A. Prior work in RL Finetuning

A.1. DAPG [20]

Preliminaries. Rajeswaran *et al*. [20] proposed DAPG, a method which incorporates demonstrations in RL, and thus quite relevant to our methodology. DAPG first pretrains a policy using behavior cloning then finetunes the policy using an augmented RL objective (shown in Eq. (4)). DAPG proposes to use different parts of demonstrations dataset during different stages of learning for tasks involving sequence of behaviors. To do so, they add an additional term to the policy gradient objective:

$$g_{aug} = \sum_{(s,a)\in\tau\sim\pi_\theta} \nabla_\theta \log_{\pi_\theta}(a|s) A^\pi(s,a) + \sum_{(s,a)\in\tau\sim\mathcal{T}} \nabla_\theta \log_{\pi_\theta}(a|s) w(s,a) \quad (4)$$

Here $\tau \sim \pi_\theta$ is a trajectory obtained by executing the current policy, $\tau \sim \mathcal{T}$ denotes a trajectory obtained by replaying a demonstration, and $w(s,a)$ is a weighting function to alternate between imitation and reinforcement learning. DAPG uses a heuristic weighting scheme to set $w(s,a)$ to decay the auxiliary objective:

$$w(s,a) = \lambda_0 \lambda_1^k \max_{(s',a')\in\tau\sim\pi_\theta} A^{\pi_\theta}(s',a') \forall (s,a) \quad (5)$$

where λ_0 and λ_1 are hyperparameters and k is the update iteration counter. The decaying weighting term λ_1^k is used to avoid biasing the gradient towards the demonstrations data towards the end of training.

Implementation Details. [20] showed results of using DAPG on dexterous hand manipulation tasks for object relocation, in-hand manipulation, tool use, *etc*. To train the policy with behavior cloning, they use 25 demonstrations for each task gathered using the Mujoco HAPTIX system [53]. The small size of the demonstrations dataset and the observation input allows DAPG to load the demonstrations dataset in system memory which makes it feasible to compute the augmented RL objective shown above.

Challenges in adopting [20]'s setup. Compared to [20], our setup uses high-dimensional visual input (256×256 RGB observations) and $77k$ ObjectNav demonstrations for training. Following DAPG's training implementation, storing the visual inputs for $77k$ demonstrations in system memory would require 2TB, which is significantly higher than what is possible on today's systems. An alternative is to leverage on-the-fly demonstration replay during RL training. However, efficiently incorporating demonstration replay with experience collection online requires solving a systems research problem. Naively switching between online experience collection using the current policy and replay demonstrations would require 2x the current experience collection time, overall hurting the training throughput.

A.2. Feasibility of Off-Policy RL finetuning

There are several methods for incorporating demonstrations with off-policy RL [35–39]. Algorithm 1 shows the general framework of off-policy RL (finetuning) methods.

Algorithm 1 General framework of off-policy RL algorithm

Require: π_θ : Policy, B: replay buffer, N: Rounds, I: Policy Update Iterations
for $k = 1$ to N **do**
  Trajectory $\tau \leftarrow$ Rollout $\pi_\theta(\cdot|s)$ to collect trajectory $\{(s_1, a_1, r_1, h_1), ..., (s_T, a_T, r_T, h_T)\}$
  $B \leftarrow \{B\} \cup \{\tau\}$
  $\pi_\theta \leftarrow$ TrainPolicy(π_θ, B) for I iterations
end for

Unfortunately, most of these methods use feedforward state encoders, which is ill-posed for partially observable settings. In partially observable settings, the agent requires a state representation that combines information about the state-action trajectory so far with information about the current observation, which is typically achieved using a recurrent network.

To train a recurrent policy in an off-policy setting, the full state-action trajectories need to be stored in a replay buffer to use for training, including the hidden state h_t of the RNN. The policy update requires a sequence input for multiple time steps $\left[(s_t, a_t, r_t, h_t), ..., (s_{t+l}, a_{t+l}, r_{t+l}, h_{t+l})\right] \sim \tau$ where l is sampled sequence length. Additionally, it is not obvious how the hidden state should be initialized for RNN updates when using a sampled sequence in the off-policy setting. Prior work DRQN [54] compared two training strategies to train a recurrent network from replayed experience:

1. **Bootstrapped Random Updates**. The episodes are sampled randomly from the replay buffer and the policy updates begin at random steps in an episode and proceed only for the unrolled timesteps. The RNN initial state is initialized to zero at the start of the update. Using randomly sampled experience better adheres to DQN's [55] random sampling strategy, but, as a result, the RNN's hidden state must be initialized to zero at the start of each policy update. Using zero start state allows for independent decorrelated sampling of short sequences which is important for robust optimization of neural networks. Although this can help RNN to learn to recover predictions from an initial state that mismatches with the hidden state from the collected experience but it might limit the ability of the network to rely on it's recurrent state and exploit long term temporal correlations.
2. **Bootstrapped Sequential Updates**. The full episode replays are sampled randomly from the replay buffer and

the policy updates begin at the start of the episode. The RNN hidden state is carried forward throughout the episode. Eventhough this approach avoids the problem of finding the correct initial state it still has computational issues due to varying sequence length for each episode, and algorithmic issues due to high variance of network updates due to highly correlated nature of the states in the trajectory.

Even though using bootstrapped random updates with zero start states performed well in Atari which is mostly fully observable, R2D2 [40] found using this strategy prevents a RNN from learning long-term dependencies in more memory critical environments like DMLab. [40] proposed two strategies to train recurrent policies with randomly samples sequences:

1. **Stored State**. In this strategy, the hidden state is stored at each step in the replay and use it to initialize the network at the time of policy updates. Using stored state partially remedies the issues with initial recurrent state mismatch in zero start state strategy but it suffers from 'representational drfit' leading to 'recurrent state staleness', as the stored state generated by a sufficiently old network could differ significantly from a state from the current policy.
2. **Burn-in**. In this strategy the initial part of the replay sequence is used to unroll the network and produce a start state ('burn-in period') and update the network on the remaining part of the sequence.

While R2D2 [40] found a combination of these strategies to be effective at mitigating the representational drift and recurrent state staleness, this increases computation and requires careful tuning of the replay sequence length m and burn-in period l.

Both [40, 54] demonstrate the issues associated with using a recurrent policy in an off-policy setting and present approaches that mitigate issues to some extent. Applying these techniques for Embodied AI tasks and off-policy RL finetuning is an open research problem and requires empirical evaluation of these strategies.

B. Prior work in Imitation Learning

In Imitation Learning (IL), we use demonstrations of successful behavior to learn a policy that imitates the expert (demonstrator) providing these trajectories. The simplest approach to IL is behavior cloning (BC), which uses supervised learning to learn a policy to imitate the demonstrator. However, BC suffers from poor generalization to unseen states, since the training mimics the actions and not their consequences. DAgger [56] mitigates this issue by iteratively aggregating the dataset using the expert and trained policy $\pi_{\hat{i-1}}$ to learn the policy $\hat{\pi_i}$. Specifically, at each step i, the new dataset D_i is generated by:

$$\pi_i = \beta\pi_{exp} + (1-\beta)\hat{\pi}_{i-1} \tag{6}$$

where, π_{exp} is a queryable expert, and $\hat{\pi}_{i-1}$ is the trained policy at iteration $i-1$. Then, we aggregate the dataset $D \leftarrow D \cup D_i$ and train a new policy $\hat{\pi}_i$ on the dataset D. Using experience collected by the current policy to update the policy for next iteration enables DAgger [56] to mitigate the poor generalization to unseen states caused by BC. However, using DAgger [56] in our setting is not feasible as we don't have a queryable human expert for policies being trained with human demonstrations.

Alternative approaches [57–61] for imitation learning are variants of inverse reinforcement learning (IRL), which learn reward function from expert demonstrations in order to train a policy. IRL methods learn a parameterized $\mathcal{R}_\phi(\tau)$ reward function, which models the behavior of the expert and assigns a scalar reward to a demonstration. Given the reward r_t, a policy $\pi_\theta(a_t|s_t)$ is learned to map states s_t to distribution over actions a_t at each time step. The goal of IRL methods is to learn a reward function such that a policy trained to maximize the discounted sum of the learned reward matches the behavior of the demonstrator. Compared to prior works [57–61], our setup uses a partially-observable setting and high-dimensional visual input for training. Following training implementation from prior works, storing visual inputs of demonstrations for reward model training would require $2TB$ system memory, which is significantly higher than what is possible on today's systems. Alternatively, efficiently replaying demonstrations during RL training with reward model learning in the loop requires solving an open systems research problem. In addition, applying these methods for tasks in a partially observable setting is an open research problem and requires empirical evaluation of these approaches.

C. Training Details

C.1. Behavior Cloning

We use a distributed implementation of behavior cloning by [1] for our imitation pretraining. Each worker collects 64 frames of experience from 8 environments parallely by replaying actions from the demonstrations dataset. We then perform a policy update using supervised learning on 2 mini batches. For all of our BC experiments, we train the policy for $500M$ steps on 64 GPUs using Adam optimizer with a learning rate 1.0×10^{-3} which is linearly decayed after each policy update. Tab. 6 details the default hyperparameters used in all of our training runs.

C.2. Reinforcement Learning

To train our policy using RL we use PPO with Generalized Advantage Estimation (GAE) [62]. We use a discount factor γ of 0.99 and set GAE parameter τ to 0.95. We do not use normalized advantages. To parallelize training, we use DD-PPO with 16 workers on 16 GPUs. Each worker collects 64 frames of experience from 8 environments parallely and then

Parameter	Value
Number of GPUs	64
Number of environments per GPU	8
Rollout length	64
Number of mini-batches per epoch	2
Optimizer	Adam
Learning rate	1.0×10^{-3}
Weight decay	0.0
Epsilon	1.0×10^{-5}
DDPIL sync fraction	0.6

Table 6. Hyperparameters used for Imitation Learning.

Parameter	Value
Number of GPUs	16
Number of environments per GPU	8
Rollout length	64
PPO epochs	2
Number of mini-batches per epoch	2
Optimizer	Adam
Weight decay	0.0
Epsilon	1.0×10^{-5}
PPO clip	0.2
Generalized advantage estimation	True
γ	0.99
τ	0.95
Value loss coefficient	0.5
Max gradient norm	0.2
DDPPO sync fraction	0.6

Table 7. Hyperparameters used for RL finetuning.

performs 2 epochs of PPO update with 2 mini batches in each epoch. For all of our experiments, we RL finetune the policy for $300M$ steps. Tab. 7 details the default hyperparameters used in all of our training runs.

C.3. RL Finetuning using VPT

To compare with RL finetuning approach proposed in VPT [21] we implement the method in DD-PPO framework. Specifically, we initialize the critic weights to zero, replace the entropy term in PPO [42] with a KL-divergence loss between the frozen IL policy and RL policy, and decay the KL divergence loss coefficient, ρ, by a fixed factor after every iteration. This loss term is defined as:

$$L_{kl\_penalty} = \rho \text{KL}(\pi_\theta^{BC}, \pi_\theta) \tag{7}$$

where π_θ^{BC} is the frozen behavior cloned policy, π_θ is the current policy, and ρ is the loss weighting term. Following, VPT [21] we set ρ to 0.2 at the start of training and decay it by 0.995 after each policy update. We use learning rate of 1.5×10^{-5} without a learning rate decay for our VPT [21] finetuning experiments.

C.4. RL Finetuning Ablations

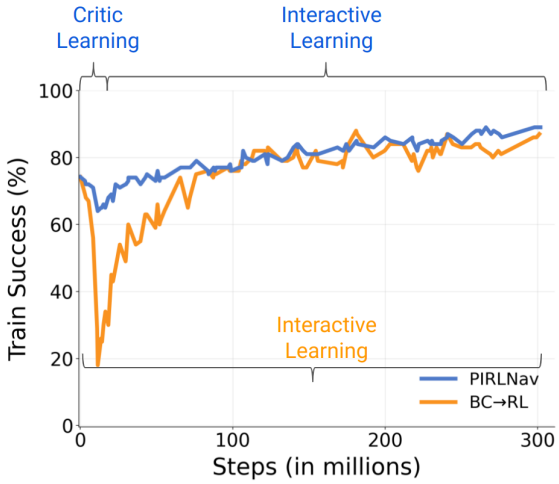

Figure 8. A policy pretrained on the ObjectNav task is used as initialization for actor weights and critic weights are initialized randomly for RL finetuning using DD-PPO. The policy performance immediately starts dropping early on during training and then recovers leading to slightly higher performance with further training.

Method	Success (↑)	SPL (↑)
1) BC	52.0	20.6
2) BC→RL-FT	53.6 ±1.01	**28.6** ±0.50
3) BC→RL-FT (+ Critic Learning)	56.7 ±0.93	27.7 ±0.82
4) BC→RL-FT (+ Critic Learning, Critic Decay)	59.4 ±0.42	26.9 ±0.38
5) BC→RL-FT (+ Critic Learning, Actor Warmup)	58.2 ±0.55	26.7 ±0.69
6) PIRLNav	**61.9** ±0.47	27.9 ±0.56

Table 8. RL-finetuning ablations on HM3D VAL [16, 32]

For ablations presented in Sec. 4.3 of the main paper (also shown in Tab. 8) we use a policy pretrained on $20k$ human demonstrations using BC and finetuned for $300M$ steps using hyperparameters from Tab. 7. We try 3 learning rates (1.5×10^{-4}, 2.5×10^{-4}, and 1.5×10^{-5}) for both BC → RL (row 2) and BC → RL (+ Critic Learning) (row 3) and we report the results with the one that works the best. For PIRLNav we use a starting learning rate of 2.5×10^{-4} and decay it to 1.5×10^{-5}, consistent with learning rate schedule of our best performing agent. For ablations we do not tune learning rate parameters of PIRLNav, we hypothesize tuning the parameters would help improve performance.

We find BC → RL (row 2) works best with a smaller learning rate but the training performance drops significantly early on, due to the critic providing poor value estimates, and recovers later as the critic improves. See Fig. 8. In contrast when using proposed two phase learning setup with the learning rate schedule we do not observe a significant drop in training performance.