Learning Proposal Distributions for Refining High Level Plans

Rohan Chitnis[1], Dylan Hadfield-Menell[2], and Pieter Abbeel[2]

***Abstract*— A key challenge in robotics is the execution of long-horizon tasks. These require the robot to reason through a long sequence of high level steps in order to reach a goal, as well as determine appropriate joint trajectories for executing each step. Recent work has been devoted to hierarchical planning, which seeks to plan for such goals by combining task and motion planning. These planning systems attempt to refine symbolic plans by searching for a feasible assignment of continuous values to the symbolic parameters. A core limitation of task and motion planning systems is the manner in which values are sampled for the search: using hand-coded discretizations that leverage specificity about the geometry of the environment and its objects. In this paper, we apply techniques in reinforcement learning to train a system that produces good proposal distributions for sampling plan parameter values. Our contributions are: 1) a machine learning approach to find sampling distributions for the refinement of symbolic plans in a hierarchical planning system; 2) a novel strategy for refining these symbolic plans based on randomization, which we found integrates well with our learning algorithm. Our experiments demonstrate comparable to improved performance in motion planning time and number of motion planner calls in a variety of test environments, when compared to the existing infrastructure that uses hand-tuned discrete sampling distributions.**

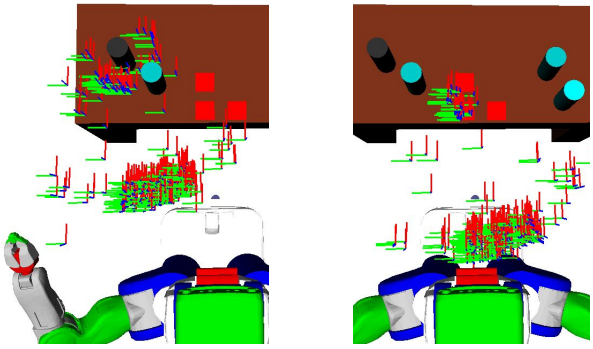

Fig. 1: Screenshots showing some distributions learned by our system in one experimental setup. The robot is tasked with grasping the black cylinder and putting it down on the central red square. The left image shows learned base motion and grasping distributions, and the right shows learned base motion and putdown distributions. The grasp distribution properly learned to avoid the region close to the blue obstruction. We sample from these distributions to refine the high level plan, rather than relying on hand-coded ones.

I. Introduction

A major long-term goal of robotics research is the introduction of intelligent household robots. Such robots would face complex tasks, such as setting dinner tables and doing laundry. Planning for these long-horizon tasks is infeasible using only motion planning due to the sheer number of steps involved, making apparent the need for a hierarchical system of reasoning.

Recent methods for hierarchical planning focus on the intersection of high level task planning and low level motion planning. In this framework, the (classical) task planner produces a symbolic plan containing a sequence of actions to reach a goal state, and an interface layer refines this plan by sampling concrete values for the abstract symbols, thus grounding the plan. A central challenge in building such an interface layer is designing good distributions from which to sample values for the symbolic plan parameters.

Our work improves directly upon recent approaches to task and motion planning. A key limitation of these systems is that they typically rely on hand-coded sampling distributions to instantiate continuous values for grounding a symbolic plan. These distributions often rely on geometric information about the object of interest, surrounding objects, and overall environment. This restriction has several negative implications: the parameters of the hand-coded distributions must be fine-tuned when running the system in a new setting, and the resulting refinement distributions are discretized, lacking robustness.

Our main contribution is a reinforcement learning approach to find good proposal distributions for symbolic plan refinement, using rollouts. We take inspiration from the work of Zhang and Dietterich, who explored applying reinforcement learning to planning problems in a job shop scheduling setting. Using methods adapted from Zucker et al., who train a configuration space sampler for motion planning using features of the discretized workspace, we train a policy that determines how to sample values for plan symbols, using $TD(\lambda)$ with linear function approximation. Our approach allows the learned distributions to be continuous, robust to changes in the environment, and trainable for any experimental setting, eliminating the need for hand-tuning distribution parameters.

A second contribution of our work is randomized refinement, a novel refinement strategy which maintains at all times a set of values for all symbolic plan variables, then randomly resamples one whose current instantiation is causing a failure. This refinement algorithm allows for a clear formulation of the reinforcement learning system and naturally guides variable resampling intelligently, so that better sampling distributions can be learned more easily for complex environments.

[1] ronuchit@berkeley.edu
[2] {dhm, pabbeel}@eecs.berkeley.edu

Our implementation builds directly off the system presented by Srivastava et al. They propose a complete backtracking search algorithm for refining the symbolic plan skeleton into a set of collision-free trajectories using motion planning. If a motion planning feasible refinement is not found within the resource limit, symbolic error information is propagated back to the task planner, and a new symbolic plan is produced. The system uses an off-the-shelf classical task planner and motion planner, both as black boxes. We evaluate our approach in a variety of test environments involving complicated grasp and putdown actions with obstructions and base motion. Our experimental results demonstrate comparable to improved performance for both motion planning time and number of calls to the motion planner, when compared to the hand-coded distributions used in the original system.

II. Related Work

Prior work has been devoted to using machine learning techniques for training heuristics to guide search algorithms. This general formulation has been applied to many domains, such as large-scale optimization, solving constraint satisfication problems, and controlling state-space beam search in classical task planners. To our knowledge, our work is the first to combine reinforcement learning with a task and motion planning system, in order to find good proposal distributions for grounding a symbolic plan.

III. Background

A. Reinforcement Learning for Planning

A reinforcement learning problem is typically formulated as a Markov decision process (MDP), defined by a state space S; action space A; transition function $P(s'|s,a)$ for $s, s' \in S, a \in A$; reward function $R(s,a,s')$; and discount factor $\gamma \in [0,1]$. The goal is to train a policy that selects which action $a \in A$ to execute given a current state $s \in S$, in order to maximize the sum of expected rewards. Zhang and Dietterich present a technique for formulating the job shop scheduling problem using the reinforcement learning framework. They define the following: S is all temporal configurations and resource allocations of the jobs; A is changing either the time or resource allocation of a single job; R is a measure of schedule goodness based on its length relative to the difficulty of the scheduling problem.

B. Task and Motion Planning

We summarize the work of Srivastava et al. A fully observed task planning problem is a tuple $(\mathcal{O}, \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A})$:

$\mathcal{O}$ is a set of symbolic references to *objects* in the environment, such as cylinders, cylinder locations, and grasping poses. (These are treated as strings.)

$\mathcal{F}$ is a set of fluents (which define relationships among objects and are either true or false in the state).

$\mathcal{I}$ is the set of fluents that hold true in the initial state (all others are assumed false).

$\mathcal{G}$ is the set of fluents defining the goal state.

$\mathcal{A}$ is a set of parametrized *actions*, defined by *preconditions* (a set of fluents characterizing what must hold true in the current state for the action to be validly peformed) and *effects* (a set of fluents that hold true after the action is performed).

A solution to the task planning problem is then a sequence of actions $a_0, a_1, ..., a_n \in \mathcal{A}$ which, when applied to the state of fluents successively starting with $\mathcal{I}$, results in a state in which all fluents of $\mathcal{G}$ hold true. We refer to this sequence of actions as a *symbolic*, or *high level*, plan: the task planner maintains no knowledge about the geometric meaning behind any of the symbols.

We emphasize that any continuous variables of this plan, such as grasping poses for objects, are abstracted into symbols representing their nature. This allows a classical planner to produce solution plans without any need to interpret these symbols. An *interface layer* is then responsible for assigning continuous values to the high level plan variables and discovering facts about the environment, sending them back to the task planner to be added into the fluent state. *Refinement* is the process of assigning such continuous values to the abstract plan variables, grounding the plan.

For example, consider a pick domain with a target object o_t to be grasped and a single obstruction. Initially, the task planner returns a high level plan to immediately grasp o_t, because it is not yet aware of the obstruction. The interface layer then samples grasp poses for o_t, but motion planning for each sample fails due to the obstruction. Eventually, this error information about the obstruction is converted into a symbolic representation and propagated back to the task planner. The new high level plan correctly grasps the obstruction before grasping o_t, and refining this plan succeeds.

Srivastava et al. introduce a refinement algorithm called TryRefine, an exhaustive backtracking routine for finding a motion planning feasible set of instantiations of the high level plan variables. The algorithm progresses by sampling from the refinement distribution associated with each high level plan variable incrementally; it backtracks to resample the previous one each time the sample space for a particular variable is exhausted. The completeness of this backtracking strategy, thus, depends on the refinement distribution being composed of a discrete and finite set of values. Furthermore, it does not take into consideration the particular causes of infeasibility at each iteration.

IV. Randomized Refinement

In order to move toward refining high level plans using continuous sampling distributions, we present one contribution of our work: a novel algorithm called randomized refinement, which resamples plan variables based on which values are leading to motion planning failures or violations of action preconditions. We thus leverage information about action precondition fluents explicitly, which backtracking refinement does not take into account. In the next section, we will show how this new refinement strategy can be easily augmented with a reinforcement learning system to train good distributions for sampling values for plan variables. Algorithm 1 shows pseudocode for randomized refinement.

The procedure takes two arguments, the high level plan object and a maximum iteration count. In Line 2, all variables in the high level plan are initialized by sampling from the corresponding distributions. For efficiency, we only initialize the symbolic pose references (such as object grasping poses) with IK-feasible values, so as to avoid unnecessary calls to the motion planner. We then call the MOTIONPLAN subroutine in Line 4, which iterates through the sequence of actions that comprise the high level plan. For each action, a call to the motion planner is made (Line 22) using the instantiated values for the parameters of that action, to attempt to find a trajectory linking the sampled poses. If this succeeds, the preconditions of the action are tested (Line 26). As part of this step, we verify that the trajectory returned by the black box motion planner is collision-free, satisfying the precondition that the trajectory is feasible in the environment. Thus, based on the returned values of MOTIONPLAN, we may distinguish a motion planning failure from a precondition violation, for the current refinement. We appropriately call the `resample` routine on the high level parameters associated with the failure; this routine picks one of the parameters at random and resamples it from its refinement distribution. Again for efficiency, we keep resampling until we have an IK-feasible value. If we reach the iteration limit (Line 18), we convert the most recent failure information into a symbolic representation, then raise it to the classical planner, which will update its fluent state and provide a new high level plan.

We emphasize the simplicity of randomized refinement over backtracking refinement. Essentially, we maintain a set of instantiations over all high level variables at all times, and we perform local resampling on the variable values based on failure information. This technique provides several benefits over a backtracking system. First, it remains unchanged in performance when moving from discretized to continuous sampling distributions, while backtracking loses completeness and requires introduction of heuristics dictating when to stop sampling for a particular variable and backtrack to a previous one. Additionally, it allows the parameter instantiations for a particular action in the plan to be influenced by those for a *future* action. (For example, in a pick-and-place task, it can make sense for the object's grasp pose to be sampled conditionally on the current instantiation of the putdown pose, even though the putdown appears after the grasp in the plan sequence.) Randomized refinement does, however, require explicit storage of predicates in the interface layer, so that action preconditions can be checked within the environment to determine which parameters get resampled next.

Algorithm 1 Randomized refinement.

```
1:  procedure RANDREF(HLP, N)
2:      init ← initRefinement(HLP)
3:      for iter = 0, 1, ..., N do
4:          failStep, failPred ←MOTIONPLAN(HLP)
5:          if failStep is null then
6:              return success
7:          else
8:              if failPred is null then
9:                  # Motion planning failure.
10:                 failAction ← HLP.ops[failStep]
11:                 resample(failAction.params)
12:             else
13:                 # Action precondition violation.
14:                 resample(failPred.params)
15:             end if
16:         end if
17:     end for
18:     Raise failure to task planner, receive new plan.
19: end procedure
20: procedure MOTIONPLAN(HLP)
21:     for op_i in HLP.ops do
22:         res ← op_i.motionPlan()
23:         if res.failed then
24:             return i, null
25:         else
26:             failPred ← op_i.checkPreconds()
27:             if failPred then
28:                 return i, failPred
29:             else
30:                 # Found successful plan refinement.
31:                 return null, null
32:             end if
33:         end if
34:     end for
35: end procedure
```

V. LEARNING REFINEMENT DISTRIBUTIONS

A. Formulation and Training Process

In this section, we present our primary contribution: a method for training good sampling distributions using reinforcement learning. Formally, our training system is defined by the following:

$\mathcal{P}$: A distribution over planning problems, encompassing both the task planning problem and the environment.

$\theta_a, a \in \mathcal{A}$: The learned feature weight vector associated with the high level action a, which defines the sampling distribution for the parameters of a. In our formulation, there is one weight vector per action *type* (e.g. "left gripper grasp," "right gripper grasp," or "base motion"), so that the learned distributions are not tied to any particular high level plan.

$\mathcal{R}$: The reward function.

$f(a, s_0, ..., s_n), a \in \mathcal{A}$: A function mapping the sampled parameter values $s_0, ..., s_n$ of action a to a feature vector representation.

N: The number of planning problems on which to train.

L: The number of samples for one planning problem.

ϵ: The number of samples comprising an episode. We perform a weight vector update after each episode.

Our system attempts to learn the feature weights θ for each action type; these weights encode a policy that maps the state (comprised of the planning problem $\Pi \in \mathcal{P}$ and the symbolic plan) to a set of actions governing plan refinement.

The training is a natural extension of randomized refinement and progresses as follows. N times, sample from $\mathcal{P}$ to obtain a complete planning problem Π. For each Π, run the randomized refinement algorithm to attempt to find a valid plan refinement, allowing the `resample` routine to be called L times. As the refinement process runs, any collected reward is appended to a global reward list. If a valid, motion planning feasible refinement is found, we collect appropriate rewards, then resample one of the high level variables at random (potentially making the refinement infeasible) and continue. Every ϵ calls to `resample`, we perform a gradient update on the weight vectors using the collected rewards and samples since the previous update. This process evinces the strength of randomized refinement for our system: only the variable instantiations responsible for failures are resampled, so the learning is naturally focused toward improving sampling distributions which are not yet well-shaped.

B. Distribution and Weight Updates

We now describe the particular distribution we use to sample and how to perform batch gradient updates for the weight vector of this distribution. Zucker et al. use features of a discretization of the workspace to train a configuration space sampler for motion planning, and we adapt their method to the case of a continuous distribution. Accordingly, we define the sampling distribution for action a as

$$q(\theta_a, s) \propto exp(\theta_a^T f(a, s))$$

(The probability distribution must be appropriately normalized so that it integrates to 1.) A useful facet of this distribution is that when θ_a is the zero vector, q is uniform over the sample space, providing a good initialization for the weight vectors. We sample from q using the Metropolis algorithm, an MCMC technique for sampling from a probability distribution using a symmetric proposal distribution. (At each iteration, the sampled value is based only on the previous one, so the samples form a Markov Chain.)

Zucker et al. define the expected reward of an episode e:

$$\eta(\theta_a) = E_q[\mathcal{R}(e)]$$

and provide an approximation for its gradient:

$$\nabla\eta(\theta_a) \approx \frac{\mathcal{R}(e)}{\epsilon} \sum_{i=0}^{\epsilon} (f(a, s_i) - E_q[f])$$

where $\mathcal{R}(e)$ is the sum over all rewards obtained throughout the episode, and $E_q[f]$ is the expected feature vector under q. Because our distribution is continuous, we cannot easily calculate $E_q[f]$, and so we approximate it by averaging together the feature vector for several samples from q. The weight vector update is then:

$$\theta_a \leftarrow \theta_a + \alpha\nabla\eta(\theta_a)$$

for appropriate step size α.

C. Reward and Feature Functions

Our reward function $\mathcal{R}$ provides small constant rewards based on the ongoing plan refinement. Because we check for IK feasibility of samples during the initialization step and when resampling, we provide a small negative reward every time we sample an IK-infeasible pose, and a small positive reward whenever an IK check succeeds. Similarly, we add negative reward upon motion planning failure or action precondition violation, and positive reward upon a successful call to the motion planner. Finally, we assign a large reward for finding a complete, valid refinement of the high level plan. This choice of reward function implies that our training optimizes the sampling distributions for motion planning feasibility, rather than explicitly for refining plans. However, the two are closely correlated – plan refinement takes fewer iterations when variable instantiations are likely to be motion planning feasible. Our feature function f considers only geometric aspects of the sample, such as distance between the sample and object being grasped, and distance from the sample to nearby obstructions.

VI. Experiments

We evaluate our reinforcement learning system in several pick-and-place tasks, varying the number of obstructions, obstruction locations, and whether the base is allowed to move. Because the distributions are optimized for motion planning successes, we do not evaluate performance on total system running time; rather, we evaluate the refinement of only the final high level plan. This is because, if we have a high level plan with no valid refinement, our system may find more IK-feasible samples (and thus trigger more calls to the motion planner) than the original system, causing increased time spent on refinement attempts before raising error information back to the task planner. The information propagation occurs when the iteration limit is reached in the randomized refinement algorithm.

We use the following reward function: -1 for an IK-infeasible sample, +3 for an IK-feasible sample, -3 for action motion planning failure, -3 for action precondition violation, and +5 for action motion planning success. Our weight vectors are initialized to 0, representing a uniform distribution across the limits of the search space. These limits are a cube of side length 30 cm around an object being grasped or location of putdown, and a square of side length 1 meter around an object to which the robot is moving.

Our experiments are in Python using the OpenRave simulator with a PR2 robot. The motion planner we use is trajopt, and the task planner is Fast-Forward. The experiments were carried out in series on an Intel Core i7-4770K machine with 16GB RAM. We test five different scenarios, all of which share the goal of grasping an object off a table and putting it down at a specific location: one grasp obstruction, two grasp obstructions, three grasp obstructions, one grasp obstruction with base motion allowed (this means the system must learn a distribution for base poses), and putdown location obstructed in the cardinal directions. Table 1 summarizes our results,

and Figure 2 shows the learned distribution for different iterations in the training process.

VII. Conclusion and Future Work

TEMP

A good direction for future work will be to train a system that decides whether to return to the high level on its own, instead of relying on reaching the iteration limit in the randomized refinement algorithm.