Learning to Bridge the Gap: Efficient Novelty Recovery with Planning and Reinforcement Learning

Alicia Li[*], **Nishanth Kumar**[*], **Tomás Lozano-Pérez, and Leslie Pack Kaelbling**
MIT CSAIL
`{aliciali, njk, tlp, lpk}`@mit.edu

Abstract

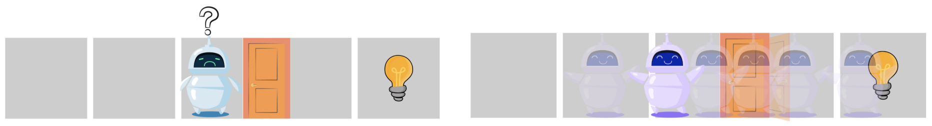
The real world is unpredictable. Therefore, to solve long-horizon decision-making problems with autonomous robots, we must construct agents that are capable of adapting to changes in the environment during deployment. Model-based planning approaches can enable robots to solve complex, long-horizon tasks in a variety of environments. However, such approaches tend to be brittle when deployed into an environment featuring a novel situation that their underlying model does not account for. In this work, we propose to learn a "bridge policy" via Reinforcement Learning (RL) to adapt to such novelties. We introduce a simple formulation for such learning, where the RL problem is constructed with a special "CallPlanner" action that terminates the bridge policy and hands control of the agent back to the planner. This allows the RL policy to learn the set of states in which querying the planner and following the returned plan will achieve the goal. We show that this formulation enables the agent to rapidly learn by leveraging the planner's knowledge to avoid challenging long-horizon exploration caused by sparse reward. In experiments across three different simulated domains of varying complexity, we demonstrate that our approach is able to learn policies that adapt to novelty more efficiently than several baselines, including a pure RL baseline. We also demonstrate that the learned bridge policy is generalizable in that it can be combined with the planner to enable the agent to solve more complex tasks with multiple instances of the encountered novelty.

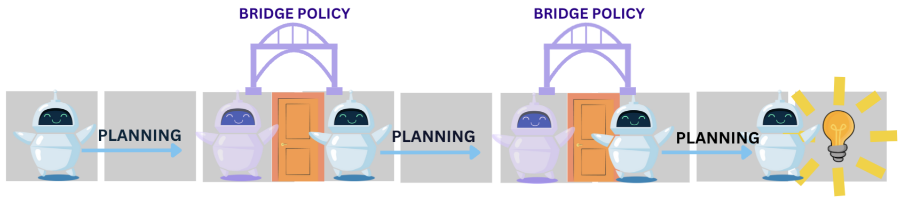
1 Introduction

Recent model-based planning approaches, such as Task and Motion Planning (TAMP), have enabled robots to perform complex and long-horizon tasks, such as setting a table, rearranging a room, or even assembling complex structures, under a wide variety of circumstances (1; 2; 3). These approaches assume access to some form of structured, abstract *model* that captures aspects of the environment important for the robot's decision making. However, useful robots must be able to cope with *novelty* that arises from being deployed in environments that neither the robot nor its designers could possibly have seen ahead of time (2). Unfortunately, following the planner's proposed action sequence fails catastrophically in such cases, which significantly limits the applicability and utility of such model-based approaches. We are thus primarily interested in efficiently and autonomously learning to cope with environment novelties encountered during an agent's deployment in a novel environment.

In this work, we assume the robot is equipped with a collection of *skills* (such as moving to a specified location, picking a specified object, or placing a held object at a location) as well as a planning model in terms of those skills that it can use to solve user-provided tasks. We are interested in situations where executing plans made according to the model leads to some kind of failure in the environment. As a simple example, consider the 'Light Switch Door' environment illustrated in Figure 1. Here, the

[*]Equal Contribution.


(a) Encountering a Failure (b) Learning to Recover



(c) Deploying RL Bridge Policy Approach

Figure 1: Our approach in Light Switch Door. (a): During deployment, the robot encounters an unknown object (door), when trying to turn on the light. This renders it unable to follow its plan. (b): The robot switches from planning to RL to learn how to overcome the novelty. (c): During evaluation, the robot switches from following its plan to following the learned 'bridge policy' each time it gets stuck at a door, and then switches back to the planner once it opens the door. It is thus able to generalize to opening an arbitrary number of doors.

robot possesses skills to move left and right between grid cells, and turn on and off the light once it is in the same cell. When provided with the task of turning on the light, it is able to make and execute a plan composed of these skills to move right until it is in the light's cell, and then turn it on. However, the robot is deployed into an environment containing one or more doors that prevent it from moving between cells separated by a door. Given it has no knowledge of the door, nor a skill to open it, the robot's planner will make a plan that will fail during execution (i.e., the robot will be unable to execute its 'move' skill between cells separated by a door, as shown in Figure 1(a)). We assume the robot is equipped with the ability to automatically detect such a failure. Since the robot has encountered this novelty in its new deployed environment, it can reasonably expect to encounter it in the future when asked to perform similar tasks. Thus, it must not only efficiently resolve the novelty temporarily to complete the task at hand, but also *learn* such that it can effectively handle such novelty in the future. Given this, our key question of interest is: how can we efficiently learn to overcome failures due to novelty and solve tasks in our deployment environment?

One straightforward approach to answering our key question is to instantiate a reinforcement learning (RL) problem starting from the state in which the planner fails and provide reward only when the ultimate task goal is achieved. Unfortunately, such a problem features a continuous action space and sparse reward, which are known to be extremely challenging and sample-inefficient for RL approaches (4; 5; 6). Our key idea is that we can make this RL problem significantly easier by exploiting the planner. In particular, we simply construct the RL problem so that it has access to a special action we call 'CallPlanner' that hands control to the planner. In this way, the policy learned by RL need not to solve the entire long-horizon task, but instead only to resolve the novelty before handing back control to the planner. Through the use of the CallPlanner action, our method implicitly learns a policy from the failed state to the *initiation set* of the planner: the set of states where following the plan returned will bring us closer to the goal state. Intuitively, in our running example, the planner already knows how to move and turn on the light: the RL policy must only learn to open the door before handing back control to the planner. Correctly learning such a "bridge policy" allows the agent to smoothly "bridge" the gap caused by planning failures. In our example, it also enables the agent to solve tasks involving arbitrary numbers of doors by simply alternating between calling the planner, running the RL policy, and then handing back control to the planner (Figure 1 (c)).

We implement our approach in an online learning setting where the agent is provided with a series of tasks in its deployment environment. In experiments, we evaluate the extent to which our approach—*Bridge Policy Learning*—is able to learn policies that can be used in conjunction with planning to overcome novelties encountered during long-horizon tasks. Experiments across three different simulated environments with varying novelty reveal that our approach is significantly more sample efficient than existing baselines from the literature. We also find that the learned bridge policy, when combined with planning, enables the agent to generalize aggressively to solving more challenging tasks than those encountered during learning.

2 Problem Setting

We consider planning and learning in deterministic, fully-observable environments with object-oriented states. Below, we first describe our assumptions about the environment model before detailing a minimal specification for the planning system, as well as our notion of environment novelty. As a simple running example, we introduce the 'Light Switch Door' environment depicted in Figure 1. In this environment, the agent's goal is to turn on the light at the end of a row of grid cells. It possesses skills and a planning model enabling it to move left and right and turn on the light when it is in the corresponding cell. However, at deployment time, it encounters a door that is not part of its planning model. The agent does not possess an explicit skill for opening the door, however, it is able to do this by combining a series of low-level actions.

2.1 Environment and Task Models

Following previous work (7), we take an *environment* to be a tuple $\mathcal{E} = \langle \Lambda, \chi, U, f, \Psi \rangle$. Λ is the set of possible object types; an object type $\lambda \in \Lambda$ has a name (e.g. `robot`, `light`, or `door` in our running example) and a set of real-valued features (e.g. `x-position`, `light-level`) represented as a tuple of continuous values of dimension $\mathtt{dim}(\lambda)$. An object o has a particular name (e.g. `robby`, `light-bulb0`) and an associated type (e.g. `robot`) in any particular problem (i.e., an instance of an environment). An object will specify particular values for each of the features of the associated type (e.g. `robot` will have `x-position`: 0). A state $x \in \chi$ is a collection of objects with assigned feature values, and the state space is defined by the set of all objects $\mathcal{O}$, and the possible feature values they could be assigned. U is the action space consisting of named *parameterized skills* (8), $u(\bar{\lambda}, \circ) \in U$ (e.g. `MoveRight(robot)`, or `ToggleLightSwitch(robot, [toggle-value])`, where `toggle-value` is a continuous parameter). Each such skill can be executed by specifying the list of objects with types $\bar{\lambda}$, which causes the environment to advance to a new state $x' \in \chi$ according to some unknown transition model $f : \chi \times U \to \chi$. We assume there always exists a particular skill named `RunLowLevelAction`, which takes in no objects and whose continuous parameters provide access to a very low-level action space (e.g. the robot's motor commands) compared to other skills. Finally, Ψ is a set of *predicates*. Each *predicate* $\psi \in \Psi$ has a name (e.g., `Light-On`) and a tuple of types (e.g., (`light`)). *Grounding* a predicate yields a *ground atom*, which is a predicate and a mapping from its type tuple to objects (e.g., `Light-On(light-bulb0)`). Given a particular state $x \in \chi$, a ground atom can be run on this state to produce a boolean value. Predicates induce a state abstraction: $\text{ABSTRACT}(x)$ denotes the set of ground atoms that hold true in x, with all others assumed false. We denote a set of ground atoms via $\bar{\psi}$. We denote the abstract state as s (i.e., $s = \text{ABSTRACT}(x)$).

Each environment is associated with some *task distribution* T. A *task* $t \in T$, is a tuple $\langle \mathcal{O}, x_0, G, H \rangle$, where $\mathcal{O}$ is some set of objects and $x_0 \in \chi$ is an initial state. G is a collection of ground atoms describing the goal (e.g. [`Light-On(light-bulb0)`]), and H is a maximum horizon (i.e., number of actions) within which the task must be solved. A solution to a task is sequence of actions $u_0, u_1, \ldots, u_n$, such that $n < H$ and taking these actions from the initial state yields a final state x_n such that the goal expression holds (i.e. $\forall g \in G, g(x_n)$).

2.2 Planning and Environment Novelty

We assume the agent has access to a planner P which takes a task t and corresponding environment $\mathcal{E}$ as input and produces a policy π_{plan}. Importantly, this planner operates over some internal model that *does not* account for environment novelty. This policy in turn takes in a state, and outputs (1) an action to take, (2) a boolean 'stuck' indicator (i.e., $\pi_{\text{plan}} : \chi \to U \times \{0, 1\}$). The stuck indicator can be interpreted as a simple measure of the robot's confidence in its output: when it is false, the robot believes its action will make progress towards the goal as intended, whereas when it is true, the policy believes some novelty has been encountered that it is not equipped to handle. For example, in a problem from the 'Light Switch Door' in which there are 3 cells, a door between the second and third cells, and a light switch in the final cell, the planner will produce a policy that outputs a 'MoveRight' action with the stuck indicator set to false in the first and second cells, and then produce any action with the stuck indicator set to true after failing to move through the door [2]. For the purposes of our

[2]Internally, the policy has a model of the expected effects of each action (computed by the planner), and realizes that it is stuck after these effects do not hold

problem, we can treat this planner and its output policy largely as black boxes, and thus we defer discussing implementation details to Appendix B. We refer to the state from which the stuck indicator is first set to true as the stuck state x_{stuck}, and are primarily interested in learning to complete the corresponding task from it.

We consider an online learning setting in which to learn to overcome this novelty. We measure both the task solves achieved during learning process and the performance on evaluation tasks after each learning update. Specifically, *tasks* are drawn from T. The agent has a finite budget of environment steps H_{train} every *episode* (i.e., before the environment is reset). Our objective is to solve as many tasks drawn from this distribution T as possible given a limited time horizon for each task. For the purposes of measuring generalization, in Section 4 we artificially split tasks into *training* and held-out *evaluation* tasks such that the evaluation tasks are constructed to have similar goals, but often involve more objects (e.g. more cells with more doors in the case of the Light Switch Door environment), requiring the agent to generalize beyond the training distribution.

3 Bridge Policy Learning

How should the agent formalize and optimize the objective introduced in Section 2.2? A simple but naive approach would be to simply behave randomly for a random amount of time and then try to follow the plan again. However, this process would need to be repeated for every new problem, and would likely lead to the task horizon being exhausted in most cases. Another approach would be to simply perform RL in a new Markov Decision Process (MDP) that models the environment, task, and novelty. However, not only would this approach need to learn to overcome whatever novelty got the planner stuck, it would also need to complete the rest of the task. This is likely to be quite sample inefficient. In particular, it does not leverage or exploit knowledge the planner already possesses. For instance, in our running example from the Light Switch Door environment, the planner would be able to achieve the task goal once the door is opened: we only need to learn how to open the door.

We propose to learn a *bridge policy* that the agent can use during execution to get to a state where replanning and following the new plan will no longer get stuck. At evaluation time, the agent will start by planning, and then executing its plan until it gets stuck. It will then use its bridge policy until this policy terminates when it calls the planner, then replan and execute the new plan once again. This iteration between the planner and the bridge policy continues until the agent achieves the task goal or the task horizon is exhausted. To learn such a bridge policy, we need to define a set of target states for the policy to reach. In this work, we propose to have the agent automatically discover such a set of states implicitly. We do this by setting up an RL problem in which one of the available actions is to hand back control of the agent to the planner, so that the agent learns the optimal state to start replanning.

Below, we first describe the construction of the RL problem, then discuss how we choose to solve it. Finally, we describe how we use the learned policy at evaluation time to generalize to problems in our evaluation set.

3.1 Constructing an RL Problem

Once the agent detects that it is in a 'stuck' state x_{stuck}, we set up an RL problem by constructing an MDP $\mathcal{M} = \langle \chi', U', f, R, \gamma \rangle$. Here, χ' represents the RL problem's state space, U' its action space, f its transition function (which is the same as the original task's environment), R its reward function, and γ its discount factor.

The state space χ' is a subspace of the corresponding environment's state space χ. In particular, we construct χ' by selecting a subset of the objects $\mathcal{O}$ that make up χ. The purposes of this feature selection are twofold: to improve the sample efficiency of RL, and to improve the generalization of the learned bridge policy. Many possible strategies for selecting such a subset are possible (e.g. (9)), but in this work, we adopt the simple strategy of selecting the single object that can be interacted with that is closest to the robot object by euclidean distance in x_{stuck}. In our running example, this will simply be the 'door' object. Learning a bridge policy that's specific to this object's state enables this policy to be reused in novel problems that may include additional variations in different objects (e.g. problems with more doors or grid cells in our running example).

The action space U' is simply the original environment's action space augmented with an extra action we call '*CallPlanner*' (i.e. $U' = U \cup \{\text{CallPlanner}\}$) [3]. As the name suggests, this action simply returns control of the agent to the planner: the agent begins executing a sequence of actions output by the planner, and terminates in a new state x' where either (1) the goal is achieved, (2) the task horizon has expired, or (3) the planner is stuck again. Importantly, even though the the planner itself executes many actions before terminating, we treat the entire CallPlanner sequence as one atomic action within the MDP $\mathcal{M}$.

Finally, the reward function is simply a sparse reward corresponding to achieving the task goal G.

$$R(x_t, a_t) = \begin{cases} 1 & \text{if } G \subseteq \textsc{abstract}(x_{t+1}), \\ 0 & \text{otherwise} \end{cases}.$$

We set the discount factor γ to be less than 1 so that the agent is encouraged to solve the MDP with the fewest possible actions. Once the robot gets to a state it can plan from, executing the CallPlanner atomic action will bring the robot to the goal in one step, so an optimal bridge policy would take the fewest possible actions to resolve the novelty before calling the planner.

As is standard for RL problems, the learning objective is to learn a policy π_{bridge} to maximize the discounted sum of rewards in expectation over all MDPs that comprise our training distribution:

$$\mathbb{E}_{\mathcal{M}\sim T} \sum_{\substack{t=0 \\ a_t \sim \pi(x_t)}}^{H} \gamma^t R(x_t, a_t).$$

3.2 Learning a Bridge Policy

Algorithm 2 Our overall meta-policy for solving tasks by leveraging a planner P and a learned bridge policy π_{bridge}

```
input: Environment $\mathcal{E} = \langle \Lambda, \chi, U, f, \Psi \rangle$, Task $t = \langle \mathcal{O}, x_0, G, H \rangle$, Goal $G$, Planner $P$, Bridge Policy $\pi_{\text{bridge}}$.
 1: $\pi_{\text{plan}} = P(t, \mathcal{E})$                                ▷ Start by planning
 2: $x = x_0$
 3: num_steps = 0
 4: while $G \subsetneq \textsc{abstract}(x)$ and num_steps $< H$ do
 5:     $u$, stuck $= \pi_{\text{plan}}(\textsc{abstract}(x))$          ▷ Follow plan
 6:     while stuck != 1 do
 7:         $x = f(u, x)$
 8:         $u$, stuck $= \pi_{\text{plan}}(\textsc{abstract}(x))$
 9:         num_steps += 1
10:     end while
11:     while u != 'CallPlanner' do
12:         $u = \pi_{\text{bridge}}(x)$ ▷ Follow bridge until CallPlanner
13:         $x = f(u, x)$
14:         num_steps += 1
15:     end while
16: end while
.
```

Given the MDPs constructed in Section 3.1, we can apply any RL algorithm capable of handling continuous states and actions to learn a policy π_{bridge}. However, in our setting, actions take the form of parameterized skills [4]. We thus take advantage of this by leveraging recently-introduced efficient architectures for RL with parameterized skills (11; 2). Specifically, we choose to use a variant of the pure RL approach introduced in (2).

Following this approach, which is related to Deep Q Learning (12), we learn a Q-function $Q(x, a)$ that maps MDP states and skills (with all parameters specified) to Q-values. At evaluation time, we sample n_{sample} fixed number of continuous parameters uniformly at random for each skill among our set of actions U' and pick the argmax. To prevent overestimation of Q-values, we maintain a separate target network in accordance with Double DQN (13).

We randomly initialize our Q-networks, and collect data for training them by performing epsilon-greedy exploration with annealing of the epsilon parameter. Here, epsilon greedy intuitively serves to balance how much we value solving the current task with how much we want to explore and learn new things about our environment.

[3]In environments where it is possible for the robot to take unsafe actions, it would be useful to further restrict or modify this action space, but that is beyond the scope of this current work.

[4]Thus, our MDP is actually a parameterized action MDP (PAMDP) (10)

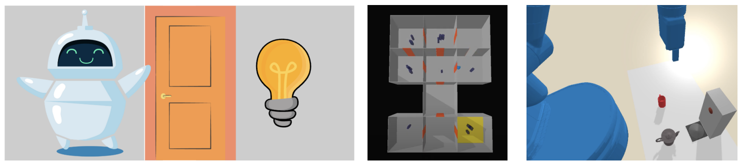

(a) Light Switch Door (b) Doorknobs (c) Coffee

Figure 2: Visualizations of our different experimental domains.

3.3 Using a Learned Bridge Policy

After training our bridge policy π_{bridge} we can leverage it to recover from novelties encountered during evaluation time by switching control of the agent between using its planner and using its learned bridge policy. Our meta policy for solving tasks via both planning and using the learned bridge policy is shown in Algorithm 1. Given a task, we first call the planner and execute its plan until a 'stuck' state is reached (if one is reached at all). We then pass control to the bridge policy, which will perform some sequence of actions before either achieving the task goal, or invoking its CallPlanner action to hand control of the agent back to the planner. In this way, a bridge policy that has correctly learned to overcome an encountered novelty can be used by the agent to generalize to testing tasks involving more instances of that novelty (assuming every instance can be resolved by the same policy) by calling the same bridge policy multiple times in sequence whenever the novelty is encountered. We evaluate this capability empirically in Section 4 below.

4 Experiments

Our experiments are designed to answer the following questions about our approach:

Q1. How sample efficient is bridge policy learning compared to alternatives?
Q2. How well can our learned bridge policy be used to generalize to harder versions of the tasks?
Q3. How important is the $CallPlanner$ action in our approach?

Environments: We provide high-level descriptions of our three simulated environments here.

- *Light Switch Door*: An implementation of our toy running example depicted in Figure 1. The robot must traverse a row of cells to turn on a light at the end of the row. However, there are doors obstructing the way that the robot's planner does not model, and thus following its output plan will result in a stuck state. The robot needs to learn how to open these doors using two low level actions. This domain is a variation on 'Light Switch' from (2).
- *Doorknobs*: Equivalent to the 'Doors' environment from (7). Here, a robot must navigate to a target room while avoiding obstacles and opening doors. The planner has no direct knowledge of the existence of doors, nor how to open them. A skill that moves the robot between configurations using a motion planner (BiRRT) is provided; door opening must be learned by a bridge policy. Tasks have 4–25 rooms with random connectivity. The goals, obstacles, and initial robot pose also vary.
- *Coffee*: Another environment taken directly from (7). The robot makes coffee by putting the jug in the coffee machine, turning the machine on, and then pouring the coffee into a cup. However, the jug is initially rotated in a way so that the robot must first rotate the jug in order to grasp it. Otherwise, if the robot tries to grasp the jug without first rotating it, it will enter a stuck state.

Approaches: We now briefly describe the various approaches we evaluate in the above environments.

- *Bridge Policy Learning:* Our main approach.
- *Ours no feature selection:* An ablation of our approach that does not perform any feature selection when constructing the RL problem, but instead passes in the entire task state (i.e., $\chi' = \chi$). Note that we do not test this ablation in the Doorknobs environment because the entire state is too large and the method times out after several learning cycles.
- *Ours no CallPlanner:* An ablation of our approach that does not include CallPlanner in the RL problem's action space. This ablation is not able to exploit the planner's knowledge.
- *Random Bridge:* Do not learn using the training tasks; behave randomly (including attempting to call the planner) whenever the planner reaches a stuck state.

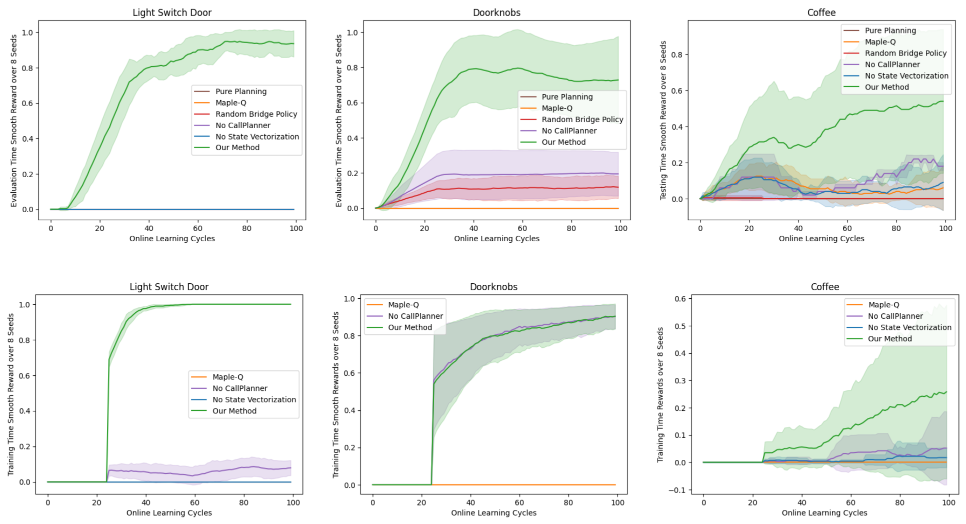


Figure 3: Experiments in our different domains. Results from both evaluation time and training time. We average smooth reward, the reward averaged over the last 25 online learning cycles, across all seeds and graph the variance.

- *Pure Planning:* Do not learn using the training tasks; keep attempting to call the planner whenever the planner reaches a stuck state.
- *Maple Q:* A pure RL approach from (2) inspired by (11). Do not use the planner at all, not even at the start. Rather, attempt to learn a policy to achieve the goal from the initial state for each of the training tasks. There is also no state space feature selection in this method.

Experimental Setup: We run 8 independent seeds of each approach in each environment. Different seeds yield different training and test problems where the numbers of objects and initial states of the task are randomized. We fix the train and evaluation horizons to be the same for all approaches in each environment. Our key quantitative metrics of interest are smooth test time reward, and smooth training time reward, to track the learning process of the agent over time. Additional details on hyperparameters and experiments are presented in Appendix A.

Results and Analysis: Figure 4 shows that our method strongly outperforms the baselines across environments. Specifically, our method is much more sample efficient, converging at around the 30th online learning cycle when it has learned from and collected only 150 trajectories in Light Switch Door and Doorknobs. Our approach achieves significantly higher training time reward in Coffee and Light Switch Door than all other methods. In Doorknobs, the ablation without the CallPlanner action is also able to reach high training rewards because the environment in those experiments is very simple with only 2x2 grids. In this environment, the ablation can simply learn the full trajectory to the goal without calling the planner. When the ablation is evaluated in a larger environment, however, our method obtains much higher reward in comparison. Without CallPlanner, we find that the ablation simply gets "stuck" after overcoming the first novelty during evaluation time in all three environments. Our results show that the ablation without feature selection is not able to learn as efficiently. CallPlanner and feature selection are both critical aspects of our approach.

5 Related Work

Open-World AI: There has been a long history of work that attempts to develop agents capable of continuously adapting to novel scenarios (14; 15; 16; 17; 18; 19). Many of these works (14; 15; 16; 17) are interested in a setting where the agent is not provided any models nor specific goals a priori but rather must simply explore in an open-ended fashion. By contrast, our work is interested in more directed exploration, where the agent is specifically trying to overcome some failure and also is equipped with a model and planner to be leveraged. The works that operate in a setting similar to ours (18; 19) typically focus on two aspects: novelty detection, and novelty adaptation. Additionally, these works often operate in environments with purely discrete action spaces. Our work assumes a relatively simple form of novelty detection and focuses on a particular form of novelty caused by a particular skill failing to achieve its affect due to particular object(s) in the environment. However, we operate in environments with fundamentally continuous action spaces, and make no assumptions about any existing recovery strategies.

Learning new skills for planning: Previous works have studied adding or adapting skills for a planning model by learning from expert demonstrations (20; 7), and from online interaction (21; 22; 23; 18). Similar to our approach, works that learn from online interaction generally assume the agent is following a provided plan until an error is encountered, at which point it switches to RL. Importantly, they assume the original computed plan is correct (but for the error) and thus aim to learn to resolve the error and simply continue the plan. This assumption enables them to leverage the original plan for reward shaping. By contrast, our approach makes no explicit commitment to the original plan, but rather enables the agent to replan after the learned policy has terminated. This enables our approach to tackle a wider range of environments as demonstrated experimentally in Section 4 by the ablation that does not contain the CallPlanner action [5]. Specifically, the set of states reachable from the planner that these methods learn is a subset of the states we learn. Therefore, our method is more general and able to find a more optimal trajectory in certain tasks. However, we must rely on only the sparse reward of achieving the agent's goal, and thus could be significantly less sample-efficient in complex environments. Intuitively, previous methods using RL can be thought of as giving a planner "access" to learning an RL policy. Our method builds upon this by also doing the converse: giving the RL agent access to a planner.

Recovering from failures in robotics: Previous approaches in diagnosis and plan repair for robotics have handled recovery via pre-specified heuristics (24; 25; 26) or prompting a human user (27; 28). More recently, several methods (23; 2) seek to autonomously *adapt* existing skills to succeed under unexpected environmental changes. This assumes that there exists a skill in the robot's repertoire that can be adapted to cope with the encountered novelty. We do not make this assumption, but rather learn a novel recovery strategy that may be composed of existing skills and low-level actions.

6 Conclusion

In this work, we proposed a method that enables a robot equipped with a model-based planner to adapt to novel deployment environments where following the returned plan fails to complete an assigned task. Experiments in simulated domains revealed that our method is able to successfully overcome this novelty with very low sample complexity and generalize to larger environments.

There are several limitations of our approach. Firstly, we rely on reducing the dimensionality of the input state space to enable generalization of the bridge policy to novel, more complex tasks. We currently use a very simple approach for this that is unlikely to work in more complex and realistic environments. Secondly, we attempt to learn a single bridge policy that is capable of coping with all encountered novelties. This suffices for environments with very few novelties that lead to planning failure, but would not work in environments featuring multiple different types of novelty. Finally, our approach performs a relatively simple form of exploration (epsilon-greedy), which works well when the desired recovery behavior is relatively short horizon, but will likely not scale well to settings that require learning a much more long-horizon bridge policy.

In the future, we hope to address these limitations and enable our approach to be realized in complex and useful robotics domains. An important direction is to integrate our approach with perception to perform decision making from camera input, both for our planner (2), and for our learned bridge policy (11). In conjunction with this, integrating pretrained vision language models (VLMs) could help perform automatic dimensionality reduction for our bridge policy by suggesting which objects to ignore or consider after a stuck state is encountered. This could also be directly learned via a graph neural network (GNN), as demonstrated in previous work (9). Additionally, it will be important to run our approach in a wider range of complex environments, and compare directly to closely-related approaches from the literature (e.g. (23; 18)). This would allow us to further test the efficacy of our CallPlanner formulation in comparison to other approaches that have achieved high performance in similar problem settings.

[5]We do not directly compare to baselines that leverage the original plan for reward shaping (e.g. (23; 18))

References

[1] C. R. Garrett, R. Chitnis, R. M. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez, "Integrated task and motion planning," *CoRR*, vol. abs/2010.01083, 2020. [Online]. Available: https://arxiv.org/abs/2010.01083

[2] N. Kumar, T. Silver, W. McClinton, L. Zhao, S. Proulx, T. Lozano-Pérez, L. P. Kaelbling, and J. Barry, "Practice makes perfect: Planning to learn skill parameter policies," 2024. [Online]. Available: https://arxiv.org/abs/2402.15025

[3] J. Chen, J. Li, Y. Huang, C. Garrett, D. Sun, C. Fan, A. Hofmann, C. Mueller, S. Koenig, and B. C. Williams, "Cooperative task and motion planning for multi-arm assembly systems," 2022. [Online]. Available: https://arxiv.org/abs/2203.02475

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *arXiv preprint arXiv:1509.02971*, 2015.

[5] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight experience replay," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[6] M. G. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

[7] T. Silver, A. Athalye, J. B. Tenenbaum, T. Lozano-Perez, and L. P. Kaelbling, "Learning neuro-symbolic skills for bilevel planning," 2022. [Online]. Available: https://arxiv.org/abs/2206.10680

[8] B. Da Silva, G. Konidaris, and A. Barto, "Learning parameterized skills," in *International Conference on Machine Learning (ICML)*, 2012. [Online]. Available: https://arxiv.org/pdf/1206.6398.pdf

[9] T. Silver, R. Chitnis, A. Curtis, J. B. Tenenbaum, T. Lozano-Pérez, and L. P. Kaelbling, "Planning with learned object importance in large problem instances using graph neural networks," in *AAAI Conference on Artificial Intelligence (AAAI)*, 2021.

[10] W. Masson and G. D. Konidaris, "Reinforcement learning with parameterized actions," *CoRR*, vol. abs/1509.01644, 2015. [Online]. Available: http://arxiv.org/abs/1509.01644

[11] S. Nasiriany, H. Liu, and Y. Zhu, "Augmenting reinforcement learning with behavior primitives for diverse manipulation tasks," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2022.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013. [Online]. Available: https://arxiv.org/abs/1312.5602

[13] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015. [Online]. Available: https://arxiv.org/abs/1509.06461

[14] P.-Y. Oudeyer, F. Kaplan, and V. V. Hafner, "Intrinsic motivation systems for autonomous mental development," in *IEEE transactions on evolutionary computation*, 2007.

[15] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," in *International Conference on Machine Learning (ICML)*, 2017.

[16] C. Colas, P. Fournier, M. Chetouani, O. Sigaud, and P.-Y. Oudeyer, "Curious: intrinsically motivated modular multi-goal reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2019. [Online]. Available: https://proceedings.mlr.press/v97/colas19a/colas19a.pdf

[17] V. G. Santucci, P.-Y. Oudeyer, A. Barto, and G. Baldassarre, "Intrinsically motivated open-ended learning in autonomous robots," 2020. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fnbot.2019.00115/full

[18] S. Vats, M. Likhachev, and O. Kroemer, “Efficient recovery learning using model predictive meta-reasoning,” in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 7258–7264.

[19] S. Goel, P. Lymperopoulos, R. Thielstrom, E. Krause, P. Feeney, P. Lorang, S. Schneider, Y. Wei, E. Kildebeck, S. Goss, M. C. Hughes, L. Liu, J. Sinapov, and M. Scheutz, “A neurosymbolic cognitive architecture framework for handling novelties in open worlds,” *Artificial Intelligence*, vol. 331, p. 104111, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S000437022400047X

[20] A. Pacheck and H. Kress-Gazit, “Physically feasible repair of reactive, linear temporal logic-based, high-level tasks,” *IEEE Transactions on Robotics*, vol. 39, no. 6, pp. 4653–4670, 2023.

[21] J. Mendez-Mendez, L. P. Kaelbling, and T. Lozano-Pérez, “Embodied lifelong learning for task and motion planning,” in *Conference on Robot Learning (CoRL)*, 2023. [Online]. Available: https://proceedings.mlr.press/v229/mendez-mendez23a/mendez-mendez23a.pdf

[22] S. Cheng and D. Xu, “League: Guided skill learning and abstraction for long-horizon manipulation,” 2023. [Online]. Available: https://arxiv.org/abs/2210.12631

[23] S. Goel, Y. Shukla, V. Sarathy, M. Scheutz, and J. Sinapov, “Rapid-learn: A framework for learning to recover for handling novelties in open-world environments,” 2022. [Online]. Available: https://arxiv.org/abs/2206.12493

[24] E. Khalastchi and M. Kalech, “On fault detection and diagnosis in robotic systems,” *ACM Comput. Surv.*, 2018. [Online]. Available: https://doi.org/10.1145/3146389

[25] M. Colledanchise and P. Ögren, “Behavior trees in robotics and AI: an introduction,” *CoRR*, vol. abs/1709.00084, 2017. [Online]. Available: http://arxiv.org/abs/1709.00084

[26] T. Matsuoka, T. Hasegawa, and K. Honda, “A dexterous manipulation system with error detection and recovery by a multi-fingered robotic hand,” in *Proceedings 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human and Environment Friendly Robots with High Intelligence and Emotional Quotients*, vol. 1, 1999, pp. 418–423 vol.1.

[27] C. Innes and S. Ramamoorthy, “Elaborating on learned demonstrations with temporal logic specifications,” 2020. [Online]. Available: https://arxiv.org/abs/2002.00784

[28] S. Tellex, P. Thaker, J. Joseph, N. Roy, and D. H. Laidlaw, “Asking for help using inverse semantics,” in *Robotics: Science and Systems (RSS)*, 2014.

[29] M. Fox and D. Long, “Pddl2. 1: An extension to pddl for expressing temporal planning domains,” *Journal of Artificial Intelligence Research (JAIR)*, 2003. [Online]. Available: https://arxiv.org/pdf/1106.4561.pdf

[30] T. Silver, R. Chitnis, J. Tenenbaum, L. P. Kaelbling, and T. Lozano-Pérez, “Learning symbolic operators for task and motion planning,” in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2021. [Online]. Available: https://arxiv.org/pdf/2103.00589.pdf

[31] T. Silver, R. Chitnis, N. Kumar, W. McClinton, T. Lozano-Pérez, L. Kaelbling, and J. B. Tenenbaum, “Predicate invention for bilevel planning,” in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2023. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/26429/26201

[32] N. Kumar, W. McClinton, R. Chitnis, T. Silver, T. Lozano-Pérez, and L. P. Kaelbling, “Learning efficient abstract planning models that choose what to predict,” in *Conference on Robot Learning (CoRL)*, 2023. [Online]. Available: https://openreview.net/pdf?id=_gZLyRGGuo

[33] M. Helmert, “The fast downward planning system,” *Journal of Artificial Intelligence Research (JAIR)*, 2006. [Online]. Available: https://www.jair.org/index.php/jair/article/view/10457/25068

A Additional Experimental Details

Table 1: Environment Hyperparameters

Env	Light Switch Door	Doorknobs	Coffee
Horizon	max(30, h+5)	200	100
Steps per learning trajectory	100	100	100
Trajectories per learning cycle	5	5	5
Number of training tasks	1	1	1
Number of test tasks	10	10	1
Number of test novelties	2–4	2–5	1
Number of test cells	10–20	4–25	NA

Note: h is the minimum horizon length needed to complete the task in Light Switch Door.

Table 2: Method Hyperparameters

Parameter	Value
Discount factor γ	0.8
Learning rate	10^{-3}
Polyak averaging	2.5^{-3}
ϵ annealing reduction per step	$3.8 \cdot 10^{-5}$
MLP hidden layer sizes	[32, 32]
MLP max iters	10^4
Replay buffer size	10^6
Optimizer	Adam
Weight decay	0

All experiments were run on Intel Xeon Gold 6130 Processor CPUs.

B Planner Implementation Details

Here, we provide additional details about the implementation of the planner used in this work. We note that this is a relatively simple implementation that satisfies the setup in Section 2 and other, more sophisticated implementations are possible as well.

We adopt a planning implementation that is identical to that of (2). This requires access to a set of extra predicates, Ψ_{planner} in addition to those specified as part of the environment. In everything that follows, assume that when we perform abstraction (i.e., calling ABSTRACT), we will use the environment predicates Ψ as well as the planner predicates Ψ_{planner}. For example, in our toy Light Switch Door environment, the environment predicates will include only 'LightOn(light)', while the agent will additionally have predicates such as 'RobotInCell(robot, cell)' and 'Adjacent(?c1:cell, ?c2:cell)'. Next, we assume access to PDDL symbolic planning operators (29) with predicate-based preconditions and effects [6]. For example, the operator `Move(robot, current_cell, target_cell)` is:

[6]Previous work (30; 31; 32) has learned operators and predicates; but we choose to manually specify them here

```
    MoveRobot(robot, current_cell,
              target_cell)
    :precondition (and
      (Adjacent current_cell
                  target_cell)
      (RobotInCell robot current_cell))
    :effect (and
      (RobotInCell robot current_cell)
      (not (RobotInCell robot
            target_cell)))
```

Importantly, note that these symbolic operators are entirely discrete and do not feature continuous parameters. Here, the preconditions characterize the initiation conditions of the skill. For example, in the above operator, we say the preconditions hold in a state x if $\{$`Adjacent(current_cell, target_cell)`, `RobotInCell(robot, current_cell)`$\} \subseteq$ ABSTRACT(x). Similarly, the effects characterize aspects of the state we expect to be true after the skill is executed. We say the operator has been executed successfully in a state x' (where x' is the result of executing a skill u from the state x; $x' = f(x, u)$) if $\{$`RobotInCell(robot, current_cell)`$\} \subseteq$ ABSTRACT(x') and also if $\{$`RobotInCell(robot, target_cell)`$\} \not\subseteq$ ABSTRACT(x').

We assume that each operator is linked with a (1) parameterized skill from our environment's action space U, as well as (2) a *parameter policy* that takes in an environment state x and outputs a distribution over continuous parameters for the associated skill.

Given a task (Section 2), we use the initial state x_0 and a goal G, we construct a PDDL planning problem with initial state ABSTRACT(x_0) and use a PDDL planner (33) to efficiently generate a sequence of planning operators (a skeleton) that chain together by precondition and effect to reach the goal. In experiments, we perform A^* search using the LM-Cut heuristic.

Once a skeleton is obtained, we construct the planner policy π_{plan} such that it sequentially executes the skill associated with each operator in the plan by greedily selecting parameters from the operator's associated parameter policy. We implement the boolean stuck indicator by simply checking whether the subsequent operator's preconditions hold in the state resulting after executing this skill with the provided continuous parameters.