Guided Imitation of Task and Motion Planning

Michael J. McDonald
University of California, Berkeley
m_j_mcdonald@berkeley.edu

Dylan Hadfield-Menell
Massachusetts Institute of Technology
dhm@csail.mit.edu

Abstract: While modern policy optimization methods can do complex manipulation from sensory data, they struggle on problems with extended time horizons and multiple sub-goals. On the other hand, task and motion planning (TAMP) methods scale to long horizons but they are computationally expensive and need to precisely track world state. We propose a method that draws on the strength of both methods: we train a policy to imitate a TAMP solver's output. This produces a feed-forward policy that can accomplish multi-step tasks from sensory data. First, we build an asynchronous distributed TAMP solver that can produce supervision data fast enough for imitation learning. Then, we propose a hierarchical policy architecture that lets us use partially trained control policies to speed up the TAMP solver. In robotic manipulation tasks with 7-DoF joint control, the partially trained policies reduce the time needed for planning by a factor of up to 2.6. Among these tasks, we can learn a policy that solves the RoboSuite 4-object pick-place task 88% of the time from object pose observations and a policy that solves the RoboDesk 9-goal benchmark 79% of the time from RGB images (averaged across the 9 disparate tasks).

1 Introduction

This paper describes a policy learning approach that leverages task-and-motion planning (TAMP) to train robot manipulation policies for long-horizon tasks. Modern policy learning techniques can solve robotic control tasks from complex sensory input [1, 2, 3], but that success has largely been limited to short-horizon tasks. It remains an open problem to learn policies that execute long sequences of manipulation actions [4, 5]. By contrast, TAMP methods readily solve problems that require dozens of abstract actions and satisfy complex geometric constraints in high-dimensional configuration spaces [6, 7]. However, their application is often limited to controlled settings because TAMP methods need to robustly track world state and budget time for planning [8, 9].

To address these concerns, two lines of work have emerged. The first uses machine learning to identify policies or heuristics that reduce planning time [10, 11, 12, 13, 14, 15]. This extends the domains where TAMP can be applied, but still relies on explicit state estimation. The second line of work learns policies that imitate the output of TAMP solvers. These approaches reduce dependence on state estimation by, e.g., learning predictors to ground the logical state [16] or training recurrent models to predict the feasibility of a task plan [17]. These learned policies operate directly on perceptual data, and do not need hand-coded state estimation. However, TAMP solvers define a highly non-linear mapping from observations to controls and this makes imitation difficult. Furthermore, the compute resources required for planning limits the availability of training data for learning.

In this paper, we build on both approaches. We take inspiration from guided expert imitation [18, 19] and train feed-forward policies to imitate TAMP. Our key insight is that Srivastava et al. [20]'s modular TAMP framework supports a distributed TAMP architecture that: 1) is optimized for throughput (as opposed to low latency, the typical focus in TAMP research); 2) trains policies to imitate the planner output; and 3) uses those policies to amortize planning across problems and reduce compute costs. The result is a virtuous cycle where learning accelerates planning, and faster planning provides more supervision. We propose a policy architecture that leverages the definition of the TAMP domain to improve learning performance. Task plans from our system supervise a task-level model that learns to predict a parameterized action. This output selects from and parameterizes a set of control networks, one for each action schema. We show that this architecture can execute manipu-

5th Conference on Robot Learning (CoRL 2021), London, UK.

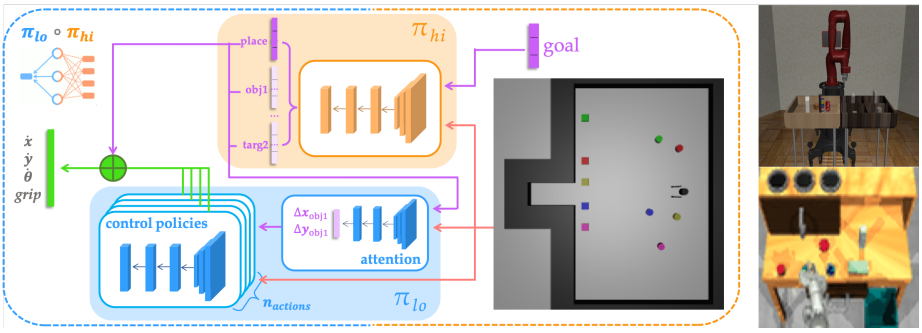

Figure 1: Left: We train hierarchical policies to imitate the output of a distributed task and motion planning system. The policy has two components, π_{hi} and π_{lo}. π_{hi} takes observations and a goal and produces a one-hot encoding of the choice of abstract action and associated parameters. These outputs and the original observation pass to the motion-level policy π_{lo}. π_{lo} consists of two stages: an attention module and action-specific control networks. The attention module maps the continuous observation and discrete action parameters to a continuous parameterization. There is one control network per action type and the choice of abstract action gates the outputs of these controllers to produce the next control. We denote this combined policy as $\pi_{lo} \circ \pi_{hi}$. Right: We evaluate in two simulated robotics domains: RoboSuite [21] (top) and RoboDesk [22] (bottom). In RoboSuite we train policies from object poses that reach 88% success on the four object variant of the domain. In RoboDesk, we train policies from RGB image data and reach 79% success on the 9-goal multitask problem.

lation skills in high-dimensional environments from complex sensory input. As a result, our system is able to compile task and motion planning into a single feed-forward policy.

The contributions of this work are as follows: 1) we show a design for a distributed, asynchronous, high-throughput task and motion planning system that leverages policy learning to speed up planning; 2) we propose a hierarchical policy architecture that leverages the TAMP problem specification; and 3) we implement and evaluate this method to show it can learn to accomplish multiple goals over (comparatively) long horizons with high-dimensional sensory data and action spaces. In a 2D pick-place task, we train policies that place 3 objects precisely onto targets 83% of the time from RGB images. In the RoboSuite [21] pick-place benchmark we train policies for 7-DoF joint control that place 4 objects 88% of the time from object pose vectors. In the multitask RoboDesk benchmark, our learned policy averages a 79% success rate across 9 diverse tasks with 7-DoF joint control from RGB images.

2 Background

Task and Motion Planning. Task and motion planning (TAMP) divides a robot control problem into two components: a symbolic representation of actions (e.g., $grasp$) and a geometric encoding of the world. Task planning operates on a logical representation of the world. It finds sequences of abstract actions to accomplish a goal (e.g., pick(obj_1), place(obj_1, $targ_1$), ...). Each action encodes a motion problem that must be *refined* (i.e., solved) to obtain a feasible trajectory that satisfies specified constraints.

We represent TAMP problems with the formalism introduced by Hadfield-Menell et al. [23]. A TAMP problem is a tuple $\langle X, F, G, U, f, x_0, A \rangle$. X is the space of valid world configurations. F is a set of *fluents*, binary functions of the world state that characterize the task space $f : X \to \{0, 1\}$ (e.g. at(obj_3, $targ_2$) or holding(obj_1)). G is the goal state, defined as a conjunction of fluents $\{g_i\}$. U is the control space of the robot. f describes the world dynamics: $f(x_t, u) = x_{t+1}$. x_0 describes the initial world configuration $x_0 \in X$. Finally, A is the set of abstract *action schemas*. Each action schema a has four components: 1) $a.params$: the parameters of the action (e.g., which object to grasp); 2) $a.pre$: a set of parameter-dependent fluents that defines the states when this action can be taken; 3) $a.mid$: a set of fluents that constrains the allowable controls for this action; and 4) $a.post$: a set of fluents that will be true after the action is executed. Solutions to such a problem are a pair of sequences $(\vec{a}, \vec{\tau})$, where $\vec{a}$ encodes abstract actions and $\vec{\tau}$ encodes refined motion trajectories. A plan is valid if 1) the initial state of each τ_i satisfies $a_i.pre$; 2) each τ_i satisfies $a_i.mid$; 3) the end state of each τ_i satisfies $a_i.post$; and 4) the final state satisfies the fluents that define the goal.

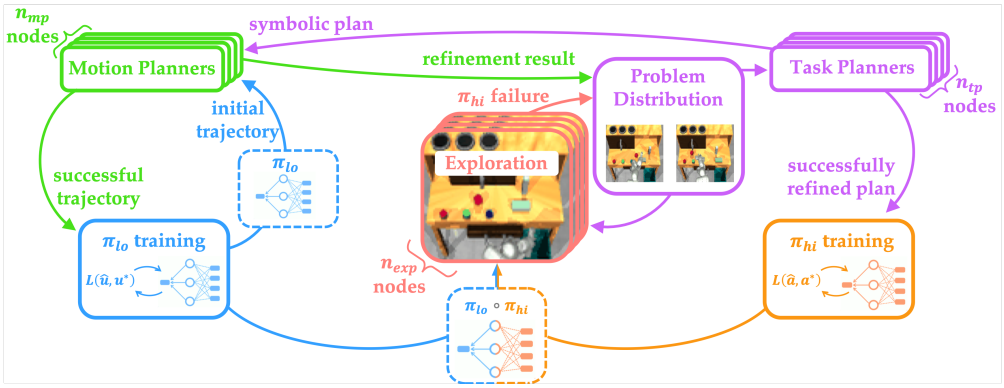

Figure 2: The system contains five core components which run in parallel and communicate through shared data structures. This division allows for the execution of arbitrary copies of motion planning, task planning, and policy rollouts and can scale to utilize all available hardware.

Modular TAMP. Our approach parallelizes the modular TAMP approach of Srivastava et al. [20]. They use a graph representation of TAMP, where nodes correspond to different task plans. A node can be either refined or expanded. Refinement uses motion planning to search for trajectories that accomplish the actions in the plan. Refinement failures are tracked along with the unsatisfiable constraints that caused the failure. When a node is expanded, one of these refinement failures is used to generate a new plan. The error information (e.g., a collision with an obstacle) is used to update the abstract domain so that the planner can identify an alternative plan (e.g., one that moves the obstruction out of the way). To solve a TAMP problem, Srivastava et al. [20] interleave refinement and expansion to find a plan that reaches the goal.

To implement plan refinement, we use the sequential convex optimization method described in Hadfield-Menell et al. [23]. This formulates motion planning as an optimization problem. The objective is a smoothing cost $\|\tau\|^2 = \sum_t \|\tau_{t+1} - \tau_t\|^2$. The constraints are determined by the pre, post, and mid conditions of an action schema: $\tau_0 \in a.pre$, $\tau_T \in a.post$, and $\tau \in a.mid$. E.g., for a grasp action the preconditions constrain the initial state to position the gripper near the object with proper orientation. The postconditions constrain the final state so that the object is in a valid grasp. The midconditions constrain the intermediate states to avoid collisions. If this optimization fails, we use the unsatisfied constraints to expand the associated plan node, as described above.

3 Method

This section has three parts. First, we propose a distributed TAMP solver. Then, we describe a hierarchical policy architecture that leverages the TAMP domain description to make it easier to model TAMP behavior. Finally, we show how to incorporate feedback from the learned policies to prevent trajectory drift.

3.1 Distributed Planning and Training Architecture

Figure 2 shows our overall design. Our system has four types of nodes that operate asynchronously: 1) policy training; 2) task planning; 3) motion planning; and 4) structured exploration. The task-level policy π_{hi} and the motion-level policy π_{lo} train within their respective nodes without backpropagation between the two. For this work we found standard supervised learning sufficient for good performance. However the modular design makes it straightforward to apply more complex procedures (e.g. generative adversarial imitation learning [24] or inverse reinforcement learning [25]). Specific network architectures and hyperparameters are included in the Supplemental Materials.

Algorithm 1 outlines our task planning procedure. It reads from a shared priority queue Q_{task} that tracks task planning problems, represented as tuples of an initial state x_0, a logical state Φ_0, a goal g, and a refined trajectory prefix τ_0. At random intervals, or when the queue is empty, we sample a new planning problem from the base problem distribution, with an empty trajectory prefix $\tau_0 = \emptyset$. A symbolic planner, such as FastFoward [26], computes a valid action sequence $\vec{a}$ that achieves g and pushes the problem (x_0, Φ_0, g, τ_0) and action sequence $\vec{a}$ to the motion queue Q_{motion}.

Algorithm 2 outlines our motion planning procedure. It pulls from Q_{motion} and applies the refinement procedure from section 2 to compute a valid trajectory segment τ^i for each a^i. If refinement fails, it computes the set of unsatisfiable constraints $\{e_i\}$ that describe why the motion planner could not refine a^i, appends these to the original logical description Φ_0, and pushes the updated problem instance $(x_0, \Phi_0 \cup \{e_i\}, g, \tau^{0:i-1})$ to Q_{task}. If refinement succeeds, each τ^i is pushed to the motion dataset D_{motion} to supervise the appropriate control policy. When the goal is reached (i.e., $\tau_T \in g$), it pushes the action sequence $\vec{a}$ to the task dataset D_{task} to supervise the task-level policy.

3.2 Hierarchical Task and Motion Policies

Learning to imitate TAMP solutions with a single policy is difficult because small changes in the observation can lead to different task plans and, thus, radically different controls. We deal with this through a hierarchical policy architecture that mirrors the TAMP domain description. The policy is split into two parts: a task-level policy π_{hi} to select parameterized actions a_t and a motion-level policy π_{lo} to select controls u_t as a function of a_t and world state. Figure 1 illustrates our design.

π_{hi} maps continuous observations and goals to the discrete space of parameterized action schemas. We use the factored encoding scheme from Van et al. [12]. π_{hi} outputs a sequence of vectors. The first specifies a one-hot encoding of the choice of action-type (e.g. place). Each subsequent vector provides a one-hot encoding of the ordered action parameters (e.g. $obj1$, $targ2$). Training data for this policy is pulled from the task dataset D_{task}.

The low-level policy π_{lo} consists of two stages: an attention module and action-specific control networks. The attention module maps the continuous observation and discrete action parameters to a continuous parameterization. This step is flexible. In our system, it represents the relevant objects in a particular geometric frame. E.g., for the action $place(obj1, targ2)$ this is the pose of $obj1$ in the $targ2$ frame. When the policy has access to state data, we hard code this step. In other situations (e.g., learning from camera images), this is learned from supervision alongside the control policy. The abstract action, the continuous parameterization, and the original observation pass to the second stage to predict the next control. This stage contains a set of separate control networks, one per action schema. Training data for this policy is pulled from the motion dataset D_{motion}.

Algorithm 1 Task Planning Node

Require: Shared queues Q_{task}, Q_{motion}
Require: Problem distribution P
while not terminated **do**
  if is_empty(Q_{task}) **then**
    $(x_0, \Phi, \text{g}, \tau) \sim P_{prob}$
  else
    $(x_0, \Phi, \text{g}, \tau) \leftarrow \text{pop}(Q_{task})$
  $\vec{a} \leftarrow$ task_plan(Φ, g)
  push(Q_{motion}, (x_0, Φ, g, $\vec{a}$))

Algorithm 2 Motion Planning Node

Require: Shared queues Q_{motion}, Q_{task}
Require: Expert datasets D_{motion}, D_{task}
Require: Motion policy π_{motion}
while not terminated **do**
  $(x, \Phi, \text{g}, \tau, \vec{a}) \leftarrow Q_{motion}$
  for $a_i \in \vec{a}$ **do**
    $\hat{\tau^i} \leftarrow \text{rollout}\big(x, \pi_{lo}(*|a_i)\big)$
    τ^i, success $\leftarrow$ motion_plan$\big(x, a_i, \hat{\tau^i}\big)$
    if success **then**
      $x \leftarrow \tau^i[-1]$
      append(τ, τ^i)
      push(D_{motion}, (a_i, τ^i))
    else
      ## add unsatisfiable constraints $\{e_i\}$
      push(Q_{task}, $(x, \Phi \cup \{e_i\}, \text{g}, \tau)$)
      break
  if $\tau[-1] \in$ g **then** push(D_{task}, $(\tau, \vec{a}$, g))

3.3 Policy-Aware Supervision

A common challenge in imitation learning is that small deviations from training supervision build up over time [27]. To account for this, we propose task and motion supervision methods based on Dataset Aggregation [28]. For controls, we bias trajectory optimization to provide supervision near states encountered by the motion policies. After an initial training period, we warm-start optimization with a rollout from the appropriate motion policy. Then, to keep supervision close to this trajectory, we modify the optimization objective with a *rollout deviation cost*, based on the acceleration kernel from Dragan et al. [29]. This uses a generalization of Dynamic Movement Primitives [30] that adapts the rollout trajectory to satisfy optimization constraints. For a sampled trajectory $\hat{\tau}^i$, the optimization cost is $\|\hat{\tau}^i - \tau^i\|^2 = \sum_t \|(\tau^i_{t+1} - \tau^i_t) - (\hat{\tau^i_{t+1}} - \hat{\tau^i_t}))\|^2$. This is conceptually similar to the regularization penalty from Guided Policy Search [18].

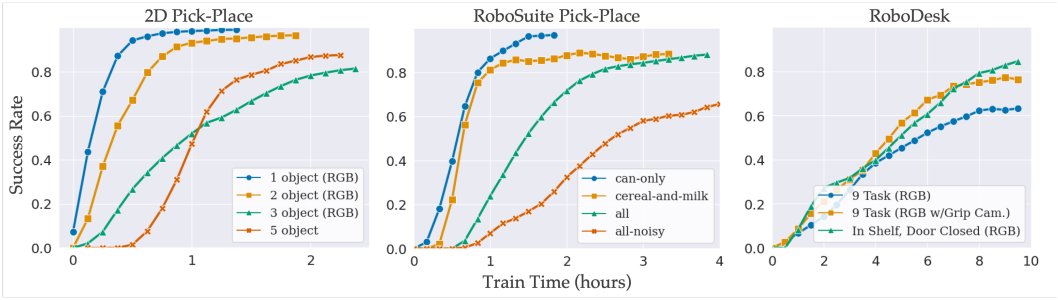

Figure 3: Average success rate over the course of training. Left: 2D pick-place domain. Training from RGB images and LIDAR-style sensors, the system learned to place 1, 2, and 3 objects 99%, 97%, and 83% of the time, respectively. Observing object positions, it learned to place 5 objects 88% of the time. Center: RoboSuite pick-place domain. Training from joint angles with object positions and orientations, the system learned to place 1, 2, and 4 objects 97%, 90%, and 88% of the time, respectively. With Gaussian observation noise it could place 4 objects 64% of the time. Right: RoboDesk domain. Training from joint angles and RGB images, the system learned to place the block in the shelf and close the door 89% of the time. Across the multitask 9-goal benchmark, the learned policy averaged a success rate of 68%, or 79% with an added gripper camera.

To generate task-level supervision, the exploration node identifies states where the task policy generates invalid actions. We sample a problem from the problem distribution and roll out the combined policy with an execution monitor that ensures that high-level action a is only started when $a.pre$ are satisfied and is executed until $a.post$ are satisfied. We use this to build up a dataset of negative examples for the task policy and to identify states where TAMP supervision may improve performance. Details and pseudo-code are in the Supplemental Materials.

4 Experimental Results

We evaluate our system in three domains: a 2D pick-place simulator, the RoboSuite benchmark [21], and the RoboDesk benchmark [22]. Screenshots of the simulation environments are shown in Figure 1. We used FastForward [26] as the task planner. Unless otherwise stated, evaluations used 2 processes for task planning, 18 for motion planning, and, when applicable, 10 for supervised exploration. Experiments ran for 5 random seeds unless otherwise noted. Policy architecture details and hyperparameters are in the Supplemental Materials.

4.1 Learning 2D Pick-Place

Setup. We begin with a simple, synthetic, pick-place domain. The robot is circular with a parallel jaw gripper and its goal is to transfer up to 5 objects each to one of 8 randomly assigned targets. The domain has three action schemas: moveto-and-grasp(obj), transfer$(obj, targ)$, place-and-retreat$(obj, targ)$. The full space includes a grounding of these for each object and target. With 5 objects and 8 targets, this gives 85 possible actions. The control space outputs x, y, and rotational velocities on the robot body and an open/close gripper signal. We used DMControl [31] for the underlying physics engine. The goal requires each object overlap with its target (i.e. $||\text{pos}_{obj} - \text{pos}_{targ}||_2 \leq \text{radius}_{obj}$). We use a simulated LIDAR sensor with 39 distance readings to facilitate collision avoidance.

Manipulated Variables and Dependent Measures. Our primary variable is the training procedure used. We compare against: 1) flat imitation learning (flat-IL), a single feedforward network trained to imitate TAMP output; 2) reinforcement learning with a dense reward function, trained with double the compute of our method (60 vCPUs); and 3) passive imitation learning (no-feedback), which uses the hierarchical architecture but turns off policy-aware supervision (20 vCPUs). We used two RL methods: PPO [32], a flat RL method, and HIRO [33], a hierarchical method. See the Supplemental Materials for training details. We used two performance measures: 1) success rate (Succ.), the rate at which trained policies achieve the goal; and 2) distance reduced (Dist), the percent reduction in the distance from each object to its target.

Results. Table 1 shows the results of these comparisons. With 1 object, we observed a 12% success rate for PPO and a 1% success rate for HIRO. Flat-IL succeeded 21% of the time in the 1 object variant. With two objects, it reliably reduced distance to the goal, but only succeeded 1.5% of the

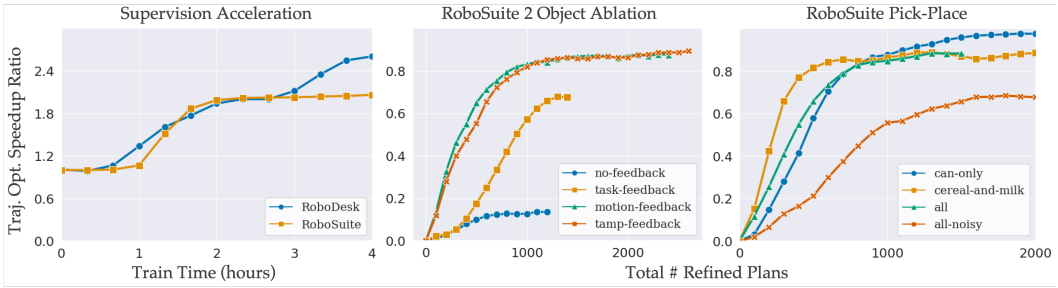

Figure 4: Our architecture utilizes partially trained motion policies to speedup plan refinement and improve data quality. Left: Speedup of plan refinement. During training, our method gave a 2x speedup of plan refinement in the RoboSuite domain with two objects and 2.6x speedup in the RoboDesk domain. Center: Ablation of performance as a function of supervision trajectories (Total # Refined Plans) for the RoboSuite domain with two objects. Right: Performance as a function of supervision trajectories across RoboSuite domains.

time. With 5 objects, we observed an 80% success rate for the hierarchical policy without feedback. This highlights the strong inductive bias our hierarchical architecture provides. With feedback, we observed success rates of 99.7%, 98.7%, and 88% for 1, 2, and 5 objects. Performance plateaued on the 5-object variant with approximately 3.6×10^5 environment transitions of supervision, generated by 1600 successful plans over 2 hours. See Figure 3 (left) for a depiction of performance over the course of training.

Camera Observations. We investigated the ability of our approach to train policies from RGB data. Figure 3 shows performance over time for solving problems with 1, 2, and 3 objects. After 4 hours of training (120 cpu hours) our policy successfully solves the 3 object task 83% of the time. In all cases, performance plateaued with less than 3.4×10^5 timesteps of supervision from the planner.

	1 obj 3HL / 39TS		2 obj 6HL / 85TS		5 obj 30HL / 223TS	
Method	Succ.	Dist.	Succ.	Dist.	Succ.	Dist.
PPO	12%	33%	0%	10%	n/a	n/a
HIRO	1%	13%	n/a	n/a	n/a	n/a
flat-IL	21%	55%	1.5%	51%	n/a	n/a
no-feedback	n/a	n/a	n/a	n/a	80%	97%
tamp-feedback	99.7%	99%	98.7%	99%	88%	97%

Table 1: Comparison of average success rate (Succ.) and distance towards goal (Dist.) at completion of training for different learning methods on state data in the 2D Pick-Place domain. Dist. denotes the percentage reduction in the distance from each object to its target. HL denotes the number of high-level actions the planner would use to solve the problem and TS denotes the number of environment transitions induced by those actions.

Generalization. We tested the learned policies' ability to generalize to novel goals and dynamics at test time. First, we introduced 6 additional obstacles to represent 'humans' in the environment. The policy observations were the same as before, so the robot had to use the LIDAR sensor to avoid the humans. At training time the humans stayed in fixed, random locations. At test time, the humans use model predictive control to take actions that move towards a random goal location. Their objective penalized collisions with other humans, but not the robot. Rollouts terminated if a human collided with the robot. We observed a 50% success rate placing 3 objects in this condition. Next, we moved the target locations for objects from training to test time. We arranged the targets in a square at the center of the free space at training time. At test time, we moved the target locations to the edges of the space. This placed all targets outside the region seen during training. We observed success rates of 99%, 97%, and 39% for problems with 1, 2, and 4 objects, respectively.

4.2 Learning Pick-Place for 7-DoF Robotic Arm Control

Setup. We measured performance in the RoboSuite pick-place benchmark with a simulated 7-DoF Sawyer arm [21]. The goal is to move a cereal box, milk carton, soda can, and loaf of bread from random initial locations in a bin on the left to their target bins on the right. The world state is the joint configuration and the positions of the objects. The action space specifies velocities for the 7 joints and a binary open/close signal for the gripper. The symbolic domain has 16 high-level actions across 4 action schemas: moveto(obj), grasp(obj), moveholding(obj), and putdown(obj). The policy observations were joint angles, gripper state, the translation and rotational displacements from the gripper to each object, and the displacement from each object to its target. Our main experiment had 4 conditions: 1) can-only, to compare with prior work; 2) cereal-and-milk, the most

difficult 2 object condition; 3) all, the standard benchmark goal; and 4) all-noisy, which added Gaussian observation noise to the policy observations with standard deviation equal to 1% of the possible deviation in initial values. We performed an ablation study in the cereal-and-milk condition to compare performance across feedback types: 1) no-feedback; 2) task-feedback (only); 3) motion-feedback (only); and 4) tamp-feedback (i.e., both feedback types).

Dependent Measures. To compare with published results in RL, we report the average dense return of the trained policies over 500-step horizons for can-only. Our primary measure of performance is success rate. We also measured the number of timesteps taken to reach the goal on successful trajectories. For the ablation study, we tracked the amount of time spent during motion optimization calls to measure the speedup from feedback.

Comparison to Prior Work. First, we compare against published results from RL and LfD. The official benchmark reports returns near 50 when using Soft-Actor Critic to move the can [21, 34]. In comparison, we observed an average return of 324 for this task. Fan et al. [35] report an average return below 600 over a 2000-step horizon. Our return for this horizon is at least 1734. We compare with LfD results on success rate. Mandlekar et al. [36] trained from the RoboTurk cans dataset [37] and observed success rates of 31% from state data and 43% from RGB camera images. In comparison, we observed a 98% success rate from state data for our can-only condition.

Results. Figure 3 (center) shows success rates over time for the four conditions described above. Our learned policies reach an 88% success rate in the all condition, 90% in the cereal-and-milk condition, and 98% in can-only condition. These correspond to average (successful) rollout lengths of 244, 120, and 55, respectively. In the all-noisy condition, we observed a success rate of 64%, compared with a 15% success rate executing plans from the TAMP solver directly. Figure 4 (center) shows success rate against training time for the ablation study. Both motion-feedback and tamp-feedback reach 80% success within 2 hours. Task-feedback shows steady improvement but does not reach the performance of the other variants within the time limit. (See the Supplemental Materials for a non-stationary condition where task-feedback improves on motion-feedback.) The no-feedback condition does not get above 20% success. We observe up to a 2x speedup of the motion optimization code with tamp-feedback compared to no-feedback. Figure 4 (left) illustrates the results.

Generalization. We ran a followup experiment to test the ability of the learned policies to generalize to unseen goals and dynamics. First, we held out goals that moved 3 and 4 objects and tested with other sizes of goals. When trained on problems that move 1 and 2 objects, the system was able to solve 3 object problems 48% of the time. When trained on 1, 2, and 4 object goals, it was able to solve 3 object problems 58% of the time. Next, we trained on 1, 2, and 3 object problems and observed a 60% success rate when tested with a goal to move all 4 objects. Note that this condition, which learns a multitask policy, is harder than the RoboSuite benchmark, which only attempts to learn a single-task policy.

Next, we used the domain randomization functionality of RoboSuite to test the ability of the previously trained policies to generalize to different dynamics. This varies properties of the physics engine such as the inertia and mass of the robot and objects, the parameters of the solver for contact forces, position and quaternion offsets of bodies, and physical properties of the joints. Using the default randomization parameters, we observed a success rate of 34%. When randomizing all parameters except the positional and quaternion parameters, we observed a 73% success rate.

4.3 Multitask Learning for 7-DoF Robotic Arm Control from RGB Images

Background. We completed the core of this work on May 19^{th}, with the above domains [38]. Our final experiment anecdotally measures the engineering effort required to adapt the system to a previously unseen environment. The RoboDesk multitask benchmark had been released 6 days prior, on May 13^{th} [22]. Our final results reflect one engineer's effort over the course of three weeks to train on the benchmark, as well as two further weeks to train on custom variants of the benchmark. The primary engineering steps were: 1) extension of existing action schemas to handle new tasks; 2) integration of the existing system with the new simulator; and 3) iteration on motion specification and ML hyperparameters.

Setup. RoboDesk provides a 7-DoF PANDA arm with 9 disparate tasks: open slide, press button, lift block, open drawer, block in bin, block off table, lift ball, stack blocks, and block in shelf. We added a composite goal: block in shelf with door closed. Our policy observations contain only RGB

camera images and joint positions. The controls are on joint velocities and the gripper. Our domain has 7 action schemas: moveto(obj), lift(obj), stack($obj1, obj2$), place($obj, targ$), press($button$), open($door$), close($door$). This gives 31 possible high-level actions.

Results. Figure 3 shows training results over 10 hours on an Nvidia DGX Station (40 vCPUs, 4 GPUs). On the composite task, the policy had a success rate of 89%. On the benchmark with the default forward-facing camera, the learned policy had a success rate of 68% averaged across the 9 tasks. That rose to 79% when we added a second camera attached to the robot gripper. We report the benchmark performance for each goal in the Supplemental Material. In this domain, we observed a 2.6x reduction in motion optimization time from feedback as shown in Figure 4 (left).

5 Related Work

Speeding up TAMP. There is a growing field of work that uses machine learning on previous experience to reduce planning time for TAMP systems. One focus is on guiding the task-level search — something we do not explicitly address in our system. Chitnis et al. [15] trained linear heuristics from expert demonstrations. Kim et al. [11] used a score-space representation to guide the search by transferring knowledge from previous plans. Wells et al. [10] learned a classifier to assess motion feasibility and incorporated it as a heuristic into the search. At the motion level, Ichnowski et al. [13] also learned policies to warm-start trajectory optimization. The key differences are that they predicted full trajectories, rather than executing the policy, and they did not penalize optimization for deviating from the policy controls. We note that our focus is on the learned policies themselves, and so we do not directly compare the speedup produced by our system to other methods.

Imitating TAMP. Several other methods distill all or part of TAMP into learned policies. Paxton et al. [14] used imitation learning to train policies in an abstract task space and deep Q-learning in a control space. Like our system, they learned feed-forward policies for both task and motion prediction and used those policies to guide supervision. The key difference is the type of planner used. Ours is more compute intensive, but scales to larger problems. Kase et al. [16] proposed a method to imitate hand-engineered trajectories to train a model to predict logical state and controllers to execute actions. Their method does not directly predict tasks, but allows a TAMP system to run online. In contrast, we train a single policy that maps observations directly to controls.

The approach most similar to ours is that of Driess et al. [17]. They also train hierarchical policies to match TAMP output. They show that their system is able to integrate geometric and high-level reasoning for a reaching task that, depending on the position of the target, potentially requires the use of tools. The primary difference is that their approach predicts the feasibility of a given action sequence, based on the initial state, while we predict actions and controls given the current state. This means that they need to search over candidate high-level action sequences at test time. In our system, this search is handled by the task planning node during training and bypassed entirely at test time. The other large difference is that they represent their control policies with learned energy functions that are minimized online. Task transitions occur when this system reaches an equilibrium state. In contrast, we learn policies that directly choose when to transition between tasks.

6 Future Work

First, this system should be deployed on physical robots. We hope that our policies will transfer more readily than those produced by, e.g., RL because TAMP solutions do not rely on detailed dynamics models. Next, it would be interesting to experiment with parameter sharing for the motion policies to save space and potentially speed up training. It would also be interesting to experiment with the output space of the high-level policy to directly output continuous parameters, which may allow policies to generalize more effectively. There is likely room to improve on both the task and motion policy learning: sequence modeling methods from NLP [39] may be better suited to represent task plans and modern imitation learning procedures, such as GAIL [24] or related works, may reduce training time or improve generalization. Note that each of these changes is relatively straightforward to implement, as it only modifies a single node of the training architecture. Finally, it would be interesting to identify theoretical guarantees on the learned policies. For example, it may be possible to formalize this as optimizing policy parameters with respect to a mixed discrete-continuous cost function defined by the TAMP domain.

7 Acknowledgements

We wish to express our deepest appreciation to Professor Anca Dragan for her support and advice throughout the course of this work, and for providing the resources needed to see it through to completion. We would also like to deeply thank Thanard Kurutach for providing his advice throughout the project. We are furthermore grateful to our anonymous reviewers from the 2021 Conference on Robot Learning, whose feedback led to significant improvements both to the text and experimental results of this paper. This work was supported in part through funding from the Center for Human-Compatible AI.

References

[1] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[2] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust region policy optimization, 2017.

[3] OpenAI, I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, J. Schneider, N. Tezak, J. Tworek, P. Welinder, L. Weng, Q. Yuan, W. Zaremba, and L. Zhang. Solving rubik's cube with a robot hand, 2019.

[4] T. Yu, P. Abbeel, S. Levine, and C. Finn. One-shot hierarchical imitation learning of compound visuomotor tasks. *ArXiv*, abs/1810.11043, 2018.

[5] A. Li, C. Florensa, I. Clavera, and P. Abbeel. Sub-policy adaptation for hierarchical reinforcement learning. *ArXiv*, abs/1906.05862, 2020.

[6] L. P. Kaelbling and T. Lozano-Pérez. Hierarchical task and motion planning in the now. In *2011 IEEE International Conference on Robotics and Automation*, pages 1470–1477, 2011.

[7] T. Ren, G. Chalvatzaki, and J. Peters. Extended task and motion planning of long-horizon robot manipulation. *ArXiv*, abs/2103.05456, 2021.

[8] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez. Integrated task and motion planning, 2020.

[9] G. Kazhoyan, S. Stelter, F. K. Kenfack, S. Koralewski, and M. Beetz. The robot household marathon experiment. *ArXiv*, abs/2011.09792, 2020.

[10] A. Wells, N. Dantam, A. Shrivastava, and L. Kavraki. Learning feasibility for task and motion planning in tabletop environments. *IEEE Robotics and Automation Letters*, PP:1–1, 01 2019.

[11] B. Kim, L. P. Kaelbling, and T. Lozano-Pérez. Learning to guide task and motion planning using score-space representation. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2810–2817, 2017.

[12] H. M. Van, O. S. Oguz, Z. Zhou, and M. Toussaint. Guided sequential manipulation planning using a hierarchical policy. RSS Workshop on Learning (in) Task and Motion Planning, 2020.

[13] J. Ichnowski, Y. Avigal, V. Satish, and K. Goldberg. Deep learning can accelerate grasp-optimized motion planning. *Science Robotics*, 5(48), 2020.

[14] C. Paxton, V. Raman, G. D. Hager, and M. Kobilarov. Combining neural networks and tree search for task and motion planning in challenging environments. *CoRR*, 2017.

[15] R. Chitnis, D. Hadfield-Menell, A. Gupta, S. Srivastava, E. Groshev, C. Lin, and P. Abbeel. Guided search for task and motion plans using learned heuristics. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 447–454. IEEE, 2016.

[16] K. Kase, C. Paxton, H. Mazhar, T. Ogata, and D. Fox. Transferable task execution from pixels through deep planning domain learning, 2020.

[17] D. Driess, J.-S. Ha, R. Tedrake, and M. Toussaint. Learning geometric reasoning and control for long-horizon tasks from visual input. 2021.

[18] S. Levine and V. Koltun. Guided policy search. In S. Dasgupta and D. McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1–9, Atlanta, Georgia, USA, 17–19 Jun 2013. PMLR.

[19] E. Groshev, A. Tamar, S. Srivastava, and P. Abbeel. Learning generalized reactive policies using deep neural networks. *CoRR*, abs/1708.07280, 2017.

[20] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel. Combined task and motion planning through an extensible planner-independent interface layer. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 639–646, 2014.

[21] Y. Zhu, J. Wong, A. Mandlekar, and R. Martín-Martín. robosuite: A modular simulation framework and benchmark for robot learning. 2020.

[22] H. Kannan, D. Hafner, C. Finn, and D. Erhan. Robodesk: A multi-task reinforcement learning benchmark. https://github.com/google-research/robodesk, 2021.

[23] D. Hadfield-Menell, C. Lin, R. Chitnis, S. Russell, and P. Abbeel. Sequential quadratic programming for task plan optimization. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5040–5047. IEEE, 2016.

[24] J. Ho and S. Ermon. Generative adversarial imitation learning. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016.

[25] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 1, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138385.

[26] J. Hoffmann. Ff: The fast-forward planning system. *AI Magazine*, 22:57–62, 09 2001.

[27] S. Ross and D. Bagnell. Efficient reductions for imitation learning. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 661–668. JMLR Workshop and Conference Proceedings, 2010.

[28] S. Ross, G. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In G. Gordon, D. Dunson, and M. Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 627–635, Fort Lauderdale, FL, USA, 11–13 Apr 2011. JMLR Workshop and Conference Proceedings.

[29] A. D. Dragan, K. Muelling, J. A. Bagnell, and S. S. Srinivasa. Movement primitives via optimization. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2339–2346. IEEE, 2015.

[30] S. Schaal. Dynamic movement primitives-a framework for motor control in humans and humanoid robotics. In *Adaptive motion of animals and machines*, pages 261–280. Springer, 2006.

[31] Y. Tassa, S. Tunyasuvunakool, A. Muldal, Y. Doron, S. Liu, S. Bohez, J. Merel, T. Erez, T. Lillicrap, and N. Heess. dm_control: Software and tasks for continuous control, 2020.

[32] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[33] O. Nachum, S. Gu, H. Lee, and S. Levine. Data-efficient hierarchical reinforcement learning. *CoRR*, abs/1805.08296, 2018. URL http://arxiv.org/abs/1805.08296.

[34] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.

[35] L. Fan, Y. Zhu, J. Zhu, Z. Liu, O. Zeng, A. Gupta, J. Creus-Costa, S. Savarese, and L. Fei-Fei. Surreal: Open-source reinforcement learning framework and robot manipulation benchmark. In *Conference on Robot Learning*, 2018.

[36] A. Mandlekar, F. Ramos, B. Boots, L. Fei-Fei, A. Garg, and D. Fox. Iris: Implicit reinforcement without interaction at scale for learning control from offline robot manipulation data. *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4414–4420.

[37] A. Mandlekar, Y. Zhu, A. Garg, J. Booher, M. Spero, A. Tung, J. Gao, J. Emmons, A. Gupta, E. Orbay, S. Savarese, and L. Fei-Fei. Roboturk: A crowdsourcing platform for robotic skill learning through imitation. In *Conference on Robot Learning*, 2018.

[38] M. McDonald. Accelerate then imitate: Learning from task and motion planning. Master's thesis, EECS Department, University of California, Berkeley, May 2021. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-96.html.

[39] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

Appendix

A. Structured Exploration

The final component of our system, the exploration node, identifies states where the task policy generates invalid actions. We adopt a procedure that enforces the structure of the action schemas A onto the policy rollouts. Note this structure is enforced only in training and not during evaluation of the policies.

At each timestep t, the task policy π_{hi} proposes a parameterized action $\hat{a}_t$. We then check the post-conditions of the last action a_{t-1} and the pre-conditions of $\hat{a}_t$. If both are satisfied, this is a valid high-level action sequence and we set $a_t = \hat{a}_t$. If the post-conditions of a_{t-1} are not met (and $\hat{a}_t \neq a_{t-1}$), we reject $\hat{a}_t$ and leave a_{t-1} unchanged. In this case, we generate a *negative* training example for the task policy: we populate a dataset D_{neg} and train the network to minimize the likelihood of those (invalid) high-level actions. We also generate negative examples when a_{t-1}'s post-conditions are satisfied and $\hat{a}_t = a_{t-1}$. This discourages actions that do not change the high-level state.

If the pre-conditions of $\hat{a}_t$ are not met, we need to select an alternative action for a_t. We do this by sampling from a softmax distribution over the raw logits of the network with temperature parameter λ. We continue sampling until 1) either a valid action is found or 2) we hit a termination condition. For each rejected candidate $\hat{a}_t$, we generate a negative training example.

If we rejected any candidate $\hat{a}_t$ at timestep t or terminate without having reached the goal, a new problem instance is added to the shared queue Q_{task}. The problem instance contains four values: a state vector x, a symbolic description Φ of the world in state x, the trajectory τ up to the timestep of x, and the current goal $goal$. If $\hat{a}_t$ was rejected due to pre-condition violations, we set x to the current state. If the rollout failed to reach the goal or $\hat{a}_t$ was rejected due to post-condition violations, we set x to the state of the most recent action transition. The assumption in this case is that π_{lo} has failed to execute properly, and the system should query from the last timestep when both π_{hi} and π_{lo} were trusted. Our full approach is outlined in algorithm 3.

Algorithm 3 Exploration Node

Require: Datasets D_{task}, D_{neg}
Require: Problem distribution P
Require: Shared queue Q_{task}
Require: Motion policy π_{lo}; Task policy π_{hi}
Require: Temperature λ; coefficient η
Require: Environment dynamics $f : X \times U \to X$

while not terminated **do**
  $x, \Phi, goal \sim P$
  $a \leftarrow \pi_{hi}(x, \lambda)$
  $x^{prev} \leftarrow x$
  $\tau \leftarrow \emptyset$
  $\vec{a} \leftarrow \emptyset$
  while not goal(x) and not timeout **do**
    $\hat{a} \leftarrow \pi_{hi}(x, \lambda)$
    if $\hat{a} \neq a$ and $a.post(x)$ **then**
      ## The previous action successfully completed
      $x^{prev} \leftarrow x$
      if not $\hat{a}.pre(x)$ **then**
        ## Get TAMP supervision from this state
        push$(Q_{task}, (x, \Phi, \tau, goal))$
      while not $\hat{a}.pre(x)$ and not max_iter **do**
        ## Penalize π_{hi} for invalid actions
        append$(D_{neg}, (\{x\}, \{\hat{a}\},$ goal$))$
        $\lambda \leftarrow \eta \cdot \lambda$
        $\hat{a} \leftarrow \pi_{hi}(x, \lambda)$
      if not $\hat{a}.pre(x)$ **then**
        ## No valid action found, reset
        break
      $a \leftarrow \hat{a}$
    else if $\hat{a} \neq a$ and not $a.post(x)$ **then**
      ## Get TAMP supervision in case a was a bad, but valid, selection
      push$(Q_{task}, (x^{prev}, \Phi, \tau, goal))$
      ## Penalize π_{hi} for a premature transition
      append$(D_{neg}, (\{x\}, \{\hat{a}\},$ goal$))$
    else if $\hat{a} = a$ and $a.post(x)$ **then**
      ## Penalize π_{hi} for a delayed transition
      append$(D_{neg}, (\{x\}, \{a\},$ goal$))$
      ## Get TAMP supervision
      push$(Q_{task}, (x, \Phi, \tau, goal))$
    $u \leftarrow \pi_{lo}(x|a)$
    $x \leftarrow f(x, u)$
    $\Phi \leftarrow$ update(x, Φ)
    $\vec{a}$.append(a); τ.append(x)

The benefit of these structured rollouts is two-fold. First, at each timestep t we can guarantee the sequence of abstract actions up to t forms a valid plan. This enables the system to isolate specific points of failure: either the high-level has provided a bad transition or the low-level has failed to properly execute. Second, we isolate configurations where the task policy must disagree with the task planner, without the cost of invoking the task planner. This enables efficient feedback from the task policy into the training.

B. Hyperparameters and Policy Training Details

In this section we provide specifics relating to policy training and the underlying neural networks.

Where appropriate, each network used the ReLU function between layers. All layer weights were initialized with Xavier initialization. Continuous network outputs (e.g. motion controls) were trained with the standard mean-squared error loss. Discrete network outputs (e.g. action parameterizations) were passed through a softmax activation function (modelling the outputs as a probability distribution over a discrete option set) and trained with the standard cross-entropy loss.

2D Pick-Place Experiments

For these experiments, a single control network was trained which took the one-hot encoding of the action schema as input. The control network consisted of two fully-connected hidden layers with 64 units each. The task network always contained 2 fully connected layers with 96 units each. Where applicable, training data was split evenly between trajectories generated from the base problem distribution and trajectories generated from problems sampled via exploration. Optimized trajectories were re-timed to a maximum velocity of 0.3 and two no-op (zero velocity) actions appended to task transitions. For variants with flat policies, the hyperparameters and network architecture of the control network were used.

Ground State Observations. The task policy was trained with a learning rate of 10^{-3} and l2 regularization with coefficient 4×10^{-4}. The motion policy was trained with a learning rate of 2×10^{-4} and l2 regularization with coefficient 10^{-4}.

The attention component was a hard-coded conversion that computed the displacement from the robot to the target object (for grasping actions) or from the grasped object to the target location (for transfer/place actions).

Camera Observations. The task policy was trained with a learning rate of 4×10^{-4} and l2 regularization with coefficient 10^{-5}. The motion policy was trained with a learning rate of 2×10^{-4} and l2 regularization with coefficient 10^{-5}.

For the task network, 80-by-80 RGB images fed through to a network containing two convolutional layers with 32 5-by-5 filters each. This then concatenated with the velocity information and passed to the fully connected layers. Between the convolutional and fully connected layers we applied a spatial softmax with feature point expectation as outlined by Levine et al. [1].

For the attention stage, the outputs were unchanged but a separately trained network replaced the hard-coded conversion. All inputs contained 80-by-80 RGB images. The network contained two convolutional layers with 32 5-by-5 filters each. This then concatenated with the LIDAR observations and one-hot task encoding and passed to the fully connected layers. Between the convolutional and fully connected layers we applied a spatial softmax with feature point expectation.

PPO Baseline. The primary hyperparameter we tuned was the environment update batch size (n_steps). We varied this value to use batch sizes from 32 to 4096. For our reported results, we settled on the default value of 128 provided by Hill et al. [32]. We also evaluated gamma values of 0.9, 0.95, and the default 0.99.

HIRO Baseline. We ran with the default TD3 hyperparameters. We varied the meta-period from 5 to 25. We tried both scaled and unscaled version of the negative distance exponential negative distance intrinsic rewards. Our reported results are with the default settings.

RoboSuite Pick-Place Experiments

For these experiments, four separate control networks were trained for each action schema. All networks in the task and motion policies consisted of two fully-connected hidden layers with 64 units each. Where applicable, training data was divided 48% into trajectories generated from the base problem distribution, 48% into trajectories generated from problems sampled via exploration, and 4% negative samples encountered via exploration. For variants with flat policies, the hyperparameters and network architecture of the control network were used.

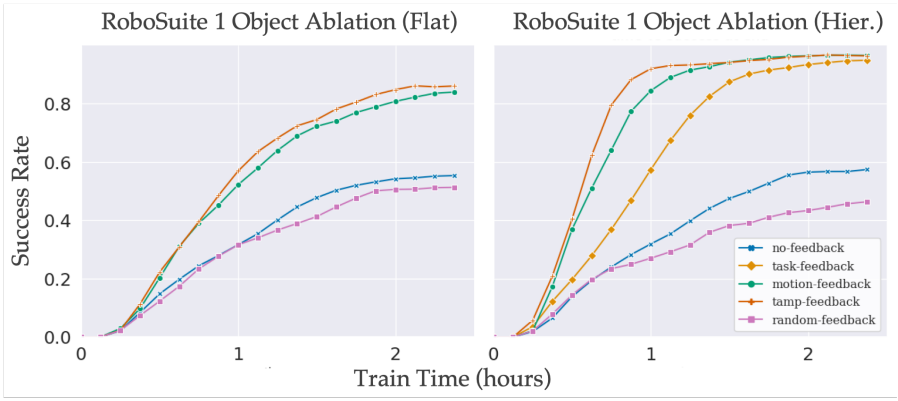

Figure 5: For the 1 object RoboSuite pick-place problem, we compare different feedback types in our system for training both flat policies (left) and our hierarchical policies (right). In both cases, motion-feedback provided the most significant performance gains, bringing flat policy performance from 52% to 88% and hierarchical policy performance from 58% to 97%. Because the task-feedback condition requires the separately trained task policy, for comparison we use an alternative (random-feedback) where randomly sampled states from failed policy rollouts are fed back to the problem distribution. This noticeably hurt performance for both types of policy, indicating how the choice of problem selection impacts overall performance.

The task policy was trained with a learning rate of 10^{-4} and l2 regularization with coefficient 10^{-4}. The motion policy was trained with a learning rate of 2×10^{-4} and l2 regularization with coefficient 10^{-5}.

The attention component was a hard-coded conversion that computed the displacement from the robot end-effector to a grasp point on the target object (for grasping related actions) or from the grasped object to the target location (for put-down related actions).

RoboDesk Experiments

For these experiments, seven separate control networks were trained for each action schema. Observations contained 64-by-64 RGB images. All control networks contained three convolutional layers: 32 7x7 filters, followed by 32 5x5 filters, followed by 16 5x5 filters. The primitive network contained two layers each with 32 5x5 filters. In each network, the final convolutional layer was followed by two fully connected layers with 64 units each. Between the convolution and fully connected layers was a spatial softmax with feature point expectation. The output of the convolutional layers was concatenated with the current joint angles and end-effector position before passing to the fully connected layers. For the control networks, the one-hot encoding of the action schemas and parameters were included in the concatenation. No separate attention module was used for these experiments.

Where applicable, training data was divided 48% into trajectories generated from the base problem distribution, 48% into trajectories generated from problems sampled via exploration, and 4% negative samples encountered via exploration.

The task policy was trained with a learning rate of 10^{-4} and l2 regularization with coefficient 10^{-5}. The motion policy was trained with a learning rate of 10^{-4} and l2 regularization with coefficient 10^{-6}.

C. Additional Results

7.1 RoboSuite Experiments

We provide additional results from our RoboSuite experiments. The first set, shown in Figure 5, provide a complete ablation for the 1 object problems. In this setting, the feedback from the motion policy to trajectory optimization (motion-feedback) provided the greatest benefit to both flat and hierarchical policy structures, bringing flat policy performance from 52% to 88% and hierarchical policy performance from 58% to 97%. Additionally, we add a random-feedback condition where

randomly sampled states from failed policy rollouts are fed back to the problem distribution. This is combined with motion-feedback to construct the tamp-feedback condition for flat policies, since task-feedback requires the separately trained task policy.

The use of random-feedback actually reduced performance for both the flat and hierarchical policies. This is likely due to increased time being spent attempting to solve problems the trajectory optimizer cannot refine (e.g. an object has been knocked into an unreachable position). These results support our claim that problem selection impacts overall performance of the system and highlights the benefit of the use of our structured problem sampling via the exploration node.

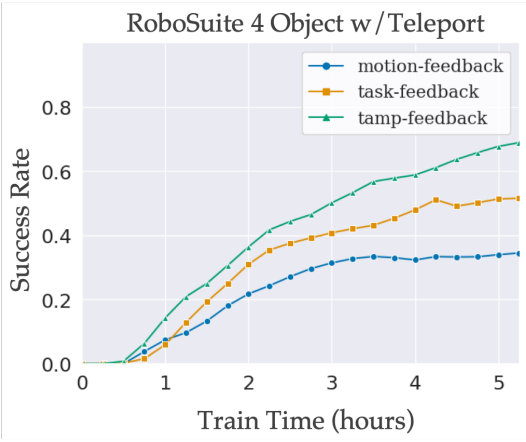

Figure 6: Comparison for the teleporting RoboSuite problems. With tamp-feedback, the policies learn to solve the problems 75% of the time. With just task-feedback, success plateaued at 54%. With just motion-feedback, success plateaued at 38%. This is an example where task-feedback provides noticeable benefit, as the policies frequently encounter states not captured in supervision from the base problem distribution.

We also show the training on the teleportation problems with 4 objects, for both the motion-feedback variant and the tamp-feedback variant. These results are in Figure 6. This is an example of a problem where supervision from the underlying problem distribution P is insufficient for generalization to the evaluation setting. Because teleportation was set not to occur during plan refinement, certain conditions (e.g. the can and cereal having been placed but the milk having reverted to the left) could be encountered by the learned policies that never appeared in supervision from P. These conditions however were sampled during exploration and fed back to the problem distribution, allowing for supervision in the new scenarios. This highlights that the primary benefit of task-feedback is to provide supervision in symbolic configuration that the policies may encounter but the planners, in isolation, would not.

7.2 RoboDesk Experiments

We have provided a breakdown of the learned policies' performance on the RoboDesk benchmark across the 9 different goals, which are shown in Table 2. These results are all from the same learned policies, each of which were trained on all 9 goals. Some tasks the policies learned easily, such as the open slide task (where the slide door is moved all the way to the right) or the press button task. Others, particularly the lift ball task, were quite difficult.

Goal	Base	W/Grip Cam.
Open Slide	100%	100%
Lift Bock	46%	65%
Push Button	93%	94%
Block off Table	59%	79%
Block in Shelf	52%	81%
Open Drawer	76%	76%
Block in Bin	84%	92%
Stack Blocks	68%	79%
Lift Ball	41%	48%

Table 2: Per-task policy performance for RoboDesk, as measured by average success rate for each task in the benchmark. Base refers to training from the default camera in the benchmark, while W/Grip cam/ adds a second camera to the gripper as an additional observation.

Lifting the ball provides an effective case study of where our method breaks down. There are two challenges here. The first is that the diameter of the ball equals the width of the fully opened gripper, making it easy to knock out of alignment. This reflects one major shortcoming of learning from TAMP: it is hard to model the world dynamics in the underlying optimization. This makes it difficult to generate supervision that explicitly takes advantage of those dynamics to solve the task. The second challenge is that the object is easily occluded by the arm, making it difficult to precisely place the gripper from the current image alone. This reflects a shortcoming of feed-forward policies in dealing with object permanence, In our system, we attempted to overcome this limitation by providing joint velocity information (roughly informing the policy of "where it was heading" prior to occlusion) and images from earlier timesteps. This highlights a scenario where sequence modelling would likely provide significant benefit.