Adversarial Actor-Critic Method for Task and Motion Planning Problems Using Planning Experience

Beomjoon Kim, Leslie Pack Kaelbling and Tomás Lozano-Pérez
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{beomjoon,lpk,tlp}@mit.edu

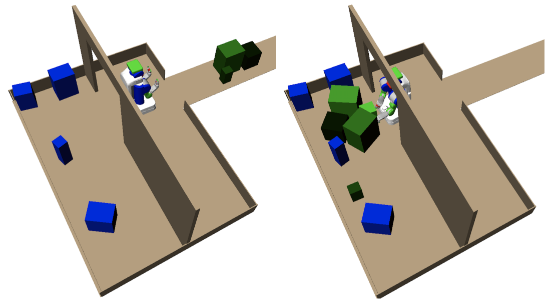
Abstract

We propose an actor-critic algorithm that uses past planning experience to improve the efficiency of solving robot task-and-motion planning (TAMP) problems. TAMP planners search for goal-achieving sequences of high-level operator instances specified by both discrete and continuous parameters. Our algorithm learns a policy for selecting the continuous parameters during search, using a small training set generated from the search trees of previously solved instances. We also introduce a novel fixed-length vector representation for world states with varying numbers of objects with different shapes, based on a set of key robot configurations. We demonstrate experimentally that our method learns more efficiently from less data than standard reinforcement-learning approaches and that using a learned policy to guide a planner results in the improvement of planning efficiency.

Introduction

A task and motion planning (TAMP) problem involves planning a sequence of low-level robot motions that achieves a high level objective, such as cooking a meal, by integrating low-level geometric reasoning with high-level symbolic reasoning. This typically requires a search in a high-dimensional hybrid space, for a sequence of high-level operators, each of which has both continuous and discrete parameters. Existing planners (Garrett, Kaelbling, and Lozano-Pérez 2017; Cambon, Alami, and Gravot 2009; Srivastava et al. 2014; Kaelbling and Lozano-Perez 2011) successfully tackle these problems, but they can be very computationally costly, because each node expansion typically requires a call to a motion planner to determine feasibility.

This paper proposes a learning algorithm that learns to guide the planner's search based on past search experience. We focus on an important subclass of TAMP called geometric-TAMP (G-TAMP), whose emphasis is on geometric reasoning rather than symbolic task reasoning. In G-TAMP, the primary interest is achieving a high-level objective by changing the poses of objects using collision-free motions, as shown in Figures 1a and 1b. Whether cooking, cleaning, or packing, all TAMP problems require geometric reasoning. Therefore, any efficiency improvements to G-TAMP problems, will benefit TAMP more broadly. For G-TAMP problems, choosing values for the continuous parameters of operators (such as grasps or object placements) is a crucial point of leverage. Choosing an infeasible value leads to a motion-planner call that will take a long time to fail. Even if the chosen value is feasible, but ultimately does not lead to a full solution, the planner will waste time searching.


(a) Conveyor belt domain: green objects must be packed in room.

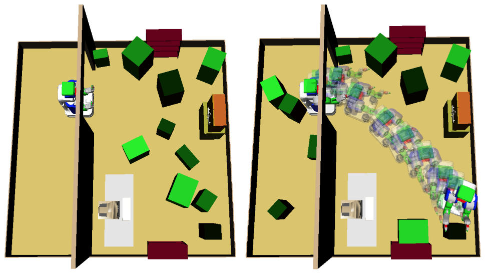

(b) Fetch domain: objects must be removed from path to target object.

Figure 1: Examples of initial (left) and goal states (right).

We propose to learn a policy that maps a representation of the planning state to the continuous parameters of an operator to be applied in that state. We leave the rest of the decision variables, such as collision-free motions and discrete operator parameters, to be filled in by the planner. This has many benefits from both learning and planning perspectives. For learning, this allows the policy to operate at a higher level, namely by specifying goals for the low-level motion planners. Such a policy can be learned from less experience than a policy for low-level robot motions. For planning, such a policy will constrain the planner's continuous search space to smaller yet promising regions, making the search more efficient. And, because we will use the policy to guide search, occasional poor action choices will not impact the correctness of the actions ultimately selected by the robot.

To learn a policy, we use a dataset of past planning experience. Our premise is that the planner generates operator sequences that yield high rewards. This is based on two observations. First, the planner only adds feasible operator instances to its search tree, which allow us to learn to exclude operator instances such as infeasible object placements. Second, a search tree usually contains states and actions that move closer to the goal based on a heuristic function. Even when the heuristic function is not optimal, this is far more effective than an unguided exploration strategy.

Therefore, we would like to learn a policy that imitates the planner's operator sequences from the search trees. However, this poses a challenge: from an episode of planning experience, we get a search tree, that contains one solution sequence and many *off-target* sequences that are feasible but did not get to a goal state (although they might have, had the planner continued expanding them). Since we have both off-target and solution sequences, we cannot simply train a policy that imitates the operator instances on these sequences.

Alternatively, we could completely rely on a reward signal and discard this planning experience dataset, but doing so would be extremely wasteful. Feasible state and operator sequences are a clearer learning signal than rewards from random exploration in a high-dimensional state-action space.

Therefore, we propose an actor-critic method that learns a policy by simultaneously maximizing the sum of rewards and imitating the planning experience. Our algorithm, Adversarial Monte-Carlo (ADMON), penalizes operator instances if they are too different from the planners sequences, while learning the Q-function of the sequences using past search trees. ADMON can be seen as a regularized policy learning procedure, where the regularization encourages imitating the planning data.

The learned policy operates on a representation of the planning state. G-TAMP problems usually involve varying number of objects with different shapes, and the ever-changing shape of the robot as it picks, places, and otherwise manipulates the objects in the environment. Thus, encoding the state as a fixed-size vector is not straightforward.

We propose a *key configuration* representation, inspired by the roadmap approach used in multi-query path planning. This representation copes with different number of objects and different robot and object shapes. Specifically, key configurations are a set of robot configurations that were used in past solutions. Given a new problem instance, the collision state at these key configurations approximates the relevant free-space for the robot. The learned policy uses this information to infer (approximate) reachability.

We evaluate ADMON and the key configuration representation in the two challenging G-TAMP problems shown in Figure 1 and described in detail in the Experiments section. We compare ADMON against several pure actor-critic methods, as well as a state-of-the-art imitation learning algorithm, and show that our method, by using both reward and demonstration signals, outperforms them in terms of data efficiency. We then apply the learned policy to guide planners, and show a significant improvement in planning efficiency.

Related work

In G-TAMP problems, a high-level objective is achieved by moving one or more objects. Several existing problem types in the literature can be seen as G-TAMP problems. The manipulation planning problems considered by Alami, Siméon, and Laumond (1989) and Siméon et al. (2004) also involve manipulating a few objects, albeit in more intricate ways. Garret et al. (2015) and Kaelbling et al. (2011) use heuristic search to solve problems with a large number of objects and a long horizon. Stilman et al. (2007) attack the "navigation among movable obstacles" (NAMO) problem, in which a robot moves objects out of the way to reach a target. Many of these approaches define high-level operators, and use random sampling to choose feasible continuous parameters of these operators. Our learning algorithm, ADMON, can be used with any of these planning algorithms to predict the continuous parameters. An alternative approach is to find the continuous parameters for a TAMP problem using optimization rather than sampling (Toussaint 2015); our method does not directly apply to such approaches.

ADMON can be seen as a variant of an actor-critic algorithm that uses extra data from past planning experience in addition to reward signals. In a standard actor-critic algorithm (Konda and Tsitsiklis 2003), a value function is first trained that evaluates the current policy, and a policy is trained by maximizing this value function. Actor-critic algorithms have traditionally been applied to problems with discrete action spaces, but recently they have been successfully extended to continuous action space problems. For example, in Proximal Policy Optimization (PPO) (Schulman et al. 2017), an off-policy actor-critic method, a value function is trained with Monte-Carlo roll-outs from the current policy. Then, the policy is updated based on an advantage function computed using this value function, with a clipping operator that prohibits a large changes between iterations. Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al. 2016) is another actor-critic algorithm that extends deep Q-learning to continuous action space by using a deterministic policy gradient. These methods have been applied to learning low-level control tasks such as locomotion, and rely solely on the given reward signal. Our method uses both past search trees and reward signals, and is applied to the problem of learning a high-level operator policy that maps a state to continuous parameters of the operators.

A number of other algorithms also use data from another source besides rewards to inform policy search. Guided policy search (GPS) (Levine and Abbeel 2014) treats data obtained from trajectory optimization as demonstrations, trains a policy via supervised learning, and enforces a constraint that makes the policy visit similar states as those seen in the trajectory optimization data. The key difference from our work is that we use search trees as guiding samples - in a search tree, not all actions are optimal, so we cannot simply use supervised learning. We instead propose an objective that simultaneously maximizes the sum of rewards while softly imitating the operator sequences in the search tree. Another important difference is that trajectory optimization requires smooth reward functions while the problems of interest to us have discontinuous reward functions. In approximate policy iteration from demonstrations (APID) (Kim et al. 2013), suboptimal demonstrations are provided in addition to reward signals, and the discrete action-space problems are solved using an objective that mixes large-margin supervised loss and policy evaluation. ADMON can be seen as an extension of this work, where the suboptimal demonstrations are provided by the past search trees, to continuous action spaces.

Since we use learning to guide a planning search process, our method is similar in spirit to AlphaGo (Silver et al. 2016), which uses learning to guide Monte-Carlo tree search in a large search space. The main difference in problem settings is that we have a continuous action space, and evaluating an edge in a search requires a call to a low-level motion planner, which can be very expensive.

There has been some work in learning to guide planning for TAMP problems as well. Kim et al. (2017) predict global constraints on the solution for generic TAMP problems using a scoring function to represent planning problem instances. In another work, Kim et al. (2018) develop an algorithm that learns a stochastic policy from past search trees using generative adversarial nets, for problems with fixed numbers of objects.

Problem formulation

We consider a subclass of TAMP whose main concern is geometric reasoning, called G-TAMP. In particular, we formulate the G-TAMP problem class as follows. Denote a robot as R, a set of fixed obstacles as $F_1, \cdots, F_p$, and a set of movable objects as $B_1, \cdots B_q$. An instance of a G-TAMP problem is defined by the given initial state s_0, the number of obstacles and objects, p and q, their shapes, and by a goal predicate $\mathcal{L}_G$ that is True if $s \in S$ is a goal state. For example, for the conveyor belt domain, $\mathcal{L}_G$ might be $Packed(S)$, that outputs True if all objects are in the storage room.

Solving this problem at the level of robot and object configurations is extremely difficult due to a long planning horizon and high dimensional state space. Due to the presence of dimensionality reducing constraints (Garrett, Kaelbling, and Lozano-Pérez 2017) such as grasping and placing, classic motion planning based on random sampling in the combined robot and object configuration space is not an option.

TAMP approaches define high-level operators, such as pick, to structure the search process. An operator $\mathfrak{o}$ is described by a set of input parameters, preconditions and effects. When the input parameters are given, the planner checks that the preconditions hold in the input state; this often involves testing for the existence of a path by calling a low-level motion planner, with a goal determined by the specified parameters.

We assume m operators $\mathcal{O} = \{\mathfrak{o}^{(1)}, \cdots, \mathfrak{o}^{(m)}\}$, where each operator $\mathfrak{o}$ is associated with discrete parameters, usually drawn from the set of movable objects, denoted by $\delta \in \Delta_{\mathfrak{o}}$, and continuous parameters, such as a base pose, denoted by $\kappa \in K_{\mathfrak{o}}$. An operator instance is defined as an operator that has fully instantiated input parameters, which we denote with $\mathfrak{o}(\delta, \kappa)$. A TAMP planner finds a sequence of operator instances that gets the robot and objects from the given initial state to a goal state.

Operator policy learning problem Suppose that we are given a planning experience dataset $\mathbf{D}_{\text{pl}} = \{\tau^{(i)}\}_{i=1}^{L}$, where each *operator sequence*, τ, from a search tree is a tuple $\{s_t, \mathfrak{o}_t(\delta_t, \kappa_t), r_t, s_{t+1}\}_{t=1}^{T}$ and $\mathfrak{o}_t \in \mathcal{O}$. Our objective is to learn an *operator policy* associated with each operator that maps a state to the continuous parameters of the operator, $\{\pi_{\theta_i}\}_{i=1}^{m}$, where θ_i is the parameters of the policy for the operator $\mathfrak{o}^{(i)}$, $\pi_{\theta_i} : S \to K_{\mathfrak{o}^{(i)}}$, that maximizes the expected sum of the rewards

$$\max_{\theta_1, \cdots, \theta_m} \mathbb{E}_{s_0 \sim P_0}\Big[\sum_{t=0}^{H} r(s_t, \kappa_t)\Big|\theta_1, \cdots, \theta_m\Big]$$

where $r(s_t, \kappa_t) = r(s_t, \mathfrak{o}_t(\delta_t, \kappa_t))$, and P_0 is the initial state distribution. Given a problem instance, we assume a task-level planner has given the operators and discrete parameters, and our goal is to predict the continuous parameters.

Adversarial Monte-Carlo

One way to formulate an objective for imitating the planning experience dataset $\mathbf{D}_{\text{pl}}$ is by using the adversarial training scheme (Goodfellow et al. 2014), where we learn a discriminator function $\hat{Q}_\alpha$ that assigns high values to operator instances from $\mathbf{D}_{\text{pl}}$ and low-values to operator instances generated by the policy. We will, for the purpose of exposition, consider a single operator setting to avoid the notational clutter. We have

$$\max_{\alpha} \sum_{s_i, \kappa_i \in \mathbf{D}_{\text{pl}}} \hat{Q}_\alpha(s_i, \kappa_i) - \hat{Q}_\alpha(s_i, \pi_\theta(s_i)) \tag{1}$$

$$\max_{\theta} \sum_{s_i, \kappa_i \in \mathbf{D}_{\text{pl}}} \hat{Q}_\alpha(s_i, \pi_\theta(s_i)) \tag{2}$$

These two objectives are optimized in an alternating fashion to train the policy that imitates the operator sequences in $\mathbf{D}_{\text{pl}}$. This can be seen as an application of Wasserstein-GAN (Arjovsky, Chintala, and Bottou 2017) to an imitation learning problem.

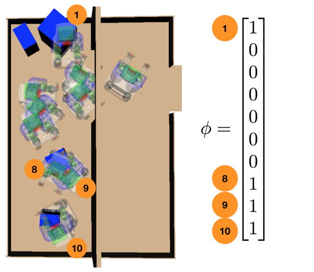
The trouble with this approach is that not all sequences in the search trees are equally desirable: we would like to generate operator instances that yield high values. So, we propose the following regularized policy evaluation objective that learns the value function from sequences in the search trees, but simultaneously penalizes the policy in an adversarial manner, in order to imitate the planner's operator sequences. We have

$$\min_{\alpha} \sum_{s_i, \kappa_i \in \mathbf{D}_{\text{pl}}} ||Q(s_i, \kappa_i) - \hat{Q}_\alpha(s_i, \kappa_i)||^2 + \lambda \cdot [\hat{Q}_\alpha(s_i, \pi_\theta(s_i))]$$

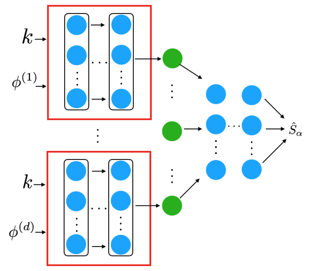
where $Q(s_i, \kappa_i) = r(s_i, \kappa_i) + \sum_{t=i+1}^{T} r(s_t, k_t)$ is the sum of the rewards of operator sequences in the search tree. We treat this as we would a value obtained from a Monte-Carlo rollout, and λ is used to trade off adversarial regularization versus accuracy in value-function estimation.

Algorithm 1 AdMon($\mathbf{D}_{\text{pl}}, \lambda, T_S, lr_\alpha, lr_\theta, n$)

```
for $t_s = 0$ to $T_s$ do
    // Train Q-value
    Sample $\{s_i, \kappa_i\}_{i=1}^n \sim \mathbf{D}_{\text{pl}}$ // a batch of data
    $dq = \nabla_\alpha \sum_{i=1}^n \left[(Q_i - \hat{Q}_\alpha(s_i, \kappa_i))^2 + \lambda \hat{Q}_\alpha(s_i, \pi_\theta(s_i))\right]$
    $\alpha = \alpha - \text{Adam}(lr_\alpha, dq)$
    // Train policy
    $dp = \nabla_\theta \left[\sum_{i=1}^n \hat{Q}_\alpha(s_i, \pi_\theta(s_i))\right]$
    $\theta = \theta + \text{Adam}(lr_\theta, dp)$
    $J_{t_s} = Evaluate(\pi_\theta)$
end for
return $\hat{Q}_\alpha, \pi_\theta$ with max $J_0, \cdots, J_{T_s}$
```



(a) An example scene and its ϕ.



(b) The architecture for $\hat{Q}_\alpha$

Figure 2

The pseudocode for our algorithm, Adversarial Monte-Carlo (AdMon), is given in Algorithm 1. The algorithm takes as inputs the planning experience dataset $\mathbf{D}_{\text{pl}}$, the parameter for AdMon, λ, the number of iterations, T_s, the learning rates for the Q-function, lr_α, and the policy, lr_θ, , and batch size n. It then takes a batch gradient descent step with the parameters of the Q-function, α, and then takes a batch gradient step with those of the policy, θ.

Adversarial training is known to have stability problems in its typical application of generating images, since evaluating image-generation policies is not simple. This is not an issue in our case. In AdMon, after each update of the policy parameters, we evaluate its performance using the $Evaluate$ function, which executes the given policy for a fixed number of time steps and returns the sum of the rewards. We then return the best performing policy.

Applying AdMon to task-and-motion planning

Key configuration based feature representation

We now describe a feature representation for a state s, ϕ, called key configuration obstacles, that captures essential collision information for many G-TAMP problems. It is constructed from data from a set of related of problem instances that have some aspects in common, such as locations of permanent obstacles, typical poses of other objects, or goals.

Critical to the success or failure of operator parameters that we generate for the planner is whether the motion plans they lead to are feasible (or solvable within a reasonable amount of time), which is ultimately determined by the free configuration space. However, representing the entire free configuration space using any regular discretization would be unthinkably expensive.

We, instead, construct a sparse, carefully sampled approximate representation of the free configuration space, by selecting a set of configurations that are important in our problem instance distribution. A distribution over problem instance induces a distribution over solutions, including particular robot configurations. Given a set of previous problem instances and plans that solve them, we consider the set of all the robot configurations that were attained and construct a subset of *key configurations* that has the property that, for any configuration in the original set, there is at least one key configuration near it, and the key configurations are not too close to one another. We denote each key configuration constructed this way as $\phi^{(i)}$.

Given a set of key configurations, we can construct a vector representation of the free configuration space in any state, as shown in figure 2a. For each key configuration $\phi^{(i)}$, shown as a semi-transparent robot, we check for collisions in that configuration in the current state: if it collides, the associated feature in a binary vector is set to 1; if not, the feature is set to 0. Some key configurations may involve the robot holding an object, which allows us to take collisions with the held object into account as well.

For $\hat{Q}_\alpha$, we use an architecture illustrated in figure 2b. For each key configuration $\phi^{(i)}$, it has a separate locally fully-connected network (shown in the red box—the number of layers and units per layer may be varied) that combines the continuous parameter κ with the free-configuration information, eventually generating a single output value shown as a green neuron; after that, the outputs of the green neurons are combined into layers of fully connected networks to generate the final output $\hat{Q}_\alpha$. Similar to a convolutional network, the weights in the local networks are "tied" so that they all represent the same function of k and $\phi^{(i)}$.

Using this network and feature representation, we find an interesting result: when the Q-function approximation $\hat{Q}_\alpha$ requires reasoning about paths to determine the value of a continuous parameter of an operator, such as an object placement, the network has learned to detect the key configurations near the relevant path. For example, one of our parameters for the conveyor belt domain is an object placement, which determines a robot base configuration for placing the object. When we select the 10 green neurons in Figure 2b with the highest activation levels for a particular object placement, we find that the corresponding key configurations, out of hundreds other key configurations, are those that would be on the path to the suggested object placement from the robot's fixed initial configuration. Some examples are shown in Figure 3. Different placements activate different sets of key configurations, suggesting that the value of a placement depends on the collision information of configurations important for reaching it.

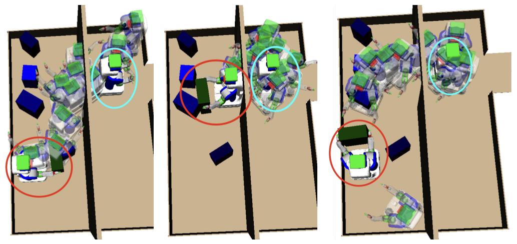


Figure 3: Key configurations that have top-10 activations for three different object placements (semi-transparent); initial configuration (cyan); final configuration (red).

Planning with the learned policy

Given a problem instance (s_0, s_g), we take an approach similar to (Lozano-Perez and Kaelbling 2014; Toussaint 2015) in that we try to find the continuous parameters of the plan conditioned on the given discrete plan, which is also known as a *plan skeleton*. The main difference is that we use the sampling-based graph search with the learned policy to guide the search.

Specifically, given a plan skeleton $\{\mathfrak{o}_1(\delta_1, \cdot), \cdots, \mathfrak{o}_T(\delta_T, \cdot)\}$, the sampling-based graph-search tries to find the continuous parameters of the operators, $\kappa_1, \cdots, \kappa_T$, to make the discrete plan successful. The graph search proceeds as follows.

At the root node, the search algorithm first generates m operator instances of $\mathfrak{o}_1(\delta_1, \cdot)$ by sampling the instances of the parameters κ_1, and adds their successor states to the search agenda. It then pops the node from the agenda with the lowest heuristic value (estimated cost to reach the goal), and the search continues until we arrive at a goal state. If at the current node we cannot sample any feasible operators, then we discard the node and continue with the next node in the agenda. The root node is always added back to the queue after expansion, in order to guarantee completeness.

Typically, a uniform stochastic policy is used as a default choice to sample the continuous parameters, which guarantees probabilistic completeness. We use instead the learned policy to sample the continuous parameters.

Experiments

We evaluate ADMON in two practical and challenging G-TAMP problems, and compare against three benchmarks: PPO, DDPG, which are actor-critic algorithms for continuous action spaces, and Generative Adversarial Imitation Learning GAIL, a state-of-the-art inverse reinforcement learning algorithm that treats the planning experience dataset as optimal demonstrations, and then uses PPO to find a policy that maximizes the learned rewards. For DDPG, we use an episodic variant that defers updates of the policy and replay buffer to the end of each episode, which makes it perform better in our inherently episodic domain. It is important to keep some level of stochasticity in any policy we learn, because there is a large volume of infeasible operator instances for which no transition occurs. So, we use a Gaussian policy with a fixed variance of 0.25, and use the learned policies to predict only the mean of the Gaussian. The hyperparameters and architectures of the actor and critic neural networks are included in the appendix.

Our hypotheses are that (a) ADMON, by using the planning experience dataset $\mathbf{D}_{\text{pl}}$, can learn more data efficiently than the benchmarks, and (b) learning these policies can improve planning efficiency. To test the first hypothesis, we show two plots. The first is the learning curve as a function of the size of $\mathbf{D}_{\text{pl}}$, with a fixed number of interactions with the simulated environment for the RL methods. For the RL methods, $\mathbf{D}_{\text{pl}}$ is used as an initial training set. For ADMON, simulations are only used for the evaluation of the current policy. For this plot, we fix the amount of RL experience at 30000 for the conveyor belt domain, and 15000 for the object fetching domain; these are obtained from 300 updates of the policy and value functions of each algorithm, where for each update, we do 5 roll-outs, each of which is 20 steps long for the first domain and 10 steps long for the second domain. We report the performance of the best policy from these 300 updates, averaged over four different random seeds. Second is the learning curve with increasing amount of simulated RL experience, with fixed $\mathbf{D}_{\text{pl}}$ size. We fix its size at 100 for the first domain and 90 for the second domain. For testing hypothesis (b), we show the improvement in planning efficiency when we use the best policy out of all the ones used to generate the first two plots to guide a planner.

Domain overview Our objective is to test the generalization capability of the learned policy across the changes in the poses and shapes of different number of objects in the environment, while the shape of the environment stays the same. In the first domain, the robot's objective is to receive either four or five box-shaped objects with various sizes from a conveyor belt and pack them into a room with a narrow entrance, already containing some immovable obstacles. A problem instance is defined by the number of objects in the room, their shapes and poses, and the order of the objects that arrive on the conveyor belt. The robot must make a plan for handling all the boxes, including a grasp for each box, a placement for it in the room, and all of the robot trajectories. The initial base configuration is always fixed at the conveyor belt. After deciding the object placement, which determines the robot base configuration, a call to an RRT motion planner is made to find, if possible, a collision-free path from its fixed initial configuration at the conveyor belt to the selected placement configuration. The robot cannot move an object once it has placed it. This is a difficult problem that involves trying a large number of infeasible motion planning problems, especially if poor sampling is used to sample continuous operator parameters.

In the second domain, the robot must move to fetch a target object and bring it back to the robot's initial location. The target object is randomly placed in the bottom half of the room, which is packed with both movable and immovable obstacles. A problem instance is defined by the poses and shapes of the target and movable obstacles. The robot's initial configuration is always fixed, but it needs to plan paths from various configurations as it picks and places objects. To reach the target object, the robot must pick and place movable obstacles along the way, each of which involves a call to the motion planner in a relatively tight environment. We have set the problem instance distribution so that the robot must move five to eight objects to create a collision-free path to fetch the target object. This problem is a generalization of the navigation among movable obstacle problem (Stilman et al. 2007) to a mobile-manipulation setting.

In both domains, if the selected operator instance is infeasible due to collision or kinematics, the state does not change. Otherwise, the robot picks or places the selected object with the given parameters. The reward function for the conveyor belt domain is 1 if we successfully place an object into the room and 0 otherwise. For the fetching domain, the reward function is 0 if we successfully pick an object, -1 if we try an infeasible pick or place, and 1 if we successfully move an object out of the way. For moving all the objects out of the way, the robot receives a reward of 10.

Task-level planner and sampling procedures At a high-level, the discrete operator parameters for both of the domains consist of the sequence of objects to be picked and placed. Specifically, two operators are given to the planner, *pick* and *place*, each of which uses two arms to grasp a large object. Table 1 summarizes the parameters of each operator.

To provide guidance, we use the policies trained with different learning algorithms. The inputs to the policy are described in the table as well. Each operator will generally require a call to a motion planner to find a collision-free path from the previous configuration.

The task-level plan for both domains is the sequence of objects to be pick-and-placed. For the conveyor domain, this is given by the problem instance definition: the objects arrive in the order to be packed. For the fetching domain, we implement a swept-volume approach similar to (Dogar and Srinivasa 2011; Stilman et al. 2007). We first plan a fetching motion to the target object assuming there are no movable obstacles in the scene. Then, we check the collision between this path and the moveable obstacles to identify objects that need to be moved.

To sample parameters for the *pick* operation using the default uniform policy:

1. Sample a collision-free base configuration, (x^o_r, y^o_r, ψ^o_r), uniformly from a circular region of free configuration space, with radius equal to the length of the robot's arm, centered at the location of the object.
2. With the base configuration fixed at (x^o_r, y^o_r, ψ^o_r) from the object, sample (d, h, χ), where d and z has a range $[0.5, 1]$, and χ has a range $[\frac{\pi}{4}, \pi]$, uniformly. If an inverse kinematics (IK) solution exists for both arms for this grasp, proceed to step 3, otherwise restart.
3. Use bidirectional RRT (biRRT), or any motion planner, to get a path from the current robot base configuration to (x^o_r, y^o_r, ψ^o_r) from the pose of the object. A linear path from the current arm configuration to the IK solution found in step 2 is then planned.

If a collision is detected at any stage, the procedure restarts. When we use the learned policy π_θ, we simply draw a sample from it, and then check for IK solution and path existence with the predicted grasp and base pose.

For the conveyor belt domain, we assume that the conveyor belt drops objects into the same pose, and the robot can always reach them from its initial configuration near the conveyor belt, so we do not check for reachability. For the object fetch domain, we do all three steps.

From a state in which the robot is holding an object, it can place it at a feasible location in a particular region. To sample parameters for *place* using the default uniform policy:

1. Sample a collision-free base configuration, (x, y, ψ), uniformly from a desired region.
2. Use biRRT from the current robot base configuration to (x, y, ψ).

To use π_θ, we sample base configurations from it in step 1.

For heuristic function for the continuous-space graph search in the fetching domain, we use the number of objects to be moved out of the way as a heuristic. For the conveyor belt domain, we use the remaining number of objects to be packed as a heuristic.

To collect a dataset $\mathbf{D}_{\text{pl}}$, we use search trees constructed while solving previous planning problems. To create a operator sequence τ from a search tree, we begin at the root and collect state, action, and rewards up to each leaf node.

Results for the conveyor belt domain Figure 4 (left) shows the learning curve as we increase the number of search trees. Each search tree from a problem instance adds at most 50 (state, reward, operator instance, next state) tuples. The RL algorithms, DDPG and PPO, have rather flat learning curves. This is because they treat the planning experience dataset $\mathbf{D}_{\text{pl}}$ as just another set of roll-outs; even with 100 episodes of planning experience, this is only about 5000 transitions. Typically, these methods require tens of thousands of data to work well. ADMON, on the other hand, makes special use of $\mathbf{D}_{\text{pl}}$ by trying to imitate the sequences. On the other hand, the results from GAIL show that it is ineffective to treat $\mathbf{D}_{\text{pl}}$ as optimal demonstrations and simply do imitation. ADMON, which uses reward signals to learn a Q-function in addition to imitating $\mathbf{D}_{\text{pl}}$, does better.

Figure 4 (middle) shows the learning curves as we increase the amount of transition data, while fixing the number of search trees at 100. Again, ADMON outperforms the RL algorithms. Note that the RL approaches, DDPG and PPO, are inefficient in their use of the highly-rewarding $\mathbf{D}_{\text{pl}}$ dataset. For instance, PPO, being an on-policy algorithm, discards $\mathbf{D}_{\text{pl}}$ after an update. Even though the transition data collected after that is much less informative, since it consists mostly of zero-reward transitions, it makes an update based solely on them. As a result, it tends to fall into bad local optima, and the learning curve saturates around 3000 steps. The situation is similar for the off-policy algorithm DDPG. It initially only has $\mathbf{D}_{\text{pl}}$ in its replay buffer, but as it collects more data it fills the buffer with zero-reward sequences, slowing the learning significantly after around 5000 steps. ADMON, on the other hand, is able to better exploit the planning experience dataset to end up at a better (local) optima. GAIL shows slightly better performance than PPO and

Operators	**Continuous Parameters**	**Inputs to π_θ (conv belt)**	**Inputs to π_θ (obj fetch)**	**call to RRT (conv belt)**	**call to RRT (obj fetch)**
Pick	$(x^o_r, y^o_r, \psi^o_r), (d, h, \chi)$	Learned π_θ not used	$\phi_{fetch}, \phi, (x_o, y_o, \psi_o), (l, w, h)$	No	Yes
Place	(x, y, ψ)	ϕ	$\phi_{fetch}, \phi, (x_o, y_o, \psi_o)$	Yes	Yes

Table 1: Operator descriptions. (x, y, ψ) refers to a robot base pose, at (x,y) location and rotation ψ in the global frame, (x^o_r, y^o_r, ψ^o_r) refers to the relative robot base pose with respect to the pose of an object o, whose pose in global frame is (x_o, y_o, ψ_o). (d, h, χ) is a grasp represented by a depth, as a portion of size of object in the pushing direction, height, as a portion of object height, and angle in the pushing direction, respectively, and (l, w, h) represents the length, width, and height of object being picked. ϕ_{fetch} is a fetching path represented with key configurations. We describe this in detail in the appendix.

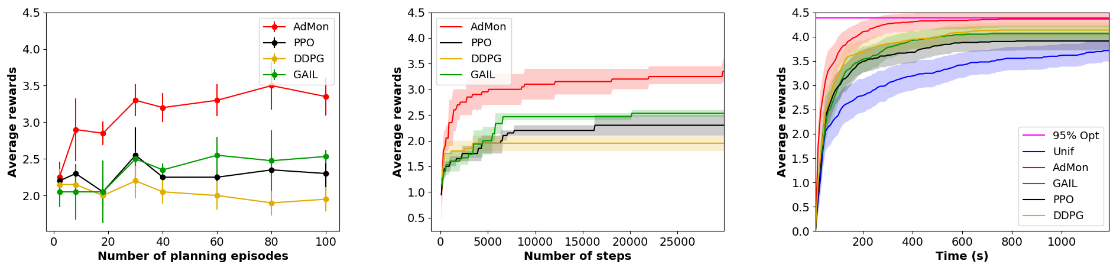


Figure 4: Plots for conveyor belt domain

DDPG. It tends to escape bad local optima by learning a reward function that assigns high rewards to the planning experience dataset $\mathbf{D}_{\text{pl}}$, but still performs worse than ADMON because it treats $\mathbf{D}_{\text{pl}}$ as optimal demonstrations.

Figure 4 (right) shows the reduction in planning time achieved by different learning algorithms. We can see that, after about 400 seconds, ADMON achieves 95% optimal performance, whereas the uniform policy still have not achieved that performance after 1200 seconds, indicating a speed up of at least 3.

Results for object fetch domain Figure 5 (left) shows the learning curve as we increase the number of search trees. Each search tree adds at most 50 (state, operator instance, next state) tuples, up to 25 of which use pick operator instances, and the remaining are place operator instances. Again, the RL methods show weaker performance than ADMON although this time they are closer. The poor performance of GAIL is due to $\mathbf{D}_{\text{pl}}$ containing many more off-target operator sequences than before, due the longer horizon. Since most of these sequences are not similar to the solution sequence, treating these data points as optimal demonstrations hurts the learning.

Figure 5 (middle) shows the learning curve as we increase the number of transitions, while fixing the number of search trees at 90. PPO shows large variance with respect to different random seeds, and on average, shows a very steep learning curve at the beginning, but it gets stuck at a bad local optima. DDPG shows good performance, but still performs worse than ADMON. GAIL fails to learn anything meaningful due to the previously stated reasons.

Figure 5 (right) shows the impact on the planning efficiency when trained policies are used to choose the continuous parameters. This time, we plot the progress, measured by the number of objects cleared from the fetching path for different time limits. We can see that ADMON clears the optimal number of objects at around 1500 seconds, and the uniform policy takes 3500 seconds, an improvement in planning efficiency by a factor of more than 2.3.

DDPG initially performs just as well as ADMON until it clears 3 objects, but its improvement stops after this point. This is due to the current-policy-roll-out exploration strategy used by DDPG. It is very unlikely with this strategy to encounter an episode where it clears more than 3 objects. When used with the planner, which uses a heuristic, the policy starts encountering states that have more than three objects cleared, leading to poor performance. This phenomenon, where there is a discrepancy between the distribution of states encountered during training and testing, is also noted in (Ross, Gordon, and Bagnell 2011). ADMON on the other hand does not have this problem because it is trained with the search trees produced by the planner. Note also the decrease in planning efficiency when using poor policies.

Conclusion

In this work, we proposed an actor-critic algorithm that learns from planning experience to guide a planner, using key configuration features. Our experiments shows that ADMON is more data efficient than benchmarks since it uses both reward signals and the past search trees. We also demonstrated that by using the learned policy, we can achieve a substantial improvement in planning efficiency in challenging and practical G-TAMP problems.

Acknowledgement

We thank KakaoBrain for the GPU support. We gratefully acknowledge support from NSF grants 1523767 and 1723381; from AFOSR grant FA9550-17-1-0165; from

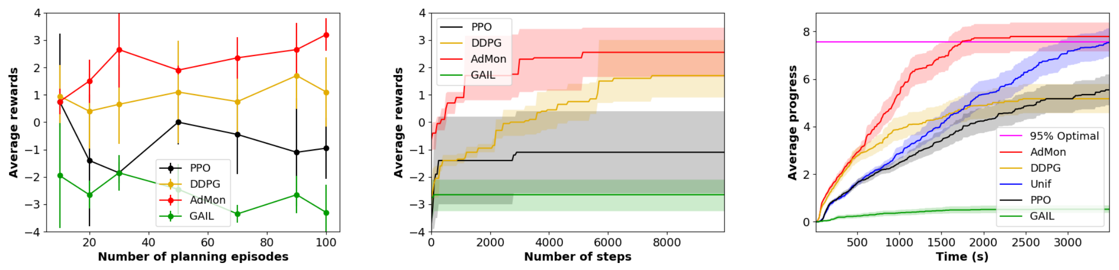


Figure 5: Plots for object fetch domain

ONR grant N00014-18-1-2847; from Honda Research; and from Draper Laboratory.

References

Alami, R.; Siméon, T.; and Laumond, J. 1989. A geometrical approach to planning manipulation tasks: The case of discrete placements and grasps. *International Symposium on Robotics Research*.

Arjovsky, M.; Chintala, S.; and Bottou, L. 2017. Wasserstein generative adversarial networks. *International Conference on Machine Learning*.

Cambon, S.; Alami, R.; and Gravot, F. 2009. A hybrid approach to intricate motion, manipulation, and task planning. *International Journal of Robotics Research*.

Dogar, M., and Srinivasa, S. 2011. A framework for push-grasping in clutter. *Robotics: Science and systems*.

Garrett, C. R.; Kaelbling, L. P.; and Lozano-Pérez, T. 2017. Sample-based methods for factored task and motion planning. *Robotics: Science and Systems*.

Garrett, C. R.; Lozano-Pérez, T.; and Kaelbling, L. P. 2015. Backward-forward search for manipulation planning. *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Goodfellow, I.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; and Bengio, Y. 2014. Generative adversarial nets. *Advances in Neural Information Processing Systems*.

Kaelbling, L. P., and Lozano-Perez, T. 2011. Hierarchical task and motion planning in the now. *IEEE Conference on Robotics and Automation*.

Kim, B.; Farahmand, A.-M.; Pineau, J.; and Precup, D. 2013. Learning from limited demonstrations. *Advances in Neural Information Processing Systems*.

Kim, B.; Kaelbling, L. P.; and Lozano-Pérez, T. 2017. Learning to guide task and motion planning using score-space representation. *IEEE Conference on Robotics and Automation*.

Kim, B.; Lozano-Perez, T.; and Kaelbling, L. 2018. Guiding search in continuous state-action spaces by learning an action sampler from off-target search experience. *AAAI Conference on Artificial Intelligence*.

Konda, V. R., and Tsitsiklis, J. N. 2003. On actor-critic algorithms. *SIAM Journal on Control and Optimization*.

Levine, S., and Abbeel, P. 2014. Learning neural network policies with guided policy search under unknown dynamics. *Advances in Neural Information Processing Systems*.

Lillicrap, T. P.; J. J. Hunt, A. P.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2016. Continuous control with deep reinforcement learning. *International Conference on Learning Representations*.

Lozano-Perez, T., and Kaelbling, L. 2014. A constraint-based method for solving sequential manipulation planning problems. *IEEE/RSJ International Conference on Intelligent Robots and Systems*.

Ross, S.; Gordon, G.; and Bagnell, D. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. *International Conference on Artificial Intelligence and Statistics*.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *arXiv*.

Silver, D.; Huang, A.; Maddison, C.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*.

Siméon, T.; Laumond, J.-P.; Cortés, J.; and Sahbani, A. 2004. Manipulation planning with probabilistic roadmaps. *International Journal of Robotics Research*.

Srivastava, S.; Fang, E.; Riano, L.; Chitnis, R.; Russell, S.; and Abbeel, P. 2014. Combined task and motion planning through an extensible planner-independent interface layer. *IEEE Conference on Robotics and Automation*.

Stilman, M.; Schamburek, J.-U.; Kuffner, J.; and Asfour, T. 2007. Manipulation planning among movable obstacles. *IEEE International Conference on Robotics and Automation*.

Toussaint, M. 2015. Logic-geometric programming: An optimization-based approach to combined task and motion planning. *International Joint Conference on Artificial Intelligence*.