Learning from Guided Play: A Scheduled Hierarchical Approach for Improving Exploration in Adversarial Imitation Learning

Trevor Ablett*, **Bryan Chan***, **Jonathan Kelly**
University of Toronto, Canada

Abstract

Effective exploration continues to be a significant challenge that prevents the deployment of reinforcement learning for many physical systems. This is particularly true for systems with continuous and high-dimensional state and action spaces, such as robotic manipulators. The challenge is accentuated in the sparse rewards setting, where the low-level state information required for the design of dense rewards is unavailable. Adversarial imitation learning (AIL) can partially overcome this barrier by leveraging expert-generated demonstrations of optimal behaviour and providing, essentially, a replacement for dense reward information. Unfortunately, the availability of expert demonstrations does not necessarily improve an agent's capability to explore effectively and, as we empirically show, can lead to inefficient or stagnated learning. We present Learning from Guided Play (LfGP), a framework in which we leverage expert demonstrations of, in addition to a main task, multiple auxiliary tasks. Subsequently, a hierarchical model is used to learn each task reward and policy through a modified AIL procedure, in which exploration of all tasks is enforced via a scheduler composing different tasks together. This affords many benefits: learning efficiency is improved for main tasks with challenging bottleneck transitions, expert data becomes reusable between tasks, and transfer learning through the reuse of learned auxiliary task models becomes possible. Our experimental results in a challenging multitask robotic manipulation domain indicate that our method compares favourably to supervised imitation learning and to a state-of-the-art AIL method.[1]

1 Introduction

Exploration is a crucial part of effective reinforcement learning (RL). A variety of methods have been developed for attempting to optimize the exploration-exploitation trade-off of RL agents [1, 2, 3], but the development of a method that generalizes across domains is still very much an open research problem. A simple, well-known approach for improving learning efficiency is to provide a dense, or "shaped" reward to learn from, but this can be very challenging to design appropriately [4]. Furthermore, the environment may not directly provide the low-level state information required for such a reward. An alternative to providing a dense reward is to learn a reward function from expert demonstrations of a task, in a process known as inverse RL (IRL). Most current, state-of-the-art approaches to IRL are in the class known as adversarial imitation learning (AIL). In AIL, rather than learning a true reward function, the policy and the discriminator form a two-player mini-max optimization problem: the policy aims to confuse the discriminator by minimizing the objective while the discriminator aims identify expert-like data by maximizing the objective [5, 6, 7].

[1] Code available at `https://github.com/utiasSTARS/lfgp`.

Deep Reinforcement Learning Workshop at NeurIPS 2021, Sydney, Australia.

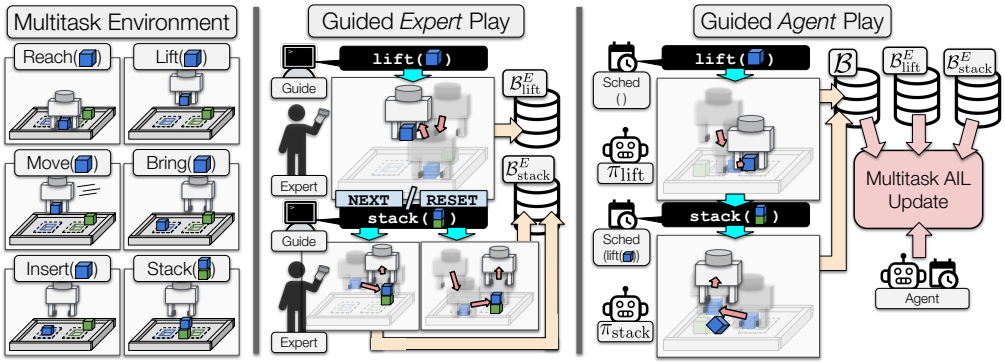

Figure 1: The main components of our system for learning from guided play. In a multitask environment, a guide prompts an expert for a mix of multitask demonstrations, after which we learn a multitask policy through scheduled hierarchical AIL.

AIL has been shown to be effective in many continuous-control environments [5, 7], but its application to long-horizon robotic manipulation tasks with a wide distribution of possible initial configurations remains challenging [7, 8, 9]. In this work, we investigate the use of AIL in a multitask robotic manipulation domain (see the left side of Figure 1), specifically investigating stacking, unstacking and then stacking, reaching followed by bringing to a target, and reaching followed by inserting. We find that a state-of-the-art AIL method, in which off-policy learning is used to maximize sample efficiency [7], performs poorly on these complex manipulation tasks. In all of our tasks, AIL even performs worse than the simpler alternative of supervised learning (often referred to as behavioural cloning (BC). We hypothesize that this is because AIL, and more generally IRL, can provide something akin to a dense reward, but because this learned reward is not necessarily optimal for teaching, AIL alone may not necessarily enforce sufficiently diverse exploration to allow effective learning.

To overcome this limitation of AIL, we present Learning from Guided Play (LfGP), in which we combine AIL with a scheduled approach to hierarchical RL (HRL), allowing an agent to "play" in the environment using the expert "guide." Using expert demonstrations of multiple auxiliary tasks, along with a main task, we demonstrate that our scheduled hierarchical agent is able to learn much more effectively than with AIL alone. This approach also affords several other benefits: learning a challenging main task becomes more efficient than supervised learning in terms of required expert data, no hand-crafted reward functions are needed as in [10], expert data on auxiliary tasks can be reused from one main task to another, and learning time can be reduced by reusing previously-learned auxiliary models on new main tasks through simple transfer learning. We choose this name in reference to the work upon which we build [10], which is in turn inspired by the "playful" phase of learning experienced by children, as opposed to goal-directed learning. In our case, *guided* represents two separate but related ideas: first, the expert guides this play, as opposed to requiring hand-crafted sparse rewards as in [10] (right side of Figure 1), and second, the expert gathering of multitask, semi-structured demonstrations is *guided* by uniform-random task selection (middle of Figure 1), rather than requiring the expert to choose transitions between goals, as in [11, 12]. Our specific contributions are:

1. A novel approach to AIL that learns a reward and policy for a challenging main task by simultaneously learning rewards and policies for auxiliary tasks.
2. A variety of manipulation experiments illustrating the viability of our approach, including comparisons with competing approaches to AIL and imitation learning.
3. An extension of our method to the domain of transfer learning.

2 Related Work

Imitation learning (IL) is often divided into two main categories: behavioural cloning (BC) [13] and inverse reinforcement learning (IRL) [14, 15]. BC recovers the expert policy via supervised learning, but it suffers from compounding errors due to covariate shift [16]. Alternatively, IRL partially alleviates the covariate shift problem by estimating the reward function and then applying RL using the learned reward. A popular approach to IRL is adversarial imitation learning (AIL)

[5, 7, 17, 18], in which the expert policy is recovered by matching the occupancy measure between the generated data and the demonstration data. Our proposed method extends existing AIL algorithms by enabling more effective exploration via the use of a scheduled multitask model.

Agents learned via hierarchical reinforcement learning (HRL) methods, which aim to learn and act over multiple levels of temporal abstractions in long planning horizon tasks, are shown to provide more effective exploration than agents operating with only a single level [10, 19, 20, 21, 22]. Our work most closely resembles hierarchical AIL methods that attempt to combine AIL with HRL [18, 23, 24, 25]. Existing works [23, 24, 25] often formulate the hierarchical agent using the Options framework [19] and recover the reward function using AIL [5]. Both [23] and [25] leverage task-specific expert demonstrations to learn options using mixture-of-experts and expectation-maximization strategies respectively. Our work focuses on expert demonstrations that include multiple tasks, each of which encodes clear semantic meaning. In the multitask setting, [18] and [24] leverage unsegmented, multitask expert demonstrations to learn low-level policies via a latent variable model. Other work has used a large corpus of unsegmented but semantically meaningful "play" expert data to bootstrap policy learning [11, 12]; we define our expert dataset as *guided* play, in that the expert completes semantically meaningful auxiliary tasks with provided transitions, reducing the burden on the expert to generate this data arbitrarily and simultaneously providing auxiliary task labels. Compared with learning from unsegmented demonstrations, the use of segmented demonstrations, as in [26], ensures that we know which auxiliary tasks our model will be learning, and opens up the possibility of expert data reuse and transfer learning without further assumptions. Finally, we deviate from the Options framework and build upon Scheduled Auxiliary Control (SAC-X) to train our hierarchical agent as it is shown to work well in complex manipulation tasks [10, 27].

3 Problem Formulation

We define a Markov Decision Process (MDP) as $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, R, \mathcal{P}, \rho_0, \gamma \rangle$. The sets $\mathcal{S}$ and $\mathcal{A}$ are respectively the state and action space, $R : S \times A \to \mathbb{R}$ is a reward function, $\mathcal{P} : S \times A \times S \to \mathbb{R}_{\geq 0}$ is the state-transition environment dynamics distribution, $\rho_0 : S \to \mathbb{R}_{\geq 0}$ is the initial state distribution, and γ is the discount factor. Actions are sampled from a stochastic policy $\pi(a|s)$. The policy π interacts with the environment to yield experience (s_t, a_t, r_t, s_{t+1}) for $t = 0, \dots, \infty$, where $s_0 \sim \rho_0(\cdot), a_t \sim \pi(\cdot|s_t), s_{t+1} \sim \mathcal{P}(\cdot|s_t, a_t), r_t = R(s_t, a_t)$.

We assume infinite-horizon, non-terminating environments where t is unbounded for notational convenience, but our method also easily applies to the finite-horizon case. We aim to learn a policy π that maximizes the expected return $J(\pi) = \mathbb{E}_\pi \left[G(\tau_{0:\infty}) \right] = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \right]$, where $\tau_{t:\infty} = \{(s_t, a_t), \dots\}$ is the trajectory starting in state s_t, and $G(\tau_{t:\infty})$ is the return of the trajectory.

In this work, we focus on imitation learning (IL) in which R is unknown and instead, we are given a finite set of expert demonstration (s, a) pairs $\mathcal{B}^E = \left\{ (s, a)^{E,(1)}, \dots \right\}$. In AIL, we attempt to simultaneously learn π and a discriminator $D : S \times A \to \mathbb{R}_{\geq 0}$ that differentiates between expert samples $(s, a)^E$ and policy samples $(s, a)^\pi$, and subsequently define R using D [5, 7]. To accommodate hierarchical learning, we augment $\mathcal{M}$ to contain auxiliary tasks, where $\mathcal{T}_{\text{aux}} = \{\mathcal{T}_1, \dots, \mathcal{T}_K\}$ are separate MDPs that share $\mathcal{S}, \mathcal{A}, \mathcal{P}, \rho_0$ and γ with the main task $\mathcal{T}_{\text{main}}$ but have their own reward functions R_k. With this modification, we refer to entities in our model that are specific to task $\mathcal{T} \in \mathcal{T}_{\text{all}}$, $\mathcal{T}_{\text{all}} = \mathcal{T}_{\text{aux}} \cup \{\mathcal{T}_{\text{main}}\}$, as $(\cdot)_\mathcal{T}$. We assume to have a set of expert data $\mathcal{B}^E_\mathcal{T}$ for each task.

4 Learning from Guided Play (LfGP)

We now introduce Learning from Guided Play (LfGP), in which we improve the performance of off-policy AIL in complex manipulation tasks that require synthesizing a composition of potentially reusable skills. Hypothesizing that better exploration will improve performance [21], our method specifically enforces the learning and exploration of these skills through a hierarchical framework [10], while still exclusively using expert data. Our primary goal is to learn a policy $\pi_{\mathcal{T}_{\text{main}}}$ that can solve the main task $\mathcal{T}_{\text{main}}$, with a secondary goal of also learning auxiliary task policies $\pi_{\mathcal{T}_1}, \dots, \pi_{\mathcal{T}_K}$ that can be used both for improved exploration and for transfer learning. More specifically, we derive a hierarchical learning objective that is decomposed into three parts: i) recovering the reward function of each task with expert demonstrations, ii) training all policies to achieve their respective goals,

and iii) using all policies for effective exploration in $\mathcal{T}_{\text{main}}$. For a summary of the algorithm, see our supplementary material.

4.1 Learning the Reward Function

We first describe how to recover the reward functions from expert demonstrations. For each task $\mathcal{T} \in \mathcal{T}_{\text{all}}$, we learn a discriminator $D_{\mathcal{T}}(s, a)$ that is used to define the reward function for policy optimization. The discriminator is trained in an off-policy manner [17]:

$$\max_{D_{\mathcal{T}}} \mathbb{E}_{\mathcal{B}} \left[\frac{\rho_{\pi_{\mathcal{T}}}(s,a)}{\rho_{\mathcal{B}}(s,a)} \log\left(1 - D_{\mathcal{T}}(s,a)\right) \right] + \mathbb{E}_{\mathcal{B}^E_{\mathcal{T}}} \left[\log\left(D_{\mathcal{T}}(s,a)\right) \right], \tag{1}$$

where $\mathcal{B}$ is the replay buffer storing transitions collected while interacting with the environment, $\rho_{\pi_{\mathcal{T}}}(s, a)$ the intention occupancy measure, and $\rho_{\mathcal{B}}$ the distribution induced by the replay buffer $\mathcal{B}$. In practice, following [17], we sample the transitions from $\mathcal{B}$ without applying importance sampling:

$$\max_{D_{\mathcal{T}}} \mathbb{E}_{\mathcal{B}} \left[\log\left(1 - D_{\mathcal{T}}(s,a)\right) \right] + \mathbb{E}_{\mathcal{B}^E_{\mathcal{T}}} \left[\log\left(D_{\mathcal{T}}(s,a)\right) \right]. \tag{2}$$

Each discriminator tries to differentiate the occupancy measure between the distributions induced by $\mathcal{B}^E_{\mathcal{T}}$ and $\mathcal{B}$. As a result, the policy $\pi_{\mathcal{T}}$ attempts to confuse the discriminator by minimizing Equation 2 through changing the distribution induced by the replay buffer to be similar to the expert demonstrations. In our implementation, the discriminators minimize the joint discriminator loss

$$\mathcal{L}(D) = -\sum_{\mathcal{T}}^{\mathcal{T}_{\text{all}}} \mathbb{E}_{\mathcal{B}} \left[\log\left(1 - D_{\mathcal{T}}(s,a)\right) \right] + \mathbb{E}_{\mathcal{B}^E_{\mathcal{T}}} \left[\log\left(D_{\mathcal{T}}(s,a)\right) \right]. \tag{3}$$

The resulting $D_{\mathcal{T}}$ can be used to define various reward functions [17]—we define the reward function for each task $\mathcal{T}$ to be $D_{\mathcal{T}}$. Thus, we define the expected return for $\pi_{\mathcal{T}}$ as

$$J(\pi_{\mathcal{T}}) = \mathbb{E}_{\pi_{\mathcal{T}}} \left[G_{\mathcal{T}}(\tau_{t:\infty}) \right] = \mathbb{E}_{\pi_{\mathcal{T}}} \left[\sum_{t=0}^{\infty} \gamma^t D_{\mathcal{T}}(s_t, a_t) \right]. \tag{4}$$

4.2 Learning the Hierarchical Agent

We adapt SAC-X [10] to learn the hierarchical agent. The agent includes two levels of policies: intention policies (low-level) and a scheduler (high-level), as well as the Q-functions and the discriminators. The intentions aim to solve their corresponding tasks (i.e. intentions are $\pi_{\mathcal{T}}$ aiming to maximize task return $J(\pi_{\mathcal{T}})$), whereas the scheduler aims to maximize the expected return for $\mathcal{T}_{\text{main}}$ by selecting a sequence of intentions to interact with the environment. For the remainder of this paper, when we refer to a policy, we are referring to an intention policy, as opposed to the scheduler, unless otherwise specified.

4.2.1 Learning the Intentions

We learn each intention using Soft Actor-Critic (SAC) [28], an actor-critic algorithm that optimizes for the entropy-regularized objective, though any off-policy RL algorithm would suffice. We augment Equation 4 with entropy regularization as

$$J(\pi_{\mathcal{T}}) = \mathbb{E}_{\pi_{\mathcal{T}}} \left[\sum_{t=0}^{\infty} \gamma^t (D_{\mathcal{T}}(s_t, a_t) + \alpha \mathcal{H}(\pi_{\mathcal{T}}(\cdot|s_t))) \right], \tag{5}$$

where the learned temperature α determines the importance of the entropy term against the reward, and $\mathcal{H}(\pi_{\mathcal{T}}(\cdot|s_t))$ is the entropy of the intention $\pi_{\mathcal{T}}$ at state s_t. The corresponding soft Q-function $Q_{\mathcal{T}}(s_t, a_t)$ for task $\mathcal{T}$ is defined as

$$Q_{\mathcal{T}}(s_t, a_t) = D_{\mathcal{T}}(s_t, a_t) + \mathbb{E}_{\pi_{\mathcal{T}}} \left[\sum_{t=0}^{\infty} \gamma^t (D_{\mathcal{T}}(s_{t+1}, a_{t+1}) + \alpha \mathcal{H}(\pi_{\mathcal{T}}(\cdot|s_{t+1}))) \right]. \tag{6}$$

The intentions maximize the joint policy improvement objective:

$$\mathcal{L}(\pi) = \sum_{\mathcal{T}}^{\mathcal{T}_{\text{all}}} \mathbb{E}_{s \sim \mathcal{B}, a \sim \pi_{\mathcal{T}}(\cdot|s)} \left[Q_{\mathcal{T}}(s,a) - \alpha \log \pi_{\mathcal{T}}(a|s) \right]. \tag{7}$$

Notice that the replay buffer $\mathcal{B}$ contains transitions generated using different intentions. This allows each intention $\pi_{\mathcal{T}}$ to learn optimal actions for its respective task starting at any state visited by any intention. In other words, other intentions $\pi_{\mathcal{T}' \neq \mathcal{T}}$ interacting with the environment provide exploration samples for $\pi_{\mathcal{T}}$.

For policy evaluation, the soft Q-functions $Q_{\mathcal{T}}$ for each $\pi_{\mathcal{T}}$ minimize the joint soft Bellman residual

$$\mathcal{L}(Q) = \sum_{\mathcal{T}}^{\mathcal{T}_{\text{all}}} \mathbb{E}_{(s,a,r,s') \sim \mathcal{B}, a' \sim \pi_{\mathcal{T}}(\cdot|s')} \left[(Q_{\mathcal{T}}(s,a) - \delta_{\mathcal{T}})^2 \right], \tag{8}$$

$$\delta_{\mathcal{T}} = r + \gamma Q_{\mathcal{T}}(s', a') - \alpha \log \pi_{\mathcal{T}}(a'|s'). \tag{9}$$

4.2.2 Learning the Scheduler

The scheduler composes a sequence of intentions to interact with the environment. The scheduler can switch between intentions every ξ timesteps. With $\gamma < 1$, we define H as the total number of possible intention switches within an episode.[2] The H intention choices made within the episode are defined as $\mathcal{T}^{0:H-1} = \left\{ \mathcal{T}^{(0)}, \ldots, \mathcal{T}^{(H-1)} \right\}$, where $\mathcal{T}^{(h)} \in \mathcal{T}_{\text{all}}$. The main task's return given the intention choices is then defined as

$$G_{\mathcal{T}_{\text{main}}}(\mathcal{T}^{0:H-1}) = \sum_{h=0}^{H-1} \sum_{t=h\xi}^{(h+1)\xi-1} \gamma^t D_{\mathcal{T}_{\text{main}}}(s_t, a_t), \tag{10}$$

where $a_t \sim \pi_{\mathcal{T}^{(h)}}(\cdot|s_t)$ is the action taken at timestep t, sampled from the chosen intention $\mathcal{T}^{(h)}$ in the hth scheduler period. We define the Q-function for the scheduler as $Q_S(\mathcal{T}^{0:h-1}, \mathcal{T}^{(h)}) = \mathbb{E}_{P_S} \left[G_{\mathcal{T}_{\text{main}}}(\mathcal{T}^{h:H-1}) | \mathcal{T}^{0:h-1} \right]$ and represent the scheduler as a Boltzmann distribution P_S over the Q-function. The scheduler maximizes the expected return of the main task following the scheduler:

$$\mathcal{L}(S) = \mathbb{E}_{P_S} \left[Q_S(\emptyset, \mathcal{T}^{(0)}) \right]. \tag{11}$$

We use Monte Carlo returns to estimate Q_S, estimating the expected return using the exponential moving average

$$Q_S(\mathcal{T}^{0:h-1}, \mathcal{T}^{(h)}) = (1-\phi) Q_S(\mathcal{T}^{0:h-1}, \mathcal{T}^{(h)}) + \phi G_{\mathcal{T}_{\text{main}}}(\mathcal{T}^{h:H}), \tag{12}$$

where $\phi \in [0, 1]$ represents the amount of discounting on older returns and $G_{\mathcal{T}_{\text{main}}}(\mathcal{T}^{h:H})$ is the cumulative discounted return of the trajectory τ starting at timestep $h\xi$.

4.3 Expert Data Collection

We assume that each $\mathcal{T} \in \mathcal{T}_{\text{all}}$ has, for evaluation purposes only, a binary indicator of success. In single-task imitation learning (IL) where this assumption is valid, expert data is typically collected by allowing the expert to control the agent until success conditions are met. At that point, the environment is reset following ρ_0 and collection is repeated for a fixed number of episodes or (s, a) pairs. In LfGP, and more generally for multitask IL, expert data can also be collected this way for each $\mathcal{T}$ separately: we refer to this strategy, which we use for most of our experiments, as *reset-based* expert data collection.

A limitation of reset-based expert data collection is that, when training LfGP, the initial state of each individual policy can be any state that $\mathcal{M}$ has been left in by a previous policy, which may include states not in the distribution of ρ_0. This "transition" initial state distribution, which we call $\rho_0(s|\mathcal{T}')$, where $\mathcal{T}' \in \mathcal{T}_{\text{all}}$ corresponds to the previously running policy $\pi_{\mathcal{T}'}$, would be challenging to sample from—it relies on the policies, and it may include states which are impractical to manually reset to (e.g. objects may start off as grasped or in mid-air). Consequently, we propose an alternative data collection strategy: *play-based* expert data collection, where we alternate between uniformly sampling the next task for an expert to complete and having the expert execute that task until success. In our implementation, we also reset the environment following ρ_0 periodically. See middle section of Figure 1 for a comparison of the two methods.

[2] We can choose H in the infinite horizon setting using the effective horizon $\frac{1}{1-\gamma}$.

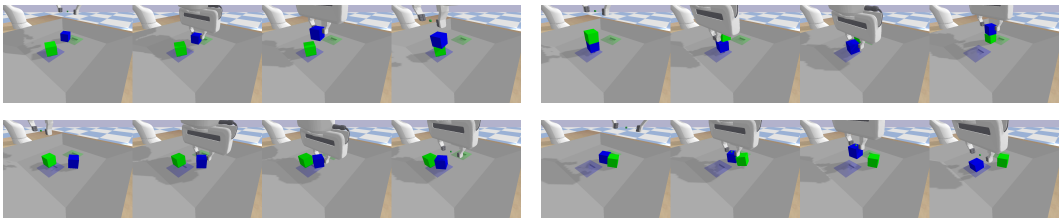

Figure 2: Example successful runs of our four main tasks. Left to right, top to bottom: Stack, Unstack-Stack, Bring, Insert.

5 Experiments

In our experiments, we attempt to answer the following questions:

1. How does the performance of LfGP compare with other existing state-of-the-art IL methods in challenging manipulation tasks, in terms of success rate, environment sample efficiency, and expert sample efficiency?
2. Are models trained using LfGP able to transfer from one main task to another? Does this transfer afford any benefits over learning the new main task from scratch?
3. Does the use of play-based expert data improve performance (see Section 4.3)?

5.1 Experimental Setup

We begin by describing our experimental setup. We complete experiments in a simulation environment containing a 7-DOF Franka Emika Panda manipulator, one green and one blue block, fixed zones corresponding to the green and blue blocks, and one slot in each zone with $<$ 1mm tolerance for fitting the blocks (see bottom right of Figure 2). The environment is designed such that several different challenging tasks can be completed with a common observation and action space. The main tasks that we investigate are Stack, Unstack-Stack, Bring, and Insert (see Figure 2 for examples). When learning Unstack-Stack, the green block is initialized on top of the blue block. For more details on our environment and definitions of task success, see Appendix B and Appendix D respectively. We also define a set of auxiliary tasks: Open-Gripper, Close-Gripper, Reach, Lift, Move-Object, and Bring (Bring is both a main task and an auxiliary task for Insert), all of which have some degree of re-usability between main tasks. Table 1 shows the relationship between main and auxiliary tasks.

We implemented several multitask and single-task baselines to compare our method to. The multitask algorithms, like our method, simultaneously attempt to learn to complete the main task as well as the auxiliary tasks. The single-task algorithms only attempt to learn to complete the main task. In general, we consider that a multitask algorithm is more useful than a single-task algorithm, given the potential to reuse trained models for learning new tasks, as we show in Section 5.3.

The first multitask algorithm that we compare to is a "No Schedule" version of our method (LfGP-NS), where the models are trained similarly to LfGP, but the scheduler only chooses the main task; this is meant to be a similar baseline to the Intentional Unintentional Agent shown in [10, 29]. The expected benefit is that the policies, the Q-functions, and the discriminators share their initial layers between tasks, so the auxiliary tasks can accelerate learning the representation for the main-task models [30]. We also compare to a multitask variant of behavioural cloning (BC), in which we simultaneously train every task by minimizing a multitask mean squared error objective

$$\mathcal{L}(\pi) = \sum_{\mathcal{T}}^{\mathcal{T}_{\text{all}}} \sum_{(s,a)\in\mathcal{B}_{\mathcal{T}}^{E}} \left(\pi_{\mathcal{T}}(s) - a\right)^2, \tag{13}$$

taking only the mean action outputs from each $\pi_{\mathcal{T}}$.

The single-task algorithms we compare to are BC and Discriminator-Actor-Critic (DAC) [7]. To make a fairer comparison and evaluate our method's expert sample efficiency, we provide BC and DAC with an equivalent amount of *total* expert data as our multitask methods as shown in Table 1. To show the possible benefits or detriments of learning only a single main task with an equivalent

Table 1: A summary of the reusability of the expert data used to generate the performance results described in Section 5.2. Each letter under "Dataset Sizes" is the first letter of a single (auxiliary) task, and bolded letters indicate that a dataset was reused for more than one main task (e.g., **O**pen-Gripper was used for all four main tasks). The numbers in this table refer to (s, a) pairs.

Algorithm Class	Task	Dataset Sizes	Reusable	Single	Total
Multitask	Stack	**OCSRLM**: 9k/task	**45k**	9k	54k
	U-Stack	**OCURLM**: 9k/task	**45k**	9k	54k
	Bring	**OCBRLM**: 9k/task	**54k**	0	54k
	Insert	**OCIBRLM**: 9k/task	**54k**	9k	63k
Single-task	Stack	S: 54k (less: 9k)	0	54k (9k)	54k (9k)
	U-Stack	U: 54k (less: 9k)	0	54k (9k)	54k (9k)
	Bring	B: 54k (less: 9k)	0	54k (9k)	54k (9k)
	Insert	I: 63k (less: 9k)	0	63k (9k)	63k (9k)

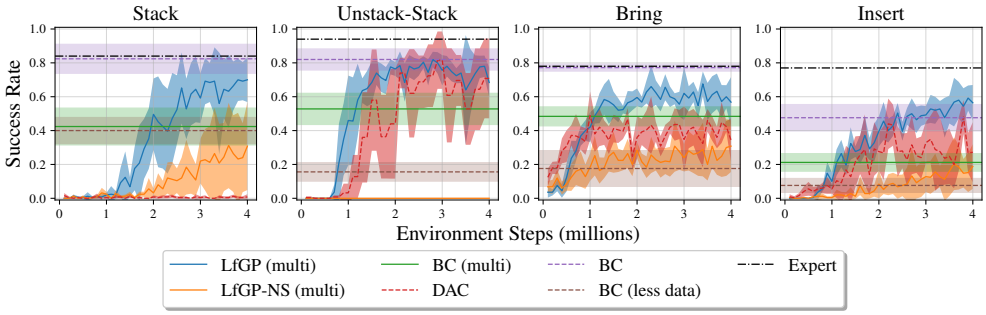

Figure 3: Performance results for our method (LfGP) and the baselines, as described in Section 5.1. Solid lines are multitask methods, dotted lines are single-task methods, and shaded area corresponds to standard deviation across five seeds.

amount of main task data as the multitask algorithm, we also show results for single-task BC with an equivalent amount of main task data to the multitask algorithms, which we call BC (less data).

We gathered expert data by first training an expert policy using SAC-X [10] (see appendix C). We then ran the expert policies to collect various amounts of expert data as described in Table 1. In these experiments, in the interest of testing the benefits of having reusable expert data, we use reset-based expert data (see Section 4.3).

5.2 Performance Results

For the AIL algorithms, we freeze the policies every 100k interactions and evaluate those policies for 50 randomized episodes, only using the mean action outputs. We trained each agent to 4 million environment steps. For the BC algorithms, we train the policies by dividing each expert buffer into a 70%/30% train/validation split, taking the policy after validation error has not improved for 100 epochs and evaluating it on 50 randomized episodes. For all algorithms, we test across five seeds and report the mean and standard deviation of all seeds. For more details of our training algorithms, including hyperparameters, see our supplementary material. The performance results for all methods with respect to the main tasks we tested are shown in Figure 3.

It is clear that our method outperforms multitask BC and LfGP-NS, demonstrating the necessity of exploring each of the policies to maximize performance. Intriguingly, even with substantially more main-task expert data, DAC alone is unable to match the performance of our method, and completely fails in the Stacking task. Single-task BC with more main-task data outperforms our method in Stack and Bring, but we emphasize that this baseline has two major limitations compared with ours: none of the collected expert data for this policy is reusable for other tasks, and the policy itself does not have any learned auxiliary policies that could be reused.

The performance results for all multitask methods and all auxiliary tasks are shown in Figure 4. One notable finding is that multitask BC tends to perform quite well on the auxiliary tasks, and, in fact, for all auxiliary tasks, outperforms LfGP. We suspect this is because, compared to the main task, the

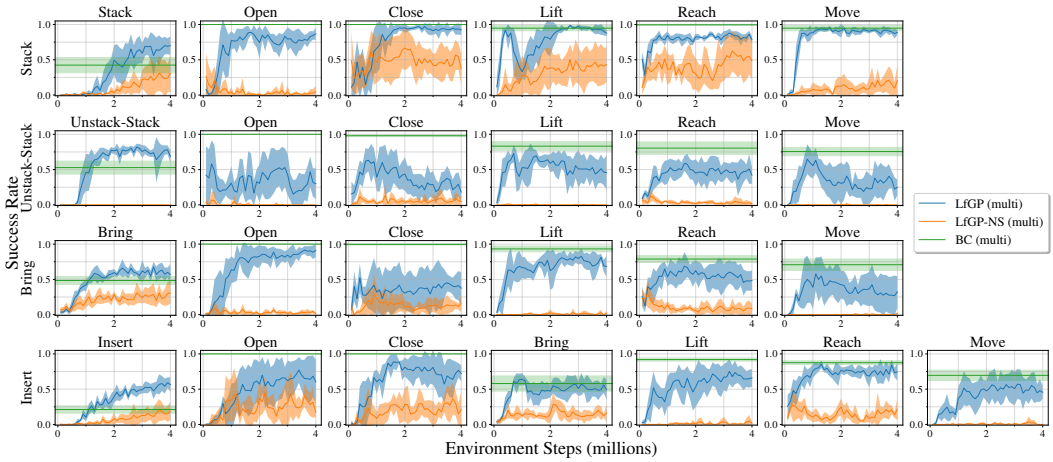

Figure 4: Performance for LfGP and the multitask baselines across all tasks, shaded area corresponds to standard deviation.

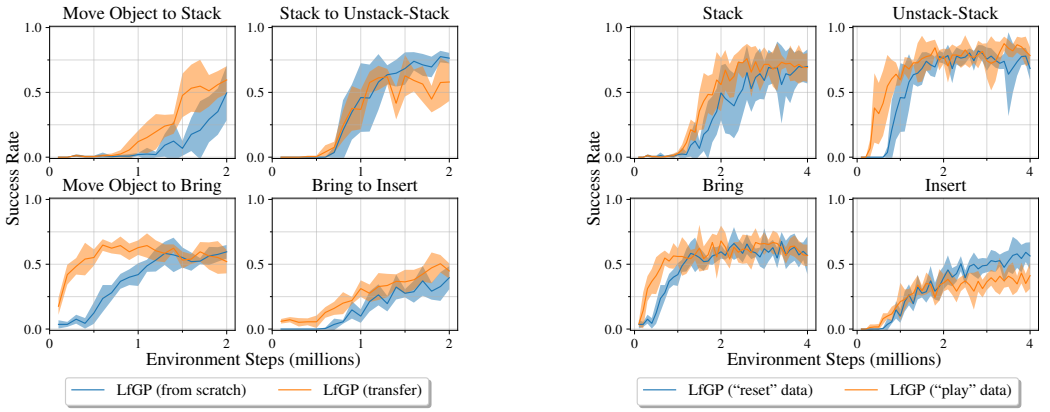

Figure 5: **Left:** The results of our transfer experiments. **Right:** The results of our play-based expert data experiments. Shaded area corresponds to standard deviation.

auxiliary tasks are shorter horizon and simpler than the main task, making them very good candidates for BC. Furthermore, since LfGP maximizes the expected return of the main task, LfGP does not necessarily perform as well on the auxiliary tasks compared to multitask BC, which simultaneously matches the expert for all tasks. A natural followup question is whether we could combine the benefits of quick learning using multitask BC on the simpler auxiliary tasks with the improved performance of LfGP on main tasks—our preliminary results were negative so we leave this as future work.

5.3 Transfer Learning Results

In this section, we show the results of experiments in which we attempt to reuse policies that learned to complete one main task to complete another main task. Specifically, we are interested in reducing the learning time for a main task compared with learning from scratch. To complete these experiments, we chose a relatively simple formulation as a proof-of-concept. For each main task, we take a high-performing existing model that uses a subset of auxiliary tasks required for the new main task and save the parameters and the replay buffer of the existing model. We then train a new model on the new task by loading the saved parameters and replay buffer, adding new model outputs for the new main task. For Stack and Bring, we transfer from a model with $\mathcal{T}_{\text{main}}$ = Move-Object and $\mathcal{T}_{\text{aux}}$ = {Open-Gripper, Close-Gripper, Reach, Lift}. For Unstack-Stack and Insert, we use $\mathcal{T}_{\text{main}}$ = Stack and $\mathcal{T}_{\text{main}}$ = Bring models respectively, with the same $\mathcal{T}_{\text{aux}}$ as in Table 1.

The results of these experiments are shown on the left side of Figure 5. It is clear that in all cases except for Unstack-Stack, transferring trained models from one main task to another is more sample efficient than from scratch. This result is particularly clear in the case of Bring. For Unstack-Stack,

training speed remains roughly consistent, but final performance is actually considerably lower with the transfer model. Given that this experiment was only meant to show that transfer learning would be possible and could provide benefits in some cases, we leave investigating this issue for future work.

5.4 Expert Data Collection Schemes

As described in Section 4.3, there are (at least) two strategies for collecting expert data to be used with LfGP. In this section, we compare the training performance of LfGP between reset-based and play-based expert data.

We show the comparison between the results of training LfGP with reset-based and with play-based data on the right side of Figure 5. For Stack, Unstack-Stack, and Bring, play-based data appears to generally increase the learning speed of LfGP, implying that matching the transition distribution appears to be beneficial for learning, although there is no significant effect on final performance. Conversely, for Insert, play-based data appears to have only a marginal effect on learning speed, while having fairly significant negative impact on final performance. This could be because the Insert task is the least forgiving in terms of the required final state of the object, and reset-based data may contain more transitions between near-insertions and complete insertions than play-based data.

Compared with reset-based data, while play-based data assures that the expert distribution better matches the learning distribution, it also has the downside of making it harder to reuse. In the case of reset-based data, one can easily add a new dataset corresponding to a new task, while keeping existing datasets the same. In play-based, and in our experiments, each individual main task has its own dataset, given that the "transition" initial state distribution should contain data from $\mathcal{T}_{\text{all}}$, which changes depending on $\mathcal{T}_{\text{main}}$.

6 Conclusion

In this work, we identified and addressed an exploration problem in adversarial imitation learning (AIL) algorithms applied to manipulation tasks by introducing Learning from Guided Play (LfGP), a scheduled hierarchical AIL method that enables more effective exploration through the use of "playful" scheduled auxiliary tasks, "guided" by expert data, as opposed to hand-crafted rewards. We demonstrated that our method substantially outperforms an existing state-of-the-art AIL algorithm. As well, with significantly less main task data, our algorithm can perform comparably to behaviour cloning (BC) and significantly outperforms multitask BC given the same data. Furthermore, our method can leverage reusable expert data and its learned intentions can be applied to transfer learning. As future work, we intend to further investigate alternative approaches to transfer learning, such as a combination of multitask BC with our method, to see if the complementary advantages of supervised learning and AIL can reduce the number of required environment interactions.

Broader Impact

In this paper, we introduced a scheduled hierarchical approach to adversarial imitation learning (AIL), and demonstrated its successful application to a simulated robotic manipulation domain. Robotic manipulation systems are inherently dangerous, though significant strides have been made in the field of *collaborative* robotics in the past few decades [31], in which lighter arms with speed and force limitations are used in close proximity with humans. The simulated arm from this work, the Franka Emika Panda, is an example of such an arm. Like any other reinforcement learning (RL) method applied to a physical system, ours shares the safety concern that a system that exhibits any degree of random exploration can act in unexpected ways, potentially causing harm through damage to the agent itself, the environment, or to present humans. In our method, this could be exacerbated by the abrupt switching from one task to another; when using this approach on a real platform, care should be taken to constrain the robot's movements through soft and/or hard boundaries. This abruptness could also be reduced by using some kind of soft switching mechanism between hierarchical members [32, 33]. Compared with RL, our method could also learn unexpected behaviours due to the existence of suboptimal expert data, but this is a well-studied issue in the field of IL [34], so there is no shortage of approaches that could be combined with ours to address potentially imperfect demonstrations.

References

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[2] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying Count-Based Exploration and Intrinsic Motivation. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[3] Ashvin Nair, Bob McGrew, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Overcoming Exploration in Reinforcement Learning with Demonstrations. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6292–6299, 2018.

[4] Andrew Y. Ng and Michael I. Jordan. *Shaping and Policy Search in Reinforcement Learning*. PhD thesis, University of California, Berkeley, 2003.

[5] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *Advances in neural information processing systems*, 29:4565–4573, 2016.

[6] Justin Fu, Katie Luo, and Sergey Levine. Learning robust rewards with adverserial inverse reinforcement learning. In *International Conference on Learning Representations*, 2018.

[7] Ilya Kostrikov, Ofir Nachum, and Jonathan Tompson. Imitation learning via off-policy distribution matching. In *International Conference on Learning Representations*, 2019.

[8] Manu Orsini, Anton Raichuk, Léonard Hussenot, Damien Vincent, Robert Dadashi, Sertan Girgin, Matthieu Geist, Olivier Bachem, Olivier Pietquin, and Marcin Andrychowicz. What matters for adversarial imitation learning? *arXiv preprint arXiv:2106.00672*, 2021.

[9] Rafael Rafailov, Tianhe Yu, Aravind Rajeswaran, and Chelsea Finn. Visual adversarial imitation learning using variational models. In *ICML 2021 Workshop on Unsupervised Reinforcement Learning*, 2021.

[10] Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Wiele, Vlad Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing solving sparse reward tasks from scratch. In *International Conference on Machine Learning*, pages 4344–4353. PMLR, 2018.

[11] Corey Lynch, Mohi Khansari, Ted Xiao, Vikash Kumar, Jonathan Tompson, Sergey Levine, and Pierre Sermanet. Learning latent plans from play. In *Conference on Robot Learning*, pages 1113–1132. PMLR, 2020.

[12] Abhishek Gupta, Vikash Kumar, Corey Lynch, Sergey Levine, and Karol Hausman. Relay policy learning: Solving long horizon tasks via imitation and reinforcement learning. In *Conference on Robot Learning (CoRL)*, 2019.

[13] Dean A. Pomerleau. ALVINN: An Autonomous Land Vehicle in a Neural Network. In D. S. Touretzky, editor, *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS'89)*, pages 305–313, Denver, CO, USA, 1989. Morgan-Kaufmann.

[14] Andrew Ng and Stuart Russell. Algorithms for inverse reinforcement learning. *Proceedings of the Seventeenth International Conference on Machine Learning*, 0:663–670, 2000.

[15] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1, 2004.

[16] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011.

[17] Ilya Kostrikov, Kumar Krishna Agrawal, Sergey Levine, and Jonathan Tompson. Addressing sample inefficiency and reward bias in inverse reinforcement learning. *CoRR*, abs/1809.02925, 2018.

[18] Karol Hausman, Yevgen Chebotar, Stefan Schaal, Gaurav Sukhatme, and Joseph Lim. Multi-modal imitation learning from unstructured demonstrations using generative adversarial nets. In *31st Annual Conference on Neural Information Processing Systems (NIPS 2017)*, pages 1236–1246. Curran Associates, Inc., 2018.

[19] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[20] Tejas D Kulkarni, Karthik Narasimhan, Ardavan Saeedi, and Josh Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Advances in neural information processing systems*, 29:3675–3683, 2016.

[21] Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *Advances in Neural Information Processing Systems*, 31:3303–3313, 2018.

[22] Ofir Nachum, Haoran Tang, Xingyu Lu, Shixiang Gu, Honglak Lee, and Sergey Levine. Why does hierarchy (sometimes) work so well in reinforcement learning? *arXiv preprint arXiv:1909.10618*, 2019.

[23] Peter Henderson, Wei-Di Chang, Pierre-Luc Bacon, David Meger, Joelle Pineau, and Doina Precup. Optiongan: Learning joint reward-policy options using generative adversarial inverse reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[24] Mohit Sharma, Arjun Sharma, Nicholas Rhinehart, and Kris M Kitani. Directed-info gail: Learning hierarchical policies from unsegmented demonstrations using directed information. In *International Conference on Learning Representations*, 2018.

[25] Mingxuan Jing, Wenbing Huang, Fuchun Sun, Xiaojian Ma, Tao Kong, Chuang Gan, and Lei Li. Adversarial option-aware hierarchical imitation learning. *arXiv preprint arXiv:2106.05530*, 2021.

[26] Felipe Codevilla, Matthias Müller, Antonio López, Vladlen Koltun, and Alexey Dosovitskiy. End-to-end driving via conditional imitation learning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4693–4700. IEEE, 2018.

[27] Tim Hertweck, Martin Riedmiller, Michael Bloesch, Jost Tobias Springenberg, Noah Siegel, Markus Wulfmeier, Roland Hafner, and Nicolas Heess. Simple sensor intentions for exploration. *arXiv preprint arXiv:2005.07541*, 2020.

[28] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *ICML*, 2018.

[29] Serkan Cabi, Sergio Gómez Colmenarejo, Matthew W Hoffman, Misha Denil, Ziyu Wang, and Nando Freitas. The intentional unintentional agent: Learning to solve many continuous control tasks simultaneously. In *Conference on Robot Learning*, pages 207–216. PMLR, 2017.

[30] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

[31] J. Edward Colgate, J. Edward, Michael A. Peshkin, and Witaya Wannasuphoprasit. Cobots: Robots For Collaboration With Human Operators, 1996.

[32] Thomas Kipf, Yujia Li, Hanjun Dai, Vinicius Zambaldi, Alvaro Sanchez-Gonzalez, Edward Grefenstette, Pushmeet Kohli, and Peter Battaglia. Compile: Compositional imitation learning and execution. In *International Conference on Machine Learning (ICML)*, 2019.

[33] Youngwoon Lee, Shao-Hua Sun, Sriram Somasundaram, Edward S. Hu, and Joseph J. Lim. Composing complex skills by learning transition policies. In *Proceedings of International Conference on Learning Representations*, 2019.

[34] Saurabh Arora and Prashant Doshi. A Survey of Inverse Reinforcement Learning: Challenges, Methods and Progress. *arXiv:1806.06877 [cs, stat]*, June 2018.

[35] Bryan Chan. Rl sandbox. `https://github.com/chanb/rl_sandbox_public`, 2020.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[37] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans. *arXiv preprint arXiv:1704.00028*, 2017.

[38] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[39] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018.

[40] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[41] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[42] Tuomas Haarnoja, Aurick Zhou, Kristian Hartikainen, George Tucker, Sehoon Ha, Jie Tan, Vikash Kumar, Henry Zhu, Abhishek Gupta, Pieter Abbeel, et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

[43] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. 2016.

[44] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] We specifically state our contributions in Section 1.
   (b) Did you describe the limitations of your work? [Yes] In Section 5, we describe several specific ways in which our method falls short of our baselines, or when things we tried didn't work out.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See our broader impact statement following our conclusion.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See our supplementary material. We had to limit how much we could include due to size restrictions, so we only included a representative set of the data.

(b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Short details are in Section 5.1, with expanded details included in the supplementary material.

(c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] We showed standard deviation across five seeds in all of our plots.

(d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See our supplementary material.

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...

(a) If your work uses existing assets, did you cite the creators? [N/A] We used our own environment and data for this work, though we do cite the creator of the underlying simulator (PyBullet).

(b) Did you mention the license of the assets? [No] Since PyBullet is well-known in the community as being free and open source, we do not mention that it has a permissive zlib license.

(c) Did you include any new assets either in the supplemental material or as a URL? [Yes] We include code for producing our environment, as well as the data and expert models for a single task. Due to space restrictions, we could not include expert data/models for all four tasks.

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Learning from Guided Play Algorithm

The complete pseudo-code is given in Algorithm 1. Our implementation builds on RL Sandbox [35], an open-source PyTorch [36] implementation for RL algorithms. For learning the discriminators, we apply gradient penalty to regularize the discriminators [37], as done in DAC [17]. We optimize the intentions via the reparameterization trick [38]. As commonly done in deep RL algorithms, we use the Clipped Double Q-Learning trick [39] to mitigate overestimation bias [40] and use a target network to mitigate learning instability [41] when training the Q-functions. We also learn the temperature parameter $\alpha_{\mathcal{T}}$ separately for each task $\mathcal{T}$ (see Section 5 of [42] for more details on learning α). The hyperparameters are provided in Appendix F. Please see attached video for a short representative example of what LfGP looks like in practice.

Algorithm 1 Learning from Guided Play (LfGP)

Input: Expert replay buffers $\mathcal{B}^E_{\text{main}}, \mathcal{B}^E_1, \ldots, \mathcal{B}^E_K$, scheduler period ξ, sample batch size N
Parameters: Intentions $\pi_{\mathcal{T}}$ with corresponding Q-functions $Q_{\mathcal{T}}$ and discriminators $D_{\mathcal{T}}$, and scheduler π_S (e.g. with Q-table Q_S)

Initialize replay buffer $\mathcal{B}$
for $t = 1, \ldots,$ **do**
  # Interact with environment
  For every ξ steps, select intention $\pi_{\mathcal{T}}$ using π_S
  Select action a_t using $\pi_{\mathcal{T}}$
  Execute action a_t and observe next state s'_t
  Store transition $\langle s_t, a_t, s'_t \rangle$ in $\mathcal{B}$

  # Update discriminator $D_{\mathcal{T}'}$ for each task $\mathcal{T}'$
  Sample $\{(s_i, a_i)\}_{i=1}^N \sim \mathcal{B}$
  for each task $\mathcal{T}'$ **do**
    Sample $\{(s'_i, a'_i)\}_{i=1}^B \sim \mathcal{B}^E_k$
    Update $D_{\mathcal{T}'}$ following equation 3 using GAN + Gradient Penalty
  end for

  # Update intentions $\pi_{\mathcal{T}'}$ and Q-functions $Q_{\mathcal{T}'}$ for each task $\mathcal{T}'$
  Sample $\{(s_i, a_i)\}_{i=1}^N \sim \mathcal{B}$
  Compute reward $D_{\mathcal{T}'}(s_i, a_i)$ for each task $\mathcal{T}'$
  Update π and Q following equations 7 and 8

  # Update scheduler π_S if necessary
  if at the end of effective horizon **then**
    Compute main task return $G_{\mathcal{T}_{\text{main}}}$ using reward estimate from D_{main}
    Update π_S (e.g. update Q-table Q_S following equation 12 and recompute Boltzmann distribution)
  end if
end for

B Environment Details

A screenshot of our environment, simulated in PyBullet [43], is shown in Figure 6. We chose this environment because we desired tasks that a) have a large distribution of possible initial states, representative of manipulation tasks in the real world, b) have a shared observation/action space with several other tasks, allowing the use of auxiliary tasks and transfer learning, and c) require a reasonably long horizon and significant use of contact to solve. The environment contains a tray with sloped edges to keep the blocks within the reachable workspace of the end-effector, as well as a green and a blue block, each of which are 4 cm $\times$ 4 cm $\times$ 4 cm and set to a mass of 100 g. The dimensions of the lower part of the tray, before reaching the sloped edges, are 30 cm $\times$ 30 cm. The dimensions of the bring boundaries (shaded blue and green regions) are 8 cm $\times$ 8 cm, while the dimensions of the insertion slots, which are directly in the center of each shaded region, are 4.1 cm $\times$ 4.1 cm $\times$ 1 cm.

Table 2: The components used in our environment observations, common to all tasks. Grip finger position is a continuous value from 0 (closed) to 1 (open).

Component	Dim	Unit	Privileged?	Extra info
EE pos.	3	m	No	rel. to base
EE velocity	3	m/s	No	rel. to base
Grip finger pos.	6	[0, 1]	No	current, last 2
Block pos.	6	m	Yes	both blocks
Block rot.	8	quat	Yes	both blocks
Block trans vel.	6	m/s	Yes	rel. to base
Block rot vel.	6	rad/s	Yes	rel. to base
Block rel to EE	6	m	Yes	both blocks
Block rel to block	3	m	Yes	in base frame
Block rel to slot	6	m	Yes	both blocks
Force-torque	6	N,Nm	No	at wrist
Total	**59**			

The boundaries for end-effector movement, relative to the tool center point that is directly between the gripper fingers, are a 30 cm $\times$ 30 cm $\times$ 14.5 cm box, where the bottom boundary is low enough to allow the gripper to interact with objects, but not to collide with the bottom of the tray.

See Table 2 for a summary of our environment observations. In this work, we use privileged state information (e.g., block poses), but adapting our method to exclusively use image-based data is straightforward since we do not use hand-crafted reward functions as in [10].

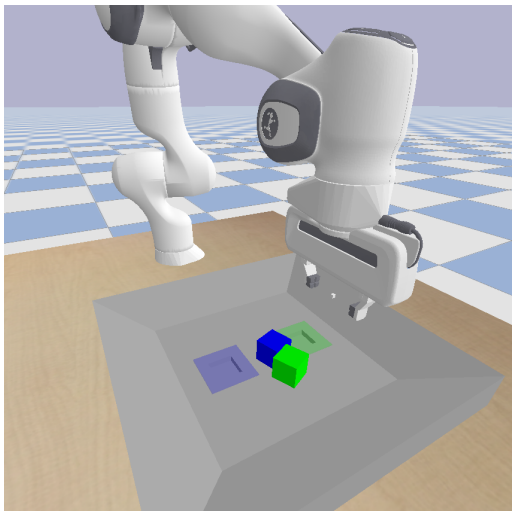

Figure 6: An image of our multitask environment immediately after a reset.

The environment movement actions are 3-DOF translational position changes, where the position change is relative to the current end-effector position, and we leverage PyBullet's built-in position-based inverse kinematics function to generate joint commands. Our actions also contain a fourth dimension for actuating the gripper. To allow for the use of policy models with exclusively continuous outputs, this dimension accepts any real number, with any value greater than 0 commanding the gripper to open, and any number lower than 0 commanding it to close. Actions are supplied at a rate of 20 Hz, and each training episode is limited to being 18 seconds long, corresponding to 360 time steps per episode. For play-based expert data collection, we also reset the environment manually every 360 time steps. Between episodes, block positions are randomized to any pose within the tray, and the end-effector is randomized to any position between 5 and 14.5 cm above the tray, within the earlier stated end-effector bounds, with the gripper fully opened. The only exception to these initial conditions is during expert data collection and agent training of the Unstack-Stack task: in this case, the green block is manually set to be on top of the blue block at the start of the episode.

C Procedure for Obtaining Experts

As stated, we used SAC-X [10] to train models that we used for generating expert data. We used the same hyperparameters as we used for LfGP (see Table 3), apart from the discriminator which, of course, does not exist in SAC-X. See Appendix D for details on the hand-crafted rewards that we used for training these models. For an example of gathering play-based expert data, please see our attached video.

We made two modifications to regular SAC-X to speed up learning. First, we pre-trained a Move-Object model before transferring it to each of our main tasks, as we did in Section 5.3 of our main paper, since we found that SAC-X would plateau when we tried to learn the more challenging tasks from scratch. The need for this modification demonstrates another noteworthy benefit of LfGP—when training LfGP, main tasks could be learned from scratch, and generally in fewer time steps, than it took to train our experts. Second, during the transfer to the main tasks, we used what we called a conditional weighted scheduler instead of a Q-Table: we defined weights for every combination of tasks, so that the scheduler would pick each task with probability $P(\mathcal{T}^{(h)}|\mathcal{T}^{(h-1)})$, ensuring that $\forall \mathcal{T}' \in \mathcal{T}_{\text{all}}, \sum_{\mathcal{T} \in \mathcal{T}_{\text{all}}} P(\mathcal{T}|\mathcal{T}') = 1$. The weights that we used were fairly consistent between main tasks, and can be found in our included code. The conditional weighed scheduler ensured that every task was still explored throughout the learning process, ensuring that we would have high-quality experts for every auxiliary task, in addition to the main task.

D Evaluation

As stated in our paper, we evaluated all algorithms by testing the mean output of the main-task policy head in our environment and generating a success rate based on 50 randomly selected resets. These evaluation episodes were all run for 360 time steps to match our training environment, and if a condition for success was met within that time, they were recorded as a success. See our included video for sample runs. The remaining section describes in detail how we evaluated success for each of our main and auxiliary tasks.

As previously stated, we also trained experts using modified SAC-X [10] that required us to define a set of reward functions for each task as well, which we also include in this section. The authors of [10] focused on sparse rewards, but also showed a few experiments in which dense rewards reduced the time to learn adequate policies, so we also used dense rewards. We would like to note that many of these reward functions are particularly complex and required significant manual shaping effort, further motivating the use of an imitation learning scheme like the one presented in this paper. It is possible that we could have gotten away with sparse rewards, such as those used in [10], but our compute resources made this impractical—for example, in [10], their agent took 5000 episodes $\times$ 36 actors $\times$ 360 time steps = 64.8 M time steps to learn their stacking task, which would have taken over a month of wall-time on our fastest machine. To see the specific values used for the rewards and success conditions described in these sections, see our included code.

Unless otherwise stated, each of the success conditions in this section had to be held for 10 time steps, or 0.5 seconds, before they registered as a success. This was to prevent registering a success when, for example, the blue block slipped off the green block during Stack.

D.1 Common

For each of these functions, we use the following common labels:

- p_b: blue block position,
- v_b: blue block velocity,
- a_b: blue block acceleration,
- p_g: green block position,
- p_e: end-effector tool center point position (TCP),
- p_s: center of a block pushed into one of the slots,
- g_1: (scalar) gripper finger 1 position,
- g_2: (scalar) gripper finger 2 position, and
- a_g: (scalar) gripper open/close action.

A block is flat on the tray when $p_{b,z} = 0$ or $p_{g,z} = 0$. To further reduce training time for SAC-X experts, all rewards were set to 0 if $\|p_b - p_e\| > 0.1$ and $\|p_g - p_e\| > 0.1$ (i.e., the TCP must be within 10 cm of either block). During training while using the Unstack-Stack variation of our environment, a penalty of -0.1 was added to each reward if $\|p_{g,z}\| > 0.001$ (i.e., there was a penalty to all rewards if the green block was not flat on the tray).

D.2 Stack/Unstack-Stack

The evaluation conditions for Stack and Unstack-Stack are identical, but in our Unstack-Stack experiments, the environment is manually set to have the green block start on top of the blue block.

D.2.1 Success

Using internal PyBullet commands, we check to see whether the blue block is in contact with the green block and is *not* in contact with both the tray and the gripper.

D.2.2 Reward

We include a term for checking the distance between the blue block and the spot above the the green block, a term for rewarding increasing distance between the block and the TCP once the block is stacked, a term for shaping lifting behaviour, a term for rewarding closing the gripper when the block is within a tight reaching tolerance, and a term for rewarding the opening the gripper once the block is stacked.

D.3 Bring/Insert

We use the same success and reward calculations for Bring and Insert, but for Bring the threshold for success is 3 cm, and for insert, it is 2.5 mm.

D.3.1 Success

We check that the distance between p_b and p_s is less than the defined threshold, that the blue block is touching the tray, and that the end-effector is *not* touching the block. For insert, the block can only be within 2.5 mm of the insertion target if it is correctly inserted.

D.3.2 Reward

We include a term for checking the distance between the p_b and p_s and a term for rewarding increasing distance between p_b and p_e once the blue block is brought/inserted.

D.4 Open-Gripper/Close-Gripper

We use the same success and reward calculations for Open-Gripper and Close-Gripper, apart from inverting the condition.

D.4.1 Success

For Open-Gripper and Close-Gripper, we check to see if $a_g < 0$ or $a_g > 0$ respectively.

D.4.2 Reward

We include a term for checking the action, as we do in the success condition, and also include a shaping term that discourages high magnitudes of the movement action.

D.5 Lift

D.5.1 Success

We check to see if $p_{b,z} > 0.06$.

D.5.2 Reward

We add a dense reward for checking the height of the block, but specifically also check that the gripper positions correspond to being closed around the block, so that the block does not simply get pushed up the edges of the tray. We also include a shaping term for encouraging the gripper to close when the block is reached.

D.6 Reach

D.6.1 Success

We check to see if $\|p_e - p_b\| < 0.015$.

D.6.2 Reward

We have a single dense term to check the distance between p_e and p_b.

D.7 Move-Object

For Move-Object, we changed the required holding time for success to 1 second, or 20 time steps.

D.7.1 Success

We check to see if the $v_b > 0.05$ and $a_b < 5$. The acceleration condition ensures that the arm has learned to move the block in smooth trajectories, rather than vigorously shaking it or continuosly picking up and dropping it.

D.7.2 Reward

We include a velocity term and an acceleration penalty, as in the success condition, but also include a dense bonus for lifting the block.

E Return Plots

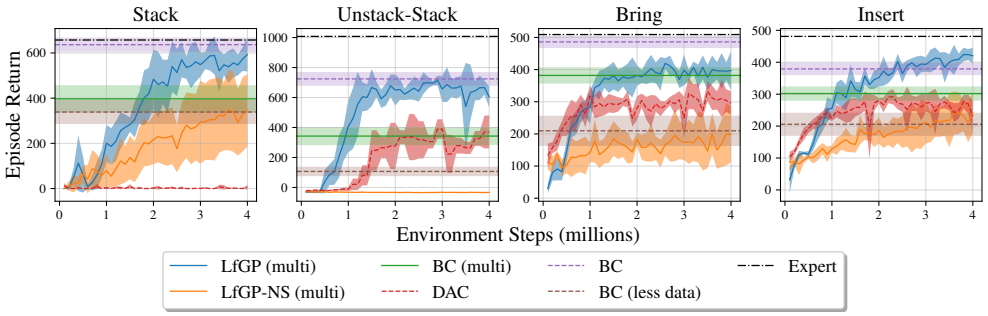

Figure 7: Episode return for LfGP compared with all baselines. Shaded area corresponds to standard deviation.

As previously stated, we generated hand-crafted reward functions for each of our tasks for the purpose of training our SAC-X experts. Given that we have these rewards, we can also generate return plots corresponding to our results to add extra insight. The episode return plots corresponding to our main task performance (Figure 3 of the main paper), multitask performance (Figure 4 of the main paper), transfer performance (Figure 5 of the main paper) and play-based expert data performance (Figure 6 of the main paper) are shown in Figure 7, Figure 8, and Figure 9 respectively. The patterns displayed in these plots are, for the most part, quite similar to the success rate plots. One notable exception was the fact that in Unstack-Stack, DAC performed far worse than LfGP as measured by return, as opposed to success rate—this can be explained by the fact that the DAC policies learned to unstack and restack the blue block continually, rather than letting the blue block rest on top of the green block (see included videos). As well, in the transfer experiments, it becomes clear that transferring from existing models did, in fact, have a notable increase in training speed for *all* tasks, which was not necessarily as evident from observing the success rate plots.

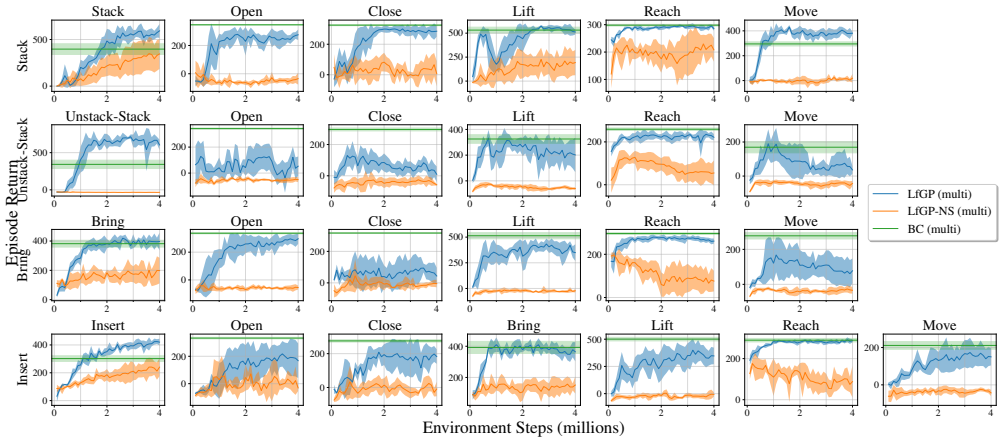

Figure 8: Episode return for LfGP compared with multitask baselines on all tasks. Shaded area corresponds to standard deviation.

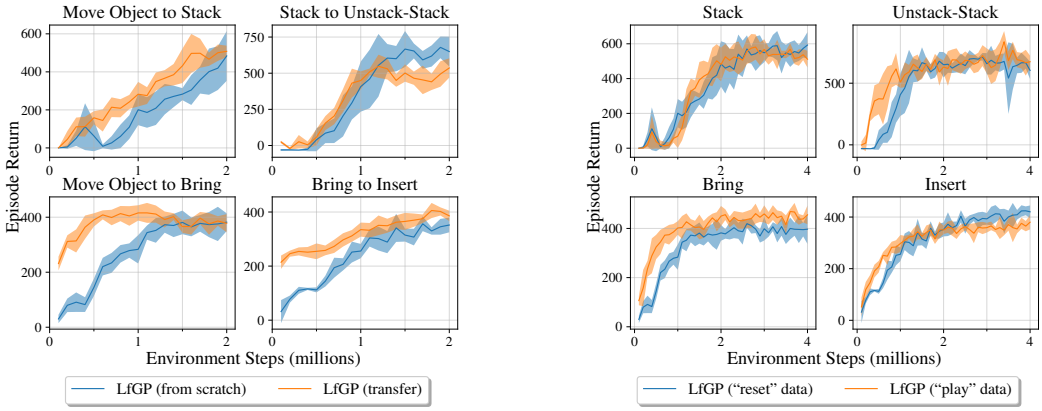

Figure 9: **Left:** Episode return for our transfer experiments. **Right:** Episode return for our play-based expert data experiments. Shaded area corresponds to standard deviation.

F Model Architectures and Hyperparameters

All the single-task models share the same network architectures and all the multitask models share the same network architectures. All layers are initialized using the PyTorch default methods [36].

For the single-task variant, the policy is a fully-connected network with two hidden layers followed by ReLU activation. Each hidden layer consists of 256 hidden units. The output of the policy is split into two vectors, mean $\hat{\mu}$ and variance $\hat{\sigma}^2$. The vectors are used to construct a Gaussian distribution (i.e. $N(\hat{\mu}, \hat{\sigma}^2\mathbf{I})$, where $\mathbf{I}$ is the identity matrix). When computing actions, we squash the samples using the tanh function, and bounding the actions to be in range $[-1, 1]$, as done in SAC [42]. The variance $\hat{\sigma}^2$ is computed by applying a softplus function followed by a sum with an epsilon $\epsilon =$ 1e-7 to prevent underflow: $\hat{\sigma}_i = \text{softplus}(\hat{x}_i) + \epsilon$. The Q-functions are fully-connected networks with two hidden layers followed by ReLU activation. Each hidden layer consists of 256 hidden units. The output of the Q-function is a scalar corresponding to the value estimate given the current state-action pair. Finally, The discriminator is a fully-connected network with two hidden layers followed by tanh activation. Each hidden layer consists of 256 hidden units. The output of the discriminator is a scalar corresponding to the logits to the sigmoid function. The sigmoid function can be viewed as the probability of the current state-action pair coming from the expert distribution.

For multitask variant, the policies and the Q-functions share their initial layers. There are two shared fully-connected layers followed by ReLU activation. Each layer consists of 256 hidden units. The output of the last shared layer is then fed into the policies and Q-functions. Each policy head and

Q-function head correspond to one task and have the same architecture: a two-layered fully-connected network followed by ReLU activations. The output of the policy head corresponds to the parameters of a Gaussian distribution, as described previously. Similarly, the output of the Q-function head corresponds to the value estimate. Finally, The discriminator is a fully-connected network with two hidden layers followed by tanh activation. Each hidden layer consists of 256 hidden units. The output of the discriminator is a vector, where the ith entry corresponds to the logit to the sigmoid function for task $\mathcal{T}_i$. The ith sigmoid function corresponds to the probability of the current state-action pair coming from the expert distribution in task $\mathcal{T}_i$.

The hyperparameters for our experiments are listed in Table 3 and Table 4. In BC, *overfit tolerance* refers to the number of full dataset training epochs without an improvement in validation error before we stop training. All models are optimized using Adam Optimizer [44] with PyTorch default values, unless specified otherwise.

Table 3: Hyperparameters for AIL algorithms across all tasks.

Algorithm	LfGP (Ours)	LfGP-NS	DAC
Total Interactions		4M	
Buffer Size		4M	
Buffer Warmup		1000	
Initial Exploration		1000	
Intention			
γ		0.99	
Batch Size		256	
Q Update Freq.		1	
Target Q Update Freq.		1	
π Update Freq.		1	
Polyak Averaging		0.005	
Q Learning Rate		3e-4	
π Learning Rate		1e-5	
α Learning Rate		3e-4	
Initial α		1	
Target Entropy		4	
Max. Gradient Norm		10	
Discriminator			
Learning Rate		3e-4	
Batch Size		256	
Gradient Penalty λ		10	
Scheduler			
Type	Q-table	Select $\mathcal{T}_{\text{main}}$	N/A
ξ	45	N/A	N/A
ϕ	0.6	N/A	N/A
Initial Temp.	360	N/A	N/A
Temp. Decay	0.9995	N/A	N/A
Min. Temp.	0.1	N/A	N/A

Table 4: Hyperparameters for BC algorithms across all tasks.

Algorithm	BC	BC (Less Data)	Multitask BC
Batch Size		256	
Learning Rate		3e-4	
Overfit Tolerance		100	

G Experimental Hardware

For a list of the software we used in this work, see our included code and instructions. We used a number of different computers for completing experiments:

1. GPU: NVidia Quadro RTX 8000, CPU: AMD - Ryzen 5950x 3.4 GHz 16-core 32-thread, RAM: 64GB, OS: Ubuntu 20.04.
2. GPU: NVidia V100 SXM2, CPU: Intel Gold 6148 Skylake @ 2.4 GHz (only used 4 threads), RAM: 32GB, OS: CentOS 7.
3. GPU: Nvidia GeForce RTX 2070, CPU: RYZEN Threadripper 2990WX, RAM: 32GB, OS: Ubuntu 20.04.

H Open-Action and Close-Action Distribution Matching

There was one exception to the "reset-based" method we used for collecting our expert data. Specifically, our Open-Gripper and Close-Gripper tasks required several additional considerations. It is worth reminding the reader that our Open-Gripper and Close-Gripper tasks were meant to simply open or close the gripper, respectively, while remaining reasonably close to either block. If we were to use the approach described above verbatim, the Open-Gripper and Close-Gripper data would contain no (s, a) pairs where the gripper actually released or grasped the block, instead immediately opening or closing the gripper and simply hovering near the blocks. Perhaps unsurprisingly, this was detrimental to our algorithm's performance: as one example, an agent attempting to learn Stack would, if Open-Gripper was selected while the blue block was held above the green block, move the currently grasped blue block *away* from the green block before dropping it on the tray. This behaviour, of course, is not what we would want, but it better matches an expert distribution collected using the method described above.

To mitigate this, our Open-Gripper data actually contain a mix of each of the other sub-tasks called first for 45 time steps, followed by a switch to Open-Gripper, ensuring that the expert dataset contains some degree of block-releasing, with the trade-off being that 25% of the Open-Gripper expert data is specific to whatever the main task is. We left this detail out of our main paper for clarity, since it corresponds to only 4-5% of the data ($2250/45000$ or $2250/54000$) that was claimed as being reusable being, in actuality, task-specific. Similarly, the Close-Gripper data calls Lift for 15 time steps before switching to Close-Gripper, ensuring that the Close-gripper dataset will contain a large proportion of data where the block is actually grasped. Given the simplicity of designing a reward function in these two cases, a natural question is whether Open-Gripper and Close-Gripper could use hand-crafted reward functions, or even hand-crafted policies, instead of these specialized datasets. In our experiments, both of these alternatives proved to be quite detrimental to our algorithm, so we leave further exploration of these options for future work.