imitation learning

behavior cloning（lecture2 part2）

定义：直接使用监督学习的方法，将专家数据看作样本输入

目标

我们在$p_{\text{data}}(\mathbf{o}_t)$下训练，但是我们用$p_{\pi_\theta}(\mathbf{o}_t)$测试，但$p_{\text{data}}(\mathbf{o}_t) \neq p_{\pi_\theta}(\mathbf{o}_t)$

所以不考虑以下目标函数

$$\max_\theta E_{\mathbf{o}_t \sim p_{\text{data}}(\mathbf{o}_t)}[\log \pi_\theta(\mathbf{a}_t|\mathbf{o}_t)]$$

首先，定义损失函数为

$$c(\mathbf{s}_t, \mathbf{a}_t) = \begin{cases} 0 \text{ if } \mathbf{a}_t = \pi^\star(\mathbf{s}_t) \\ 1 \text{ otherwise} \end{cases}$$

我们关心的是π_θ情况，我们需要做的是

$$\text{minimize } E_{\mathbf{s}_t \sim p_{\pi_\theta}(\mathbf{s}_t)}[c(\mathbf{s}_t, \mathbf{a}_t)]$$

假设：$\pi_\theta(\mathbf{a} \neq \pi^\star(\mathbf{s})|\mathbf{s}) \leq \epsilon$，对于$\mathrm{s} \in \mathcal{D}_{\text{train}}$

因此有

$$E\left[\sum_t c(\mathbf{s}_t, \mathbf{a}_t)\right] \leq \underbrace{\epsilon T + (1-\epsilon)(\epsilon(T-1) + (1-\epsilon)(\ldots))}_{T \text{ terms, each } O(\epsilon T)} = O(\epsilon T^2)$$

这个式子表明BC的cost会随着决策步数的增加而呈现平方次增加

更加泛化的说，对于$\mathbf{s} \sim p_{\text{train}}(\mathbf{s})$，我们假设$E_{p_{\text{train}}(\mathbf{s})}[\pi_\theta(\mathbf{a} \neq \pi^\star(\mathbf{s})|\mathbf{s})] \leq \epsilon$

对于$p_{\text{train}}(\mathbf{s}) \neq p_\theta(\mathbf{s})$，我们有

$$p_\theta(\mathbf{s}_t) = (1-\epsilon)^t p_{\text{train}}(\mathbf{s}_t) + (1-(1-\epsilon)^t))p_{\text{mistake}}(\mathbf{s}_t)$$

式子前一项表明不犯错（即train data）的概率，后一项表明其他分布

然后有

$$|p_\theta(\mathbf{s}_t) - p_{\text{train}}(\mathbf{s}_t)| = (1-(1-\epsilon)^t)|p_{\text{mistake}}(\mathbf{s}_t) - p_{\text{train}}(\mathbf{s}_t)| \leq 2(1-(1-\epsilon)^t)$$

其中$(1-\epsilon)^t \geq 1-\epsilon t, \quad for\ \epsilon \in [0,1]$，所以小于$2\epsilon t$

对于我们的目标函数来说

$$\begin{aligned}\sum_t E_{p_\theta(\mathbf{s}_t)}[c_t] = \sum_t \sum_{\mathbf{s}_t} p_\theta(\mathbf{s}_t) c_t(\mathbf{s}_t) &\leq \sum_t \sum_{\mathbf{s}_t} p_{\text{train}}(\mathbf{s}_t) c_t(\mathbf{s}_t) + |p_\theta(\mathbf{s}_t) - p_{\text{train}}(\mathbf{s}_t)| c_{\max} \\ &\leq \sum_t \epsilon + 2\epsilon t \leq \epsilon T + 2\epsilon T^2 \\ &O(\epsilon T^2)\end{aligned}$$

problem of Imitation learning

Non-Markovian behavior

人类在标注数据进行决策的时候，考虑到的并不仅仅是当前这一帧的observation，而是根据过去的许多信息来做出判断的，对于同样一帧图像，可能由于前面的信息不同，人类会做出不一样的决策。对于这样的数据，算法在学习的时候是有可能产生混淆的，这也就导致算法对于数据的拟合效果不够好。

所以解决问题的关键就是如何将缺失的信息补充回来。最直观的方法是不仅仅使用当前帧作为输入，而是使用过去若干帧作为输入并从中提取有用的信息，这也正是原始DQN中使用的方法。RNN是一个不得不提到的方法，它将前面的信息作为hidden state传输给后面的状态，从而作为缺失信息的补充。

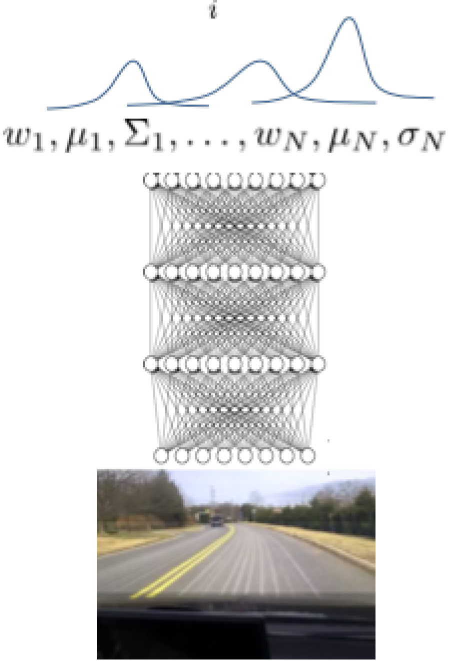
但是**添加的信息越多则更可能导致泛化性能变差**。但是学习中由于信息过多，加上现有的学习方法大部分对于因果关系的提取仍然比较差，所以会将这种共现因子当作是决策的原因，从而产生了所谓的causal confusion

Multimodal behavior

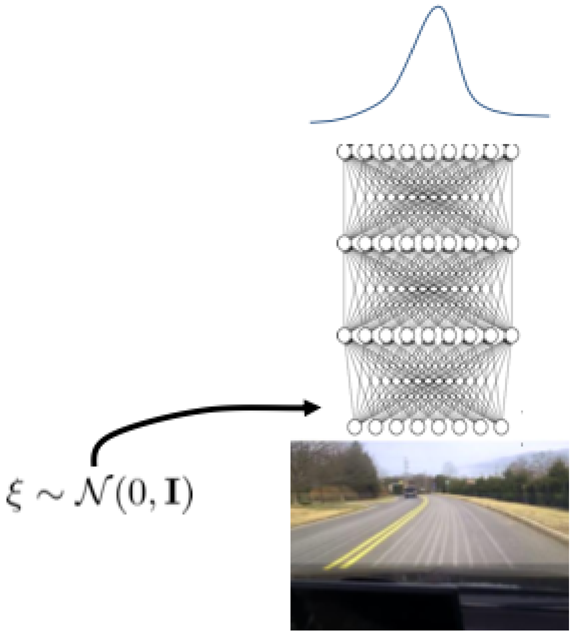
在很多时候对于一个情景的确是存在多个可行解

Mixture of Gaussian

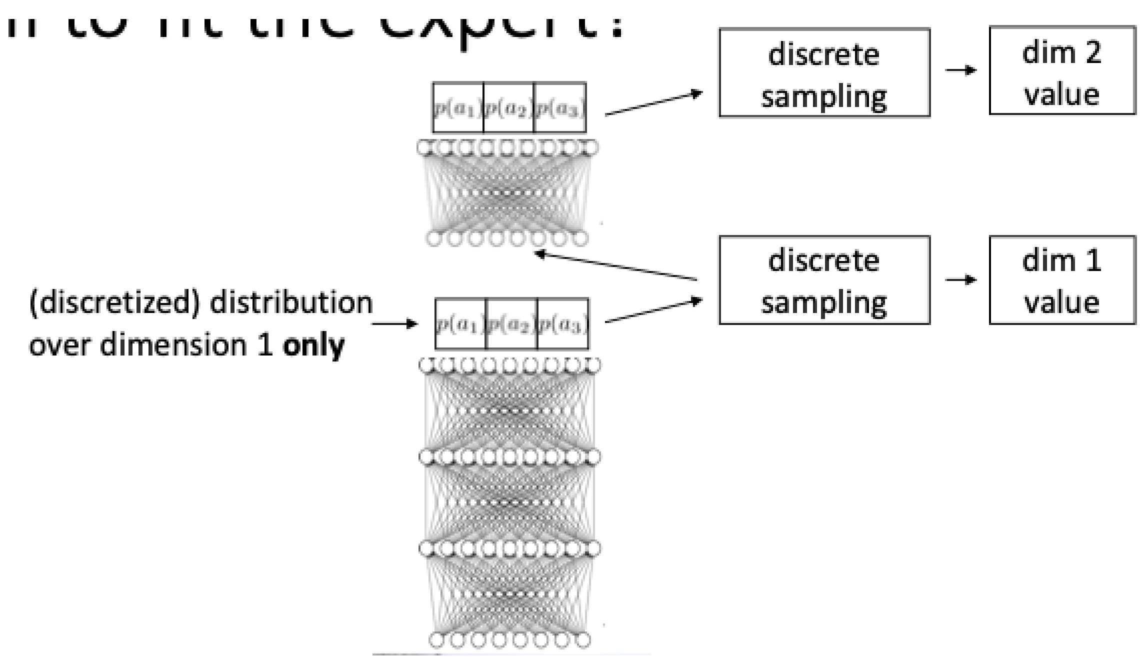
$$\pi(\mathbf{a}|\mathbf{o}) = \sum_i w_i \mathcal{N}(\mu_i, \Sigma_i)$$



Latent varable model



Autoregressive discretization



Extra notes

连续动作vs离散动作空间

特征	连续动作空间 (`Box`)	离散动作空间 (`Discrete`)
动作类型	实数向量	整数编号
Gym 类型	`gym.spaces.Box`	`gym.spaces.Discrete`

特征	连续动作空间 (`Box`)	离散动作空间 (`Discrete`)
策略分布类型	`Normal(mean, std)`	`Categorical(probs)`
适用算法举例	DDPG, PPO (continuous), SAC	DQN, REINFORCE, PPO (discrete)
处理方式（log_prob）	`dist.log_prob(actions).sum(-1)`	`dist.log_prob(action)`

homework

analysis

第一问

对于其中任意某个状态来说，有

$$\mathbb{E}_{p_{\pi^*}(s)}\pi_\theta\left(a \neq \pi^*(s) \mid s\right) = \frac{1}{T}\sum_{t=1}^{T}\mathbb{E}_{p_{\pi^*}(s_t)}\pi_\theta\left(a_t \neq \pi^*\left(s_t\right) \mid s_t\right) \leq \varepsilon$$

这个式子表明在状态s_t下policyπ_θ dispute policy π^*的概率小于ε

用题目提示的公式有

$$\operatorname{Pr}\left(\bigcup_{t=1}^{T} E_t\right) \leq \sum_{t=1}^{T}\operatorname{Pr}\left(E_t\right) \leq T\epsilon$$

其中E_t代表the event that π_θ dispute with π^* at time t

故得(a conservative bound)

$$\sum_{s_t}\left|p_{\pi_\theta}\left(s_t\right) - p_{\pi^*}\left(s_t\right)\right| \leq 2T\varepsilon$$

第二问

2.a

因为reward only depend on the last state,故有

$$J\left(\pi^*\right) - J\left(\pi_\theta\right) \leq R_{\max}T\epsilon = O(T\epsilon)$$

2.b

根据题目所给公式$J(\pi) = \sum_{t=1}^{T}\mathbb{E}_{p_\pi(s_t)}r\left(s_t\right)$,

代入可得

$$J\left(\pi^*\right) - J\left(\pi_\theta\right) \leq R_{\max}\sum_{t=1}^{T}t\epsilon = R_{\max}\frac{T(T+1)}{2}\epsilon = O\left(T^2\epsilon\right)$$

Editing code

- 在安装环境时，如果直接`pip install -r requirements.txt`,会出现报错无法安装box2d-py依赖项，可以先通过`conda install conda-forge::box2d-py`安装box2d-py,再`pip install -r requirements.txt`
- ```
  INTEL oneMKL ERROR: 找不到指定的模块。 mkl_intel_thread.2.dll.
  Intel oneMKL FATAL ERROR: Cannot load mkl_intel_thread.2.dll.
  ```

  后续报错找不到numpy，查找到应该mkl，mkl-service没有正确安装

  `conda install mkl`

  `pip install mkl-service`

  至此环境搭配完毕

MLP_policy.py

forward

- 给定 observation（状态），**输出一个分布**，表示该状态下采取各个动作的概率。

```python
def forward(self, observation: torch.FloatTensor) -> Any:
        """
        Defines the forward pass of the network

        :param observation: observation(s) to query the policy
        :return:
            action: sampled action(s) from the policy
        """
        # TODO: implement the forward pass of the network.
        # You can return anything you want, but you should be able to differentiate
        # through it. For example, you can return a torch.FloatTensor. You can also
        # return more flexible objects, such as a
        # `torch.distributions.Distribution` object. It's up to you!
        observation=observation.to(ptu.device)
        mean = self.mean_net(observation)#调用mean_net进行前向传播
        logstd = self.logstd.expand_as(mean)  # logstd is a single value
        std = torch.exp(logstd)  # convert logstd to std
        dist = distributions.Normal(mean, std)  # create a normal distribution
        action = dist.rsample()  # sample an action from the distribution,使用rsample可以使得梯度可以通过采样传递
        action = torch.tanh(action)  # apply tanh to the action限制
        return action, dist
```

- observation是一个 **PyTorch 张量**（`torch.FloatTensor`），形状是（batch_size,ob_dim)
- mean_net是一个神经网络，输出是动作分布的均值
- [torch.Tensor.expand_as — PyTorch 2.7 documentation]()：将logstd拓展成mean一样的大小
- [Probability distributions - torch.distributions — PyTorch 2.7 documentation]()：.Normal创建一个多维独立正态分布

$$\pi(a \mid s) = \mathcal{N}\left(\mu(s), \sigma(s)^2\right)$$
```

- 使用 `rsample()` 而不是 `sample()` 是为了支持 **重参数化（reparameterization trick）**，即允许梯度从 loss 反传到 `mean` 和 `logstd`，常用于策略梯度或 VAE。
- 将动作经过 `tanh` 激活函数映射到 `(-1, 1)` 范围，使得输出动作符合大多数连续控制环境（如 MuJoCo）要求的动作空间限制。

update

- 给定一批 `(observation, action)` 样本（专家数据），
- 调用 `forward` 得到策略在这些状态下的动作分布；
- 计算这些 **分布对 expert 动作的 log-probability**；
- 最大化该 log-prob（或最小化负对数似然）作为 loss，反向传播并优化网络。

```python
def update(self, observations, actions):
        """
        Updates/trains the policy

        :param observations: observation(s) to query the policy
        :param actions: actions we want the policy to imitate
        :return:
            dict: 'Training Loss': supervised learning loss
        """
        # TODO: update the policy and return the loss
        observations = observations.to(ptu.device)
        actions = actions.to(ptu.device)
        self.optimizer.zero_grad()
        _, dist = self.forward(observations)
        # Calculate the loss as the negative log likelihood of the actions under the policy
        log_probs = dist.log_prob(actions)
        log_probs = log_probs.sum(dim=-1)  # sum over action dimensions
        loss = -log_probs.mean()  # mean over batch
        # Backpropagate the loss
        loss.backward()
        self.optimizer.step()
        # Return a dictionary with the training loss
        # You can add extra logging information here, but keep this line
        return {
            # You can add extra logging information here, but keep this line
            'Training Loss': ptu.to_numpy(loss),
        }
```

- `self.optimizer.zero_grad()` 梯度清零，为当前batch的梯度计算做准备
- 计算专家动作 `actions` 在当前策略分布下的对数概率（log likelihood）,如果动作是多维的，求和得到整体动作向量的对数概率 `.sum(dim=-1)`
- `loss = -log_probs.mean()` 计算负对数似然损失（negative log likelihood，NLL）

  损失越小，说明策略分布越能“解释”专家动作，即策略输出的动作更接近专家动作
- `loss.backward()` 反向计算梯度
- `self.optimizer.step()` 优化器根据梯度更新模型参数，提升策略

utils.py

sample_trajectory

- 用当前策略采取动作，并记录每个时间步的observation、action、reward、next_observation、terminal

```python
def sample_trajectory(env, policy, max_path_length, render=False):
    """Sample a rollout in the environment from a policy."""

    # initialize env for the beginning of a new rollout
    ob =  env.reset() # TODO: initial observation after resetting the env
    print("observation shape: ", ob.shape)
    # init vars
    obs, acs, rewards, next_obs, terminals, image_obs = [], [], [], [], [], []
    steps = 0
    while True:

        # render image of the simulated env
        if render:
            if hasattr(env, 'sim'):
                img = env.sim.render(camera_name='track', height=500, width=500)[::-1]
            else:
                img = env.render(mode='single_rgb_array')
            image_obs.append(cv2.resize(img, dsize=(250, 250),
interpolation=cv2.INTER_CUBIC))

        # TODO use the most recent ob to decide what to do
        ac,_ = policy.forward(ptu.from_numpy(ob)) # HINT: this is a numpy array
        ac = ac[0]#取出单个动作,去除batch维度

        # TODO: take that action and get reward and next ob
        next_ob, rew, done, _ = env.step(ac) # HINT: this is a numpy array

        # TODO rollout can end due to done, or due to max_path_length
        steps += 1
        rollout_done = done or  steps>=max_path_length# HINT: this is either 0 or 1

        # record result of taking that action
        obs.append(ob)
        acs.append(ac)
        rewards.append(rew)
        next_obs.append(next_ob)
        terminals.append(rollout_done)

        ob = next_ob # jump to next timestep

        # end the rollout if the rollout ended
        if rollout_done:
            break

    return {"observation" : np.array(obs, dtype=np.float32),
            "image_obs" : np.array(image_obs, dtype=np.uint8),
            "reward" : np.array(rewards, dtype=np.float32),
```

```
            "action" : np.array(acs, dtype=np.float32),
            "next_observation": np.array(next_obs, dtype=np.float32),
            "terminal": np.array(terminals, dtype=np.float32)}
```

- `if hasattr(env, 'sim')`检查环境是否有sim属性，有的话通过sim.render渲染，[::-1]将图像上下翻转，因为MuJoCo 渲染默认是上下颠倒的，cv2.INTER_CUBIC（Bicubic Interpolation）双三次插值
- `env.step(ac)`让环境执行一个动作，并返回它对这个动作的"回应"

sample_trajectories和sample_n_trajectories

```
def sample_trajectories(env, policy, min_timesteps_per_batch, max_path_length, render=False):
    """Collect rollouts until we have collected min_timesteps_per_batch steps."""

    timesteps_this_batch = 0
    paths = []
    while timesteps_this_batch < min_timesteps_per_batch:

        #collect rollout
        path = sample_trajectory(env, policy, max_path_length, render)
        paths.append(path)

        #count steps
        timesteps_this_batch += get_pathlength(path)

    return paths, timesteps_this_batch
```

- **多次调用** `sample_trajectory`，直到采集的时间步（step）数 ≥ `min_timesteps_per_batch`,max_path_length是单条路径的最大步数，`paths` 是所有采集到的 trajectories 列表

同理有

```
def sample_n_trajectories(env, policy, ntraj, max_path_length, render=False):
    """Collect ntraj rollouts."""

    paths = []
    for i in range(ntraj):
        # collect rollout
        path = sample_trajectory(env, policy, max_path_length, render)
        paths.append(path)
    return paths
```

convert_listofrollouts

将这些多个轨迹中的对应数据拼接（concatenate）起来，变成一个大数组，方便后续处理

```
def convert_listofrollouts(paths, concat_rew=True):
    """
        Take a list of rollout dictionaries
        and return separate arrays,
```

```python
        where each array is a concatenation of that array from across the rollouts
    """
    observations = np.concatenate([path["observation"] for path in paths])
    actions = np.concatenate([path["action"] for path in paths])
    if concat_rew:
        rewards = np.concatenate([path["reward"] for path in paths])
    else:
        rewards = [path["reward"] for path in paths]
    next_observations = np.concatenate([path["next_observation"] for path in paths])
    terminals = np.concatenate([path["terminal"] for path in paths])
    return observations, actions, rewards, next_observations, terminals
```

- `concat_rew=True` — 返回所有轨迹奖励的拼接数组（扁平的，合并的） `concat_rew=False` — 返回奖励的列表（每个元素是单个轨迹的奖励序列）

compute_metrics

根据**训练轨迹和评估轨迹**（即 paths 和 eval_paths）计算训练日志中常见的一些评估指标

```python
def compute_metrics(paths, eval_paths):
    """Compute metrics for logging."""

    # returns, for logging
    train_returns = [path["reward"].sum() for path in paths]
    eval_returns = [eval_path["reward"].sum() for eval_path in eval_paths]

    # episode lengths, for logging
    train_ep_lens = [len(path["reward"]) for path in paths]
    eval_ep_lens = [len(eval_path["reward"]) for eval_path in eval_paths]

    # decide what to log
    logs = OrderedDict()
    logs["Eval_AverageReturn"] = np.mean(eval_returns)
    logs["Eval_StdReturn"] = np.std(eval_returns)
    logs["Eval_MaxReturn"] = np.max(eval_returns)
    logs["Eval_MinReturn"] = np.min(eval_returns)
    logs["Eval_AverageEpLen"] = np.mean(eval_ep_lens)

    logs["Train_AverageReturn"] = np.mean(train_returns)
    logs["Train_StdReturn"] = np.std(train_returns)
    logs["Train_MaxReturn"] = np.max(train_returns)
    logs["Train_MinReturn"] = np.min(train_returns)
    logs["Train_AverageEpLen"] = np.mean(train_ep_lens)

    return logs
```

- 使用 `OrderedDict` 是为了让输出的指标顺序稳定（可选，不影响结果，但便于日志文件查看）

run_hw1

行为克隆（Behavior Cloning）与 DAgger 的主训练脚本，作用是训练一个模仿学习的策略网络(**核心**)

tensorboard --logdir=你的日志目录路径，查看具体日志

```python
# how many rollouts to save as videos to tensorboard
MAX_NVIDEO = 2#每次迭代最多保存两个视频
MAX_VIDEO_LEN = 40  # we overwrite this in the code below

MJ_ENV_NAMES = ["Ant-v4", "Walker2d-v4", "HalfCheetah-v4", "Hopper-v4"]


def run_training_loop(params):
    """
    Runs training with the specified parameters
    (behavior cloning or dagger)

    Args:
        params: experiment parameters
    """

    #############
    ## INIT
    #############

    # Get params, create logger, create TF session
    logger = Logger(params['logdir'])

    # Set random seeds
    seed = params['seed']
    np.random.seed(seed)
    torch.manual_seed(seed)
    ptu.init_gpu(
        use_gpu=not params['no_gpu'],
        gpu_id=params['which_gpu']
    )

    # Set logger attributes
    log_video = True
    log_metrics = True

    #############
    ## ENV
    #############

    # Make the gym environment
    env = gym.make(params['env_name'], render_mode=None)
    env.reset(seed=seed)

    # Maximum length for episodes
    params['ep_len'] = params['ep_len'] or env.spec.max_episode_steps
    MAX_VIDEO_LEN = params['ep_len']
```

```python

    assert isinstance(env.action_space, gym.spaces.Box), "Environment must be continuous"#连续动作空间
    # Observation and action sizes
    ob_dim = env.observation_space.shape[0]
    ac_dim = env.action_space.shape[0]

    # simulation timestep, will be used for video saving
    if 'model' in dir(env):
        fps = 1/env.model.opt.timestep
    else:
        fps = env.env.metadata['render_fps']

    #############
    ## AGENT
    #############

    # TODO: Implement missing functions in this class.
    actor = MLPPolicySL(
        ac_dim,
        ob_dim,
        params['n_layers'],
        params['size'],
        learning_rate=params['learning_rate'],
    )

    # replay buffer
    replay_buffer = ReplayBuffer(params['max_replay_buffer_size'])

    #######################
    ## LOAD EXPERT POLICY
    #######################

    print('Loading expert policy from...', params['expert_policy_file'])
    expert_policy = LoadedGaussianPolicy(params['expert_policy_file'])
    expert_policy.to(ptu.device)
    print('Done restoring expert policy...')

    #######################
    ## TRAINING LOOP
    #######################

    # init vars at beginning of training
    total_envsteps = 0
    start_time = time.time()

    for itr in range(params['n_iter']):
        print("\n\n********** Iteration %i ************"%itr)

        # decide if videos should be rendered/logged at this iteration
        log_video = ((itr % params['video_log_freq'] == 0) and (params['video_log_freq'] != -1))
        # decide if metrics should be logged
```

```python
        log_metrics = (itr % params['scalar_log_freq'] == 0)

        print("\nCollecting data to be used for training...")
        if itr == 0:
            # BC training from expert data.
            paths = pickle.load(open(params['expert_data'], 'rb'))
            envsteps_this_batch = 0
        else:
            # DAGGER training from sampled data relabeled by expert
            assert params['do_dagger']
            # TODO: collect `params['batch_size']` transitions
            # HINT: use utils.sample_trajectories
            # TODO: implement missing parts of utils.sample_trajectory
            paths, envsteps_this_batch = utils.sample_trajectories(
                env, actor, params['batch_size'], params['ep_len'],render=log_video)

            # relabel the collected obs with actions from a provided expert policy
            if params['do_dagger']:
                print("\nRelabelling collected observations with labels from an expert policy...")

                # TODO: relabel collected obsevations (from our policy) with labels from expert policy
                # HINT: query the policy (using the get_action function) with paths[i]["observation"]
                # and replace paths[i]["action"] with these expert labels
                for i in range(len(paths)):
                    paths[i]["action"] = expert_policy.get_action(
                        paths[i]["observation"])
                    #利用专家数据替代

        total_envsteps += envsteps_this_batch
        # add collected data to replay buffer
        replay_buffer.add_rollouts(paths)

        # train agent (using sampled data from replay buffer)
        print('\nTraining agent using sampled data from replay buffer...')
        training_logs = []
        for _ in range(params['num_agent_train_steps_per_iter']):

          # TODO: sample some data from replay_buffer
          # HINT1: how much data = params['train_batch_size']
          # HINT2: use np.random.permutation to sample random indices
          # HINT3: return corresponding data points from each array (i.e., not different indices from each array)
          # for imitation learning, we only need observations and actions.
          ob_batch, ac_batch = replay_buffer.sample(
              params['train_batch_size'])

          # use the sampled data to train an agent
          train_log = actor.update(ob_batch, ac_batch)
          training_logs.append(train_log)

```

```
        # log/save
        print('\nBeginning logging procedure...')
        if log_video:
            # save eval rollouts as videos in tensorboard event file
            print('\nCollecting video rollouts eval')
            eval_video_paths = utils.sample_n_trajectories(
                env, actor, MAX_NVIDEO, MAX_VIDEO_LEN, True)

            # save videos
            if eval_video_paths is not None:
                logger.log_paths_as_videos(
                    eval_video_paths, itr,
                    fps=fps,
                    max_videos_to_save=MAX_NVIDEO,
                    video_title='eval_rollouts')

        if log_metrics:
            # save eval metrics
            print("\nCollecting data for eval...")
            eval_paths, eval_envsteps_this_batch = utils.sample_trajectories(
                env, actor, params['eval_batch_size'], params['ep_len'])

            logs = utils.compute_metrics(paths, eval_paths)
            # compute additional metrics
            logs.update(training_logs[-1]) # Only use the last log for now
            logs["Train_EnvstepsSoFar"] = total_envsteps
            logs["TimeSinceStart"] = time.time() - start_time
            if itr == 0:
                logs["Initial_DataCollection_AverageReturn"] = logs["Train_AverageReturn"]

            # perform the logging
            for key, value in logs.items():
                print('{} : {}'.format(key, value))
                logger.log_scalar(value, key, itr)
            print('Done logging...\n\n')

            logger.flush()

        if params['save_params']:
            print('\nSaving agent params')
            actor.save('{}/policy_itr_{}.pt'.format(params['logdir'], itr))


def main():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--expert_policy_file', '-epf', type=str, required=True)  # relative to where you're running this script from
    parser.add_argument('--expert_data', '-ed', type=str, required=True) #relative to where you're running this script from
    parser.add_argument('--env_name', '-env', type=str, help=f'choices: {", ".join(MJ_ENV_NAMES)}', required=True)
```

```python
    parser.add_argument('--exp_name', '-exp', type=str, default='pick an experiment name', required=True)
    parser.add_argument('--do_dagger', action='store_true')
    parser.add_argument('--ep_len', type=int)

    parser.add_argument('--num_agent_train_steps_per_iter', type=int, default=1000)  # number of gradient steps for training policy (per iter in n_iter)
    parser.add_argument('--n_iter', '-n', type=int, default=1)

    parser.add_argument('--batch_size', type=int, default=1000)  # training data collected (in the env) during each iteration
    parser.add_argument('--eval_batch_size', type=int,
                        default=1000)  # eval data collected (in the env) for logging metrics
    parser.add_argument('--train_batch_size', type=int,
                        default=100)  # number of sampled data points to be used per gradient/train step

    parser.add_argument('--n_layers', type=int, default=2)  # depth, of policy to be learned
    parser.add_argument('--size', type=int, default=64)  # width of each layer, of policy to be learned
    parser.add_argument('--learning_rate', '-lr', type=float, default=5e-3)  # LR for supervised learning

    parser.add_argument('--video_log_freq', type=int, default=5)
    parser.add_argument('--scalar_log_freq', type=int, default=1)
    parser.add_argument('--no_gpu', '-ngpu', action='store_true')
    parser.add_argument('--which_gpu', type=int, default=0)
    parser.add_argument('--max_replay_buffer_size', type=int, default=1000000)
    parser.add_argument('--save_params', action='store_true')
    parser.add_argument('--seed', type=int, default=1)
    args = parser.parse_args()

    # convert args to dictionary
    params = vars(args)

    ##################################
    ### CREATE DIRECTORY FOR LOGGING
    ##################################

    if args.do_dagger:
        # Use this prefix when submitting. The auto-grader uses this prefix.
        logdir_prefix = 'q2_'
        assert args.n_iter>1, ('DAGGER needs more than 1 iteration (n_iter>1) of training, to iteratively query the expert and train (after 1st warmstarting from behavior cloning).')
    else:
        # Use this prefix when submitting. The auto-grader uses this prefix.
        logdir_prefix = 'q1_'
        assert args.n_iter==1, ('Vanilla behavior cloning collects expert data just once (n_iter=1)')

```

```python
    # directory for logging
    data_path = os.path.join(os.path.dirname(os.path.realpath(__file__)), '../../data')
    if not (os.path.exists(data_path)):
        os.makedirs(data_path)
    logdir = logdir_prefix + args.exp_name + '_' + args.env_name + '_' + time.strftime("%d-%m-%Y_%H-%M-%S")
    logdir = os.path.join(data_path, logdir)
    params['logdir'] = logdir
    if not(os.path.exists(logdir)):
        os.makedirs(logdir)

    ###################
    ### RUN TRAINING
    ###################

    run_training_loop(params)


if __name__ == "__main__":
    main()

```

- `logger = Logger(params['logdir'])` 通过Logger类设置日志目录

```python
class Logger:
    def __init__(self, log_dir, n_logged_samples=10, summary_writer=None):
        self._log_dir = log_dir
        print('########################')
        print('logging outputs to ', log_dir)
        print('########################')
        self._n_logged_samples = n_logged_samples
        self._summ_writer = SummaryWriter(log_dir, flush_secs=1, max_queue=1)

```

这个是 `tensorboardX.SummaryWriter` 的实例，用于将数据写入 TensorBoard 可读取的 `.event` 文件中。

参数	含义
`log_dir`	保存路径
`flush_secs=1`	每1秒就刷新一次日志（即时可见）
`max_queue=1`	不使用队列缓存，数据写入尽可能立即进行，防止数据延迟出现

- ```python
      seed = params['seed']
      np.random.seed(seed)
      torch.manual_seed(seed)
  ```

设置 **NumPy** 的随机种子，使得：
  ```

```
np.random.rand(), np.random.randint(), np.random.permutation()
```

等 NumPy 的随机函数每次运行时的结果**固定**，不会随着时间或运行顺序变化。

设置 **PyTorch** 的随机种子，使得：

```
pythonCopyEdittorch.rand(), torch.randn(), torch.randint()
torch.nn.init.xavier_uniform_()
```

这些涉及到模型参数初始化或训练过程中的随机操作时，每次运行都保持一致。

- 使用 OpenAI Gym 创建一个环境并设定随机种子

  `render_mode` 是 Gym API 从 v0.26+ 引入的新参数，用于指定渲染方式：

  - `'human'`: 实时窗口可视化
  - `'rgb_array'`: 以 NumPy 格式返回帧图像（适用于视频记录）
  - `None`: 不渲染（节省资源）

```
    env = gym.make(params['env_name'], render_mode=None)
    env.reset(seed=seed)
```

-
```
    assert isinstance(env.action_space, gym.spaces.Box), "Environment must be continuous"#连续动作空间
    # Observation and action sizes
    ob_dim = env.observation_space.shape[0]
    ac_dim = env.action_space.shape[0]
```

  断言（assert）用于确保当前环境的动作空间是 **`gym.spaces.Box` 类型**。

  `gym.spaces.Box` 表示的是 **连续动作空间**，比如动作是实数，如 `[1.3, -0.2]`。

  如果不是（比如是离散动作空间 `gym.spaces.Discrete`），就会报错。

-
```
    # simulation timestep, will be used for video saving
    if 'model' in dir(env):
        fps = 1/env.model.opt.timestep
    else:
        fps = env.env.metadata['render_fps']
```

  获取环境的视频帧率（fps, frames per second），用于后续在 TensorBoard 中保存视频。

  判断 `env`（环境）对象是否包含 `model` 这个属性。

  通常 `model` 是 **Mujoco 环境特有的属性**。

  如果环境是基于 MuJoCo（如 `HalfCheetah-v4`, `Ant-v4`），那么它们内部有一个低层次的 `model`，包含仿真的物理属性。

  对于非 Mujoco 环境（如 Atari、CartPole 等），一般在 `env.metadata` 字典里直接定义了渲染帧率。

  `render_fps` 就是推荐的可视化帧率

- `replay_buffer = ReplayBuffer(params['max_replay_buffer_size'])`

  `ReplayBuffer` 类是一个用来**缓存 agent 与环境交互的样本**的模块，结构一般是一个大表格

用于**打破样本之间的时间相关性**，提升训练稳定性（如 DQN、SAC）。

```
    def add_rollouts(self, paths, concat_rew=True):

        # add new rollouts into our list of rollouts
        for path in paths:
            self.paths.append(path)

        # convert new rollouts into their component arrays, and append them onto
        # our arrays
        observations, actions, rewards, next_observations, terminals = (
            convert_listofrollouts(paths, concat_rew))

        if self.obs is None:
            self.obs = observations[-self.max_size:]
            self.acs = actions[-self.max_size:]
            self.rews = rewards[-self.max_size:]
            self.next_obs = next_observations[-self.max_size:]
            self.terminals = terminals[-self.max_size:]
        else:
            self.obs = np.concatenate([self.obs, observations])[-self.max_size:]
            self.acs = np.concatenate([self.acs, actions])[-self.max_size:]
            if concat_rew:
                self.rews = np.concatenate(
                    [self.rews, rewards]
                )[-self.max_size:]
            else:
                if isinstance(rewards, list):
                    self.rews += rewards
                else:
                    self.rews.append(rewards)
                self.rews = self.rews[-self.max_size:]
            self.next_obs = np.concatenate(
                [self.next_obs, next_observations]
            )[-self.max_size:]
            self.terminals = np.concatenate(
                [self.terminals, terminals]
            )[-self.max_size:]
```

`concat_rew=True`：是否将 reward 拼接成一维数组（方便 supervised learning）

行为	说明
`self.rews += rewards`	批量扩展
`self.rews.append(rewards)`	单个追加
`self.rews = self.rews[-self.max_size:]`	剪裁到最大容量

调用 `convert_listofrollouts()` 将多个 path 中的 `obs`、`act` 等字段分别拼接成统一的大数组

```
def convert_listofrollouts(paths, concat_rew=True):
```

```
    """
        Take a list of rollout dictionaries
        and return separate arrays,
        where each array is a concatenation of that array from across the rollouts
    """
    observations = np.concatenate([path["observation"] for path in paths])
    actions = np.concatenate([path["action"] for path in paths])
    if concat_rew:
        rewards = np.concatenate([path["reward"] for path in paths])
    else:
        rewards = [path["reward"] for path in paths]
    next_observations = np.concatenate([path["next_observation"] for path in paths])
    terminals = np.concatenate([path["terminal"] for path in paths])
    return observations, actions, rewards, next_observations, terminals
```

参数 `concat_rew`	输出结构	使用场景说明
`True`	`np.array` (N,)	模型训练（统一结构）
`False`	`List[List]`	日志、评估（保持轨迹结构）

- `LoadedGaussianPolicy` 是一个类，表示 **一个从文件加载的高斯策略模型**，通常是预训练好的专家模型。
-
```
    # decide if videos should be rendered/logged at this iteration
    log_video = ((itr % params['video_log_freq'] == 0) and (params['video_log_freq'] != -1))
    # decide if metrics should be logged
    log_metrics = (itr % params['scalar_log_freq'] == 0)
```

  设置video_log_freq和scalar_log_freq判断是否记录日志

-
```
    if itr == 0:
        # BC training from expert data.
        paths = pickle.load(open(params['expert_data'], 'rb'))
        envsteps_this_batch = 0
    else:
        # DAGGER training from sampled data relabeled by expert
        assert params['do_dagger']
        # TODO: collect `params['batch_size']` transitions
        # HINT: use utils.sample_trajectories
        # TODO: implement missing parts of utils.sample_trajectory
        paths, envsteps_this_batch = utils.sample_trajectories(
            env, actor, params['batch_size'], params['ep_len'],render=log_video)

        # relabel the collected obs with actions from a provided expert policy
        if params['do_dagger']:
            print("\nRelabelling collected observations with labels from an expert policy...")

            # TODO: relabel collected obsevations (from our policy) with labels from expert policy
```

```
            # HINT: query the policy (using the get_action function) with paths[i]["observation"]
            # and replace paths[i]["action"] with these expert labels
            for i in range(len(paths)):
                paths[i]["action"] = expert_policy.get_action(
                    paths[i]["observation"])
                #利用专家数据替代
```

DAGGER的**核心**：利用当前策略收集状态，但用**专家指导动作**，保证训练数据的质量。

调用sample_trajectories去重新采样包括obs和action，然后利用expert_policy.get_action去重新得到专家指导的action

- ```
      for _ in range(params['num_agent_train_steps_per_iter']):

        # TODO: sample some data from replay_buffer
        # HINT1: how much data = params['train_batch_size']
        # HINT2: use np.random.permutation to sample random indices
        # HINT3: return corresponding data points from each array (i.e., not different indices from each array)
        # for imitation learning, we only need observations and actions.
        ob_batch, ac_batch = replay_buffer.sample(
            params['train_batch_size'])

        # use the sampled data to train an agent
        train_log = actor.update(ob_batch, ac_batch)
        training_logs.append(train_log)
  ```

调用sample函数随机选取数据进行训练，`train_log` 是一个字典，可能包含update返回的 `loss`, `accuracy` 等训练信息，用于后续记录

```
    def sample(self, batch_size):
        """Sample given batch size of observations and actions. """
        indices = np.random.randint(0, len(self.acs), size=(batch_size,))
        return self.obs[indices], self.acs[indices]
```

从 `[0, len(self.acs))` 范围内随机选择 `batch_size` 个整数索引。

- ```
      if log_video:
          # save eval rollouts as videos in tensorboard event file
          print('\nCollecting video rollouts eval')
          eval_video_paths = utils.sample_n_trajectories(
              env, actor, MAX_NVIDEO, MAX_VIDEO_LEN, True)

          # save videos
          if eval_video_paths is not None:
              logger.log_paths_as_videos(
                  eval_video_paths, itr,
                  fps=fps,
                  max_videos_to_save=MAX_NVIDEO,
                  video_title='eval_rollouts')

  ```

重新采样几条 trajectory，并将 `render=True` 打开，采集 `image_obs` 并写入日志文件，观察actor是否学会

```
    def log_paths_as_videos(self, paths, step, max_videos_to_save=2, fps=10, video_title='video'):#从路径中记录视频

        # reshape the rollouts
        videos = [np.transpose(p['image_obs'], [0, 3, 1, 2]) for p in paths]

        # max rollout length
        max_videos_to_save = np.min([max_videos_to_save, len(videos)])
        max_length = videos[0].shape[0]
        for i in range(max_videos_to_save):
            if videos[i].shape[0]>max_length:
                max_length = videos[i].shape[0]#选取最长的视频

        # pad rollouts to all be same length
        for i in range(max_videos_to_save):
            if videos[i].shape[0]<max_length:
                padding = np.tile([videos[i][-1]], (max_length-videos[i].shape[0],1,1,1))
                videos[i] = np.concatenate([videos[i], padding], 0)

        # log videos to tensorboard event file
        videos = np.stack(videos[:max_videos_to_save], 0)
        self.log_video(videos, video_title, step, fps=fps)

```

`videos = [np.transpose(p['image_obs'], [0, 3, 1, 2]) for p in paths]`:

格式	说明	用于
`[T, H, W, C]`	原始格式，常见于 OpenCV/Numpy	环境返回的图像序列
`[T, C, H, W]`	PyTorch/TF 格式，适用于神经网络	**TensorBoardX 日志**、网络输入

`padding = np.tile([videos[i][-1]], (max_length-videos[i].shape[0],1,1,1))`:

- `videos[i][-1]`：当前视频最后一帧（shape: `[C, H, W]`）。

  `[videos[i][-1]]`：变成 `[1, C, H, W]`，便于 `tile` 复制。

  `np.tile(..., (N, 1, 1, 1))`：沿第0维（时间轴）复制最后一帧，使得视频长度从当前长度补齐到 `max_length`。

`videos[i] = np.concatenate([videos[i], padding], 0)`从第0个维度进行拼接，即通过时间帧数

`videos = np.stack(videos[:max_videos_to_save], 0)`:`np.stack(..., 0)` 的作用是把这些视频沿着一个**新的第0维度**堆叠起来，得到一个形状为 `[N, T, C, H, W]` 的五维数组

- ```
      if log_metrics:
          # save eval metrics
          print("\nCollecting data for eval...")
          eval_paths, eval_envsteps_this_batch = utils.sample_trajectories(
              env, actor, params['eval_batch_size'], params['ep_len'])

  ```
  ```

```python
        logs = utils.compute_metrics(paths, eval_paths)
        # compute additional metrics
        logs.update(training_logs[-1]) # Only use the last log for now
        logs["Train_EnvstepsSoFar"] = total_envsteps
        logs["TimeSinceStart"] = time.time() - start_time
        if itr == 0:
            logs["Initial_DataCollection_AverageReturn"] = logs["Train_AverageReturn"]

        # perform the logging
        for key, value in logs.items():
            print('{} : {}'.format(key, value))
            logger.log_scalar(value, key, itr)
        print('Done logging...\n\n')

        logger.flush()

    if params['save_params']:
        print('\nSaving agent params')
        actor.save('{}/policy_itr_{}.pt'.format(params['logdir'], itr))

```

`logs.update(training_logs[-1]) # Only use the last log for now`只取最后一次训练日志是为了**简洁、准确地反映当前训练状态，方便后续分析和展示**

`logs["Initial_DataCollection_AverageReturn"] = logs["Train_AverageReturn"]`记录训练开始时（第0次迭代）从专家数据收集到的初始平均回报

```python
    def log_scalar(self, scalar, name, step_):
        self._summ_writer.add_scalar('{}'.format(name), scalar, step_
```

利用写好的log_scalar把数据写入tensorboard

`logger.flush()` 确保所有日志数据都写入硬盘

- ```python
  import argparse
  parser = argparse.ArgumentParser()
  parser.add_argument('--env_name', '-env', type=str, help=f'choices: {", ".join(MJ_ENV_NAMES)}', required=True)
  args = parser.parse_args()

  # convert args to dictionary
  params = vars(args)
  ```

导入 Python 标准库中的 `argparse` 模块。它用于从命令行解析参数，让你可以很方便地定义程序运行时的参数选项

`help` 是帮助信息，告诉用户可选项是 `MJ_ENV_NAMES` 列表里的字符串，用逗号连接显示

`required=True` 表示这个参数是必须要传的，否则程序会报错

`args = parser.parse_args()`
这行代码会从命令行解析参数，返回一个 `Namespace` 对象，里面的每个属性对应一个命令行参数及其值
  ```

`vars()` 是 Python 内置函数，用于将对象转换为字典。
它会把 `args` 中的属性和值，转换成一个标准的 Python 字典

- ```
    data_path = os.path.join(os.path.dirname(os.path.realpath(__file__)),
'../../data')
    if not (os.path.exists(data_path)):
        os.makedirs(data_path)
    logdir = logdir_prefix + args.exp_name + '_' + args.env_name + '_' +
time.strftime("%d-%m-%Y_%H-%M-%S")
    logdir = os.path.join(data_path, logdir)
    params['logdir'] = logdir
    if not(os.path.exists(logdir)):
        os.makedirs(logdir)
  ```

  可以通过args.name调用，`time.strftime("%d-%m-%Y_%H-%M-%S")`：获取当前时间，并格式化为字符串，格式为“日-月-年_时-分-秒”

## switchdagger

在一个具有时间跨度 T 的离散马尔可夫决策过程（MDP）中，存在一个专家策略$\pi^*$。在算法的每个迭代步骤$n$=1,...,$N$中，都有一个当前策略$\pi^n$；当基于该策略生成轨迹时，会在某个随机时间步将控制权转交给专家，并由专家完成轨迹的剩余部分，在$\pi^n$移交给专家，我们称为$\tilde{\pi}^n$

$$\hat{\pi}^n \leftarrow \text{fit to expert actions } \pi^*(s) \text{ across } s \sim p_{\tilde{\pi}^{n-1}}$$
$$\tilde{\pi}^n \leftarrow S^{X_n}(\hat{\pi}^n, \tilde{\pi}^{n-1}) \text{ where } X_n + 1 \sim \text{Geom}(1-\alpha)$$

1. 在n-1的分布下拟合专家策略
2. n+1步及以后每一步是否移交专家控制符合几何分布$\text{Geom}(1-\alpha)$，参数是移交概率

### 第一问

设定：其中t表示一共t时间跨度（从$\tilde{\pi}^n$之后），n表示考虑第n步（初始）是否移交

$$A(0,n) = 0,$$
$$A(t,0) = 0,$$
$$A(t,n) = \alpha\varepsilon t + \alpha(1-\varepsilon)A(t-1,n) + (1-\alpha)A(t,n-1)$$

第三个式子：第一项不移交时犯错的cost（一步错步步错所以*t），第二项不移交但这一步不犯错故时间跨步缩小，第三项表示第n步已移交，所以回退一个n-1（需要从起始开始考虑）

### 第二问

归纳法：

1. 当n=0时成立
2. 假设n=k成立
3. $C(\tilde{\pi}^{n+1}) \leq \varepsilon\alpha T + (1-\alpha)C(\tilde{\pi}^n)$

   右式第一项表示不移交专家，后一项表示移交专家回退一步

   故得证
  ```

第三问

π^n与$\tilde{\pi}^n$的成本差异源于X^*的取值：

- 若X^*>T：由于 MDP 的时间跨度为*T*，无额外成本。
- 若X^*≤*T*：会产生额外的犯错期望,最大为1

$T \cdot Pr[X^* \leq T] \leq T \cdot e^{\frac{-n}{(1-\alpha)T}}$

故有$C(\pi^n)$≤$C(\tilde{\pi}^n) + Te^{\frac{-n}{(1-\alpha)T}}$

第四问

$C(\pi^n) \leq Tn\alpha\varepsilon + Te^{\frac{-n}{(1-\alpha)T}}$

取$\alpha = \frac{1}{\log(1/\varepsilon)}$，$N = O(T\log(1/\varepsilon))$

结合$\alpha \leq 1/T$即可