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.

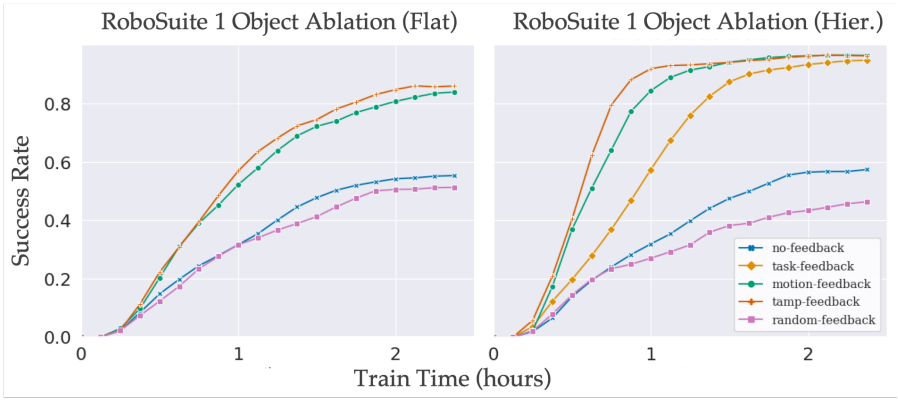

Figure 5: For the 1 object RoboSuite pick-place problem, we compare different feedback types in our system for training both flat policies (left) and our hierarchical policies (right). In both cases, motion-feedback provided the most significant performance gains, bringing flat policy performance from 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.

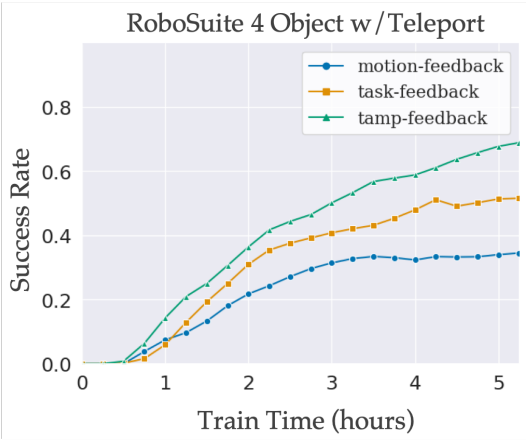

Figure 6: Comparison for the teleporting RoboSuite problems. With tamp-feedback, the policies learn to solve the problems 75% of the time. With just task-feedback, success plateaued at 54%. With just motion-feedback, success plateaued at 38%. This is an example where task-feedback provides noticeable 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.