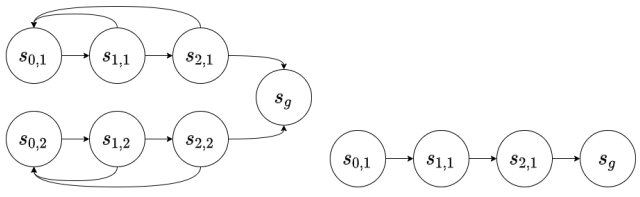
Learning from Guided Play: A Scheduled Hierarchical Approach for Improving Exploration in Adversarial Imitation Learning – Appendix

Anonymous Author(s)
Affiliation
Address
email

A Toy Example

In this section, we show an example to provide intuition for the exploration problem by exploiting the out-of-expert-distribution problem. Consider a sample MDP with seven states: $s_{0,1}, s_{1,1}, s_{2,1}, s_{0,2}, s_{1,2}, s_{2,2}, s_g$, where s_g is the goal state and $s_{0,n}$ are the initial states. The MDP has two actions: left and right (see Figure 1). Upon taking the *right* action, the agent transitions to the state on the right, and upon taking the *left* action, the agent transitions to the initial state $s_{0,n}$ depending on whether the agent is on the top or bottom half of the MDP. Assuming a uniformly random policy π, we derive the optimal discriminator from Equation 1 with on-policy data (i.e., expectation of first term is taken over π): $D(s_{m,1}, \text{right}) = \frac{2}{3}, D(s_{m,1}, \text{left}) \approx 0, D(s_{m,2}, \text{left or right}) \approx 0$. Using the common reward function $r(s,a) = -\log(1 - D(s,a))$, the rewards for state-action pairs not covered by the expert demonstration are 0, recovering a sparse reward function which provides no information for effective exploration towards the expert states. Using another reward formulation $r(s,a) = \log D(s,a)$, the rewards for the uncovered state-action pairs are the same negative values approaching $-\infty$. Although the reward function is now dense, it still does not provide any information for effective exploration of the bottom half of the MDP. In general, it requires taking the *right* action consecutively, with a probability of $\frac{1}{2^N}$ where N is the length of the chain, in order to reach the goal state (in which we obtain higher reward). This problem is exacerbated when the chains become longer or when there are more chains.



(a) MDP (b) Expert demonstration

Figure 1: a) An MDP with seven possible states, each with two possible actions – left and right. s_g is the goal state. b) Expert demonstration covering the top half of the chain.

B Learning from Guided Play Algorithm

The complete pseudo-code is given in 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 commonly done in deep RL algorithms, we use the

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

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

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). The hyperparameters are provided in Appendix G. Please see attached video for a short representative example of what LfGP looks like in practice.

C Environment Details

A screenshot of our environment, simulated in PyBullet [10], is shown in Figure 2. 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. 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.

Table 1: 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**			

See Table 1 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].

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.

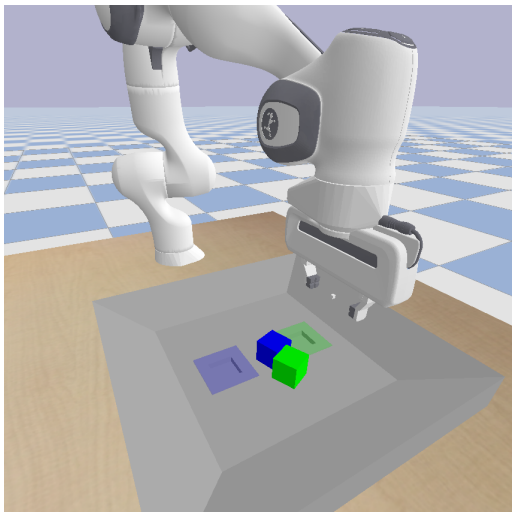


Figure 2: An image of our multitask environment immediately after a reset.

D 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 hyperparameters as we used for LfGP (see Table 2), apart from the discriminator which, of course, does not exist in SAC-X. See Appendix E 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.

E 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 [11] 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 [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.

E.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).

E.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.

E.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.

E.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.

E.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.

E.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.

E.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.

E.4 Open-Gripper/Close-Gripper

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

E.4.1 Success

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

E.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.

E.5 Lift

E.5.1 Success

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

E.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.

E.6 Reach

E.6.1 Success

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

E.6.2 Reward

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

E.7 Move-Object

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

E.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.

E.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.

F Return Plots

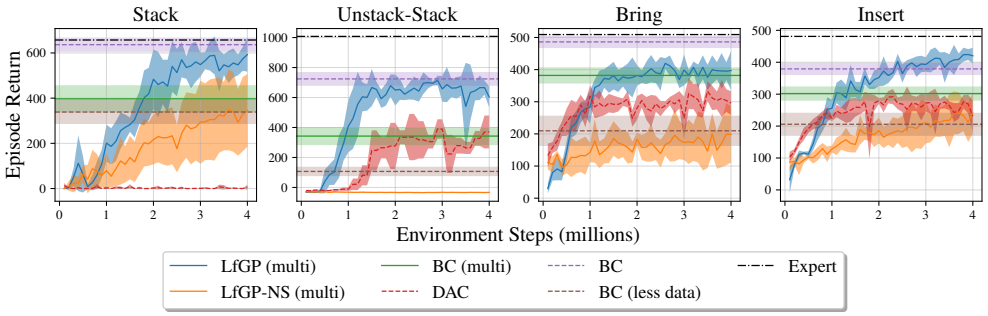


Figure 3: Episode return for LfGP compared with all baselines. Shaded area corresponds to standard deviation.

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 3, Figure 4, and Figure 5 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.

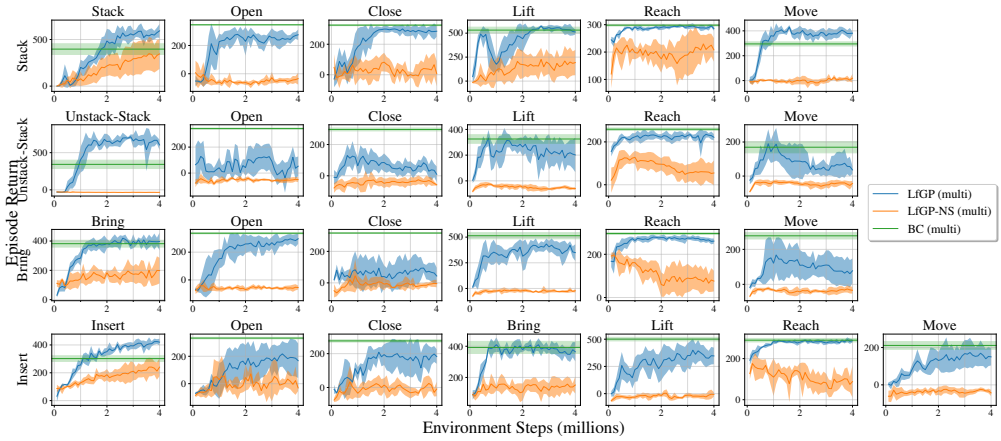


Figure 4: Episode return for LfGP compared with multitask baselines on all tasks. Shaded area corresponds to standard deviation.

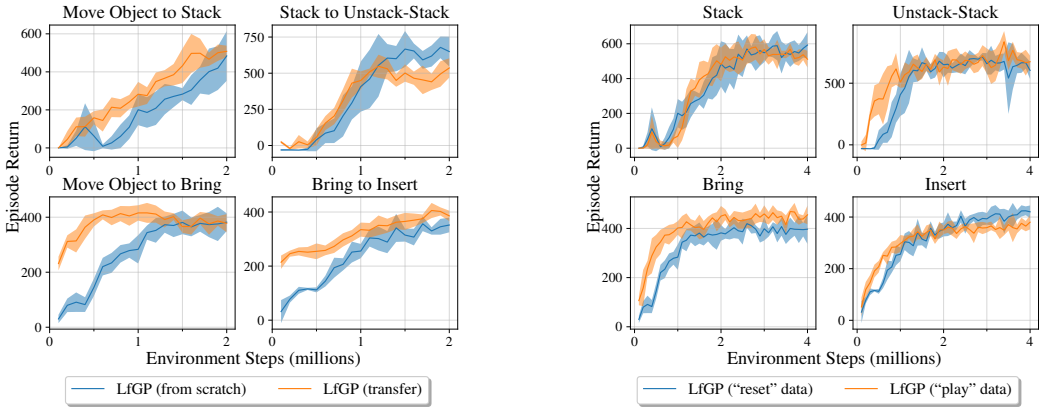


Figure 5: **Left:** Episode return for our transfer experiments. **Right:** Episode return for our play-based expert data experiments. Shaded area corresponds to standard deviation.

G 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 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 [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 2 and Table 3. 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 [12] with PyTorch default values, unless specified otherwise.

Table 2: 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 3: Hyperparameters for BC algorithms across all tasks.

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

H 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.

I 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.

References

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

[2] 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.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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