Learning from Guided Play: Improving Exploration for Adversarial Imitation Learning with Simple Auxiliary Tasks

Trevor Ablett[1], Bryan Chan[2], and Jonathan Kelly[1]

Appendix I
Learning from Guided Play Algorithm

The complete pseudo-code is given in Algorithm Algorithm 1. Our implementation builds on RL Sandbox [1], an open-source PyTorch [2] implementation for RL algorithms. For learning the discriminators, we apply gradient penalty to regularize the discriminators [3], as done in DAC [4]. We optimize the intentions via the reparameterization trick [5]. As is commonly done in deep RL algorithms, we use the Clipped Double Q-Learning trick [6] to mitigate overestimation bias [7] and use a target network to mitigate learning instability [8] when training the Q-functions. We also learn the temperature parameter $\alpha_{\mathcal{T}}$ separately for each task $\mathcal{T}$ (see Section 5 of [9] for more details on learning α). For Generative Adversarial Imitation Learning (GAIL), we use a commonly used open-source PyTorch implementation [10]. The hyperparameters used for all methods are provided in Section VII. Please see videos at papers.starslab.ca/lfgp for examples of what LfGP looks like in practice.

A. Learning the Scheduler

As stated in our paper, our main experiments used a simple weighted random scheduler with handcrafted trajectories. In this section, we provide the details of our learned scheduler. Following [11], let H be the total number of possible intention switches within an episode and let each chosen intention execute for ξ timesteps. The H intention choices made within the episode are defined as $\mathcal{T}^{0:H-1} = \{\mathcal{T}^{(0)}, \ldots, \mathcal{T}^{(H-1)}\}$, where $\mathcal{T}^{(h)} \in \mathcal{T}_{\text{all}}$. The main task's return given chosen intentions is then defined as

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

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 h^{th} scheduler period. We further define the Q-function for the scheduler as $Q_S(\mathcal{T}^{0:h-1}, \mathcal{T}^{(h)}) = \mathbb{E}_{\mathcal{T}^{h:H-1} \sim P_S^{h:H-1}}\left[G_{\mathcal{T}_{\text{main}}}(\mathcal{T}^{h:H-1}) | \mathcal{T}^{0:h-1}\right]$ and represent the scheduler for the h^{th} period as a softmax distribution P_S^h over $\{Q_S(\mathcal{T}^{0:h-1}, \mathcal{T}_{\text{main}}), Q_S(\mathcal{T}^{0:h-1}, \mathcal{T}_1), \ldots, Q_S(\mathcal{T}^{0:h-1}, \mathcal{T}_K)\}$.

The scheduler maximizes the expected return of the main task following the scheduler:

$$\mathcal{L}(S) = \mathbb{E}_{\mathcal{T}^{(0)} \sim P_S^0}\left[Q_S(\emptyset, \mathcal{T}^{(0)})\right]. \tag{2}$$

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

$$\begin{aligned} Q_S(\mathcal{T}^{0:h-1}, \mathcal{T}^{(h)}) = (1-\phi) Q_S(\mathcal{T}^{0:h-1}, \mathcal{T}^{(h)}) \\ + \phi G_{\mathcal{T}_{\text{main}}}(\mathcal{T}^{h:H}), \end{aligned} \tag{3}$$

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 starting at timestep $h\xi$.

Appendix II
Environment Details

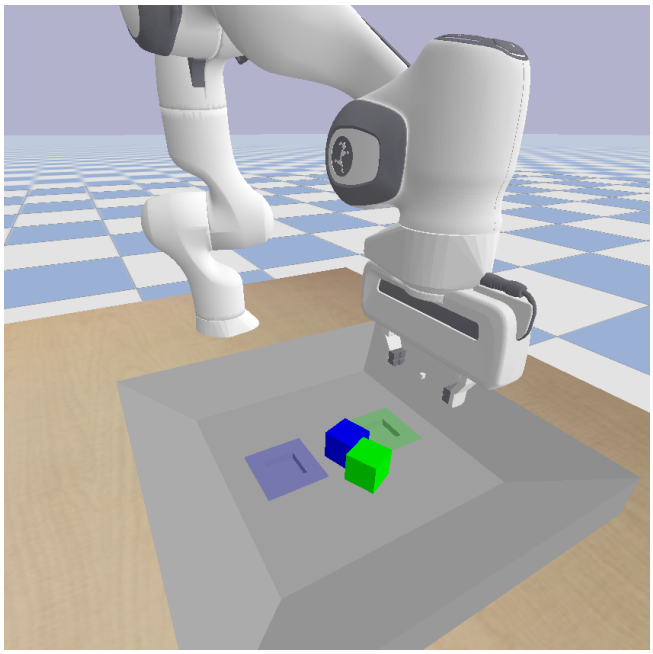

Fig. 1: An image of our multitask environment immediately after a reset.

A screenshot of our environment, simulated in PyBullet [12], is shown in Fig. 1. 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

[1]Space & Terrestrial Autonomous Robotic Systems (STARS) Laboratory at the University of Toronto Institute for Aerospace Studies (UTIAS), Toronto, Ontario, Canada, M3H 5T6. Email: <first name>.<last name>@robotics.utias.utoronto.ca

[2]Department of Computing Science at the University of Alberta, Edmonton, Alberta, Canada, T6G 2E8. Email: bryan.chan@ualberta.ca

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

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. 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 I 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 [11].

TABLE I: 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 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.

Appendix III
Performance Results for Auxiliary Tasks

The performance results for all multitask methods and all auxiliary tasks are shown in Figure Fig. 2. Multitask BC has gradually decreasing performance on many of the auxiliary tasks as the number of updates increases, which is consistent with mild overfitting. Intriguingly, however, multitask BC does get quite reasonable performance on many of the auxiliary tasks (such as Lift) without needing any of the extra environment interactions required by an online method like LfGP or DAC. An interesting direction for future work is whether pretraining via multitask BC could provide any improvements in environment sample efficiency. We did attempt to do this, but found that it resulted in poorer final performance than training from scratch.

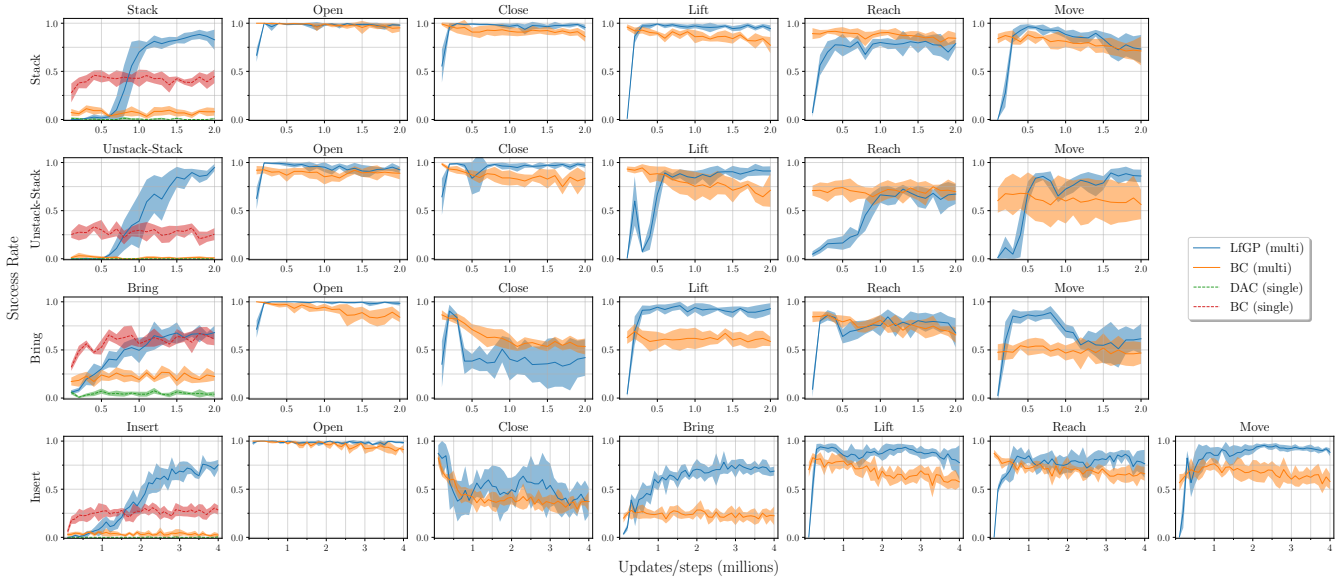

Fig. 2: Performance for LfGP and the multitask baselines across all tasks, shaded area corresponds to standard deviation.

Appendix IV
Procedure for Obtaining Experts

As stated, we used SAC-X [11] to train models that we used for generating expert data. We used the same hyper-parameters as we used for LfGP (see Table II), apart from the discriminator which, of course, does not exist in SAC-X. See Section V 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 weighted 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. This scheduler can be considered a more complex alternative to the weighted random scheduler or the addition with hand-crafted trajectories from our main paper, and again shows the flexibility of using a semantically meaningful multitask policy with a common observation and action space.

Appendix V
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. The rest of this section describes in detail how we evaluated success for each of our main and auxiliary tasks.

As previously stated, we trained experts using modified SAC-X [11] that required us to define a set of reward functions for each task, which we include in this section. The authors of [11] 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 [11], but our compute resources made this impractical—for example, in [11], 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.

A. 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).

B. 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.

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.

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.

C. 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.

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.

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. Open-Gripper/Close-Gripper

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

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

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.

E. Lift

1) Success: We check to see if $p_{b,z} > 0.06$.

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.

F. Reach

1) Success: We check to see if $\|p_e - p_b\| < 0.015$.

2) Reward: We have a single dense term to check the distance between p_e and p_b.

G. Move-Object

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

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.

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.

APPENDIX VI
RETURN PLOTS

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 (see Fig. 3 and Fig. 4) The patterns displayed in these plots are, for the most part, quite similar to the success rate plots. One notable exception is that there is an eventual increase in performance when training DAC on Insert, indicating that, perhaps for certain tasks, DAC alone can eventually make progress. Nevertheless, it is clear that LfGP improves learning efficiency, and it is unclear whether DAC would still plateau even if it was trained for longer in Insert.

APPENDIX VII
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 [2].

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 for LfGP and DAC is split into two

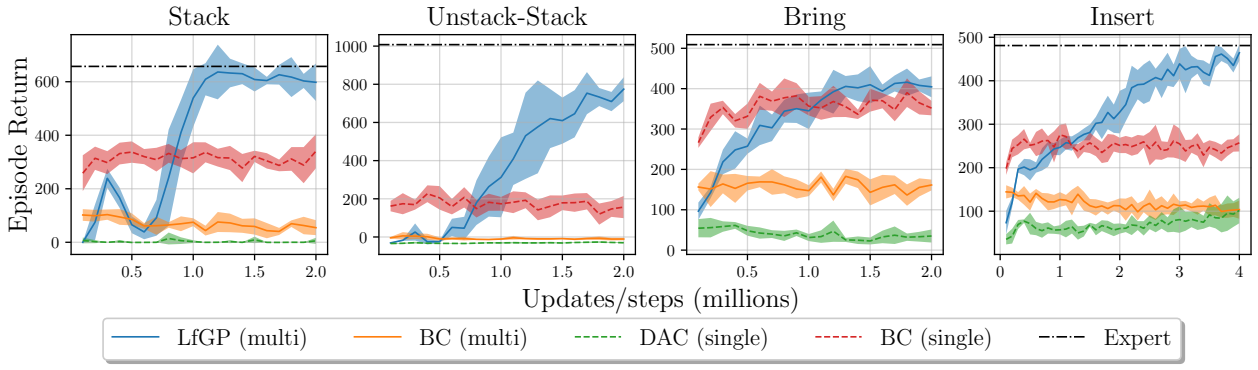

Fig. 3: Episode return for LfGP compared with all baselines. Shaded area corresponds to standard deviation.

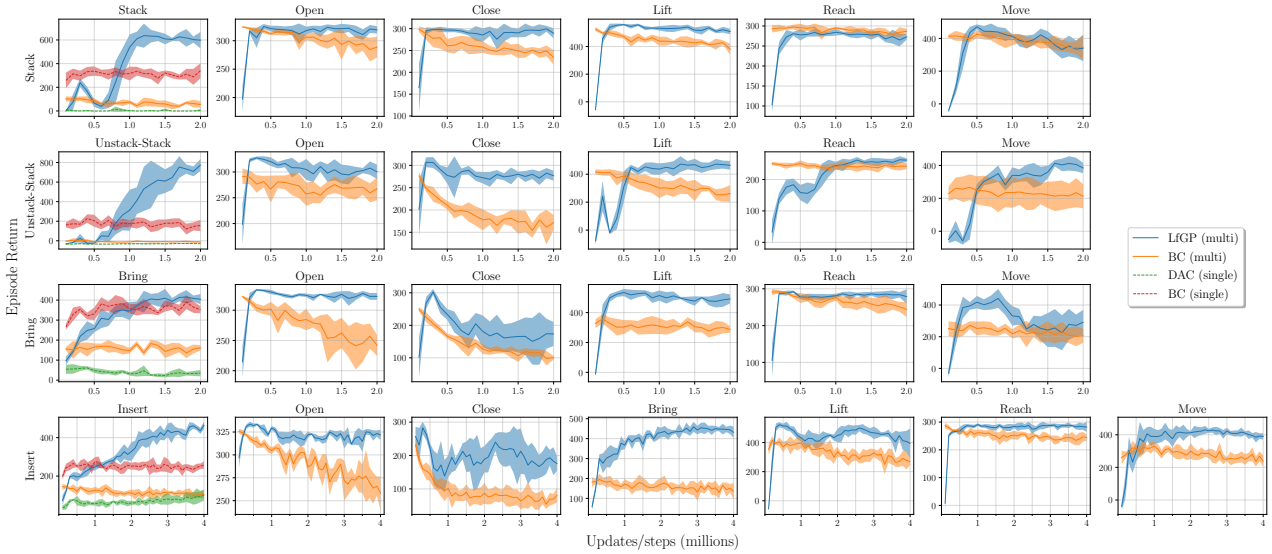

Fig. 4: Episode return for LfGP compared with multitask baselines on all tasks. Shaded area corresponds to standard deviation.

vectors, mean $\hat{\mu}$ and variance $\hat{\sigma}^2$. For both variants of BC, only the mean $\hat{\mu}$ output is used. 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 [9]. 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 i^{th} entry corresponds to the logit to the sigmoid function for task $\mathcal{T}_i$. The i^{th} 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 II and Table III. In the early stopping variant of 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 [13] with PyTorch default values, unless specified otherwise.

TABLE II: Hyperparameters for AIL algorithms across all tasks. Features that don't appear in the original version of DAC are shown in blue.

Algorithm	LfGP	DAC
Total Interactions	2M (4M for Insert)	
Buffer Size	2M (4M for Insert)	
Buffer Warmup	25k	
Initial Exploration	50k	
Evaluations per task	50	
Evaluation frequency	100k interactions	
Intention		
γ	0.99	
Batch Size	256	
Q Update Freq.	1	
Target Q Update Freq.	1	
π Update Freq.	1	
Polyak Averaging	1e-4	
Q Learning Rate	3e-4	
π Learning Rate	1e-5	
α Learning Rate	3e-4	
Initial α	1e-2	
Target Entropy	$-\dim(a) = -4$	
Max. Gradient Norm	10	
π Weight Decay	1e-2	
Q Weight Decay	1e-2	
$\mathcal{B}^E$ sampling proportion	0.1	
$\mathcal{B}^E$ sampling decay	0.99999	
Discriminator		
Learning Rate	3e-4	
Batch Size	256	
Gradient Penalty λ	10	
Weight Decay	1e-2	
$(s_T, \mathbf{0})$ sampling bias	0.95	

TABLE III: Hyperparameters for BC algorithms (both single-task and multitask) across all tasks.

Version	Main Results	Early Stopping
Batch Size	256	
Learning Rate	1e-5	
Weight Decay	1e-2	
Total Updates	2M (4M for Insert)	N/A
Overfit Tolerance	N/A	100

Appendix VIII
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.

Appendix IX
Open-Action and Close-Action Distribution Matching

There was one exception to the method we used for collecting our expert data. Specifically, our Open-Gripper and Close-Gripper tasks required 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 when the environment is reset in between each task execution.

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 50% 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 a small portion of the expert data (every other auxiliary task was fully reused). 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. For the Closer-gripper data, however, this modification still allowed data to be reused between main tasks.

Appendix X
Attempted and Failed Experiments

In this section, we provide a list of experiments and modifications that didn't help, in addition to the alternatives that did.

1) **Pretraining with BC:** We attempted to pretrain LfGP using multitask BC, and then transition to online learning with LfGP, but we found that this tended to produce significantly poorer final performance. Some existing work [14], [15] has investigated transitioning from BC to online RL, but achieving this consistently, especially with off-policy RL, continues to be an open research problem.
2) **Handcrafted Open-Gripper/Close-Gripper policies:** 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.

3) **Penalizing Q values:** In our early experiments, we found that LfGP training progress was harmed by exploding Q values. This problem was particularly exacerbated when we added $\mathcal{B}^E$ sampling to our Q and π updates. It appeared that this occurred because, at the beginning of training, the differences between discriminator outputs for expert data and non-expert data were so large that the bootstrap Q updates quickly jumped to unrealistic values. We attempted to use various forms of Q penalties to resolve this, akin to Conservative Q Learning (CQL) [16], but found that all of our modifications ultimately harmed final performance. Some of the things we tried, in addition to the CQL loss, were reducing γ (.95, .9), clipping Q losses to -5, +5, smooth L1 loss, huber loss, increased gradient penalty λ for D (50, 100), decreased reward scaling (.1), more discriminator updates per each π/Q update (10), and weight decay in D only (as is done in [17]). We ultimately resolved these exploding Q values by i) decreasing polyak averaging to a significantly lower value than is used in much other work (1e-4 as opposed to the SAC default of 5e-3), and ii) adding in weight decay (with a significantly higher value used than is used in other work) to π, Q, and D training (which was required to not overfit with the reduced polyak averaging value). Without the added weight decay, performance started to plateau and eventually decrease.
4) **Higher Update-to-Data (UTD) Ratio:** Recent work in RL has started increasing the UTD ratio (i.e. increase the number of policy/Q updates per environment interaction), with the goal of improving environment sample efficiency [18]. We were actually able to increase this from 1 to 2 and get a marginal improvement in environment sample efficiency, but this also nearly doubled the running time of our experiments, so we opted not to include this modification in our final results. Even higher values of the UTD ratio also caused our Q values to explode.

REFERENCES

[1] B. Chan, "RL sandbox," 2020.

[2] A. Paszke, *et al.*, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[3] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved Training of Wasserstein GANs," in *Proc. 31st Ann. Conf. Neural Information Processing Systems (NIPS'17)*, I. Guyon, *et al.*, Eds. Long Beach, USA: Curran Associates, Inc., Dec. 4–9 2017, pp. 5767–5777.

[4] I. Kostrikov, K. K. Agrawal, D. Dwibedi, S. Levine, and J. Tompson, "Discriminator-Actor-Critic: Addressing Sample Inefficiency and Reward Bias in Adversarial Imitation Learning," in *Proc. Int. Conf. Learning Representations (ICLR'19)*, New Orleans, USA, May 2019.

[5] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," *arXiv:1312.6114 [cs, stat]*, Dec. 2013.

[6] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing Function Approximation Error in Actor-Critic Methods," in *Proc. 35th Int. Conf. Machine Learning (ICML'18)*, Stockholm, Sweden, Jul. 10–15 2018, pp. 1582–1591.

[7] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," 2015.

[8] V. Mnih, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[9] T. Haarnoja, *et al.*, "Soft Actor-Critic Algorithms and Applications," *arXiv:1812.05905 [cs, stat]*, Jan. 2019.

[10] I. Kostrikov, "PyTorch Implementations of Reinforcement Learning Algorithms," 2018.

[11] M. Riedmiller, *et al.*, "Learning by Playing Solving Sparse Reward Tasks from Scratch," in *Proc. 35th Int. Conf. Machine Learning (ICML'18)*. Stockholm, Sweden: PMLR, July 2018, pp. 4344–4353.

[12] E. Coumans and Y. Bai, "PyBullet, a Python module for physics simulation for games, robotics and machine learning," 2016.

[13] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *Proc. Int. Conf. Learning Representations (ICLR'15)*, San Diego, USA, May 7–9 2015.

[14] A. Rajeswaran*, *et al.*, "Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations," in *Proc. Robotics: Science and Systems (RSS'18)*, Pittsburgh, USA, Jun. 26–30 2018.

[15] Y. Wu, M. Mozifian, and F. Shkurti, "Shaping Rewards for Reinforcement Learning with Imperfect Demonstrations using Generative Models," *arXiv:2011.01298 [cs]*, Nov. 2020.

[16] A. Kumar, A. Zhou, G. Tucker, and S. Levine, "Conservative Q-Learning for Offline Reinforcement Learning," *arXiv:2006.04779 [cs, stat]*, Aug. 2020.

[17] M. Orsini, *et al.*, "What Matters for Adversarial Imitation Learning?" in *Advances in Neural Information Processing Systems (NeurIPS'21)*, Virtual, June 2021.

[18] X. Chen, C. Wang, Z. Zhou, and K. Ross, "Randomized Ensembled Double Q-Learning: Learning Fast Without a Model," *arXiv:2101.05982 [cs]*, Mar. 2021.