ICLR #18

A. OBJECTNAV and Imitation Learning

A.1. OBJECTNAV

In OBJECTNAV an agent is tasked with searching for an instance of the specified object category (*e.g.*, 'bed') in an unseen environment. The agent must perform this task using only egocentric perceptions. Specifically, a RGB camera, Depth sensor[2], and a GPS+Compass sensor that provides location and orientation relative to the start position of the episode. The action space is discrete and consists of MOVE_FORWARD ($0.25m$), TURN_LEFT ($30°$), TURN_RIGHT ($30°$), LOOK_UP ($30°$), LOOK_DOWN ($30°$), and STOP actions. An episode is considered successful if the agent stops within $1m$ Euclidean distance of the goal object within 500 steps and is able to view the object by taking turn actions [14].

We use scenes from the HM3D-Semantics v0.1 dataset [16]. The dataset consists of 120 scenes and 6 unique goal object categories. We evaluate our agent using the train/val/test splits from the 2022 Habitat Challenge[3].

A.2. OBJECTNAV Demonstrations

Ramrakhya *et al*. [1] collected OBJECTNAV demonstrations for the Matterport3D dataset [15]. We begin our study by replicating this effort and collect demonstrations for the HM3D-Semantics v0.1 dataset [16]. We use Ramrakhya *et al*.'s Habitat-WebGL infrastructure to collect $77k$ demonstrations, amounting to $\sim$2378 human annotation hours.

A.3. Imitation Learning from Demonstrations

We use behavior cloning to pretrain our OBJECTNAV policy on the human demonstrations we collect. Let $\pi_\theta^{BC}(a_t \mid o_t)$ denote a policy parametrized by θ that maps observations o_t to a distribution over actions a_t. Let τ denote a trajectory consisting of state, observation, action tuples: $\tau = \left(s_0, o_0, a_0, \ldots, s_T, o_T, a_T\right)$ and $\mathcal{T} = \left\{\tau^{(i)}\right\}_{i=1}^{N}$ denote a dataset of human demonstrations. The optimal parameters are

$$\theta^* = \arg\min_\theta \sum_{i=1}^{N} \sum_{(o_t, a_t) \in \tau^{(i)}} -\log\left(\pi_\theta^{BC}(a_t \mid o_t)\right) \quad (1)$$

We use inflection weighting [27] to adjust the loss function to upweight timesteps where actions change (*i.e.* $a_{t-1} \neq a_t$).

Our **ObjectNav policy** architecture is a simple CNN+RNN model from [28]. To encode RGB input ($i_t = \text{CNN}(I_t)$), we use a ResNet50 [29]. Following [28], the CNN is first pre-trained on the Omnidata starter dataset [30] using the self-supervised pretraining method DINO [31] and then fine-tuned during OBJECTNAV training. The GPS+Compass inputs, $P_t = (\Delta x, \Delta y, \Delta z)$, and $R_t = (\Delta\theta)$, are passed through fully-connected layers $p_t = \text{FC}(P_t), r_t = \text{FC}(R_t)$ to embed them to 32-d vectors. Finally, we convert the object goal category to one-hot and pass it through a fully-connected layer $g_t = \text{FC}(G_t)$, resulting in a 32-d vector. All of these input features are concatenated to form an observation embedding, and fed into a 2-layer, 2048-d GRU at every timestep to predict a distribution over actions a_t - formally, given current observations $o_t = [i_t, p_t, r_t, g_t]$, $(h_t, a_t) = \text{GRU}(o_t, h_{t-1})$. To reduce overfitting, we apply color-jitter and random shifts [32] to the RGB inputs.

[2] We don't use this sensor as we don't find it helpful.

[3] https://aihabitat.org/challenge/2022/

B. RL Finetuning

Our motivation for RL-finetuning is two-fold. First, finetuning may allow for higher performance as behavior cloning is known to suffer from a train/test mismatch – when training, the policy sees the result of taking ground-truth actions, while at test-time, it must contend with the consequences of its own actions. Second, collecting more human demonstrations on new scenes or simply to improve performance is time-consuming and expensive. On the other hand, RL-finetuning is trivially scalable (once annotated 3D scans are available) and has the potential to reduce the amount of human demonstrations needed.

B.1. Setup

The RL objective is to find a policy $\pi_\theta(a|s)$ that maximizes expected sum of discounted future rewards. Let τ be a sequence of object, action, reward tuples (o_t, a_t, r_t) where $a_t \sim \pi_\theta(\cdot \mid o_t)$ is the action sampled from the agent's policy, and r_t is the reward. For a discount factor γ, the optimal policy is

$$\pi^* = \underset{\pi}{\text{argmax}}\, \mathbb{E}_{\tau\sim\pi}[R_T], \text{ where } R_T = \sum_{t=1}^{T} \gamma^{t-1} r_t. \quad (2)$$

To solve this maximization problem, actor-critic RL methods learn a state-value function $V(s)$ (also called a critic) in addition to the policy (also called an actor). The critic $V(s_t)$ represents the expected value of returns R_t when starting from state s_t and acting under the policy π, where returns are defined as $R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i$. We use DD-PPO [33], a distributed implementation of PPO [34], an on-policy RL algorithm. Given a θ-parameterized policy π_θ and a set of rollouts, PPO updates the policy as follows. Let $\hat{A}_t = R_t - V(s_t)$, be the advantage estimate and $p_t(\theta) = \frac{\pi_\theta(a_t|o_t)}{\pi_{\theta_{\text{old}}}(a_t|o_t)}$ be the ratio of the probability of action a_t under current policy and under the policy used to collect rollouts. The parameters are updated by maximizing:

$$J^{PPO}(\theta) = \mathbb{E}_t\left[\min\left(p_t(\theta)\hat{A}_t, \text{clip}(p_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \quad (3)$$

ICLR #18

Figure 5. OBJECTNAV trajectories for policies trained with IL+RL on 1) Human Demonstrations, 2) Shortest Paths, and 3) Frontier Exploration Demonstrations.

We use a sparse success reward. Sparse success is simple (does not require hyperparameter optimization) and has fewer unintended consequences (*e.g.* Maksymets *et al.* [17] showed that typical dense rewards used in OBJECTNAV actually *penalize* exploration, even though exploration is necessary for OBJECTNAV in new environments). Sparse rewards are desirable but typically difficult to use with RL (when initializing training from scratch) because they result in nearly all trajectories achieving 0 reward, making it difficult to learn. However, since we pretrain with IL, we do not observe any such pathologies.

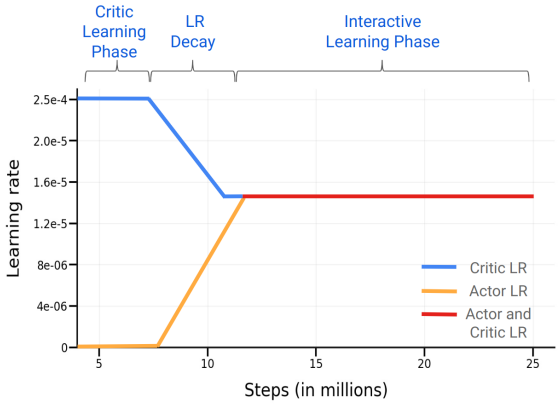

Figure 6. Learning rate schedule for RL Finetuning.

B.2. Finetuning Methodology

We use the behavior cloned policy π_θ^{BC} weights to initialize the actor parameters. However, notice that during behavior cloning we do not learn a critic nor is it easy to do so – a critic learned on human demonstrations (during behavior cloning) would be overly optimistic since all it sees are successes. Thus, we must learn the critic from scratch during RL. Naively finetuning the actor with a randomly-initialized critic leads to a rapid drop in performance[4] (see Fig. 8) since the critic provides poor value estimates which influence the actor's gradient updates (see Eq.(3)). We address this issue by using a two-phase training regime:

Phase 1: Critic Learning. In the first phase, we rollout trajectories using the frozen policy, pre-trained using IL, and use them to learn a critic. To ensure consistency of rollouts collected for critic learning with RL training, we sample actions (as opposed to using `argmax` actions) from the pre-trained IL policy: $a_t \sim \pi_\theta(s_t)$. We train the critic until its loss plateaus. In our experiments, we found $8M$ steps to be sufficient. In addition, we also initialize the weights of the critic's final linear layer close to zero to stabilize training.

Phase 2: Interactive Learning. In the second phase, we unfreeze the actor RNN[5] and finetune both actor and critic weights. We find that naively switching from phase 1 to phase 2 leads to small improvements in policy performance at convergence. We gradually decay the critic learning rate from 2.5×10^{-4} to 1.5×10^{-5} while warming-up the policy learning rate from 0 to 1.5×10^{-5} between $8M$ to $12M$ steps, and then keeping both at 1.5×10^{-5} through the course of training. See Fig. 6. We find that using this learning rate schedule helps improve policy performance. For parameters that are shared between the actor and critic (*i.e.* the RNN), we use the lower of the two learning rates (*i.e.* always the actor's in our schedule). To summarize our finetuning methodology:

– First, we initialize the weights of the policy network with the IL-pretrained policy and initialize critic weights close to zero. We freeze the actor and shared weights. The only learnable parameters are in the critic.
– Next, we learn the critic weights on rollouts collected

[4] After the initial drop, the performance increases but the improvements on success are small.

[5] The CNN and non-visual observation embedding layers remain frozen. We find this to be more stable.

from the pretrained, frozen policy.
– After training the critic, we warmup the policy learning rate and decay the critic learning rate.
– Once both critic and policy learning rate reach a fixed learning rate, we train the policy to convergence.

B.3. Results

Comparing with the RL-finetuning approach in VPT [21]. We start by comparing our proposed RL-finetuning approach with the approach used in VPT [21]. Specifically, [21] proposed initializing the critic weights to zero, replacing entropy term with a KL-divergence loss between the frozen IL policy and the RL policy, and decay the KL divergence loss coefficient, ρ, by a fixed factor after every iteration. Notice that this prevents the actor from drifting too far too quickly from the IL policy, but does not solve uninitialized critic problem. To ensure fair comparison, we implement this method within our DD-PPO framework to ensure that any performance difference is due to the fine-tuning algorithm and not tangential implementation differences. Complete training details are in the Appendix F.3. We keep hyperparameters constant for our approach for all experiments. Table 3 reports results on HM3D VAL for the two approaches using $20k$ human demonstrations. We find that PIRLNav achieves $+2.1\%$ Success compared to VPT and comparable SPL.

Method	Success (↑)	SPL (↑)
1) IL	52.0%	20.6%
2) IL→RL-FT w/ VPT [21]	60.1%	**29.1%**
3) PIRLNav (Ours)	**62.2%**	**28.7%**

Table 3. Comparison with VPT on HM3D VAL [16, 35]

Method	Success (↑)	SPL (↑)
1) IL	52.0%	20.6%
2) IL→RL-FT	54.8%	**29.1%**
3) IL→RL-FT (+ Critic Learning)	55.5%	26.7%
4) IL→RL-FT (+ Critic Learning, Critic Decay)	59.6%	26.5%
5) IL→RL-FT (+ Critic Learning, Actor Warmup)	58.7%	25.8%
6) PIRLNav	**62.2%**	28.7%

Table 4. RL-finetuning ablations on HM3D VAL [16, 35]

Ablations. Next, we conduct ablation experiments to quantify the importance of each phase in our RL-finetuning approach. Table 4 reports results on the HM3D VAL split for a policy IL-pretrained on $20k$ human demonstrations and RL-finetuned for $300M$ steps, complete training details are in Appendix F.4. First, without a gradual learning transition (row 2), *i.e.* without a critic learning and LR decay phase, the policy improves by 2.8% on success and 8.5% on SPL. Next, with only a critic learning phase (row 3), the policy improves by 3.5% on success and 6.1% on SPL. Using an LR decay schedule only for the critic after the critic learning phase improves success by 7.6% and SPL by 5.9%, and using an LR warmup schedule for the actor (but no critic LR decay) after the critic learning phase improves success by 6.7% and SPL by 5.3%. Finally, combining everything (critic-only learning, critic LR decay, actor LR warmup), our policy improves by 10.2% on success and 8.1% on SPL.

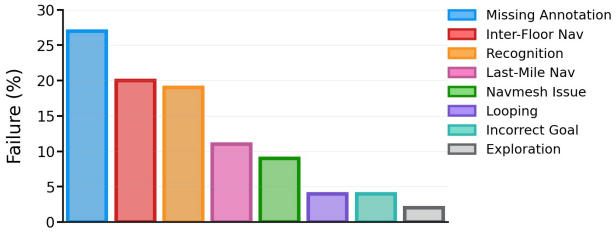

Figure 7. Failure modes of our best IL+RL OBJECTNAV policy

Method	TEST-STD Success (↑)	TEST-STD SPL (↑)	TEST-CHALLENGE Success (↑)	TEST-CHALLENGE SPL (↑)
1) Stretch [24]	60.0%	34.0%	56.0%	29.0%
2) ProcTHOR-Large [36]	54.0%	32.0%	-	-
3) Habitat-Web [1]	55.0%	22.0%	-	-
4) DD-PPO [37]	26.0%	12.0%	-	-
5) Populus A.	66.0%	32.0%	60.0%	30.0%
6) ByteBOT	68.0%	37.0%	64.0%	35.0%
7) PIRLNav[6]	**65.0%**	33.0%	**65.0%**	33.0%

Table 5. Results on HM3D TEST-STANDARD and TEST-CHALLENGE [16, 37]. Unpublished works submitted only to the OBJECTNAV leaderboard have been grayed out.

ObjectNav Challenge 2022 Results. Using our overall two-stage training approach of IL-pretraining followed by RL-finetuning, we achieve state-of-the-art results on OBJECTNAV– 65.0% success and 33.0% SPL on both the TEST-STANDARD and TEST-CHALLENGE splits. Table 5 compares our results with the top-4 entries to the Habitat OBJECTNAV Challenge 2022 [37]. Our approach outperforms Stretch [24] on success rate on both TEST-STANDARD and TEST-CHALLENGE and is comparable on SPL (1% worse on TEST-STANDARD, 4% better on TEST-CHALLENGE). ProcTHOR [36], which uses $10k$ procedurally-generated environments for training, achieves 54% success and 32% SPL on TEST-STANDARD split, which is 11% worse at success and 1% worse at SPL than ours. For sake of completeness, we also report results of two unpublished entries uploaded to the leaderboard – Populus A. and ByteBOT. Unfortunately, there is no associated report yet with these entries, so we are unable to comment on the details of these approaches, or even whether the comparison is meaningful.

C. Failure Modes

To better understand the failure modes of our IL+RL OBJECTNAV policies, we manually annotate 592 failed HM3D VAL episodes from our best OBJECTNAV agent. See Fig. 7. We include videos of various failures modes in the supplement. The most common failure modes are:

[6]The approach is called "BadSeed" on the HM3D leaderboard: eval.ai/web/challenges/challenge-page/1615/leaderboard/3899

ICLR #18

Missing Annotations (27%): Episodes where the agent navigates to the correct goal object category but the episode is counted as a failure due to missing annotations in the data.
Inter-Floor Navigation (21%): The object is on a different floor and the agent fails to climb up/down the stairs.
Recognition Failure (20%): The agent sees the object in its field of view but fails to navigate to it.
Last Mile Navigation [38] (12%). Repeated collisions against objects or mesh geometry close to the goal object preventing the agent from reaching close to it.
Navmesh Failure (9%). Hard-to-navigate meshes blocking the path of the agent. *E.g.* in one instance, the agent fails to climb stairs because of a narrow nav mesh on the stairs.
Looping (4%). Repeatedly visiting the same location and not exploring the rest of the environment.
Semantic Confusion (5%). Confusing the goal object with a semantically-similar object. *E.g.* 'armchair' for 'sofa'.
Exploration Failure (2%). Catch-all for failures in a complex navigation environment, early termination, semantic failures (*e.g.* looking for a chair in a bathroom), *etc*.

As can be seen in Fig. 7, most of the failures (∼36%) are due to issues in the ObjectNav dataset – 27% due to missing object annotations + 9% due to holes / issues in the navmesh. 21% failures are due to the agent being unable to climb up/-down stairs. We believe this happens because climbing up / down stairs to explore another floor is a difficult behavior to learn and there are few episodes that require this. Oversampling inter-floor navigation episodes during training can help address this. Another failure mode is failing to recognize the goal object – 20% failures where the object is in the agent's field of view but it does not navigate to it, and 5% where the agent navigates to another semantically-similar object. Advances in the visual backbone and object recognition can help address these. To this end, prior works [1, 24] have used explicit semantic segmentation modules to recognize objects at each step of navigation. Incorporating this within the IL+RL training pipeline could help. 11% failures are due to last mile navigation, suggesting that equipping the agent with better goal-distance estimators could help. Finally, only ∼6% failures are due to looping and lack of exploration, which is promising!

D. Related Work

ObjectGoal Navigation. Prior works on ObjectNav have used end-to-end reinforcement learning (RL) [17, 39, 40], modular learning [24, 41, 42], and imitation learning [1, 28]. Works that use end-to-end RL have proposed improved visual representations [39, 43], auxiliary tasks [40], and data augmentation techniques [17] to improve generalization to unseen environments. Improved visual representations include object relation graphs [43] and semantic segmentations [39]. Ye *et al*. [40] use auxiliary tasks like predicting environment dynamics, action distributions, and map coverage in addition to ObjectNav and achieve promising results. Maksymets *et al*. [17] improve generalization of RL agents by training with artificially inserted objects and proposing a reward to incentivize exploration.

Modular learning methods for ObjectNav have also emerged as a strong competitor [24, 35, 41]. These methods rely on separate modules for semantic mapping that build explicit structured map representations, a high-level semantic exploration module that is learned through RL to solve the 'where to look?' subproblem, and a low-level navigation policy that solves 'how to navigate to (x, y)?'.

The current state-of-the-art methods on ObjectNav [1, 28] make use of imitation learning (IL) on a large dataset of $80k$ human demonstrations. with a simple CNN+RNN policy architecture. In this work, we improve on them by developing an effective approach to finetune these imitation-pretrained policies with RL.

Imitation Learning and RL Finetuning. Prior works have considered a special case of learning from demonstration data. These approaches initialize policies trained using behavior cloning, and then fine-tune using on-policy reinforcement learning [18, 20–22, 44, 45], On classical tasks like cart-pole swing-up [18], balance, hitting a baseball [44], and underactuated swing-up [45], demonstrations have been used to speed up learning by initializing policies pretrained on demonstrations for RL. Similar to these methods, we also use a on-policy RL algorithm for finetuning the policy trained with behavior cloning. Rajeswaran *et al*. [20] (DAPG) pretrain a policy using behavior cloning and use an augmented RL finetuning objective to stay close to the demonstrations which helps reduce sample complexity. Unfortunately DAPG is not feasible in our setting as it requires solving a systems research problem to efficiently incorporate replaying demonstrations and collecting experience online at our scale. [20] show results of the approach on a dexterous hand manipulation task with a small number of demonstrations that can be loaded in system memory and therefore did not need to solve this system challenge. This is not possible in our setting, just the 256×256 RGB observations for the $77k$ demos we collect would occupy over 2 TB memory, which is out of reach for all but the most exotic of today's systems. There are many methods for incorporating demonstrations/imitation learning with off-policy RL [46–50]. Unfortunately these methods were not designed to work with recurrent policies and adapting off-policy methods to work with recurrent policies is challenging [51]. See the Appendix E for more details. The RL finetuning approach that demonstrates results with an actor-critic and high-dimensional visual observations, and is thus most closely related to our setup is proposed in VPT [21]. Their approach uses Phasic Policy Gradients (PPG) [52] with a KL-divergence loss between the current policy and

ICLR #18

the frozen pretrained policy, and decays the KL loss weight ρ over time to enable exploration during RL finetuning. Our approach uses Proximal Policy Gradients (PPO) [34] instead of PPG, and therefore does not require a KL constraint, which is compute-expensive, and performs better on OBJECTNAV.

E. Prior work in RL Finetuning

E.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.

E.2. Feasibility of Off-Policy RL finetuning

There are several methods for incorporating demonstrations with off-policy RL [46–50]. 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

ICLR #18

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 [51] found using this strategy prevents a RNN from learning long-term dependencies in more memory critical environments like DMLab. [51] 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 [51] 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 [51, 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.

F. Training Details

F.1. Imitation Learning

We use a distributed implementation of behavior cloning by [1] for our imitation learning 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 IL 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.

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.

F.2. Reinforcement Learning

To train our policy using RL we use PPO with Generalized Advantage Estimation (GAE) [56]. 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 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.

F.3. RL Finetuning using VPT

To compare with RL finetuning approach proposed in VPT [21] we implement the method in DD-PPO framework.

ICLR #18

Specifically, we initialize the critic weights to zero, replace the entropy term in PPO [34] 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:

$$L_{kl\_penalty} = \rho \text{KL}(\pi_\theta^{BC}, \pi_\theta) \tag{6}$$

where π_θ^{BC} is the frozen IL 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.

F.4. RL Finetuning Ablations

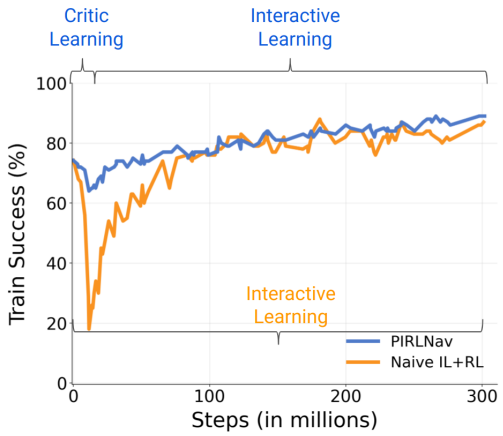

Figure 8. A policy pretrained on the OBJECTNAV task is used as initialization for actor weights and critic weights are initialized randomly for RL finetuning using DD-PPO. The policy performance immediately starts dropping early on during training and then recovers leading to slightly higher performance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

Method	Success (↑)	SPL (↑)
1) IL	52.0%	20.6%
2) IL→RL-FT	54.8%	**29.1%**
3) IL→RL-FT (+ Critic Learning)	55.5%	26.7%
4) IL→RL-FT (+ Critic Learning, Critic Decay)	59.6%	26.5%
5) IL→RL-FT (+ Critic Learning, Actor Warmup)	58.7%	25.8%
6) PIRLNav	**62.2%**	28.7%

Table 8. RL-finetuning ablations on HM3D VAL [16, 35]

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 IL 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 IL → RL (row 2) and IL → 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 IL → 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.