# Guided Imitation of Task and Motion Planning

**Anonymous Author(s)**
Affiliation
Address
email

**Abstract:** While modern policy optimization methods can do complex manipulation from sensor 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 tries to get the best of both worlds: 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 let's us use partially trained control policies to speed up the TAMP solver. In robotic manipulation tasks, the partially trained policies reduce the time needed for planning by a factor of up to 2.5. We show results in a range of Pick-Place tasks, solving the 4 object Pick-Place task from RoboSuite 88% of the time from object pose observations and solving the RoboDesk 9-goal multitask benchmark 67% of the time from RGB images.

## 1 Introduction

In the past 5 years, (deep) *policy learning* has demonstrated an impressive ability to accomplish robotic control tasks from complex sensory input [1, 2, 3]. But that success has largely been limited to short-horizon tasks; learning policies that reason over extended sequences of actions with complex manipulation remains a challenge [4, 5]. By contrast, *task and motion planning* (TAMP) provides tools to solve problems that require dozens of abstract actions, each consisting of complex geometric constraints in high-dimension configuration spaces [6, 7]. The need to robustly track world state and budget time for planning constrains the application of these methods to controlled settings [8, 9].

To address these concerns, two lines of work have emerged. In the first, machine learning is used to learn domain specific policies or heuristics that reduce planning time [10, 11, 12, 13, 14, 15]. As a result, they make it easier to apply TAMP in (more) dynamic environments. However, they still need to be appropriately combined with explicit state tracking. The second line of work uses machine learning to train policies that imitate TAMP solvers. These approaches reduce dependence on state estimation by learning predictors to ground the logical state [16] or training recurrent models to predict full action sequences [17]. This work can run at test time with only perceptual data, increasing the range of applications. However, training is hampered by the inherent difficulty of TAMP and these approaches often rely on custom solvers and policy architectures.

In this paper, we take a step to integrate the two lines of work. 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 allows us to design 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, in turn, amortize planning across solutions and reduce compute costs. The result is a virtuous cycle where learning accelerates planning, and faster planning provides more supervision.

On its own, more data is not enough. We propose a policy architecture that leverages the TAMP hierarchy to improve performance. Task plans from our system supervise a task-level model that learns to predict the current action. This output is fed to the motion-level policy. It selects from a set of control networks, one for each action schema. We show this architecture can execute manipulation

Submitted to the 5th Conference on Robot Learning (CoRL 2021). Do not distribute.

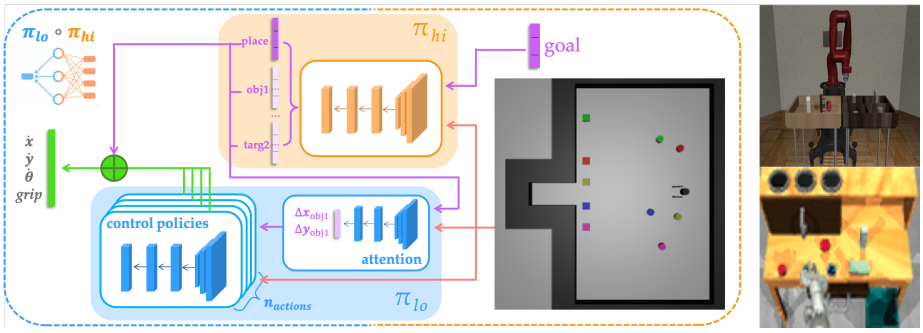


**Figure 1:** Left: We train hierarchical policies to imitate the output of a distributed task and motion planning system. The policy has two components, $\pi_{hi}$ and $\pi_{lo}$. $\pi_{hi}$ takes observations and a goal and produces a one-hot encoding of the choice of abstract action and associated parameters. These outputs and the original observation pass to the motion-level policy $\pi_{lo}$. It 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 based on object locations that reach 88% success on the four object variant of the domain. In RoboDesk, we train policies from RGB data and reach 67% success on the 9-goal multitask problem.

skills in dynamic, high-dimensional environments from complex sensory input. As a result, our system is able to compile task and motion planning into a set of reactive feed-forward policies.

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 leverage 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 that it can learn to accomplish multiple goals over (comparatively) long horizons with high-dimensional sensor data and action spaces. In a 2D Pick-Place task, we train policies that can place 3 objects precisely onto targets $83\%$ of the time from RGB images. In the RoboSuite [21] Pick-Place benchmark we train policies for joint control that can place all 4 objects $88\%$ of the time from object pose vectors. And in the RoboDesk [22] benchmark we reach 67% success on the 9-goal multitask problem from RGB images.

## 2 Background

**Task and Motion Planning.** Task and motion planning (TAMP) breaks the 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($obj_1$, $targ_1$), etc. Each action encodes a motion problem that must be refined to obtain an executable trajectory that satisfies specified constraints.

In this work, we use a slight modification of the formalism introduced in Chitnis et al. [15]. A task and motion planning 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 $f : X \rightarrow \{0, 1\}$, 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) each $a_i.pre$ are satisfied in the state $a_i$ is executed in; 2) each $\tau_i$ satisfies $a_i.mid$; 3) the end state of each $\tau_i$ satisfies $a_i.post$; and 4) the final end state satisfies the fluents that define the goal.

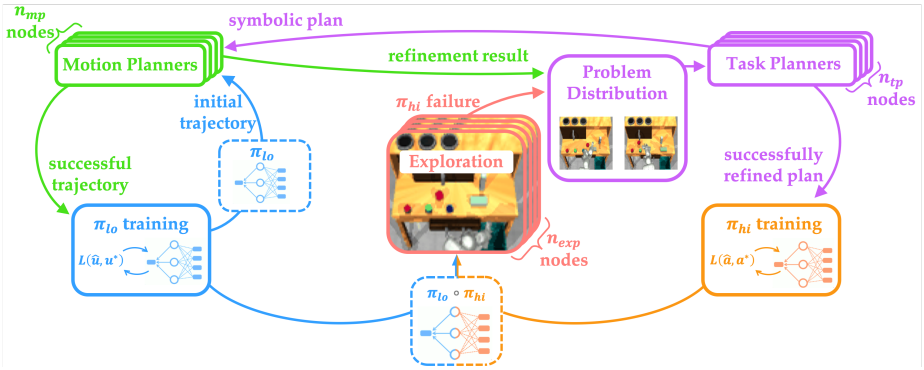


**Figure 2:** The system contains five core components which run in parallel and communicate through shared data structures. This division allows for the execution of arbitrary copies of motion planning, task planning, and policy rollouts and can scale to utilize all available hardware.

Our approach parallelizes the modular interface outlined in Srivastava et al. [20] and Chitnis et al. [15]. The approach relies on a *plan refinement graph.* In this graph, nodes correspond to task plans. They can be either refined or expanded. Refinement uses motion planning to search for trajectories that implement the actions in the plan. We track refinement failures and the unsatisfied constraints that caused 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 and the planner can identify an alternative solution. To solve a TAMP problem, [20, 15] search this graph, interleaving refinement and expansion, to find a plan that reaches the goal.

To implement plan refinement, we use the sequential convex optimization method described in [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: $\tau_0 \subseteq a.pre$, $\tau_T \subseteq a.post$, and $\tau \subseteq a.mid$. E.g., for a grasp action the preconditions would constrain the initial state to position the gripper near the object with proper orientation. The postconditions would constrain the final state so that the object is in a valid grasp. The midconditions would constrain the intermediate states to avoid collisions.

**Active Imitation Learning.** In addition to TAMP, our work builds on active imitation learning, which uses interaction with the expert to improve learning [24]. A common challenge in imitation learning is that small deviations from training supervision build up over time. E.g., Ross and Bagnell [25] showed that the expected cost of passive imitation learning under the 0-1 loss may scale quadratically with the time horizon, instead of the linear scaling one would observe in standard prediction settings. The core issue is one of distribution shift: the distribution of states encountered by a learned policy will generally differ from that of the expert. In long-horizon problems, this is especially challenging. Dataset Aggregation (DAgger) [26] is an elegant solution to this problem that mixes the learned and expert policies to align the train and test distributions. This teaches the policy to recover from failures and allows the method to recover a linear scaling of loss with respect to the time horizon. While direct DAgger is computationally impractical to couple with a TAMP solver, we adapt a similar approach to target supervision.

# 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. 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, (4) structured exploration. The task-level policy $\pi_{hi}$ and the motion-level policy $\pi_{lo}$ train within their respective nodes without backpropagation between the two. For this work we found standard supervised learning sufficient for good

performance, however the system's modularity makes it straightforward to apply more complex procedures (e.g. generative adversarial imitation learning [27] or inverse reinforcement learning [28]). Specific network architectures and hyperparameters are included in the Supplemental Materials.

Algorithm 1 outlines our procedure for task planning. This node reads from a shared priority queue $Q_{task}$ that tracks encountered problems. A problem is defined as a tuple containing an initial state $x_0$, an associated logical state $\Phi_0$, a goal $g$, and a refined trajectory prefix $\tau_0$. 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 Fast-Fowardčiteff, computes a valid action sequence $\vec{a}$ such that $g \subset \vec{a}(\Phi)$ and pushes both the problem $(x_0, \Phi_0, g, \tau_0)$ and the action sequence $\vec{a}$ to a separate priority queue $Q_{motion}$.

**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$(\Phi, g)$
  push($Q_{motion}$, $(x_0, \Phi, \text{g}, \vec{a})$)

Algorithm 2 outlines our procedure for motion planning. This node pulls the problems and associated action sequences $\vec{a}$ from $Q_{motion}$. It iteratively attempts to refine each action $a^i$ into a valid trajectory segment $\tau^i$, using the refinement procedure outlined in section 2.1. If refinement fails, it computes a set of the geometric facts $\{e_i\}$ that describe why the motion planner could not refine $a^i$ and appends these to the original logical description $\Phi_0$. An updated problem instance $(x_0, \Phi_0 \cup \{e_i\}, g, \tau^{0:i-1})$ is then pushed back to $Q_{task}$. If refinement succeeds, $\tau^i$ is pushed to the motion dataset $D_{motion}$ to supervise the appropriate control policy. When the goal is reached, $\tau_T \subset g$, the entire trajectory $\tau$ is pushed to the task dataset $D_{task}$ to supervise the task-level policy.

### 3.2 Hierarchical Task and Motion Policies

Learning to imitate TAMP solver output with flat policies is difficult. It is hard to distinguish the controls for different actions. Small changes in observations 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 $\pi_{hi}$ to select parameterized actions $a_t$ and a motion-level policy $\pi_{lo}$ to select controls $u_t$ as a function of $a_t$ and world state.

**Algorithm 2** Motion Planning Node

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

$\pi_{hi}$ maps continuous observations to the discrete space of parameterized action schemas. We use the factored encoding scheme from Van et al. [12]. This is shown in Figure 1. $\pi_{hi}$ 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 $\pi_{lo}$ is illustrated in Figure 1. It 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, 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 optimal control. This stage contains a set of separate control networks, one per action type. Training data for this policy is pulled from the motion dataset $D_{motion}$.

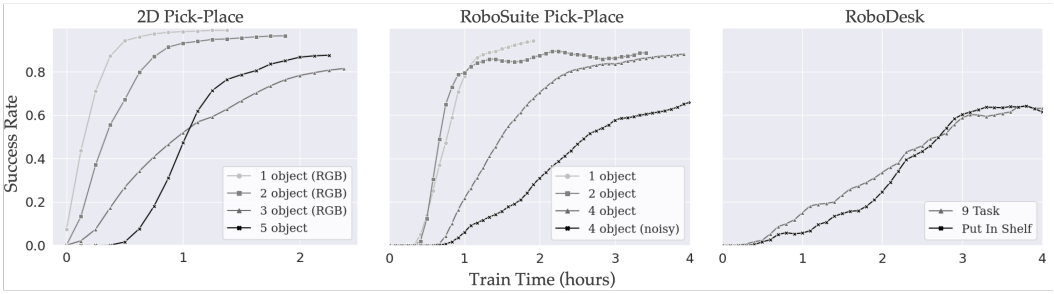


**Figure 3:** Average binary success rate over the course of training. Left: 2D pick-place domain. Training from RGB camera images and LIDAR-style sensors, the system learned to successfully place 1, 2, and 3 objects $99\%$, $97\%$, and $83\%$ of the time, respectively. Center: robosuite pick-place domain. Training from robot joint angles and object positions and orientations. The system learned to successfully place 1, 2, and 4 objects $97\%$, $90\%$, and $88\%$ of the time respectively. With Gaussian noise on the observations the system learned to successfully place 4 objects $64\%$ of the time. Right: RoboDesk domain. The system learned to place the block in the shelf and close the door $67\%$ of the time, and to complete a random selection from the 9 benchmark goals $67\%$ of the time. NB: Robodesk ran with 2 random seeds.

### 3.3 Policy-Aware Supervision

We implement active imitation learning to prevent trajectory drift during policy execution. Our system generates two types of policy-aware supervision. For the task policy $\pi_{hi}$, the exploration node identifies states where the policy violates either preconditions or postconditions. For $\pi_{lo}$, we intialize trajectory optimization from policy rollouts and penalize deviations from that trajectory.

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. Within this node, at each timestep $t$ $\pi_{hi}$ proposes a parameterized action $\hat{a}_t$. If both the pre-conditions of $\hat{a}_t$ and the post-conditions of $a_{t-1}$ are met, we set $a_t = \hat{a}_t$. In the event of failed post-conditions we continue the previous action, i.e. $a_t = a_{t-1}$. If the post-conditions of $a_{t-1}$ have been met but not the pre-conditions of $\hat{a}_t$, we sample from a distribution over the raw logits of the network. We continue sampling until 1) either a valid action is found or 2) we hit a termination condition. For each rejected proposal we push a new problem instance to the shared queue $Q_{task}$ and generate a point of negative supervision. Details and pseudo-code are in the Supplemental Materials.

The motion planning node uses rollouts of the motion-level policy $\pi_{lo}$ to align plan refinement with the learned controllers. To initialize trajectory optimization for an action $a_i$, we roll out the associated controllers $\pi_{lo}(*|a_i)$ to produce a sampled trajectory $\hat{\tau}^i$. To further encourage the planner to produce controls that $\pi_{lo}$ can imitate, we augment the objective with a *rollout deviation cost* based on the acceleration kernel from Dragan [29]: $\|\hat{\tau}^i - \tau_{opt}\|^2 = \sum_t \|(\tau^i_{t+1} - \tau^i_t) - (\hat{\tau^i_{t+1}} - \hat{\tau^i_t}))\|^2$.

## 4 Experimental Results

We evaluate our system in three domains: a 2D pick-place simulator, the RoboSuite benchmark [21], and the RoboDesk benchmark [22]. These domains are shown in Figure 1. We used Fast-Forward [30] 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 2D Pick-Place Experiments

**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. DMControl provides the underlying physics engine [31]. The goal requires each object overlap with its target (i.e. $||\text{pos}_{obj} - \text{pos}_{targ}||_2 \leq \text{radius}_{obj}$). For both state and image based learning, the observations contain the robot's velocity and a LIDAR-style sensor. This sensor performs ray checking in 39 directions evenly spaced around the robot and emanating away from its center of mass.

**Manipulated Variables and Dependent Measures.** Our primary variable is the training procedure used. We compare against flat imitation learning (flat-IL), reinforcement learning (RL), and passive imitation learning (no-feedback). For flat-IL, we trained a single policy within our system to map observations and goals directly to controls. For RL, we ran proximal policy optimization (PPO) [32]. We used a dense reward function that sums the distance from the robot to the closest object and from each object to its goal. We ran PPO with the default hyperparameters and parallelized training with double the compute of our method (60 vCPUs). For no-feedback, we ran our system without feedback between the policies and planners (20 vCPUs). We used two measures. Our primary measure of performance is the rate at which trained policies achieve their goal. To add a softer measure, we also tracked each object's progress: $||\text{pos}_{obj}[t=0] - \text{pos}_{targ}||_2 - ||\text{pos}_{obj}[t = T] - \text{pos}_{targ}||_2$. We denote these as Succ. and Dist. respectively.

| | 1 object | | 2 object | | 5 object | |
|---|---|---|---|---|---|---|
| Method | Succ. | Dist. | Succ. | Dist. | Succ. | Dist. |
| RL | 1.3% | 3.27 | 0% | 0.44 | n/a | n/a |
| flat-IL | 21% | 4.85 | 1.5% | 4.47 | n/a | n/a |
| no-feedback | n/a | n/a | n/a | n/a | 80% | 8.56 |
| tamp-feedback | 99.7% | 8.79 | 98.7% | 8.78 | 88% | 8.60 |

**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 difference between the distance of each object to its target at the start and end of training.

**Results.** Table 1 shows the results of these comparisons. PPO learned to move 1 object toward its target but lacked the precision to achieve the goal. This is because it never identified a grasping policy and settled on nudging objects with the gripper. PPO did not handle the 2 object variant well, making little progress on either metric. Flat-IL succeeded 20% of the time in the 1 object variant. With two objects, it made progress on Dist. but rarely succeeded. No-feedback does almost as well as the full system. With 5 objects present, it succeeds 80% of the time and nearly matches the full system on Dist. This highlights the strong inductive bias our hierarchical architecture provides. The full system get near perfect performance on the 1 and 2 object variants, and succeed 88% of the time on the 5 object problem. This performance is plotted in Figure 3 (left).

**Camera Observations.** We next 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.

**Non-Stationary Environment.** For the final experiment, we added 6 "human" actors. Each actor had the same radius and average speed as the robot. The actors performed model predictive control (MPC) at each simulation step. For this, 20 random 3-step trajectories were proposed sampled. The objective was to make progress towards a randomized goal location while avoiding collisions. Rollouts terminated if an actor collided with the robot. The actors only became mobile at evaluation time and were kept fixed at random locations during supervision. The robot could sense object locations directly, but not directly track human locations. Instead it avoided them with the LIDAR sensor. The trained policies were able to successfully place 3 objects 50% of the time.

### 4.2 RoboSuite Experiments

**Setup.** Our next experiments trained a 7-DoF Sawyer arm in the RoboSuite Pick-Place benchmark [21]. Four objects of different shapes/sizes are initialized at random locations. The goal is to place each in its target bin on the right. Our 1 object variants uses only the cereal box, and the 2 object variants adds the milk. Controls are on joint velocities plus the gripper. The domain has four action schemas: moveto$(obj)$, grasp$(obj)$, moveholding$(obj)$, and putdown$(obj)$. This leads to 16 possible high-level actions. The observations for these experiments were joint angles as well as translation and rotational displacements from the gripper to each object. Additionally, we considered two challenge conditions. One with noisy observations, where observations were corrupted by Gaussian noise with standard deviation equal to $1\%$ of the possible deviation in initial values. In the second, we tested the reactivity of the system by having a random object "teleport" back to its initial location with a $1\%$ probability at each timestep. We capped this to occur at most once per episode.

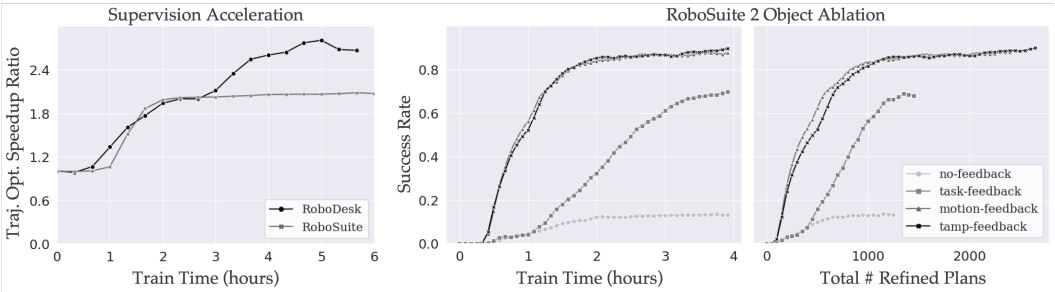


**Figure 4:** One of our contributions is a distributed TAMP architecture that can take advantage of partially trained motion policies to speedup plan refinement and improve data quality. Left: Speedup of plan refinement. Over the course of training, our method resulted in a 2x speedup of plan refinement in the RoboSuite domain with two objects and 2.6x speedup in the RoboDesk putaway task. Right: Performance as a function of train time and of total supervision (Total # Refined Plans) for the RoboSuite domain with two objects. The variants with feedback performed better even with the same amount of supervision, indicating the benefits of feedback for system performance are not solely from the increased rate of data generation.

**Dependent Measures.** Our primary measure of performance is the rate with which the trained policies successfully place all objects. For comparison against the benchmarks we also provide the average dense return of the trained policies over 500-step horizons with 1 object present.

**Results.** We compare against published results from RL and LfD. For the 1 object variant, our system achieves an average return of 390. The official benchmark for this task achieves returns near 50 with Soft-Actor Critic [21, 33]. Fan et al. [34] tried several methods, including PPO and deep determinisitc policy gradients [35], and were unable to scale them beyond the 1 object variant of this task. To compare with LfD, we refer to the results of Mandlekar et al. [36]. The authors trained from the RoboTurk cans dataset [37] and were able to successfully transfer the object to its bin 31% of the time from state data and 43% of the time from RGB camera images. By comparison, our systems reaches a 97% success rate from state data.

Figure 3 shows success rates for the problem with 1-4 objects and the noisy. Our system is able to successfully solve the task 88% of the time with all 4 objects. In the noisy observations condition, the trained policies were able to achieve a success rate of $64\%$ on the 4 object variant. By comparison, plans from the TAMP solver only executed successfully $15\%$ of the time when operating from the same distribution of noise. In the non-stationary, teleportation, condition, we ran training for 6 hours and were able to successfully move all 4 objects to their target locations 75% of the time. This experiment highlights the value of high-level feedback. When the system did not use the structured exploration procedure (i.e., the motion-feedback condition) performance topped out at $38\%$.

### 4.3 RoboDesk Experiments

**Background.** We completed the core of this work on May $19^{th}$, with the above domains [38]. We then asked what it would take to train in a new environment the system had not been engineered for. We chose to use the RoboDesk benchmark, which was released on May $13^{th}$ [22]. The results in this section reflect a single engineer working for three weeks. The steps to reach these results included: 1) extension of existing action schemas to handle new tasks; 2) integration of the existing system with the new simulator; and 3) iterating on motion specification and ML hyperparameters.

**Setup.** RoboDesk provides a 7 DoF PANDA arm with 9 goals: open slide, press button, lift block, open drawer, block in bin, block off table, lift ball, stack blocks, and block in shelf. We further designed a composite task to place a block in the shelf and close the door. Our observations are RGB camera images and joint positions. The controls are on joint velocities and the gripper. Our domain contains 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.** Our primary measure of performance is the overall success of the trained policies on a given problem distribution. Figure 3 shows the results of training from 2 random seeds for both tasks. The system learned to, coincidentally, complete both tasks with an average success rate of $67\%$. Training ran for 4 hours on a Nvidia DGX Station (40 vCPUs, 4 GPUs).

### 4.4 Ablation Study

Our final experiments explore the value of different types of feedback in our system. Generally, feedback can help in two ways: by speeding up planning, so more data is available, and by allowing policy-aware supervision to protect against trajectory drift [25]. We use two types of feedback: task-level feedback from structured exploration (task-feedback) and motion-level feedback (motion-feedback) from traj. opt. initialization and the deviation cost. We show these results in Figure 4.

First, we look at the increase in the rate of data generation. In RoboSuite and RoboDesk, we tracked time spent in optimization calls, with feedback and the no-feedback baseline. The tamp-feedback method gets a 2x speedup over the baseline in RoboSuite and a 2.6x speedup in RoboDesk.

Next, we looked at the performance of no-feedback, motion-feedback, task-feedback, and tamp-feedback in the RoboSuite 2 object domain. Figure 4 (middle) shows success rate against training time. 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 feedback within the 4 hour time limit. The no-feedback condition does not get above 20% success. To isolate the effect of data quality, Figure 4 (right) shows success rate vs the number of plans solved. Note that the gap between task-feedback and the other two methods shrinks. This makes sense, as task-feedback does not meaningfully accelerate planning. In this experiment, motion-feedback and tamp-feedback performed comparably. See the non-stationary RoboSuite variant in Section 4.2 for an example where task-feedback improves on motion-feedback.

## 5 Related Work

**Speeding up TAMP.** There is a growing field of work that uses machine learning on previous experience to reduce planning for TAMP systems. Much of this work focuses 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 they predicted full trajectories rather than executing the policy, and they did not penalize optimization for deviating from the policy controls. We don't make any claims about our method with respect to theirs, our focus is on the learned policies themselves.

**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 planners they use. Our are more compute intensive, but scale 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]. From pre-computed TAMP solutions, the authors trained a high-level model to predict action sequences and associated continuous values in conjunction with a set of controllers. They predicted full sequences with recurrent networks and then use an learned execution monitor to execute the plan with energy-based policies. A key difference of our approach is that we integrate with a more sophisticated TAMP solver and scale to more complex task spaces.

## 6 Future Work

There are several limitations to our method which can be improved on in future work. First, we could apply sequence modelling to the task policy to better handle long horizons. This may help reduce issues we encountered where the policy transitioned before accomplishing postconditions. Similarly, it would be straightforward to experiment with other imitation learning procedures beyond supervised learning. Finally, our results are in simulated environments. A clear next step is to deploy our approach on physical robots.

# 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. pages 447–454, 05 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)*, page 5040–5047. IEEE Press, 2016.

[24] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, X. Chen, and X. Wang. A survey of deep active learning, 2020.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] A. Dragan. *Legible Robot Motion Planning*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, June 2015.

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

[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] 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.

[34] 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.

[35] T. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2016.

[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] Anonymous. Technical report, May 2021.