Bandit-Based Planning and Learning in Continuous-Action Markov Decision Processes

Ari Weinstein and **Michael L. Littman**
Rutgers University
110 Frelinghuysen Road
Piscataway, NJ 08854 USA *

Abstract

Recent research leverages results from the continuous-armed bandit literature to create a reinforcement-learning algorithm for continuous state and action spaces. Initially proposed in a theoretical setting, we provide the first examination of the empirical properties of the algorithm. Through experimentation, we demonstrate the effectiveness of this planning method when coupled with exploration and model learning and show that, in addition to its formal guarantees, the approach is very competitive with other continuous-action reinforcement learners.

Introduction

Continuous-valued states and actions naturally arise in the reinforcement-learning setting when the agent is interacting with a dynamic physical system. Swimming, driving, flying, and walking all involve observing positions and velocities and applying forces to control them. A common approach when learning to act in continuous environments is to enforce a (necessarily coarse) discretization over the state and action dimensions, and make decisions in that simpler, discrete Markov Decision Process (MDP). This approach requires the system designer to make a hard choice to avoid overdiscretizing (drastically increasing the resources needed to plan and learn) or underdiscretizing (risking failure if critically different actions or states are conflated).

Reinforcement-learning (RL) algorithms that avoid coarse discretization often focus on domains with continuous state spaces but *discrete* action spaces—algorithms that function in continuous action spaces have been examined less thoroughly. In many ways, working in a continuous state space is much easier than working in a continuous action space. In both cases, algorithms must generalize information from one point in the space elsewhere. However, continuous action spaces also require optimization over the continuous action dimension to plan effectively.

The few algorithms designed for use in continuous action spaces can be divided between those that attempt to build a value function and those that search the policy space directly. To obtain reliable results and worst-case guarantees, methods that build a value function in continuous spaces are limited to classes of function approximators (FAs) that can safely be used, such as averagers (Gordon 1995). This restriction aside, the approach often fails because noise in value estimates can lead to a systematic overestimation of the value function, causing a degenerate policy to be computed (Thrun and Schwartz 1994). Another source of difficulty is that of choosing which features to use when representing the value function; the wrong set of features can cause failure due to either inexpressiveness (with too few features), or overfitting (too many features) (Kolter and Ng 2009). Policy search methods have their own set of limitations. Their applicability is restricted to episodic domains and to searching the space of policies representable by the function approximator. Additionally, policy-gradient methods, a popular subset of policy search methods, converge only to locally optimal policies (Sutton et al. 1999).

In this paper, we examine the empirical performance of a method that functions in continuous action spaces by planning over the set of possible *sequences* of actions that can be taken within a finite horizon. By doing so, the problem can be reformulated as one of optimization, where the total return with respect to the action sequence starting from a particular state is the quantity being optimized, and the cumulative reward is the only quantity relevant to planning. After optimization is performed, the estimated initial optimal action is taken in the domain and planning is started anew from the resulting state. Although this method is a form of policy search, it is different from the most common approach of representing the policy by an FA, which is incrementally improved at the end of each episode.

When used in the deterministic setting, the algorithm has a number of appealing theoretic properties, such as a fast rate of convergence to optimal behavior. It also does not suffer from the limitations of other classes of algorithms. While algorithms that bootstrap estimates of FAs to build an estimated value function may suffer from divergence (Boyan and Moore 1995), the algorithm discussed here does not use that approach and cannot have a divergent estimate of value. Likewise, because policy is represented by a sequence of actions, as opposed to parameterization of an FA, the algorithm will always be able to represent the optimal D-step policy (in deterministic settings). Finally, any method that tries to find a global mapping from states to actions must

*This research was supported in part by National Science Foundation DGE-0549115.

Copyright © 2012, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

search a hypothesis space the size of the state and action space. Because the method discussed here plans in an open-loop manner (a sequence of actions as opposed to a mapping of states to actions), it is state agnostic, and therefore is not impacted by the size of the state space. As such, it also has the property of functioning identically in discrete, hybrid, or partially observable MDPs, as long as a generative model is available.

In the next section, background information will be outlined, including the Hierarchical Optimistic Optimization (HOO) strategy we use for stochastic optimization. Next, the extension of HOO to sequential decision making and its theoretical properties are discussed. Since the algorithm requires access to a generative model, a method of exploration and model building will be presented that allows a high-quality model to be built in the absence of a generative model. Experimental results will be presented in the settings where a generative model is available, as well as where a model must be constructed from direct interaction with a domain. In both cases, our method consistently outperforms other methods, even in the presence of large amounts of noise. Following this material will be the conclusion with proposals for future extensions and improvements.

Background

This section discusses the bandit setting, along with hierarchical optimistic optimization, a bandit algorithm that functions when there is a continuum of arms. A small modification to this algorithm will allow it to function as a planner in Markov decision processes, which is also described formally.

Hierarchical Optimistic Optimization

The simplest possible setting for RL is the multi-armed bandit, where an agent makes a decision as to which of the finite number arms to pull at each time step, attempting to maximize reward. If we call the arm pulled at time step n a_n, the agent receives a reward drawn from the payoff distribution for that arm, $r_n \sim R(a_n)$. A common goal is to find an algorithm with cumulative regret that grows as slowly as possible with time. If we call the optimal arm $a^* = \mathrm{argmax}_a E[R(a)]$, the cumulative regret at n is defined as $\sum_{t=0}^{n}(E[R(a^*)] - E[R(a_t)])$.

In some settings, however, there is a continuum of arms as opposed to a small, finite set of arms. The Hierarchical Optimistic Optimization (Bubeck et al. 2008) or HOO strategy is a bandit algorithm that assumes the set of arms forms a general topological space. HOO operates by developing a piecewise decomposition of the action space, which is represented as a tree (Figure 1). When queried for an action to take, the algorithm starts at the root and continues to the leaves by taking a path according to the maximal score between the two children, called the B-value (to be discussed shortly). At a leaf node, an action is sampled from any part of the range that the node represents. The node is then bisected at any location, creating two children. The process is repeated each time HOO is queried for an action selection.

A description of HOO is shown in Algorithm 1, with some functions defined below. A node ν is defined as having a

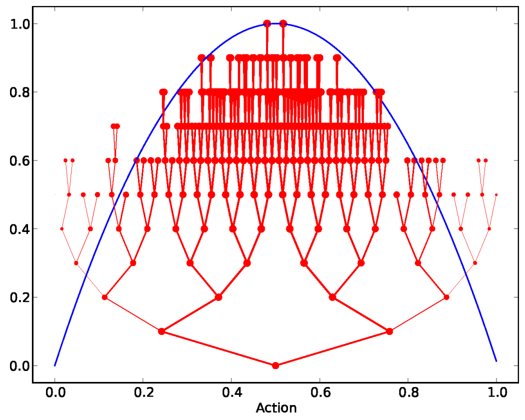


Figure 1: Illustration of the the tree built by HOO (red) in response to a particular continuous bandit (blue). Thickness of edges indicates the estimated mean reward for the region each node covers. Note that samples are most dense (indicated by a deeper tree) near the maximum.

number of related pieces of data. Unless it is a leaf, each node has two children $C(\nu) = \{C_1(\nu), C_2(\nu)\}$. All nodes cover a region of the arm space $A(\nu)$. If ν is the root of the HOO tree, $A(\nu) = A$. For any non-leaf node ν $A(\nu) = A(C_1(\nu)) + A(C_2(\nu))$. The number of times a path from root to leaf passes through ν is $N(\nu)$, and the average reward obtained as a result of those paths is $\hat{R}(\nu)$. The upper bound on the reward is

$$U(\nu) = \hat{R}(\nu) + \sqrt{\frac{2\ln n}{N(\nu)}} + v_1\rho^h$$

for $v_1 > 0$ and $0 < \rho < 1$, where v_1 and ρ are parameters to the algorithm. If the dissimilarity metric between arms x and y of dimension $|A|$ is defined as $||x-y||^\alpha$, setting $v_1 = (\sqrt{|A|}/2)^\alpha$, $\rho = 2^{-\alpha/|A|}$ will yield minimum possible regret. Finally, N, $\hat{R}$, and U are combined to compute the B-value, defined as

$$B(\nu) = \min\{U(\nu), \max\{B(C_1(\nu)), B(C_2(\nu))\}\}$$

which is a tighter estimate of the upper bound than U because it is the minimal value of the upper bound on the node itself, and the maximal B-values of its children. Taking the minimum of these two upper bounds produces a tighter bound that is still correct but less overoptimistic. Nodes with $N(\nu) = 0$ must be leaves and have $U(\nu) = B(\nu) = \infty$.

Given the assumption that the domain is locally Hölder around the maximum, HOO has regret $\tilde{O}(\sqrt{n})$, which is independent of the dimension of the arms and is tight with the lower bound of regret possible. Additionally, it is one of the few available algorithms designed to perform global optimization in noisy settings, a property we build on to perform sequential planning in stochastic MDPs.

Markov Decision Processes

An MDP M is described by a five-tuple $\langle S, A, T, R, \gamma\rangle$, where $S \subseteq \mathbb{R}^{|S|}$ is the state space, $A \subseteq \mathbb{R}^{|A|}$ is the action

Algorithm 1 HOO (see text for details)

```
 1: function PULL
 2:     $n \leftarrow 0$
 3:     loop
 4:         $a \leftarrow$NEXTACTION
 5:         $r \sim R(a)$
 6:         INSERT$(a, r)$
 7: function NEXTACTION
 8:     $n \leftarrow n + 1$
 9:     $\nu \leftarrow root$
10:     while $\nu$ is not a leaf do
11:         $\nu \leftarrow \mathrm{argmax}_{c \in C(\nu)} B(c)$
12:     return $a \in A(\nu)$
13: function INSERT$(a, r)$
14:     $\nu \leftarrow root$
15:     while $\nu$ is not a leaf do
16:         Update $R(\nu), N(\nu)$
17:         $\nu \leftarrow c \in C(\nu)$ such that $a \in A(c)$
18:     Update $R(\nu), N(\nu)$
19:     Create children $C(\nu) = \{C_1(\nu), C_2(\nu)\}$
```

space, T is the transition function, with $T(s'|s, a)$ denoting the distribution over next states s' by taking action a in state s.

The reward function $R(s, a)$ denotes the deterministic reward from taking action a in state s and $\gamma \in [0, 1)$ is the discount factor. A policy π is a mapping $\pi : S \rightarrow A$ from states to actions. The value of a policy, $V^\pi(s)$, is defined as the expected sum of discounted rewards starting at state s and following policy π. Likewise, the action-value function $Q^\pi(s, a) = R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')V^\pi(s')$, which is the value of taking a from s and then following π afterwards. The discounted return from time $t = 0$ to $t = n$ is $\sum_{t=0}^{n} \gamma^t r_t$, where r_t is the reward obtained at step t. The optimal policy, π^*, is the policy π that maximizes $V^\pi(s), \forall s \in S$. The goal in reinforcement learning is to find a policy π as close as possible to optimal, as measured by $\max_s |V^\pi(s) - V^{\pi^*}(s)|$.

Bandit-Based Planning in Continuous Action MDPs

The approach of extending bandit algorithms to sequential planning problems has been examined in Bayesian (Duff and Barto 1997) and PAC-MDP (Strehl and Littman 2004) settings. The specific analysis of HOO applied to sequential planning in terms of simple regret has also been explored theoretically (Bubeck and Munos 2010). The simple regret of an action a in a state s is defined as the difference between taking a in s and then following the optimal path and taking the optimal path starting from s. When we apply the HOO bandit algorithm to MDPs for sequential planning, we refer to it as Hierarchical Open-Loop Optimistic Planning (HOLOP), which borrows part of its name from the Open-Loop Optimistic Planning approach (Bubeck and Munos 2010).

HOLOP

Whereas HOO optimizes regret in the bandit setting (maximizing immediate reward), HOLOP optimizes regret in MDPs by optimizing the return of an D-step rollout. HOLOP is a sample-based algorithm, which assumes access to a generative model. Given a query of any (s, a), a generative model will return $R(s, a)$, and $s' \sim T(s, a)$. The requirement of such a generative model is weaker than a complete description of the MDP needed by planning approaches like linear programming but stronger than the assumption used in on-line RL where information is only obtained by direct interaction with the MDP, as generative models must return samples for any possible query (s, a). Sample-based methods can be used in any domain for which a generative model is available or a model can be built.

Given a query state s, HOLOP creates a HOO agent. The role of this HOO agent is to find the best D-step action sequence through interaction with the generative model. HOLOP then executes the first action of this sequence in the actual domain. HOO is unaware the optimization has a temporal component. It computes B-values for all nodes in the tree and takes a path from the root to a leaf by following the maximum B-value of each node's children along the way. Once execution reaches a leaf node l, an action sequence of dimension $|A|D$, drawn from $A(l)$, is returned. The return r of this action is computed by doing a rollout simulation using this action sequence from the root state s_0 obtaining $r = \sum_{d=0}^{D} \gamma^d R(s_d, a_d)$, where $|a_d| = |A|$, and corresponds to the d^{th} action in the sequence.

The value of r is then included in l and all its ancestors, and l is split into two children, as in HOO. Planning continues until a planning cutoff (such as time) is reached. At that point, HOO returns a recommended greedy action, which is executed in the actual MDP. HOLOP repeats this planning process for each new state encountered.

Algorithm 2 HOLOP

```
 1: function PLAN(s)
 2:     loop
 3:         $a \leftarrow$NEXTACTION$(s)$
 4:         $s \sim T(s, a)$
 5: function NEXTACTION$(s)$
 6:     repeat
 7:         $\langle a_0...a_D \rangle \leftarrow$HOO.NEXTACTION
 8:         $s_0 \leftarrow s$
 9:         $r \leftarrow 0$
10:         for $d = 0 \rightarrow D$ do
11:             $s_{d+1} \sim T(s_d, a_d)$
12:             $r \leftarrow r + \gamma^d R(s_d, a_d)$
13:         HOO.INSERT$(\langle a_0...a_D \rangle, r)$
14:     until planning cutoff
15:     $\langle a_0...a_D \rangle \leftarrow$HOO.GREEDYACTION
16:     return $a_0$
```

As a regret-based algorithm, HOLOP is used in an anytime manner. That is, as more queries are used, performance continually increases. This behavior is in contrast to other methods of planning like those that satisfy the PAC-MDP conditions (Strehl and Littman 2004), which cannot be interrupted prematurely and only learn an approximately optimal solution.

In the implementation used in this paper[1], the dimension at which a node is split is done probabilistically according to the maximum value the action at that step j in the sequence could contribute to the return: $\gamma^j / \sum_{d=0}^{D} \gamma^d$. Once j is selected, a cut is made in one of the $|A|$ dimensions uniformly at random. To more thoroughly leverage data resulting from planning, all (r, a) samples are passed down the tree into the appropriate leaves as the tree is grown. Finally, greedy actions are chosen by following the $\hat{R}$-values from root to leaf, and the action returned is the action stored in the leaf found by following $\hat{R}$.

HOLOP in Deterministic MDPs

One important aspect of HOLOP is that it has sound theoretical backing. Bubeck and Munos [2010] show that if the search space is smooth—that is, nearby sequences have nearby rewards[2]—the HOO agent has simple regret of $\tilde{O}\big(n^{-1/(\delta+2)}\big)$, $\delta = \frac{\log \kappa}{\log 1/\gamma}$. Here, κ is the number of near-optimal paths. Because the bound goes to zero asymptotically, as the number of trajectories grows, the HOO agent converges to finding the optimal open-loop plan.

From a theoretical perspective, it is noteworthy that this bound is independent of the size of the state and action spaces. In practical terms, however, the quantity $D|A|$ (the dimension of a D-step policy) does determine the rate of learning. In particular, the regret bound considers the case when the number of samples grows very large compared to the dimensionality, which is often not feasible in practice.

An open-loop plan like the kind HOLOP finds is essentially "blind"—it commits to a sequence of actions and then executes them regardless of the states encountered along the way. In deterministic environments, state transitions are completely predictable in advance. Thus, the best open-loop plan will match the performance of the best possible policy (given the same planning horizon). In general, however, there may be a significant gap between the best open-loop plan and the best policy.

HOLOP in Stochastic MDPs

Since HOO is an optimization method that functions in stochastic domains, HOLOP can be used for planning in stochastic MDPs. As mentioned, open-loop plans may not produce optimal decisions in such domains. Consider the MDP in Figure 2. There are four different open-loop plans. The solid-solid and solid-dashed sequences have an expected reward of 1.0, whereas both sequences beginning with the dashed transition get 0.0 on average. Thus, the best open-loop plan is solid-solid. However, the best first action

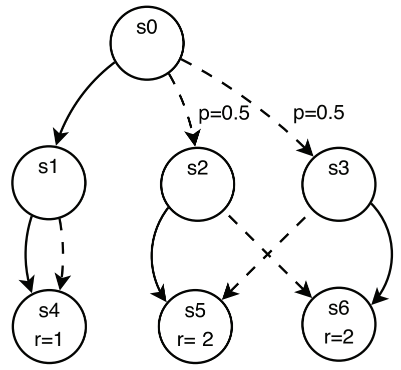


Figure 2: An MDP with structure that causes suboptimal behavior when open-loop planning is used.

is dashed as the agent will get to observe whether it is in state s_2 or s_3 and choose its next action accordingly to get a reward of 2.0, regardless.

In spite of this performance gap, HOLOP has two properties that can mitigate the limitations of open-loop planning. First, the expected reward that HOLOP computes for a given D-step sequence reflects the fact that a single action roll-out can lead to different sequences of states as a result of the stochastic transition function. Thus, it searches for open-loop sequences with high *expected* reward.

The second property is that, although the planning performed by HOLOP is open loop, the policy it executes is closed loop—replanning occurs at every step from whichever state the agent is in at that time. Thus, the cumulative reward obtained by HOLOP in the environment is guaranteed to be no worse (and can be considerably higher) than the return predicted during planning. As we will see in the experimental section, these two properties help HOLOP achieve very strong performance in stochastic domains in spite of its worst-case limitations.

Exploration and Model Building

Sample-based planners like HOLOP require a generative model, which may not be available in all situations. When an agent only has access to (s, a, r, s') samples resulting from direct interactions with an MDP, model building can be used to *construct* a generative model. In general, two components are necessary to build an accurate model in such an RL scenario: exploration and supervised learning, described next.

Multi-Resolution Exploration (Nouri and Littman 2008) or MRE is a method that can be used inside almost any RL algorithm to encourage adequate exploration of a domain, assuming R and T are Lipschitz continuous. Originally concerned with domains with discrete actions and continuous state, the version here is modified to explore the continuous space $S \times A$. MRE functions by decomposing $S \times A$ via a tree structure. Each leaf of the tree represents a region of $S \times A$ within which "knownness", $\kappa(s, a)$ is computed. As samples are experienced in the MDP, they are added to the corresponding leaf that covers the sample. Once the leaf contains a certain number of samples, it is bisected.

[1]Some technical details are omitted here due to space constraints. As such, a complete implementation of the algorithms and experimental domains has been made available at `http://code.google.com/p/holop/`

[2]A natural distance metric on sequences is γ^Δ where Δ is the depth at which the sequences diverge.

When a query is made for a predicted reward and next state, MRE may intervene and return a transition to s_{max} (a state with maximal possible value) with probability $1 - \kappa(s, a)$. The presence of this artificial state draws the agent to explore and improve its model of that region. This increases κ, which in turn means it will be less likely to be explored in the future. In terms of model building, the tree used by MRE can also be used as a regression tree. Instead of inserting only (s, a) samples, inserting (s, a, r, s') allows estimates $\hat{R}(s, a) \rightarrow r$ and $\hat{T}(s, a) \rightarrow s'$ to be constructed.

We note that other methods of exploration and model building are available. In terms of randomized exploration, the most popular method is ϵ-greedy, but as an undirected search method it can fail to explore efficiently in certain MDPs (Strehl and Littman 2004), a limitation MRE does not have. In terms of model building, any supervised learning algorithm can be used to build estimates of R and T. Once MRE is chosen as the method of exploration, however, the same tree can also be used as a regression tree with only constant-time additional costs.

In the implementation tested, we used a variant of MRE where $\kappa = \min(1, g/(k(|S| + |A|)))$, as we found this definition of κ to yield better performance. The parameter k controls the level of exploration, and g is the depth of the leaf in the MRE tree. Model building was performed by using linear regression over the samples in each leaf in the MRE tree.

Comparisons in the Planning Setting

Here, we empirically compare HOLOP to UCT (Kocsis and Szepesvári 2006). Like HOLOP, UCT plans by performing rollouts and selects actions determined by a a bandit algorithm. But, unlike HOLOP, UCT is a closed-loop planner, and functions only in discrete domains. The closed-loop policy of UCT selects actions conditionally based on estimates of the returns for each action at a particular state and depth during the rollout.

As a planner for discrete state and action spaces, UCT requires a discretization over continuous spaces. In the case that a good discretization is not known beforehand, these are additional parameters that must be searched over to ensure effectiveness from the planning algorithm. Here, it will be demonstrated that even searching over a large number of possible discretizations for UCT still yields performance worse than HOLOP, which adaptively discretizes the action space and is agnostic to state.

We refer to an episode as the result of an algorithm interacting with the actual environment, and rollouts as being the result calculations based on a generative model (either given or learned). In all empirical comparisons, the performance metric used is mean cumulative reward per episode. In both domains, episodes are 200 steps long and planners are limited to 200 rollouts from the generative model (ending at depth 50 or a terminal state). The discount factor $\gamma = 0.95$.

The first experimental domain is the the double integrator (Santamaría, Sutton, and Ram 1996), which models the motion of an object along a surface. The state is represented by a velocity v and position p. The action sets the acceleration of the object, which is constrained to the range $(-1.5, 1.5)$ units. The reward function is defined as $R(p, a) = -(p^2 + a^2)$, and the initial state is set to $(p, v) = (1, 0)$. Noise is introduced by perturbing all actions taken by ± 0.1 units uniformly distributed.

In this domain, all discretizations of the state and action spaces used by UCT result in poorer performance than HOLOP with statistical significance (Figure 3(a), 95% confidence bounds not rendered, but used to establish significance). The best discretization for UCT is when both state and action dimensions are discretized into 10 units, resulting in a mean cumulative reward of -3.15 with an *upper* confidence bound of -3.08. HOLOP, on the other hand, has a mean cumulative reward of -2.72 and a *lower* bound of -2.76.

The second domain tested is the inverted pendulum, which models the physics of a pendulum balancing on a pivot (Pazis and Lagoudakis 2009), where actions are in the form of force applied to the pendulum. Noise is introduced by perturbing the actions by ± 10 newtons uniformly distributed from a full action range of $(-50, 50)$ newtons. The reward function in this formulation favors keeping the pendulum as close to upright as possible using low magnitude actions.

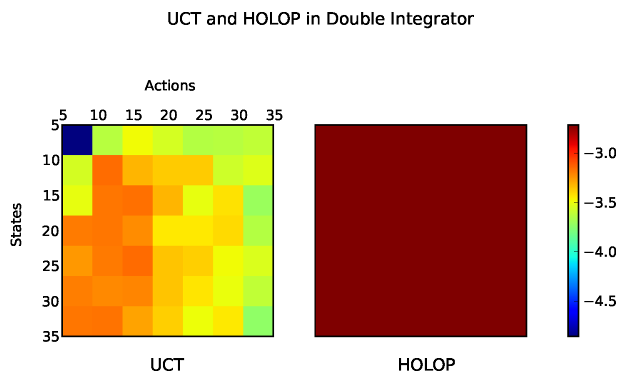
In the inverted pendulum domain (see Figure 3(b)), 3 parameterizations of UCT result in estimates of cumulative reward that are not statistically significantly different than HOLOP, and, in all other 46 parameterizations of UCT, HOLOP has significantly better performance. The best parameterization of UCT found consists of 20 discretizations per state dimension and 5 per action dimension, which results in a mean reward of -49.62 which was not statistically significantly different from the estimated mean value of -47.45 achieved by HOLOP.

Both experiments demonstrate the advantages of HOLOP over UCT. Although the most significant result of the experiments here is that HOLOP can perform better than UCT in continuous domains, there are two other important issues. First, while we mention the performance of UCT with the best parameterization found in our search over 49 possible settings for each domain, the performance for the worst parameterizations were extremely poor. For example, with the worst parameterizations, UCT was not even able to consistently prevent the pendulum from falling, which is quite simple in the pure planning setting. The second issue is the complement of this point; HOLOP produces excellent results without parameter tuning.

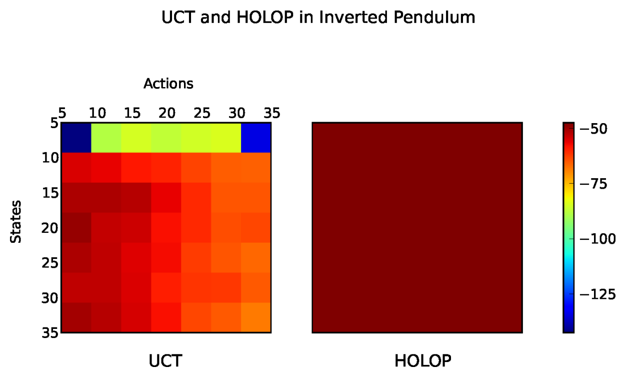
Comparisons in the Learning Setting

In the case where a generative model is not available, exploration and model building can be used in conjunction with a planner to learn through interaction with a domain. Here, the empirical performance of HOLOP with various exploration methods, as well as other learning algorithms, will be presented.

Performance of algorithms is framed between that of an agent executing a random policy and that of an agent executing the optimal policy. In any domain where the performance of the agent was not statistically significantly better



(a) The double integrator domain.



(b) The inverted pendulum domain.

Figure 3: Performance of HOLOP and UCT in continuous domains.

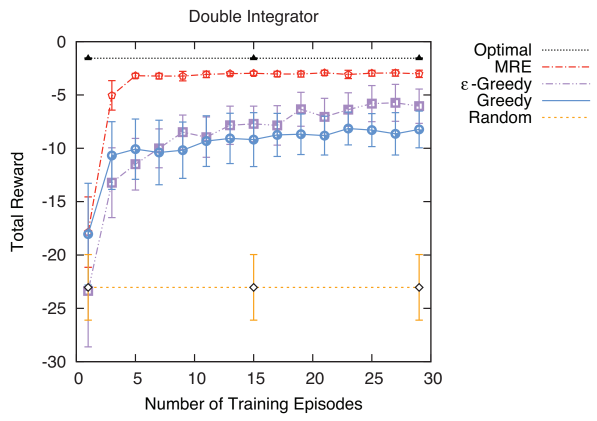


Figure 4: The impact of poor exploration on performance.

than the random policy, the performance of that agent was not plotted. Parameterization of HOLOP is the same as in the previous section. For MRE, $k = 2$ in the double integrator, and $k = 1$ in the inverted pendulum.

The domains used here are the same as those in the previous section, with the difference that in the inverted pendulum domain the reward signal is simplified to 1 while balance was maintained.

Methods of Exploration

Figure 4 demonstrates the impact of poor exploration policies in the double integrator. The methods compared are: MRE, ϵ-greedy, and pure greedy (no exploration). Here, ϵ was initialized to 1 and degraded by a factor of 0.9 at the end of each episode. Aside from exploration, model building and planning were performed identically. In this setting, MRE yields good results; the performance by the end of the 5^{th} episode is already not statistically significantly different from HOLOP with a perfect generative model, whereas the other exploration methods yield results that are still statistically significantly worse at the 30^{th} episode.

Comparison Learning Algorithms

We now introduce a number of other RL algorithms for continuous domains that will be compared to HOLOP with MRE for exploration and model building.

Ex$\langle a \rangle$ (Martín H. and De Lope 2009) extends TD(λ) to continuous action MDPs by using a k-nearest neighbor mechanism. The choice of action selection and updating of values is done according to a weighted eligibility trace.

The Continuous Actor-Critic Learning Automaton (Van Hasselt and Wiering 2007) or CACLA is an extension of actor-critic methods to continuous MDPs. Function approximators are used to estimate the value function computed according to TD errors, as well as the policy. Because of poor sample complexity, experience replay (Lin 1992) is added in an attempt to speed learning.

Policy search (PS) deals with finding the parameters of some function approximator that defines $\pi(s)$ without computing an estimation of the value function. In the experiments here, PyBrain's (Schaul et al. 2010) implementation of direct policy search is tested, which attempts to find weights of a neural network by performing black box hill climbing to maximize accumulated reward. In all domains, the size of the hidden layer is set to twice the size of the state dimension.

The only algorithm tested that functions in discrete domains, R-max (Brafman and Tennenholtz 2002) is a model-based algorithm. It has polynomial sample complexity and has been shown to be one of the most effective algorithms on general discrete MDPs. Like MRE, directed exploration is conducted based on whether an (s, a) pair is considered known or unknown. Whenever an (s, a) becomes known, the value function of the estimated MDP is solved.

Fitted Q-iteration (Ernst, Geurts, and Wehenkel 2005) or FQi is a batch mode reinforcement-learning algorithm that operates by iteratively using function approximators to build estimates of the Q-function, $\hat{Q}$. Since an FA is used, extending the algorithm to the continuous action case is possible, and any optimization method can be used in place of the $\max$ operator. For these experiments, $\hat{Q}$ is estimated using a forest of totally random trees, and hill climbing with restarts is used for optimization. The value function is reestimated at the end of each episode.

Whenever possible, implementations written directly by the author of the algorithm are tested. When computationally feasible, an automated coarse search for parameter values is performed over the range of parameters from published empirical results. For the more computationally expensive algorithms, a coarse manual search is performed over the same range of values. In the absence of published parameters, best judgment is used to find values that work as well as possible. This parameter search is performed for each algorithm independently for each experiment.

Results

Most learning algorithms perform well in the double integrator (Figure 5(a)). The structure of the dynamics and rewards means that the optimal policy is a linear combination of the two state features. As such, many algorithms exhibit behavior with good performance after 30 episodes. All algorithms aside from policy search are able to at least find policies statistically significantly better than random. HOLOP performs statistically significantly better than all algorithms at the end of the experiment aside from Ex$\langle a \rangle$, which is statistically significantly lower than HOLOP from episodes 3–23, and after which there was no significant difference.

In the inverted pendulum with simplified rewards, HOLOP is the only algorithm that approached a near optimal cumulative reward(Figure 5(b)). Performance of FQi is significantly worse, but still improving at the the end of the experiment, so convergence may have been to a better policy given more samples. Ex$\langle a \rangle$ and CACLA both perform statistically significantly better than random, but improvement is slow, still employing a poor policy by episode 30.

Discussion of Experiments

In all the domains tested, HOLOP showed the fastest improvement of its policy while learning as well as the best policy at the end of all experiments. The success of the approach can be attributed to the efficiency of exploration and the accuracy of the model built, as well as the ability of the planner to quickly find action sequences that yield high reward.

This point aside, there was another, perhaps more significant factor, that distinguished HOLOP from the other algorithms tested—how difficult was it to obtain the "best" performance from each of them. Based on previously published results, it has been demonstrated that Ex$\langle a \rangle$ is a powerful learning algorithm (Whiteson, Tanner, and White 2010; Wingate et al. 2009). While we were able to reproduce the performance described by Martín H. and De Lope [2009], we noticed small changes to a single parameter could cause large variations in effectiveness, as well as sensitivity to noise. Ex$\langle a \rangle$, however, was not alone with this issue as most of the algorithms had numerous parameters whose values significantly impact performance. Thus, parameter tuning is both time consuming and difficult.

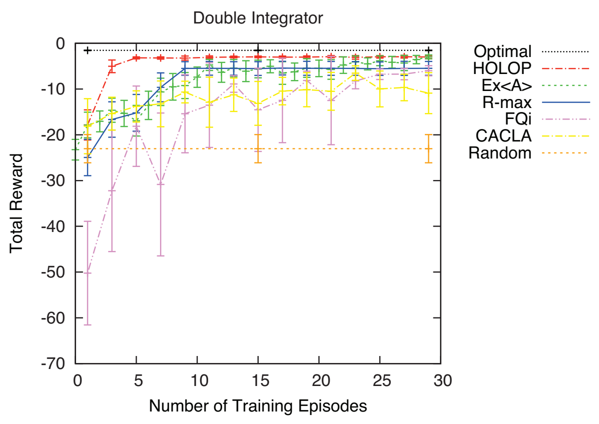
HOLOP, on the other hand, had no parameter change across experiments, and only one parameter was changed for MRE in the learning experiments. Additionally, the parameters are also simple to interpret. In general, the algorithm performs best with longer rollouts, with more rollouts, and with more exploration (if the domain is smooth, exploration can be reduced to enable faster convergence). When using the algorithm, all that needs to be done is choose values of these parameters that allow planning to terminate quickly enough. This robustness makes the algorithm much easier to use in practice than the comparison algorithms.

Of all the algorithms, the implementation of policy search had the poorest results. We can partially attribute this result to the fact that the algorithm can only learn from the result of an entire episode, as opposed to the other algorithms tested that improve their policies each time a new (s, a, r, s') is experienced. For example, in the double integrator, HOLOP reaches near-optimal performance in 5 episodes, by which time it has updated its policy 1000 times. In contrast, policy search has only done so 5 times during this interval. Although policy search performance was always poor at the end of each of our experiments, allowing the algorithm access to more episodes (up to an order of magnitude) led to convergence to near optimal policies.

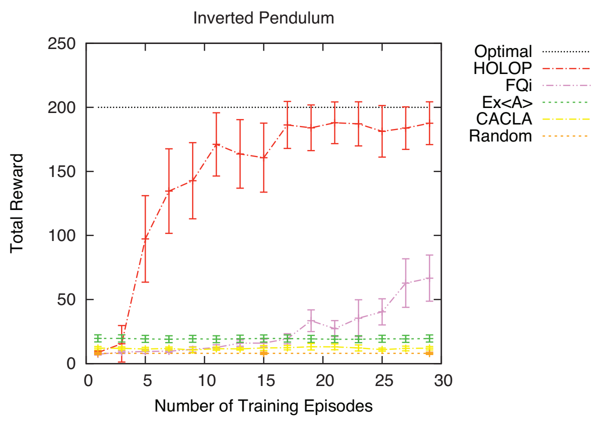
In general though, the less sample efficient an algorithm was, the faster its computation. The performance metrics here compared the number of episodes experienced against cumulative reward achieved during each episode. Another metric could be running time as compared to the cumulative reward after a certain amount of running time. Policy search, which seemed to have the poorest sample complexity, only updates its policy once every episode with a simple update rule and as such is extremely fast to execute; the cost of computation of one episode in policy search is the same as one rollout in HOLOP. Ex$\langle a \rangle$ and CACLA are only slightly more computationally expensive, doing a quick update to the policy and value function for each step. Compared to all these, the computational cost of planning with HOLOP is large. Unlike R-max or fitted Q-iteration, which has an expensive planning step on occasion, HOLOP has to replan at each step in the environment. Although HOLOP can learn from a very low number of samples in the environment, if certain factors such as response time are critical, other algorithms may yield better performance.

Related Work

Aside from the algorithms discussed, there are a small number of algorithms that operate in the continuous-action setting. HOOT (Hierarchical Optimistic Optimization applied to Trees) and WHOOT (Weighted HOOT) also utilize HOO, but perform *closed*-loop planning (Mansley, Weinstein, and Littman 2011). HOOT is designed to be used in domains with a discrete state space, whereas WHOOT functions in continuous state spaces. While closed-loop planning allows the algorithms to represent effective plans in domains that are problematic for open-loop planners, the algorithms are difficult to analyze because action-value estimates are non-stationary estimates due to policy change that occurs lower in the tree over time. They were not included in the comparison here because of the large computational complexity of the algorithms; for a computation cost of c for HOLOP, the costs of these closed loop planners are cD, as they must query a HOO agent at each step of the rollout as opposed to just at the start.



(a) The double integrator domain.



(b) The inverted pendulum domain.

Figure 5: Performance of HOLOP with MRE and other learning algorithms.

Tree Learning Search (Van den Broeck and Driessens 2011) is another method of performing open-loop planning in continuous action spaces. In this method, action-value estimates generated from rollouts are given to an on-line regression tree learner, which determines which regions of the action space may contain high reward. These estimates then inform the policy when performing rollouts at the next iteration. Unfortunately, Tree Learning Search also lacks any theoretical guarantees of performance, and was not shown to perform better than planners that used *a priori* discretization.

Binary Action Search (Pazis and Lagoudakis 2009) or BAS is a method that can be applied to any discrete action, continuous-state learner or planner to make it almost indistinguishable from a continuous-action learner. It transforms the state space S into a new state space $S' = S \times A$, and the actions are reduced to a binary decision in each original dimension in A, corresponding to incrementing or reducing each action value encoded in the state. Similar to HOO, this method dynamically divides the continuous action space into regions of smaller size. Although we did not compare directly against this algorithm here, the perfomance of HOLOP presented here in the double intergrator is superior to the performance of BAS according to Pazis and Lagoudakis [2009].

Conclusion

HOLOP combined with MRE is a learning and planning system that has a number of advantageous properties when compared to other methods designed for use in continuous MDPs. By using directed exploration and regression, high quality models can be built efficiently in domains regardless of the structure of the domain as long as it is smooth (a property undirected exploration, such as ϵ-greedy, does not have). By not building a global estimate of the value function, there is no risk of value-function divergence, with accompanying degenerate policies. Additionally, by searching over the space of action sequences, as opposed to parameters of a function describing a policy, HOLOP is guaranteed to quickly approach the optimal *global* policy, and cannot become caught in local optima, or fail to represent the optimal policy in deterministic domains.

The empirical performance of HOLOP was compared to several related algorithms on a set of benchmark reinforcement-learning problems. Although open-loop planning can lead to suboptimal behavior in stochastic settings, the performance of HOLOP was consistently strong—nearly optimal in all cases.

As mentioned, planning in HOLOP is more expensive than most of the other methods discussed. One factor mitigating this issue is that HOLOP, like most sample-based planners, parallelizes easily (Chaslot, Winands, and Van den Herik 2008); the implementation here used root parallelization in order to speed planning. Another method could utilize storing the results of previous policy queries to speed up the running time of the algorithm (Silver, Sutton, and Müller 2008). Both approaches, however, may remove the performance guarantees of HOLOP.

The approach discussed here is one concrete example of an algorithm that combines particular methods of exploration, model building, and sequential action planning. Although MRE and HOLOP do have attractive theoretical properties, which are also borne out in practice, it is worth investigating what other methods can be substituted for each of these components, potentially with a tighter coupling. At the moment, each is agnostic to the existence of the other. For example, HOLOP does not react differently when used in the planning or learning setting, and MRE has no knowledge of whether it is being used in a model-based setting. It is worth investigating whether giving each component information about the others may enhance performance.

References

Boyan, J. A., and Moore, A. W. 1995. Generalization in reinforcement learning: Safely approximating the value function. In Tesauro, G.; Touretzky, D. S.; and Leen, T. K., eds., *Advances in Neural Information Processing Systems 7*, 369–376.

Brafman, R. I., and Tennenholtz, M. 2002. R-MAX—a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research* 3:213–231.

Bubeck, S., and Munos, R. 2010. Open loop optimistic planning. In *Conference on Learning Theory*.

Bubeck, S.; Munos, R.; Stoltz, G.; and Szepesvári, C. 2008. Online optimization of X-armed bandits. In Koller, D.; Schuurmans, D.; Bengio, Y.; and Bottou, L., eds., *Advances in Neural Information Processing Systems*, volume 22, 201–208.

Chaslot, G. M. J.-b.; Winands, M. H. M.; and Van den Herik, H. J. 2008. Parallel Monte-Carlo tree search. In *Proceedings of the Conference on Computers and Games 2008*.

Duff, M. O., and Barto, A. G. 1997. Local bandit approximation for optimal learning problems. In *Advances in Neural Information Processing Systems*, volume 9, 1019–1025.

Ernst, D.; Geurts, P.; and Wehenkel, L. 2005. Tree-based batch mode reinforcement learning. *Journal of Maching Learning Research* 6:503–556.

Gordon, G. J. 1995. Stable function approximation in dynamic programming. In Prieditis, A., and Russell, S., eds., *Proceedings of the Twelfth International Conference on Machine Learning*, 261–268.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, 282–293.

Kolter, J. Z., and Ng, A. 2009. Regularization and feature selection in least-squares temporal difference learning. In Bottou, L., and Littman, M., eds., *Proceedings of the 26th International Conference on Machine Learning*, 521–528.

Lin, L. J. 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning* 8:293–321.

Mansley, C.; Weinstein, A.; and Littman, M. L. 2011. Sample-based planning for continuous action markov decision processes. In *Twenty-First International Conference on Automated Planning and Scheduling*, 335–338.

Martín H., J. A., and De Lope, J. 2009. Ex $< a >$: An effective algorithm for continuous actions reinforcement learning problems. In *In Proceedings of 35th Annual Conference of the IEEE Industrial Electronics Society*, 2063–2068.

Nouri, A., and Littman, M. L. 2008. Multi-resolution exploration in continuous spaces. In *Advances in Neural Information Processing Systems 21*, 1209–1216.

Pazis, J., and Lagoudakis, M. 2009. Binary action search for learning continuous-action control policies. In Bottou, L., and Littman, M., eds., *Proceedings of the 26th International Conference on Machine Learning*, 793–800.

Santamaría, J. C.; Sutton, R. S.; and Ram, A. 1996. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*.

Schaul, T.; Bayer, J.; Wierstra, D.; Sun, Y.; Felder, M.; Sehnke, F.; Rückstieß, T.; and Schmidhuber, J. 2010. PyBrain. *Journal of Machine Learning Research*.

Silver, D.; Sutton, R.; and Müller, M. 2008. Sample-based learning and search with permanent and transient memories. In McCallum, A., and Roweis, S., eds., *Proceedings of the 25th Annual International Conference on Machine Learning*, 968–975.

Strehl, A. L., and Littman, M. L. 2004. An empirical evaluation of interval estimation for Markov decision processes. In *Tools with Artificial Intelligence (ICTAI-2004)*.

Sutton, R. S.; McAllester, D.; Singh, S.; and Mansour, Y. 1999. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12*, 1057–1063.

Thrun, S., and Schwartz, A. 1994. Issues in using function approximation for reinforcement learning. In Mozer, M. C.; Smolensky, P.; Touretzky, D. S.; Elman, J. L.; and Weigend, A. S., eds., *Proceedings of the 1993 Connectionist Models Summer School*.

Van den Broeck, G., and Driessens, K. 2011. Automatic discretization of actions and states in monte-carlo tree search. In *International Workshop on Machine Learning and Data Mining in and around Games*.

Van Hasselt, H., and Wiering, M. A. 2007. Reinforcement learning in continuous action spaces. In *IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning, 2007*, 272–279.

Whiteson, S.; Tanner, B.; and White, A. 2010. The reinforcement learning competitions. *AI Magazine* 31(2):81–94.

Wingate, D.; Diuk, C.; Li, L.; Taylor, M.; and Frank, J. 2009. Workshop summary: Results of the 2009 reinforcement learning competition. In *Proceedings of the 26th International Conference on Machine Learning*, 166.