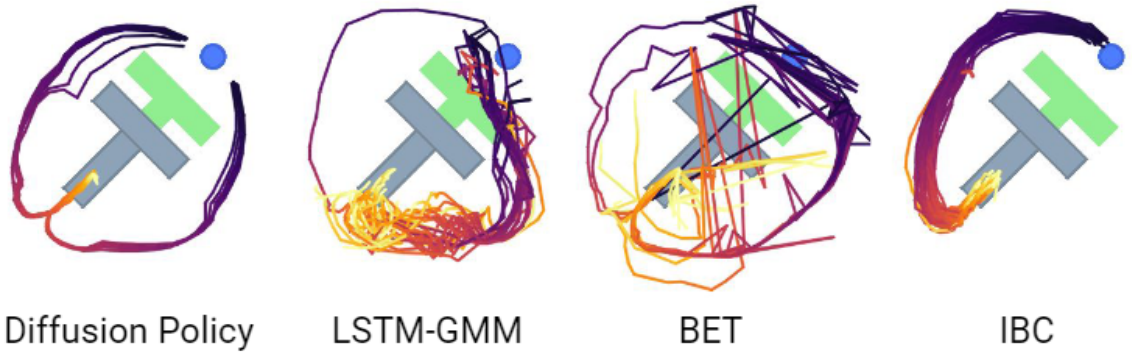
INTRIGUING PROPERTIES OF DIFFUSION POLICY

这一章主要是提供了一些很直觉的理解，讲解了其扩散模型对于其他策略表示形式的优势。

建模多模态动作分布

扩散策略能够自然且精确地表达多模态分布，

直观上，扩散策略中的动作生成的多模态性来源于两个方面——基础的随机采样过程和随机初始化。



代码部分

训练的入口在diffusion_policy/train.py

训练的本体在
diffusion_policy/diffusion_policy/workspace/train_diffusion_unet_hybrid_workspace.py

训练部分的计算损失

```
raw_loss = self.model.compute_loss(batch)
loss = raw_loss / cfg.training.gradient_accumulate_every
loss.backward()
```

训练部分的evaluation

```
                policy = self.model
                if cfg.training.use_ema:
                    policy = self.ema_model
                policy.eval()
                # 运行验证
                if (self.epoch % cfg.training.val_every) == 0:
                    with torch.no_grad():
                        val_losses = list()
                        with tqdm.tqdm(val_dataloader, desc=f"Validation epoch
{self.epoch}",
                                leave=False,
mininterval=cfg.training.tqdm_interval_sec) as tepoch:
                            for batch_idx, batch in enumerate(tepoch):
                                batch = dict_apply(batch, lambda x: x.to(device,
non_blocking=True))
                                loss = self.model.compute_loss(batch)
                                val_losses.append(loss)
                                if (cfg.training.max_val_steps is not None) \
                                    and batch_idx >= (cfg.training.max_val_steps-
1):
                                    break
                        if len(val_losses) > 0:
                            val_loss =
torch.mean(torch.tensor(val_losses)).item()
                            # 记录 epoch 平均验证损失
                            step_log['val_loss'] = val_loss
```

```
                # 在训练批次上运行扩散采样
                if (self.epoch % cfg.training.sample_every) == 0:
                    with torch.no_grad():
                        # 从训练集中采样轨迹，并评估差异
                        batch = dict_apply(train_sampling_batch, lambda x:
x.to(device, non_blocking=True))
                        obs_dict = batch['obs']
                        gt_action = batch['action']

                        result = policy.predict_action(obs_dict)
                        pred_action = result['action_pred']
                        mse = torch.nn.functional.mse_loss(pred_action,
gt_action)
                        step_log['train_action_mse_error'] = mse.item()
                        del batch
                        del obs_dict
                        del gt_action
                        del result
                        del pred_action
                        del mse
```