Online Learning for Optimistic Planning

Lucian Buşoniu[a], Alexander Daniels[b], Robert Babuška[b]

[a] *Department of Automation, Technical University of Cluj-Napoca, Romania (lucian.busoniu@aut.utcluj.ro)*
[b] *Delft Center for Systems and Control, Delft University of Technology, the Netherlands (alexanderdaniels87@gmail.com, r.babuska@tudelft.nl)*

Abstract

Markov decision processes are a powerful framework for nonlinear, possibly stochastic optimal control. We consider two existing optimistic planning algorithms to solve them, which originate in artificial intelligence. These algorithms have provable near-optimal performance when the actions and possible stochastic next-states are discrete, but they wastefully discard the planning data after each step. We therefore introduce a method to learn online, from this data, the upper bounds used to guide the planning process. Five different approximators for the upper bounds are proposed, one of which is specifically adapted to planning, and the other four coming from the standard toolbox of function approximation. Our analysis characterizes the influence of the approximation error on the performance, and reveals that for small errors, learning-based planning performs better. In detailed experimental studies, learning leads to improved performance with all five representations, and a local variant of support vector machines provides a good compromise between performance and computation.

Keywords: optimal control, machine learning, Markov decision processes, optimistic planning, near-optimality analysis.

1. Introduction

Markov decision processes (MDPs) are often used to model sequential decision-making problems in artificial intelligence [1, 2], but also work for optimal control problems in engineering, economics, operations research, medicine, etc. [3]. Online planning methods solve MDPs locally, by using a model to run an explorative search through the space of solutions (e.g. action sequences) from the current state, after which a first action is selected and applied. The system then reaches its new state and the process is repeated. A computationally effective strategy is to search the solution space optimistically [4], focusing on regions most likely to contain an optimal solution. The resulting *optimistic planning* (OP) algorithms [4, 5] integrate insights from several fields of artificial intelligence: reinforcement learning, bandit theory, and planning / graph search [3, 6], as well as from optimization. A variety of OP methods are available, including Upper Confidence Trees [7] which produced a competitive Go player [8], OP for Deterministic systems (OPD) [9], OP for stochastic MDPs with discrete next-state distributions (OPMDP) [10] or with general distributions [11], OP for continuous actions [12, 13], etc.

We focus here on OPD and OPMDP, which explore a tree representation of the possible solutions from the current state. Every node on the planning tree is labeled by a state and an upper bound (called b-value) on that state's optimal value. Exploiting the b-values, at each iteration the algorithms refine further an optimistic partial solution, with the largest upper bound on the value. OPD and OPMDP guarantee near-optimality bounds as a function of the computation invested [9, 10]. We refer to both algorithms collectively as 'OP'.

An important drawback of OP methods in their original form is that they discard the current tree right after applying the current action, and start over from scratch at the next step. However, the trees at consecutive steps will cover similar states, so the data can be reused to improve the quality of the search. Therefore, in this paper we propose to reuse data by learning online a b-function from the state–b-value pairs on the trees developed at previous steps. At the current step, the b-function learned in this way is used to initialize newly created leaves with informed b-values, rather than the uninformed values used in the original OP. The resulting approach is called OP with learning (L-OP). We provide a general analysis of the planning performance with learned b-values, taking into account the tradeoff between two novel effects that did not appear in standard OP: the b-values are closer to the optimal values than in OP (improving performance) – but due to approximation errors, they may no longer be true upper bounds and some nodes may be expanded non-optimistically (decreasing performance).

We continue by describing several b-value approximators. The first is specific to OP, and given a Lipschitz value function it guarantees that b-values always remain true upper bounds. The other approximators are standard: neural networks, local linear regression, and least-squares support vector machines; and for the latter, we also introduce a local, less computationally intensive variant. All techniques except neural networks are memory-based, so a procedure for memory management is additionally provided. We explain how these techniques can be applied to online L-OP, discuss their computational complexity, and evaluate them in practically relevant experiments on deterministic and stochastic problems. These experiments show that learning can achieve better performance with less computation, so it is useful for online control.

The idea of combining learning with model-based algorithms is well-known, see e.g. Dyna [14]. A variety of related work can be identified in classical planning by noticing that OPD and OPMDP extend the A* [15] and AO* [16] algorithms to infinite-horizon control. The b-value in OP plays a similar role to the heuristic in A* and AO*, and deriving good heuristics is recognized as essential for performance. Heuristics are often found offline, before planning starts, based e.g. on relaxed versions of the problem [17, 18]. Other methods learn heuristics incrementally from consecutive planning tasks [19, 20, 21], and these have some common elements with our work, such as generalizing function approximation [22, 19]. Nevertheless, most of these methods compute complete, optimal plans offline, whereas our focus is near-optimally solving infinite-horizon problems online. Closer to our online setting are the classical algorithms 'learning real-time A*' [23] and 'real-time dynamic programming' [24], which learn heuristics while planning online. In [25], an approximator is combined with RTDP – but this technique works for goal-based MDPs with logical states, and is not applicable in our general-MDP setting. Compared to [23, 24, 25], we introduce function approximation to deal with large, continuous state spaces (as suggested for RTDP in [24]), provide a near-optimality analysis in the context of OP, and a thorough empirical study with several types of approximators.

Our technique is to our knowledge the first to learn b-values online in optimistic planning. A value function computed *offline* was used to improve the performance of the Upper Confidence Trees algorithm in [26]. Our previous work on OP, such as [10, 13, 27], never uses learning – with a single exception: [28], where the model (transition probabilities) is learned rather than the b-values.

Next, Section 2 introduces the optimal control problem and the two OP methods. Section 3 describes learning for OP, including the methods for the deterministic and stochastic cases, their analysis, the five function approximators, and memory management. Section 4 gives the experimental results, and Section 5 concludes the paper.

2. Background

2.1. Problem Setting

We consider discrete-time optimal control problems with states $x \in X$ and actions $u \in U$. When applying u_k, the state changes from x_k to x_{k+1} with probability $f(x_k, u_k, x_{k+1})$, where $f : X \times U \times X \to [0, 1]$ is the state transition function. A reward function $\rho : X \times U \times X \to \mathbb{R}$ measures the quality of transitions, $r_{k+1} = \rho(x_k, u_k, x_{k+1})$. We assume the following: (i) the state space X is compact and included in $\mathbb{R}^p$; (ii) the action space U consists of a finite number K of actions; (iii) rewards are bounded, and without loss of generality they are in $[0, 1]$; finally, (iv) applying any action in any state can only lead to a finite number M of possible next states. These assumptions are typical in artificial intelligence, where X, U, f and ρ are said to form a Markov decision process (MDP). In control engineering, (i) is not restrictive in practice, while action discretization (ii) and reward saturation (iii) reduce performance, but the loss is often manageable. Deterministic transitions are common, in which case (iv) holds implicitly; otherwise, it restricts the random disturbance to discrete phenomena such as switches.

A policy $\pi : X \to U$ describes the control behavior: which action $u = \pi(x)$ to apply in each state x. The policy's value $V^\pi : X \to \mathbb{R}$ is defined for each state x as the expected return obtained by following the policy from x:

$$V^\pi(x) = \mathrm{E}\big\{ \sum_{k=0}^{\infty} \gamma^k r_{k+1} \big\} = \mathrm{E}\big\{ \sum_{k=0}^{\infty} \gamma^k \rho(x_k, \pi(x_k), x_{k+1}) \big\} \tag{1}$$

where $x_0 = x$, $x_{k+1} \sim f(x_k, \pi(x_k), \cdot)$, the expectation is taken over trajectories, and $\gamma \in (0, 1)$ is the discount factor. Since rewards are at most 1, an upper bound on any state value under any policy is $V_{\max} = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma}$. Next to the state value function V, the state-action value function Q is defined as:

$$Q^\pi(x, u) = \mathrm{E}_{x' \sim f(x,u,\cdot)}\{\rho(x, u, x') + \gamma V^\pi(x')\} \tag{2}$$

The objective is to *optimally* control the system, so that the value function is maximized for all $x \in X$. This maximal, optimal value function is denoted $V^*(x)$, an optimal policy that achieves these values is denoted $\pi^*(x)$, and the corresponding optimal state-action value function is $Q^*(x, u)$. The relationship $V^*(x) = \max_u Q^*(x, u)$ holds.

2.2. Optimistic Planning for Deterministic Systems

We focus first on the deterministic case where the transition function reduces to $x_{k+1} = f(x_k, u_k)$, since a single state x_{k+1} is reached with probability 1; and the reward function to $r_{k+1} = \rho(x_k, u_k)$, since the next state x_{k+1} – and hence the reward – are fully determined by x_k and u_k.[1]

We consider an online model-based planning algorithm called Optimistic Planning for Deterministic systems (OPD) [9], which at each step k explores the set of possible action sequences from the current state x_k. Such sequences are able to represent an optimal solution local to x_k. They are more general than the state-feedback policies $\pi(x)$, which can also represent optimal solutions, but sequences are convenient in planning so we will use them. At a high level, OPD iteratively refines promising action sequences until a computational budget n, related to the number of evaluations of the model f, is exhausted. Based on the return information accumulated about these sequences, OPD then chooses an action u_k that is as good as possible. This action is applied to the system and the procedure is repeated from the new state.

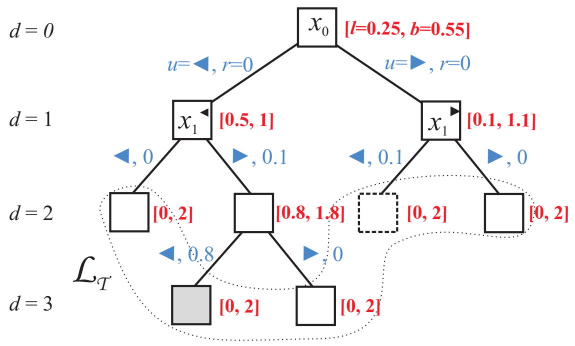

Figure 1: An OPD tree for $K = 2$ actions, symbolically denoted ◀ and ▶. Nodes are labeled by states, with the depth as a subscript and the last action leading to the state as a superscript. Arcs are labeled by actions and rewards, shown in blue. Near the nodes, l and b-values are shown in red boldface. The leaves $\mathcal{L}_\mathcal{T}$ are enclosed by a dotted line. The tree is shown after 4 expansions, and $\gamma = 0.5$. (Figure best viewed in color.)

To formalize the algorithm, we relabel by convention the current time k to 0, so that x_k becomes x_0. Of course, the procedure works at any time step. The planning process can be visualized using a tree structure $\mathcal{T}$, exemplified in Figure 1 where a (partial) planning tree for a problem with two control actions ($K = 2$) is shown. Tree nodes are denoted z and are labeled by states x. We use $x(z)$ to denote the state label of node z. Planning begins with a single, root node labeled by x_0, and proceeds by iteratively expanding nodes. The expansion of a node z consists of simulating all K actions from the associated state $x(z)$, and adding for each resulting successor state x' a new node z' as a child of z. The children of z are denoted $\mathcal{C}(z)$. An arc between a parent z and a child z' corresponds to a transition between states x and x', and is itself labeled by the action that caused the transition and by the resulting reward. E.g. in Figure 1, the leftmost node at $d = 1$ is labeled by $x_1^◀ = f(x_0, ◀)$ and the arc leading to it by action $u =$◀ and

[1]Here as well as in the sequel (e.g. for the b-values), we slightly abuse the notation by using the same symbols to denote analogous but mathematically different objects in the stochastic and deterministic case. For example, the deterministic $\rho(x_k, u_k)$ is obtained by plugging x_{k+1} in the stochastic reward function, $\rho(x_k, u_k, x_{k+1})$. It will usually be clear from the context to which variant the text refers; when it is not, we make it explicit.

reward $r = \rho(x_0, ◀)$, the latter being 0. The nodes in $\mathcal{T}$ are split in two categories: inner nodes, and leaf (unexpanded) nodes. The set of leaves is denoted $\mathcal{L}_{\mathcal{T}}$.

Each node z at some depth $d(z)$ in the tree is reached via a unique path through the tree, associated to an action sequence $\mathbf{u}(z) = [u_0, u_1, \dots, u_{d(z)-1}]$, and the obtained rewards along that path are denoted $r_1, r_2, \dots, r_{d(z)}$. E.g. the gray node in Figure 1 – denote it by y – has sequence $\mathbf{u}(y) = [◀, ▶, ◀]$ and the rewards along its path are $0, 0.1, 0.8$. Note that the depth d also has a meaning of time step. While action sequences are unique, state labels may be duplicated; the algorithm keeps all such duplicates on the tree.

OPD relies on so-called b-values, defined for the current tree as follows:

$$b(z) = \begin{cases} \dfrac{1}{1-\gamma} = V_{\max} & \text{if } z \text{ is a leaf,} \\ \max\limits_{z' \in \mathcal{C}(z)} [r(z') + \gamma b(z')] & \text{if } z \text{ is an inner node} \end{cases} \tag{3}$$

where $r(z')$ is the reward obtained for the transition to node z' (from z). The formula for inner nodes is a dynamic programming update, and we show by induction that $b(z)$ is an upper bound on $V^*(x(z))$. The property holds at the leaves due to $V_{\max}$. Consider an inner node, and assume $b(z') \geq V^*(x(z'))$ for its children z'. Then, $\max_{z'} [r(z') + \gamma b(z')] \geq \max_{z'} [r(z') + \gamma V^*(z')] = V^*(z)$, where the final equality is the Bellman equation. The induction is complete.

At each iteration of OPD, an optimistic leaf is found by starting at the root and navigating the tree downwards, always towards a child that achieved the maximum in the second branch of (3), until a leaf is reached. OPD then expands this optimistic leaf, growing the tree, and the whole procedure is repeated until the algorithm has exhausted its computational budget of n node expansions. Then, on the last tree obtained, l-values are computed like the b-values but starting from zero at the leaves:

$$l(z) = \begin{cases} 0 & \text{if } z \text{ is a leaf,} \\ \max\limits_{z' \in \mathcal{C}(z)} [r(z') + \gamma l(z')] & \text{if } z \text{ is an inner node} \end{cases} \tag{4}$$

Because rewards are positive, $l(z)$ is a lower bound on $V^*(x(z))$, by an induction similar to that for the b-values. OPD returns a root action that achieves the maximum in (4). Thus, the b-values optimistically guide the sequence refinement along upper bounds, while the final action selection is performed safely, based on pessimistic lower bounds.

Algorithm 1 summarizes OPD. Even though the algorithm explores finite-length sequences, it is near-optimal with respect to the infinite-horizon value V^*, as the guarantees in Section 3.2 will show (we postpone these guarantees until there to compare with our new method).

To exemplify the algorithm, consider e.g. the node of $x_1^{▶}$ in Figure 1. Its b-value is $\max\{0.1 + \gamma \cdot 2, 0 + \gamma \cdot 2\} = 1.1$ (recall that its children's b-values are $\frac{1}{1-\gamma} = 2$ since they are leaves and $\gamma = 0.5$). Similarly, at the root, the b-value is $\max\{0 + \gamma \cdot 1, 0 + \gamma \cdot 1.1\} = 0.55$. Then, the maximum at the root is attained via the right child $x_1^{▶}$, and in turn the maximizer child at this node is the left, dashed-outline child, which is therefore the optimistic leaf and will be expanded next. Finding l-values works similarly, e.g. at the root we have $\max\{0 + \gamma \cdot 0.5, 0 + \gamma \cdot 0.1\} = 0.25$, and if OPD were to terminate with this tree, ◀ would be returned since the maximum was achieved by this action.

Algorithm 1 Optimistic planning for deterministic systems

```
Input: model $f, \rho$, discount factor $\gamma$, state $x_0$, budget $n$
 1: initialize tree $\mathcal{T}$ with root $z_0$, labeled by $x_0$
 2: for $i = 1, \ldots, n$ do
 3:     $z \leftarrow z_0$
 4:     while $z \notin \mathcal{L}_{\mathcal{T}}$ do
 5:         $z \leftarrow \arg\max_{z' \in \mathcal{C}(z)} [r(z') + \gamma b(z')]$
 6:     end while
 7:     optimistic leaf found: $z^\dagger \leftarrow z$
 8:     expand $z^\dagger$, adding all its children to $\mathcal{T}$
 9:     update $b$ and $l$-values on the tree, with (3) and (4)
10: end for
Output: $u_0$, corresponding to $\max_{z' \in \mathcal{C}(z_0)} [r(z') + \gamma l(z')]$
```

While the implementation works with l and b, more insight can be obtained by defining lower and upper bounds on the discounted returns of action *sequences*, which will additionally be useful in the analysis. Define the partial return $L(z) = \sum_{j=0}^{d(z)-1} \gamma^j r_{j+1}$, which is a lower bound on the values of all infinitely-long sequences starting with $\mathbf{u}(z)$, and $B(z) = L(z) + \frac{\gamma^{d(z)}}{1-\gamma} = L(z) + \gamma^{d(z)} b(z)$, which is an upper bound on these values. Selecting an optimistic leaf $z^\dagger$ is equivalent to selecting a maximal $B(z^\dagger)$ among the leaves – so intuitively, we refine further a sequence that has the best chance of being part of an optimal solution. At the end, the chosen action u_0 is on a greedy leaf sequence $\mathbf{u}(z_g)$ with a maximal lower bound $L(z_g)$, so it is a safe choice. Note that the difference between the upper and lower bounds, equal to $\gamma^{d(z)} b(z) = \frac{\gamma^{d(z)}}{1-\gamma}$, is an uncertainty on the sequence values. E.g. for the dashed node y in Figure 1, $B(y) = 0 + \gamma \cdot 0.1 + \frac{\gamma^2}{1-\gamma} = 0.55$, which is maximal among all leaves, so y is the optimistic node. Denote the gray node by w, then $L(w) = 0 + \gamma \cdot 0.1 + \gamma^2 \cdot 0.8 = 0.3$, again maximal, and w is the greedy node.

2.3. Optimistic Planning for Stochastic MDPs

In the stochastic case, applying an action may lead to different states with certain probabilities. We assume that the number of nonzero-probability next states is at most M for any transition, so continuous distributions are not allowed. Each node z in the planning tree now has up to KM children. Specifically, a node z labeled by state x is expanded by adding, for each action u, and for any state x' for which $f(x, u, x') > 0$, a new child node z' labeled by x'. Figure 2 shows such a tree, explicitly including also action nodes in circles. The children of x along action u are denoted $\mathcal{C}(z, u)$. Like in the OPD tree, state labels may be duplicated. Such a tree represents many possible stochastic evolutions of the system, e.g. for the sequence $[◀, ▶]$ there are four possible state trajectories in the tree of Figure 2: those ending in the 3rd, 4th, 7th, and 8th leaves at depth 2.

As before, b-values are associated to the nodes, but now they are computed with the

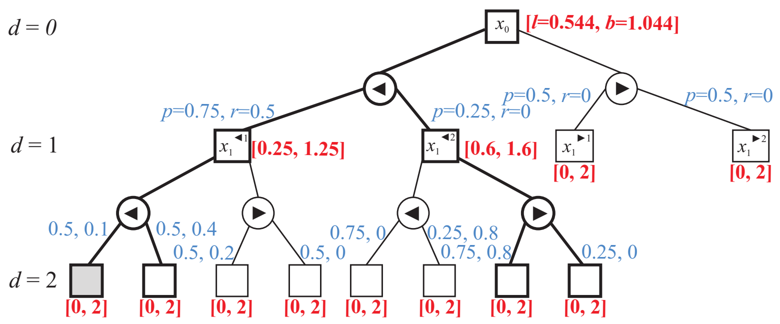

Figure 2: Example of OPMDP tree for $M = K = 2$. The squares are state nodes labeled by states, where superscripts index the possible actions and state outcomes, while subscripts are depths, which increase only with the state node levels. The actions $u \in \{\blacktriangleleft, \blacktriangleright\}$ are included as circle nodes. Transition arcs are labeled by probabilities and rewards in blue, and l and b-values are shown near the nodes in red. The thick subtree shows a tree policy. γ is again 0.5. (Figure best viewed in color.)

stochastic dynamic programming update:

$$b(z) = \begin{cases} \dfrac{1}{1-\gamma} = V_{\max} & \text{if } z \text{ is a leaf,} \\ \max\limits_{u} \sum\limits_{z' \in \mathcal{C}(z,u)} p(z')\left[r(z') + \gamma b(z')\right] & \text{otherwise} \end{cases} \tag{5}$$

where $p(z') = f(x(z), u, x(z'))$ and $r(z') = \rho(x(z), u, x(z'))$. E.g. in Figure 2, the b-value of the node of $x_1^{\blacktriangleleft,2}$ is $\max\{0.75(0 + \gamma \cdot 2) + 0.25(0.8 + \gamma \cdot 2), 0.75(0.8 + \gamma \cdot 2) + 0.25(0 + \gamma \cdot 2)\} = \max\{1.2, 1.6\} = 1.6$. The l-values are computed similarly but starting from 0 at the leaves. Quantities $b(z)$ and $l(z)$ keep their meaning of upper and lower bounds on $V^*(x(z))$.

To expand a node, an *optimistic tree policy* $h^\dagger$ is constructed, by starting from the root z_0 and recursively adding at each node z the M children along one action, the one achieving the maximal expected b-value: $\arg\max_u \sum_{z' \in \mathcal{C}(z,u)} p(z')\left[r(z') + \gamma b(z')\right]$. In contrast to an action sequence, a tree policy specifies an action choice for each inner state node reached under the previous choices, and so it defines a closed-loop control under the trajectory realizations up to the (varying) horizon of the tree. In Figure 2, the optimistic policy is shown by thick lines, e.g. the optimistic action branch at $x_1^{\blacktriangleleft,2}$ is $\blacktriangleright$ because, as seen in the calculations above, this action achieved the maximum in the b-value update. Then, a node among the leaves of this policy is selected by maximizing the contribution:

$$c(z) = P(z)\frac{\gamma^{d(z)}}{1-\gamma} \tag{6}$$

where $P(z)$ is the probability of reaching z, the product of the individual transition probabilities along the path. E.g. in Figure 2, the gray node – call it y – has a probability $P(y) = 0.75 \cdot 0.5 = 0.375$, and contribution $c(y) = P(y)\frac{\gamma^2}{1-\gamma} = 0.1875$. The reader can verify this contribution is maximal on the optimistic policy $h^\dagger$, and equal to that of y's immediate sibling, so one of these two nodes is expanded next. Finally, after n nodes have been expanded, an action that maximizes the expected l-value from the root is returned. The resulting method is called OP for stochastic MDPs (OPMDP) [10], and shown in Algorithm 2.

Algorithm 2 OP for stochastic MDPs

```
Input: state $x_0$, model $f, \rho$, budget $n$
 1: initialize tree $\mathcal{T}$ with root $z_0$, labeled by $x_0$
 2: for $i = 1, \dots, n$ do
 3:     build optimistic policy subtree, starting from $h^\dagger \leftarrow z_0$
 4:     while $\mathcal{L}(h^\dagger) \nsubseteq \mathcal{L}(\mathcal{T})$ do
 5:         retrieve a node $z \in \mathcal{L}(h^\dagger) \setminus \mathcal{L}(\mathcal{T})$
 6:         $u^\dagger = \arg\max_u \sum_{z' \in \mathcal{C}(z,u)} p(z') \left[ r(z') + \gamma b(z') \right]$
 7:         add children $\mathcal{C}(z, u^\dagger)$ to $h^\dagger$
 8:     end while
 9:     select leaf to expand: $z^\dagger \leftarrow \arg\max_{z \in \mathcal{L}(h^\dagger)} c(z)$
10:     expand $z^\dagger$, adding all its children to $\mathcal{T}$
11:     update $b$ and $l$-values on the tree
12: end for
Output: $u_0 = \arg\max_u \sum_{z' \in \mathcal{C}(z_0,u)} p(z') \left[ r(z') + \gamma l(z') \right]$
```

To get more insight into these choices, consider any tree policy h (not necessarily the optimistic one) on the current tree. Define:

$$B(h) = \sum_{z \in \mathcal{L}(h)} P(z)[L(z) + \gamma^{d(z)} b(z)] = \sum_{z \in \mathcal{L}(h)} P(z)[L(z) + \frac{\gamma^{d(z)}}{1-\gamma}]$$

where the meaning of $L(z)$ is unchanged from the deterministic case: the partial return up to z. $B(h)$ is an upper bound on the expected value of any infinite-horizon policy h_∞ starting with h. The optimistic policy $h^\dagger$ constructed as above maximizes B among all tree policies on the tree – so, like in OPD, the most promising partial solution is refined further. Also, note that:

$$L(h) = \sum_{z \in \mathcal{L}(h)} P(z) L(z)$$

is a lower bound on the values of policies starting with h, and that $B(h) = L(h) + \sum_{z \in \mathcal{L}(h)} c(z) =: L(h) + \delta(h)$. So the meaning of $c(z)$ (6) becomes clear: it is the contribution of leaf z to the uncertainty $\delta(h)$ on the value of tree policy h, and the algorithm expands the leaf that contributes the most to the uncertainty of the most promising solution. Finally, the action choice at the root is interpreted as choosing a policy h that maximizes $L(h)$ – a safe choice – and then applying the first action of h. So the main line of the algorithm is the same as in the deterministic case: it always refines an optimistic solution, and at the end chooses a safe one.

3. Online Learning for Optimistic Planning

To make the development easier to follow, we start by fully describing and analyzing the algorithm in the simpler, deterministic case. Building on this, we then describe more briefly the extension to the stochastic case.

3.1. Algorithm for the Deterministic Case

In its basic variant presented above, OPD discards the planning data right after using it to choose the current action, and then starts over from scratch at the next step. When the algorithm is applied online, the next state will be close to the current one, so much of the planning data remains relevant and discarding it is wasteful. Therefore, in this paper we introduce an approach to *reuse* the data. In particular, we focus on reusing b-values since they are crucial in guiding tree expansion. The l-values are only needed at the end, and if the tree was well developed it already has deep branches along near-optimal sequences, leading to good, near-optimal l-values.

Specifically, at the first step of online control, $k = 0$,[2] no data was seen so the b-values at the leaves are initialized using the uninformed upper bound $V_{\max}$, as in (3). So at the first step, we just run regular OPD. Once it terminates, we collect the state–b-value pairs $(x(z), b(z))$ of inner nodes on the tree that OPD developed. We use this set of pairs to learn an approximate value function $\hat{V}(x), \hat{V} : X \to [0, V_{\max}]$, using any function approximation technique. At the end of this section we will explain the connection to value iteration, which will clarify that the function learned is in fact the value function, justifying the notation.

At the next step, $k = 1$, when OPD develops the new tree the b-values at the leaves are initialized with $\hat{V}(x)$ rather than with the uninformed upper bound $V_{\max}$, thereby computing at each inner iteration of OPD *generalized* b-values at all nodes:

$$\hat{b}(z) = \begin{cases} \hat{V}(x(z)) & \text{if } z \text{ is a leaf,} \\ \max\limits_{z' \in \mathcal{C}(z)} [r(z') + \gamma \hat{b}(z')] & \text{otherwise} \end{cases} \tag{7}$$

Then, the state–b-value pairs on the new tree are collected, the approximator $\hat{V}$ is updated using these pairs, and so on at steps $k = 2, 3, \ldots$. To make it easier to distinguish, denote the algorithm that uses generalized b-values by OPD$^+$. It is obtained from Algorithm 1 by changing b-values to $\hat{b}$, and the update equation from (3) to (7). Of course, OPD is a special case of OPD$^+$, for $\hat{V}(\cdot) = V_{\max} = \frac{1}{1-\gamma}$. Note that even though a value function is learned online, the algorithm still uses the model to generate the tree.

To make the update of $\hat{V}$ more precise, denote the set of state–b-value pairs at the end of an application of OPD by $\mathcal{M}_{\text{new}}$ – a set of new training samples for the approximator:

$$\mathcal{M}_{\text{new}} = \{(x(z), \hat{b}(z)) \mid z \in \mathcal{T} \setminus \mathcal{L}_{\mathcal{T}}\} \tag{8}$$

Note that leaf b-values are *not* included in this set, because they were directly initialized with the previous approximate values, see (7), so they add no new information. Then, $\hat{V}$ is updated with these new samples. Exactly how this update is implemented depends on the particular approximator used, and we provide details in Section 3.4 for several memory-based approximators and a parametric one.

Algorithm 3 summarizes OPD with learning, the result of applying OPD$^+$ in the control loop. For conciseness, notation $\hat{V}$ is used from the first step, initialized to $\frac{1}{1-\gamma}$.

[2]To describe the overall algorithm, we need to distinguish between time steps, so in Section 3.1 we do *not* relabel each k to 0 anymore.

Algorithm 3 OPD with Learning (L-OPD)

```
Input: model $f, \rho$, discount factor $\gamma$, budget $n$,
    experiment length $T$
1: initialize $\hat{V}(x) = \frac{1}{1-\gamma}$, $\forall x$
2: for $k = 0, \dots, T-1$ do
3:     measure state $x_k$
4:     at state $x_k$, apply OPD$^+$ with $\hat{V}$, returning $u_k$
5:     apply action $u_k$
6:     learn: create $\mathcal{M}_{\text{new}}$ with (8) and use it to update $\hat{V}$
7: end for
```

The b-value updates (3) or (7) implement a local, asynchronous form of value iteration (VI) [3], where the updates are applied backwards in the tree towards the root. This VI is local to the root state x_0, because it only concerns states reachable from it – those on the current tree. Moreover, these states are well-chosen by the optimistic criterion so that they are likely to be along optimal trajectories. As the tree grows, the b-values near the root converge to the optimal values V^* [24], with the b-values closer to V^* when leaf (initial) values are more accurate. Unlike in standard VI where the initial values can be arbitrary, in OPD, to ensure the nodes are expanded in a proper optimistic order, the leaf b-values $V_{\max}$ or $\hat{V}$ should be upper bounds on the optimal value of the leaf state, $V^*(x(z))$, which by induction implies the same remains true at any node. By learning $\hat{V}$ and reusing it in the next trees, we effectively connect the VI updates together, whereas original OPD would run VI from scratch at each step. We therefore expect convergence close to V^* *along* the online experiment.

If the approximator is chosen well, it will also provide *generalization* of the b-value across states, as in standard approximate VI [29]. This is because $\hat{V}$ is a function of the underlying state x, not of the node z like the b-values. Therefore, due to generalization the approximator will identify and collate information from nearby states. This is an additional benefit over the original OPD, which did not generalize even if it encountered the same state again in the tree.

Next, we analyze the algorithm in general, for any approximator.

3.2. Analysis in the Deterministic Case

In this section we characterize the performance of OPD$^+$, with learned b-values, comparing it with the original OPD. There are two novel elements in this analysis: on the one hand, the b-values may be closer to the optimal values V^* than the uniformed bounds in OPD, which improves performance; on the other hand, due to approximation error b-values may no longer be admissible (true upper bounds), which means some nonoptimistic nodes are expanded and performance decreases. As it will turn out, for small enough approximation error it pays off to use the approximate b-values.

Since online planning algorithms choose actions locally, at each state they encounter during their interaction with the system, a good way to measure their performance is the simple regret, see e.g. [9]:

$$\mathcal{R}(x_0) = V^*(x_0) - Q^*(x_0, u_0) \tag{9}$$

where u_0 is the action chosen by the algorithm at the root state x_0. This measures the loss incurred by taking action u_0 and then acting optimally, with respect to acting optimally from the first step, thereby isolating the quality of the action taken. Acting optimally realizes a regret of 0.

To keep the analysis generic, we abstract away the details of the approximator and define ε so that the approximation is uniformly accurate up to ε for any state appearing on the tree, $|\hat{V}(x) - V^*(x)| \leq \varepsilon$ (since the largest possible error is $\frac{1}{1-\gamma}$, such an ε always exists). The analysis provides confidence that when ε is small, OPD$^+$ performs well, and in particular that it outperforms the original OPD. Although the value of ε is unknown to the algorithm, analyzing the case of small ε is (informally speaking) reasonable, since as explained above L-OPD is a variant of approximate value iteration, which is known to converge close to V^* [29]. An explicit analysis of ε is left for future work.

The performance of OPD$^+$ is dictated by the depth of the explored tree, so, following the line of analysis of OPD [9], we first characterize this tree. Define $\hat{B}(z) = L(z) + \gamma^d(z)\hat{b}(z)$, an approximate upper bound on values of sequences starting with $\mathbf{u}(z)$, and also $v(z) = L(z) + \gamma^d(z)V^*(x(z))$, the best infinite-horizon return among these sequences.

Lemma 1. OPD$^+$ *only expands nodes z for which $v(z) \geq V^*(x_0) - \varepsilon(1 + \gamma^{d(z)})$.*

Proof: At any iteration i, denote by z_i^* the deepest node on the current tree that lies along some optimal sequence (which achieves $V^*(x_0)$). Since the expanded node $z_i^\dagger$ has the largest B-value, $\hat{B}(z_i^\dagger) \geq \hat{B}(z_i^*)$. Further, observe that for any leaf z, the B-value satisfies $|\hat{B}(z) - v(z)| = \gamma^{d(z)}|\hat{b}(z) - V^*(x(z))| \leq \gamma^{d(z)}\varepsilon$ by the uniform-error property. Applying this inequality to z_i^* and $z_i^\dagger$, we obtain $\hat{B}(z_i^*) \geq V^*(x_0) - \gamma^{d(z_i^*)}\varepsilon$ and $v(z_i^\dagger) + \gamma^{d(z_i^\dagger)}\varepsilon \geq \hat{B}(z_i^\dagger)$. Combining these with $\hat{B}(z_i^\dagger) \geq \hat{B}(z_i^*)$ obtained above, we get $v(z_i^\dagger) + \gamma^{d(z_i^\dagger)}\varepsilon \geq V^*(x_0) - \gamma^{d(z_i^*)}\varepsilon$, or equivalently:

$$v(z_i^\dagger) \geq V^*(x_0) - \varepsilon(\gamma^{d(z_i^*)} + \gamma^{d(z_i^\dagger)}) \geq V^*(x_0) - \varepsilon(1 + \gamma^{d(z_i^\dagger)})$$

■

We can now give the main result for OPD$^+$.

Theorem 1. *The regret of* OPD$^+$ *satisfies:*

$$\mathcal{R}(x_0) \leq \varepsilon + \gamma^{d(z^\dagger)}\hat{V}(x(z^\dagger)) \leq \varepsilon + \frac{\gamma^{d^+_{\max}}}{1-\gamma}$$

where $z^\dagger$ is the node minimizing $\gamma^{d(z)}\hat{V}(x(z))$ among all expanded nodes, and $d^+_{\max}$ is the depth of the deepest expanded node.

*Proof:*Consider any iteration i, and denote by z_i^g the greedy leaf, maximizing $L(z)$ on the current tree. Then:

$$L(z_i^\dagger) \leq L(z_i^g) \leq L(z_n^g) \leq Q^*(x_0, u_0) \leq V^*(x_0) \leq B(z_i^\dagger) + \gamma^{d(z_i^*)}\varepsilon \qquad (10)$$

where z_i^* keeps the same meaning as in the proof of Lemma 1. These inequalities hold, in order, because: (i) z_i^g maximizes L on the current tree; (ii) $L(z_i^g)$ increases with i; (iii) the action sequence of z_n^g starts with u_0, and by the definition of Q^*; (iv) by definition; (v) and finally from Lemma 1, $B(z_i^\dagger) \geq V^*(x_0) - \gamma^{d(z_i^*)}\varepsilon$. Therefore:

$$V^*(x_0) - Q^*(x_0, u_0) \leq B(z_i^\dagger) + \gamma^{d(z_i^*)}\varepsilon - L(z_i^\dagger) \leq \varepsilon + \gamma^{d(z_i^\dagger)}\hat{V}(x(z_i^\dagger)) \leq \varepsilon + \frac{\gamma^{d(z_i^\dagger)}}{1-\gamma}$$

and because this is true at any i, the proof is complete. ■

Remarks: Lemma 1 limits the number of nodes that must be expanded, and therefore, given a budget n, implicitly leads to a minimal depth reached by OPD$^+$. The regret bound of Theorem 1 shrinks with this depth.

To simplify the statements and the upcoming comparison with OPD, a conservative step was made both in the lemma and the theorem: $d(z_i^*)$, the largest depth along optimal sequences, was replaced with 0. In reality, as the tree grows the term $\gamma^{d(z_i^*)}\varepsilon$ shrinks. Further, the better the b-values, the more likely OPD$^+$ is to expand optimal sequences deeper, and the faster $d(z_i^*)$ will grow. Also, replacing $\gamma^{d(z^\dagger)}\hat{V}(x(z^\dagger))$ by $\frac{\gamma^{d^+_{\max}}}{1-\gamma}$ is conservative. □

We next overview the guarantees of OPD [9] as a baseline for our algorithm. In OPD, the nodes that may be expanded are characterized by $v(z) \geq V^*(x_0) - \frac{\gamma^{d(z)}}{1-\gamma}$ (compare to Lemma 1), while the regret bound simplifies to $\mathcal{R}(x_0) \leq \frac{\gamma^{d_{\max}}}{1-\gamma}$ (compare to Theorem 1, noting that the maximal depths are usually different between OPD and OPD$^+$). Further, since the regret bound shrinks to 0 as n (and thus $d_{\max}$) grows large, an asymptotic analysis of the convergence rate can be provided, see [9].

In OPD$^+$, since asymptotically the constant error ε dominates, a large-n analysis is less informative. The advantage of OPD$^+$ is manifested in the practical case of small budgets n, when the uncertainty terms due to the shallower tree dominate. To characterize this, let us first investigate when Lemma 1 leads to a smaller set of expandable nodes than in OPD. This happens when:

$$\varepsilon(1 + \gamma^{d(z)}) \leq \frac{\gamma^{d(z)}}{1-\gamma}, \text{ or: } d(z) \leq \frac{\log \frac{1/(1-\gamma)-\varepsilon}{\varepsilon}}{\log 1/\gamma}$$

So, as long as the expanded tree is shallower than this bound, OPD$^+$ is expected to reach larger depths than OPD given the same budget.

Finally, the overall regret bound is better for OPD$^+$ when:

$$\varepsilon \leq \frac{\gamma^{d_{\max}} - \gamma^{d^+_{\max}}}{1-\gamma}$$

i.e. when ε and n are small, because then the left-hand side of the equation is small (due to ε) and the right-hand side is large (because $d^+_{\max} > d_{\max}$ for small ε and n, as shown above).

3.3. Stochastic Case

In stochastic problems, using OPMDP, the b-values keep their meaning of approximate value function, and the b-value updates remain a local form of VI on the current tree. Therefore, the extension to learn b-values entirely parallels the deterministic case, and is summarized next. After each step, the set of new training samples $\mathcal{M}_{\text{new}}$ is constructed with (8) and the approximator $\hat{V}$ is updated. During planning, the b-values are computed using:

$$\hat{b}(z) = \begin{cases} \hat{V}(x(z)) & \text{if } z \text{ is a leaf,} \\ \max\limits_u \sum\limits_{z' \in \mathcal{C}(z,u)} p(z') \left[r(z') + \gamma \hat{b}(z') \right] & \text{otherwise} \end{cases} \tag{11}$$

rather than with (5). The resulting algorithm is called OPMDP$^+$. Closed-loop planning is the same as Algorithm 3, except it uses OPMDP$^+$ instead of OPD$^+$.

For the analysis, define:

$$\alpha(z) = \sup_{h, z \in \mathcal{L}(h), c(z) = \max_{z' \in \mathcal{L}(h)} c(z')} \delta(h)$$

namely, the largest uncertainty of any tree policy h among whose leaves z has the largest contribution $c(z)$, see (6). Intuitively $\alpha(z)$ quantifies the global impact of z on the planning tree. Also, denote $v(h)$ the best value among policies starting with h, formally defined as $v(h) = \sum_{z \in \mathcal{L}(h)} P(z)[L(z) + \gamma^{d(z)} V^*(x(z))]$. We recall the approximator error ε, defined so that $|\hat{V}(x) - V^*(x)| < \varepsilon$.

Lemma 2. OPMDP$^+$ *only expands nodes z belonging to some policy h so that $v(h) \geq V^*(x_0) - \varepsilon[1 + \alpha(z)(1-\gamma)]$.*

Theorem 2. *The regret* (9) *of* OPMDP$^+$ *satisfies:*

$$\mathcal{R}(x_0) \leq \varepsilon + \alpha_{\min}$$

where $\alpha_{\min} = \min_i \alpha(z_i^\dagger)$ is the smallest impact among the expanded nodes $z_i^\dagger$ at all iterations i.

Remarks: The regret is given by the smallest impact among expanded nodes. Lemma 2 implicitly bounds the number of nodes that must be expanded in order to reach one with a certain (small) impact, and so for a given n leads to a (small) regret.

The proofs are rather technical extensions of the deterministic case, and are omitted here. Instead, to understand the relationship with the OPD$^+$ results, introduce the notion of *effective depth* $\tilde{d}(z)$ so that $\alpha(z) = \frac{\gamma^{\tilde{d}(z)}}{1-\gamma}$. By using $\tilde{d}(z)$, the condition in Lemma 2 is clearly analogous to that in Lemma 1; and similarly to Theorem 1, $\alpha_{\min} = \frac{\gamma^{\tilde{d}_{\max}}}{1-\gamma}$, where $\tilde{d}_{\max}$ is the largest effective depth among expanded nodes. Note that the impact (uncertainty) $\alpha(z)$ is spread over leaves at many depths, $\alpha(z) = \sum_{z \in \mathcal{L}(h)} P(z) \frac{\gamma^{d(z)}}{1-\gamma}$ for some h, so $\tilde{d}(z)$ is fictitious in this sense; it would only correspond to a real depth if all leaves $\mathcal{L}(h)$ were at the same depth. □

In the original OPMDP, the condition on expanded nodes simplifies to $v(h) \geq V^*(x_0) - \alpha(z)$, and the regret bound to $\mathcal{R}(x_0) \leq \alpha_{\min}$. As long as the algorithm expands nodes with large impacts compared to the error ε, i.e. in the practically relevant case of relatively small budgets, OPMDP$^+$ will reach nodes with smaller impacts. So it will lead to a smaller $\alpha^+_{\min}$, and (since ε is comparatively small) to a better regret bound. This can be seen by reexamining the conditions from the deterministic case, and replacing the real depths there by effective depths and their corresponding impacts.

3.4. Learning Methods

Five approximators are selected to be representative, as follows. First, since the data is naturally organized as a set of points, we favor memory-based methods over parametric ones. The first method, Lipschitz approximation, is novel and specifically adapted for the features of OP. Further, we select local linear regression (LLR) [30] because it is simple and computationally inexpensive; and at the other end of the spectrum, an advanced nonparametric technique called least-squares support vector regression (LSSVR), known for its good generalization [31, 32]. The fourth approximator is a simple empirical extension of LSSVR that is newly proposed here, and imports the locality of LLR. A hybrid, local LSSVR (L-LSSVR) is obtained with computational complexity similar to LLR but hopefully preserving approximation quality from LSSVR. Finally, we do consider one parametric method as a baseline, again chosen for good generalization: feedforward neural networks (NNs) [33].

Denote the current memory by $\mathcal{M} = \{(x_i, b_i) \mid i = 1, \ldots, N\}$, which stores a number N of state–b-value pairs. For the parametric approximator, denote the vector of parameters by $\varphi \in \mathbb{R}^W$. Below, we briefly present every method, including also its list of *parameters*, a description of the *training* process, and an investigation of the computational *complexity.* The memory-based methods additionally involve a memory-management process, which will be described in the next section. The discussion on complexity does not consider this process, and it does not include either the 'regular' steps of the OP algorithms (which take between $O(n \log n)$ and $O(n^2)$ computation, depending on how difficult the problem is, see [10]). Instead, we focus on the complexity of the learning part: training and using the approximator. We emphasize that the methods can be applied without any changes in both deterministic and stochastic problems.

Lipschitz approximation relies on the assumption that the optimal value function is Lipschitz continuous with constant $\ell \geq 0$:

$$|V^*(x) - V^*(x')| \leq \ell \|x - x'\|$$

where $\|\cdot\|$ is a norm. The property holds under appropriate conditions when the transitions and rewards are Lipschitz continuous [34]. Given some b-value $b(x) \geq V^*(x)$, we then have $b(x) + \ell\|x - x'\| \geq V^*(x) + \ell\|x - x'\| \geq V^*(x')$. Taking the most favorable left hand side among all the samples, we construct the following approximate b-function:

$$\hat{V}(x) = \min \left\{ \min_{i=1,\ldots,N} \left[b_i + \ell\|x - x_i\| \right], \frac{1}{1-\gamma} \right\} \tag{12}$$

If ℓ is not underestimated (i.e. the smoothness of b is not overestimated), then $\hat{V}$ remains a true upper bound for any x, ensuring that node expansions are strictly optimistic. *Parameters:* The true Lipschitz constant ℓ is difficult to find in practice, so it is treated instead as a tuning parameter of the algorithm. *Training:* consists of adding the new samples $\mathcal{M}_{\text{new}}$ to the memory $\mathcal{M}$. *Complexity:* is dominated by calling the approximator at every node expansion, where each call involves a linear search[3] for the minimum b-

[3]The following applies to the memory search and matrix inversion steps of all the memory-based learning methods. Efficient algorithms are available to search the memory, such as kd-trees, which reduce the $O(N)$ factor to $O(\log N)$, but in our Matlab implementation we use linear searches, which run fast using built-in code. The complexity $O(m^3)$ for inverting a matrix of size m holds for e.g. Gauss-Jordan elimination, and can be brought down to approximately $O(m^{2.3})$ with the Coppersmith-Winograd method.

value, leading to an overall complexity $O(nN)$.

LLR estimates the value of a query point x by fitting a hyperplane trough a given number k_{nn} of samples from the memory that are closest to x according to some metric on X. The predicted value $\hat{V}(x)$ is taken as the value of the hyperplane at x. *Parameters:* Once the metric is chosen (we use the standard Euclidean norm), LLR's only parameter is the number of neighbors k_{nn} taken into account for regression. *Training:* consists, as for the Lipschitz appproximator, of adding the new samples to the memory. *Complexity:* is again dominated by calling the approximator. This time, a call involves a nearest-neighbor search and solving a linear system of $k_{nn}+1$ equations (where the extra equation is for an affine term used to offset the hyperplane along the vertical axis), leading to the overall complexity of $O(n(N + (k_{nn} + 1)^3))$.

LSSVR maps the input space onto a higher dimensional feature space, then in this feature space a hyperplane is fitted trough the samples. In order to avoid making calculations in that high dimensional space the kernel trick is used, which makes the mapping implicit [35]. We use LSSVR with Gaussian kernels. *Parameters:* the kernel width σ and a regularization factor C. *Training:* is performed after each memory update, with all available data. *Complexity:* is dominated by this training step, which involves the inversion of a data-size (plus one) matrix having a complexity $O((N + 1)^3)$.

L-LSSVR avoids the large-matrix inversion required by LSSVR, by training a local model to estimate $\hat{V}$ on demand, using (like LLR) only the nearest neighbors from the memory. *Parameters:* kernel-width σ, regularization factor C, number of nearest neighbors k_{nn}. *Training:* consists of adding the new data to the memory. *Complexity:* the single inversion of a large matrix from LSSVR is traded for many inversions of smaller matrices plus a nearest-neighbor search, leading to $O(n(N + (k_{nn} + 1)^3))$ overall.

NNs are parametric so, differently from the other approximators, they do not need to store and maintain a memory. We use single hidden layer NNs with sigmoidal activation functions and a linear output function. Before the network is used, it is pre-trained such that $\hat{b}(x) \approx \frac{1}{1-\gamma}$, $\forall x \in X$ (the uninformed bound $\frac{1}{1-\gamma}$ is still explicitly used at the first step, as described in Algorithm 3). *Parameters:* include the number of neurons in the hidden layer n_{hl}, a regularization factor C and the number of epochs n_{epoch} used for training. *Training:* is performed at every step with the Levenberg-Marquardt algorithm in batch mode, using the new batch of data $\mathcal{M}_{\text{new}}$. *Complexity:* is dominated by the training process, which is $O(n_{epoch}W^3)$ with W the number of parameters in the network (related to the dimensionality of X and the number of hidden neurons). A similar cost is paid to perform the pretraining.

Table 1: Complexity of learning methods

Method	Complexity
Lipschitz	$O(nN)$
LLR	$O(n(N + (k_{nn} + 1)^3))$
LSSVR	$O((N + 1)^3)$
L-LSSVR	$O(n(N + (k_{nn} + 1)^3))$
NN	$O(n_{epoch}W^3)$

Table 1 collects the computational complexity of all the methods. From this table, the Lipschitz learner is expected to have the lowest complexity, smaller than the three other memory-based methods, LLR, LSSVR and L-LSSVR. Usually $k_{nn} \ll N$, in which case LLR and L-LSSVR invert much smaller matrices and have a lower complexity than LSSVR. NNs are difficult to compare with the other methods, since the method is very different and depends on other variables.

3.5. Memory Management

The memory-based learning methods must manage the contents of the memory, for several reasons:

- Computation time: nonparametric function approximators must perform various operations on the memory, such as detecting the nearest neighbors of a sample. Since the computational complexity of these operations scales with the memory size (see again Table 1), maintaining a large memory will lead to high computational costs.
- Non-stationarity: let us consider the application of OPD at two subsequent time steps, at which algorithm explores the same state. During the first time step the b-value estimate of that state is stored. When an improved estimate becomes available at the second time step, without memory management the previous estimate is also kept in the memory. Then, an estimate that is made later of that state's value will use outdated information.
- Size: when unmanaged, the size of the memory will grow rapidly. Since memory is cheap nowadays, this is a smaller problem than those above. Nevertheless, physical limits can be reached.

New samples are always presented in batches corresponding to the interior of the developed trees. To derive a good memory management procedure, note first that trees developed at subsequent steps will contain many identical states, since the state resulting from applying the chosen action is by definition one of the nodes at depth 1 in the planning tree. Such duplicate states should be merged into a single point, keeping the smallest b-values across all duplicates. Duplicate removal is always applied, in all memory-based methods.

The other steps are different between the Lipschitz method and the other approximators. The specialized form of the Lipschitz approximator allows deriving an adapted memory management algorithm. Specifically, if a pair (x_i, b_i) in the memory satisfies $b_i \geq \hat{V}_{-i}(x_i)$, where $\hat{V}_{-i}$ is computed with (12) *without* including pair i in the minimum, then the point does not contribute to $\hat{V}$ at any state, and it can be removed.[4]

For the other three memory-based methods (LLR, LSSVR, and L-LSSVR), we propose a heuristic management technique based on the assumption that old samples are redundant in regions where new samples become available. Define $D(\mathcal{M}_{new})$ as the smallest hyperrectangle in $\mathbb{R}^p$ that contains all the states x in $\mathcal{M}_{new}$. All the samples of $\mathcal{M}$ that lie inside $D(\mathcal{M}_{new})$ are removed, and then the new samples are added to

[4]If the memory requirements are still too large, an extra heuristic step can be taken, by extending the redundancy formula to assign a score $s_i = \hat{V}_{-i}(x_i) - b_i$ to each point, and removing points in increasing order of their score; this removes the least-contributing points first, preserving as much information in the memory as possible. We will not use this heuristic in the sequel.

the memory. The choice of using a hyperrectangle domain to assess redundancy makes this memory management method quite rough. It could be refined e.g. by using the convex hull of the new samples. However, calculating a convex hull is computationally expensive and preliminary experiments showed no significant improvement, so we use the hyperrectangle method in the paper. Algorithm 4 presents the memory management method.

Algorithm 4 Memory management

```
Input: current memory $\mathcal{M}$, new batch of samples $\mathcal{M}_{new}$
1: if approximator is LLR, LSSVR, or L-LSSVR then
2:     compute hyperrectangle $D(\mathcal{M}_{new})$
3:     $\mathcal{M} \leftarrow \mathcal{M} \setminus D(\mathcal{M}_{new})$
4: end if
5: $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{M}_{new}$
6: remove duplicates in $\mathcal{M}$, keeping the smallest b-value
7: if approximator is Lipschitz then
8:     remove all redundant points $i$, for which $b_i \geq \hat{V}_{-i}(x_i)$
9: end if
Output: $\mathcal{M}$, the updated memory
```

Recall that OP with NNs does not use a memory management mechanism, relying instead on the incremental training process to track the time-varying b-function.

4. Experimental Study

4.1. Deterministic Case: Resonating Arm

We investigate the effectiveness of learning for OPD in experimental studies on practically relevant problems involving two variants of a resonating robot arm [36]. In these experiments, the original OPD is compared to L-OPD with all five approximators, for different values of the node expansion budget n.

The resonating arm (RA) is designed to perform pick-and-place tasks in an energy-efficient manner, exploiting a nonlinear spring mechanism. We use variants of the RA with 1 or 2 degrees of freedom (DOF). A schematic representation of the 1DOF variant is shown in Figure 3. The mechanism consists of a large pulley (radius r_1) and a small pulley (radius r_2), connected by a belt and a spring. The elongation Δ_l of this spring, given with respect to the angle θ of the arm, is calculated as:

$$\Delta_l = \left(r_2 \sin\left(\frac{r_1}{r_2}\theta\right) + r_1 \sin(\theta)^2 + r_2 \cos(\frac{r_1}{r_2}\theta) - r_1 - r_2 - d_l)^2\right)^{\frac{1}{2}} - l_0 \qquad (13)$$

The physical parameters of the system are declared in Table 2. With the elongation known, the potential energy stored in the spring is $P(\Delta l) = \frac{1}{2}k\Delta l^2 + F_0 \cdot \Delta l$, while the torque working on the arm, τ_{spring}, is the derivative of P. When seen as a function of θ, P has two local minima at $\theta = \pm 0.85$ rad. At these two working points, almost no torque needs to be applied by the motor to stabilize the arm, nevertheless there is energy stored in the spring. When the arm needs to traverse from one of these working points to the other, this stored energy is used.

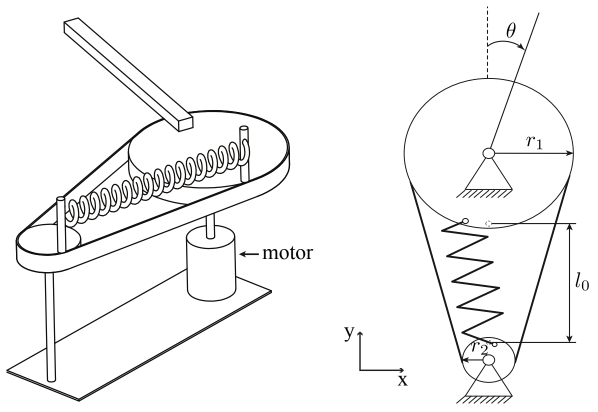

Figure 3: The resonating arm with one degree of freedom.

Table 2: Parameters of the resonating arm

Symbol	Description	Value
r_1	radius of large pulley	0.105 m
r_2	radius of small pulley	0.02 m
d_l	distance between pulleys	0.1 m
l_0	initial length of spring	0.1 m
k	spring constant	150 N/m
F_0	pre-stress of spring	5 N

The continuous-time model of the RA dynamics is:

$$\ddot{\theta} = \frac{1}{J}(\tau_{\text{friction}} + \tau_{\text{spring}} + u) \tag{14}$$

where the friction torque τ_{friction} is modeled as a combination of viscous and Coulomb friction and u is the torque delivered by the motor. In our experiments, the discretized actions $U = \{-2, 0, 2\}$ are used, since they are sufficient to solve the task, and planning algorithms work best with a small action space. The system state is $x = [\theta, \dot{\theta}]^T$ and the sampling time is $T_s = 0.05$ s. The objective is to control the arm from the first working point, $x_0 = [-0.85, 0]^T$, to the second working point, $x_{\text{goal}} = [0.85, 0]^T$. This objective is represented by the reward function:

$$\rho(x') = e^{-(x'-x_{\text{goal}})^T \mathbf{S}(x'-x_{\text{goal}})} \tag{15}$$

where $\mathbf{S}$ is a scaling matrix that assigns importance to the deviation from the goal state for each state variable. In the 1DOF experiments $\mathbf{S}$ is taken as $\mathbf{S} = \text{diag}(2, 0.04)$. The discount factor is $\gamma = 0.95$.

To obtain the 2DOF RA, a second link (lower arm) is added to the the 1DOF RA, actuated by a motor in the 'elbow'. The resulting configuration is shown in Figure 4. In the experiments with the 2DOF variant, the arm is controlled from $x_0 = [-0.85\ 0\ 0\ 0]^T$ to $x_{\text{goal}} = [0.85\ 0\ 0\ 0]^T$, where the state vector is $x = [\theta_1\ \theta_2\ \dot{\theta_1}\ \dot{\theta_2}]^T$. Nine discretized actions $U = \{-2, 0, 2\} \times \{-2, 0, 2\}$ are used. The reward function has the same form as in (15), but now with the weighting matrix $\mathbf{S} = \mathrm{diag}(10, 5, 0, 0)$. Thus both links must reach the goal position, but the first link is considered more important so it is assigned a larger weight; note that stabilizing the links at the goal position implicitly requires zero velocity even though the velocities are not directly considered in the rewards.

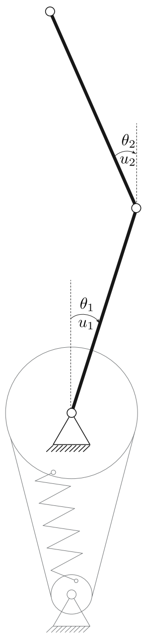

Figure 4: The 2DOF resonating arm.

Results for the 1DOF Resonating Arm. In the experiments for the 1DOF RA, we compare L-OPD with original OPD, and also with several other state-of-the-art planning methods: SOO for planning (SOOP) [13], Lipschitzian planning (LP) [37, 13], and hierachical OLOP (HOLOP) [12], for details see the respective references. We must note here that these methods find continuous-valued actions; that LP uses Lipschitz continuity of the dynamics and rewards, not of the value function like the learning variant; and that HOLOP samples solutions stochastically. The experiments are performed for planning budgets $n = 25,\ 50,\ 75,\ 100$, and for each value of n, the method-dependent parameters are varied as follows:

- L-OPD with Lipschitz: $\ell \in \{0.5, 1, 1.5, \ldots, 7.5\}$.
- L-OPD with LLR: $k_{nn} \in \{3, 5, 7, 9, 11\}$.
- L-OPD with LSSVR: $C \in \{500, 550, 600, \ldots, 1000\}$ and $\sigma \in \{0.1, 0.2, 0.3, 0.4\}$.
- L-OPD with L-LSSVR: $C \in \{400, 450, 500, \ldots, 1000\}$, $\sigma \in \{0.1, 0.2, 0.3, 0.4\}$, and $k_{nn} \in \{11, 13, 15, 17\}$.
- L-OPD with NN: $n_{hl} \in \{10, 20, 30, 40, 50\}$, $C \in \{0, 0.2, \ldots, 0.8\}$, and $n_{epoch} \in \{250, 300, 350, \ldots, 1000\}$.

- SOOP: parameter $\alpha \in \{0.1, 0.2, \dots, 0.9\}$.
- LP: Lipschitz constant $G \in \{0.1, 0.2, \dots, 1.5\}$.
- HOLOP: sequence length $K \in \{2, 3, 4, 5, 10, 15, 20, 25, 30\}$.

Experiments are performed for all the combinations of parameter values. The results shown below for each n are the best across these parameter grids (those with largest discounted returns along the trajectory).

Table 3: 1DOF resonating arm, returns

Method	$n = 25$	$n = 50$	$n = 75$	$n = 100$
OPD	5.38	5.42	5.43	5.43
L-OPD with Lipschitz	5.44	5.44	5.44	5.44
L-OPD with LLR	5.41	5.43	5.44	5.43
L-OPD with LSSVR	5.40	5.44	5.44	5.43
L-OPD with L-LSSVR	5.40	5.44	5.44	5.44
L-OPD with NN	5.44	5.44	5.44	5.44
SOOP	4.90	5.12	5.17	5.19
LP	4.52	4.77	4.79	4.90
HOLOP	4.35	4.60	4.63	4.72
HOLOP uncertainty	±0.21	±0.12	±0.18	±0.19

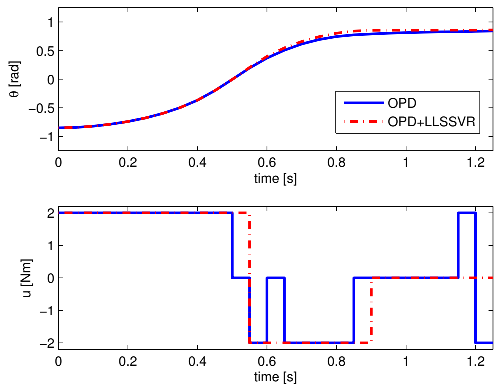

Figure 5: 1DOF resonating arm, controlled trajectory.

Table 3 shows the returns obtained. Looking first at OPD with and without learning, the learning methods all form an improvement over the original algorithm. Both Lipschitz learning and NNs achieve the largest return for all computational budgets. However, the improvements are only marginal, which is also seen in Figure 5 where the controlled trajectories and control signals of OPD and L-OPD with L-LSSVR are compared for $n = 50$. L-OPD with L-LSSVR obtains a bang-bang solution, which is near-optimal in this problem.[5] The small improvements are partly due to the nature of the solution to this problem; Figure 5 shows that the control signals of both methods are equal until $t = 0.55$ s, which means that possible differences in rewards appear after that time. Because of the discounted setting, these rewards have a small influence on the return. The second reason for the small improvements is simply that OPD is already able to find a good solution in this not too difficult problem. Comparing now to the state-of-the-art methods, they all obtain poorer returns than OPD and therefore than the learning variants. This is likely because they search for continuous actions while the bang-bang solution favors discrete actions. Note that, since HOLOP is a stochastic algorithm, mean values are shown with the half-length of the 95% confidence interval.

Table 4: 1DOF resonating arm, average tree depths and horizons (rounded to integer)

Method	$n = 25$	$n = 50$	$n = 75$	$n = 100$
OPD	8	13	26	28
L-OPD with Lipschitz	12	18	22	27
L-OPD with LLR	13	24	37	41
L-OPD with LSSVR	13	22	35	43
L-OPD with L-LSSVR	13	22	37	49
L-OPD with NN	13	26	35	46
SOOP	3	3	4	5
LP	9	20	23	27
HOLOP	3	3	3	3

From Table 4, which shows the planning depths averaged across all steps of the trajectory, it can be seen that the learning methods lead to increases in the planning depth (with an outlier for the Lipschitz approximator when $n = 75$ and $n = 100$). This means that despite the small performance difference, learning does increase the exploration efficiency. SOOP, LP, and HOLOP do not build trees, and the quantity most similar to depth is a maximal search horizon, which is shown in the table (and for HOLOP in fact equals K). The LP horizons are comparable to the depths of OPD, and much smaller for SOOP and HOLOP.

Table 5 shows the execution times. The state-of-the-art methods take similar time to OPD, with HOLOP more expensive for low n. The execution times of learning methods largely follow the expectations from the computational complexity study of Section 3.4.[6] Note that the time needed by the Lipschitz approximator is comparable to LLR, due to the more computationally intensive sample redundancy test used by Lipschitz learning, which was not taken into account in Section 3.4. Further, even if complexity is theoretically the same for LLR and L-LSSVR, the execution times of L-LSSVR are higher because of the need to compute the Gaussian kernels. NNs are very slow. We postpone a definitive discussion on the choice of approximator until the results in the 2DOF case, where the performance differences are clearer.

Next, the performance of the learning methods for OPD is further investigated for the 2DOF RA, a more challenging control problem.

[5]This was verified by computing a solution using an approximate VI algorithm with an accurate approximator [3].

[6]Note also that in our Matlab implementation, L-OPD cannot yet be used in real-time control.

Table 5: 1DOF resonating arm, execution times in seconds

Method	$n = 25$	$n = 50$	$n = 75$	$n = 100$
OPD	0.59	1.17	1.97	3.28
L-OPD with Lipschitz	0.86	1.84	2.78	3.73
L-OPD with LLR	0.95	2.04	3.21	4.02
L-OPD with LSSVR	0.78	1.61	2.39	3.42
L-OPD with L-LSSVR	1.00	2.09	3.28	4.73
L-OPD with NN	14.49	20.83	27.61	36.55
SOOP	0.87	1.34	1.91	2.53
LP	0.62	1.15	1.79	2.26
HOLOP	1.16	1.84	2.41	2.03
HOLOP uncertainty	±0.06	±0.10	±0.04	±0.12

Results for the 2DOF Resonating Arm. For the 2DOF RA, experiments are performed for OPD with and without learning, for budgets $n = 250,\ 500,\ 750,\ 1000$ and the following method-dependent parameters:

- L-OPD with the Lipschitz approximator and NN: the parameters are the same as in the 1DOF case.
- L-OPD with LLR: $k_{nn} \in \{13, 17, 21, 25, 29\}$.
- L-OPD with LSSVR: $C \in \{400, 500, 600, \dots, 1000\}$ and $\sigma \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$.
- L-OPD with L-LSSVR: $C \in \{200, 300, 400, \dots, 1000\}$, $\sigma \in \{0.1, 0.2, 0.3, \dots, 0.6\}$ and $k_{nn} \in \{9, 11, \dots, 17\}$.

The reason for the increased number of neighbors in LLR is that for a higher dimensional space, the method needs more samples to create a good fit. The grids of LSSVR and L-LSSVR differ from the 1DOF case because each time a method's best solution was found at the edge of a grid, we increased the grid. Like before, the best results across all combinations of parameters are reported.

Table 6: 2DOF resonating arm, returns

Method	$n = 250$	$n = 500$	$n = 750$	$n = 1000$
OPD	2.11	2.11	2.11	2.33
L-OPD with Lipschitz	3.50	3.99	4.14	4.39
L-OPD with LLR	3.59	4.43	3.98	3.92
L-OPD with LSSVR	4.61	4.98	5.03	4.78
L-OPD with L-LSSVR	5.05	5.07	5.05	5.04
L-OPD with NN	5.12	5.12	5.04	5.06

Table 6 shows the returns obtained by each method, while Table 7 shows the average planning depths. It can directly be seen that all learning methods achieve a significant improvement over original OPD in this more challenging 2DOF system. Figure 6 confirms this conclusion via a controlled trajectory for OPD and L-OPD with NN for

Table 7: 2DOF resonating arm, average depths (rounded to integer)

Method	$n = 250$	$n = 500$	$n = 750$	$n = 1000$
OPD	4	4	4	5
L-OPD with Lipschitz	10	9	20	18
L-OPD with LLR	18	45	94	55
L-OPD with LSSVR	11	87	12	8
L-OPD with L-LSSVR	45	116	166	97
L-OPD with NN	32	36	111	215

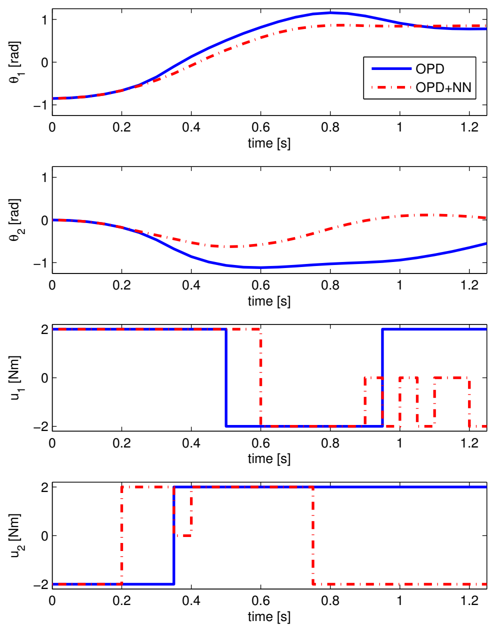

Figure 6: 2DOF resonating arm, controlled trajectories.

$n = 250$. Whereas in the 1DOF case Lipschitz learning showed good performance, we now notice that it underperforms all other learning methods. This might happen because the smoothness of the b-function varies across the state-action space, in which case the global Lipschitz constant will be conservative (too large) in smoother regions, leading to poor b-value accuracy in these regions.

Returns are usually larger for larger depths, as predicted by the analysis, although there are exceptions (e.g. for NNs $n = \{750,\ 1000\}$ have less performance than $n = \{250,\ 500\}$ despite larger depths). Such exceptions are not surprising, as sometimes the returns of explored sequences can be nonmonotonic in the planning horizon, as exemplified in [5]. A more interesting fact is that depths are often smaller for $n = 1000$ than 750, leading to slightly worse performance. The reasons are not entirely clear, but may be related to the fact that approximator errors ε start dominating at large depths, as indicated in Section 3.2, which could bias parameter optimization towards shallower trees.

Table 8: 2DOF resonating arm, execution times in seconds

Method	$n = 250$	$n = 500$	$n = 750$	$n = 1000$
OPD	13.70	30.03	49.36	70.76
L-OPD with Lipschitz	21.00	66.04	84.30	137.47
L-OPD with LLR	21.87	53.13	96.75	136.59
L-OPD with LSSVR	37.32	89.59	103.57	272.81
L-OPD with L-LSSVR	30.32	72.48	133.26	182.08
L-OPD with NN	117.52	229.45	343.92	472.01

Table 8 shows execution times, which follow similar trends to the 1DOF case. For the *same* budget, L-OPD will cost more time than OPD, since using an approximated b-function requires additional operations. Nevertheless, learning allows using *smaller* budgets while attaining better performance: e.g. all learning methods have larger returns for $n = 250$ than OPD at the maximal budget $n = 1000$. All learning methods except NNs are also faster at $n = 250$ than OPD at $n = 1000$, so they are better for online use.

Summarizing the results so far, NNs offer the best returns but are very computationally demanding and therefore poorly suited for online use. L-LSSVR offers a good compromise between performance and computation, and is recommended as the method of choice.

4.2. Stochastic Case: Inverted Pendulum Swingup

The stochastic problem chosen is an inverted pendulum with a weak, unreliable actuator [5]. The states are the angle θ and the angular velocity $\dot{\theta}$, and the goal is to stabilize them at 0 (pointing up), starting from the pointing-down state. The DC motor actuator is weak in the sense its voltage u is limited to $[-3, 3]$ V, which is insufficient to push the pendulum up in one go; instead, the pendulum needs to be swung back and forth to gather energy. It is unreliable as it only applies the intended action u with probability 0.6, and with probability 0.4 it applies only $0.7u$ (when the intended action is 0 it stays 0). This makes it more difficult to perform the swingup, and corresponds to $M = 2$ possible stochastic next states. The pendulum is a relevant problem due to its highly nonlinear solution and the difficulty in planning the long-horizon swings. Unnormalized, quadratic rewards are defined:

$$\rho_{\text{unnorm}}(x_k, u_k, x_{k+1}) = -x^T Q x - u^T R u$$

with $Q = \text{diag}[5, 0.1]$ and $R = 1$, and are then shifted and scaled into $[0, 1]$. The discount factor is $\gamma = 0.95$. Actions are discretized into $K = 3$ values, namely $-3, 0$, and 3 V.

For L-OPMDP, we test the Lipschitz approximator (the only one with the upper-bound property), and local LSSVR (since it provided the best computation-performance tradeoff in the experiments above). We also apply standard OPMDP and HOLOP, as SOOP and LP only work in deterministic systems. The budgets are $n = 50,\ 100,\ 200$, and 400, with the following method-dependent parameters:

- L-OPMDP with the Lipschitz approximator: $\ell \in \{0.01, 0.1, 0.5, 1, 5\}$.
- L-OPMDP with L-LSSVR: $C \in \{500, 750, 1000\}$, $\sigma \in \{0.1, 0.2, 0.3, 0.4\}$ and $k_{nn} \in \{5, 10, 10\}$.
- HOLOP: $K \in \{5, 10, 15, 20, 25, 30\}$.

For each combination of parameters, 20 independent runs of the experiment are performed, and we report means and 95% confidence intervals on these means. For every n, the other parameters are optimized by selecting the combination with the largest upper confidence bound.[7]

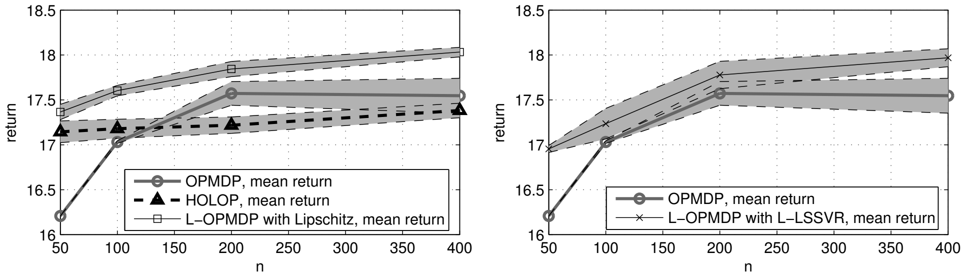

Figure 7: Returns obtained by L-OPMDP with the Lipschitz and L-LSSVR approximator, compared to OPMDP and HOLOP. For readability, results are separated into two graphs, with OPMDP on both graphs for reference.

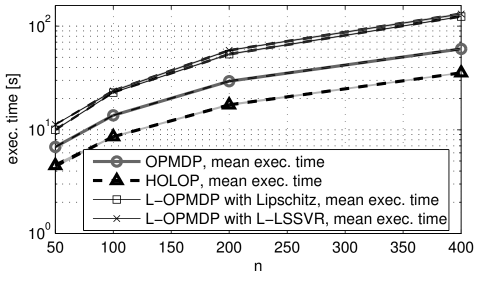

Figure 8: Execution times for the inverted pendulum.

Figure 7 shows the returns obtained. Compared to the original OPMDP, adding learning is nearly always beneficial – results are only inconclusive for L-LSSVR and $n = 200$. Note that the different returns correspond to real differences in task-solving performance: for $n = 100$, OPMDP cannot swing up the pendulum, while adding Lipschitz-based learning always results in successful swingups. The Lipschitz approximator performs better than L-LSSVR in this problem, indicating that a good estimate of the Lipschitz constant can be found. Looking now at HOLOP, with small budgets it is able to get ahead of OP-MDP and even achieve similar performance to L-LSSVR, likely because its fixed planning depth allows it to examine longer sequences early on. However, its performance does not improve with increased budget, which makes it overall less interesting.

[7]We use smaller parameter grids since the repeated experiments in the stochastic case require more computation than in the deterministic case.

Finally, from Figure 8, there is little difference in execution time between the two learning methods. Mirroring the deterministic case, for the same budget learning incurs additional costs compared to just planning. However, a well-chosen approximator can still attain at least as good performance for smaller budgets as just planning with larger budgets, while remaining comparatively cheaper (compare e.g. Lipschitz approximation at $n = 100$ with OPMDP at $n = 200$ or 400, or with HOLOP at $n = 400$).

5. Conclusions and Future Work

We introduced a method to learn b-values online in optimistic planning for deterministic and stochastic Markov decision processes. We analyzed the performance of OP with learned b-values, characterizing the dependence on the accuracy of the representation used. The analysis brings out that for small errors, the learning variant performs better. We compared several different b-function representations, for two versions of a deterministic resonating robotic arm, as well as a stochastic pendulum swingup problem. The Lipschitz approximator is a good option when the value function is well-behaved, but otherwise may provide poor performance. Neural networks perform very well, but have too high computational cost to be used online. A good compromise between performance and complexity is offered by a local variant of least-squares support vector machines.

A first extension of the work in this paper would be to learn b-values online in other flavors of OP, such as Upper Confidence Trees [7] or OP for continuous-action systems [12]. On the analytical side, it would be interesting to exploit techniques from approximate value iteration [3] to analyze the convergence of the b-function to the optimal values across *multiple* steps of online control, and couple this with the existing guarantees.

Acknowledgments

The work of L. Buşoniu was supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI, project number PNII-RU-TE-2012-3-0040.

References

[1] R. Sutton, A. Barto, Reinforcement Learing, MIT Press, 1998.

[2] O. Sigaud, O. Buffet (Eds.), Markov Decision Processes in Artificial Intelligence, Wiley, 2010.

[3] D. P. Bertsekas, Dynamic Programming and Optimal Control, 4th Edition, Vol. 2, Athena Scientific, 2012.

[4] R. Munos, The optimistic principle applied to games, optimization and planning: Towards foundations of Monte-Carlo tree search, Foundations and Trends in Machine Learning 7 (1) (2014) 1–130.

[5] L. Buşoniu, R. Munos, R. Babuška, A review of optimistic planning in Markov decision processes, in: F. Lewis, D. Liu (Eds.), Reinforcement Learning and Adaptive Dynamic Programming for Feedback Control, Wiley, 2012.

[6] S. M. La Valle, Planning Algorithms, Cambridge University Press, 2006.

[7] L. Kocsis, C. Szepesvári, Bandit based Monte-Carlo planning, in: Proceedings 17th European Conference on Machine Learning (ECML), Springer, 2006, pp. 282–293.

[8] S. Gelly, Y. Wang, R. Munos, O. Teytaud, Modification of UCT with patterns in Monte-Carlo Go, Tech. rep., INRIA (2006).

[9] J.-F. Hren, R. Munos, Optimistic planning of deterministic systems, in: Proceedings 8th European Workshop on Reinforcement Learning (EWRL-08), 2008, pp. 151–164.

[10] L. Buşoniu, R. Munos, Optimistic planning for Markov decision processes, in: Proceedings 15th International Conference on Artificial Intelligence and Statistics (AISTATS-12), Vol. 22 of JMLR Workshop and Conference Proceedings, 2012, pp. 182–189.

[11] S. Bubeck, R. Munos, Open loop optimistic planning, in: Proceedings 23rd Annual Conference on Learning Theory (COLT), 2010, pp. 477–489.

[12] A. Weinstein, M. L. Littman, Bandit-based planning and learning in continuous-action Markov decision processes, in: Proceedings 22nd International Conference on Automated Planning and Scheduling (ICAPS-12), São Paulo, Brazil, 2012.

[13] L. Buşoniu, A. Daniels, R. Munos, R. Babuška, Optimistic planning for continuous–action deterministic systems, in: Proceedings 2013 IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL-13), 2013.

[14] R. S. Sutton, Integrated architectures for learning, planning, and reacting based on approximating dynamic programming, in: Proceedings 7th International Conference on Machine Learning (ICML-90), Austin, US, 1990, pp. 216–224.

[15] P. E. Hart, N. J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems Science and Cybernetics 4 (1968) 100–107.

[16] N. J. Nilsson, Principles of Artificial Intelligence, Tioga Publishing, 1980.

[17] M. Helmert, P. Haslum, J. Hoffmann, Flexible abstraction heuristics for optimal sequential planning, in: Proceedings 17th International Conference on Automated Planning and Scheduling (ICAPS), 2007, pp. 176–183.

[18] S. Yoon, Learning heuristic functions from relaxed plans, in: Proceedings 16th International Conference on Automated Planning and Scheduling (ICAPS), 2006, pp. 162–171.

[19] J. T. Thayer, A. J. Dionne, W. Ruml, Learning inadmissible heuristics during search, in: Proceedings 21st International Conference on Automated Planning and Scheduling (ICAPS-11), 2011.

[20] S. J. Arfaee, S. Zilles, R. C. Holte, Bootstrap learning of heuristic functions, in: Proceedings 3rd Annual Symposium on Combinatorial Search (SOCS-10), 2010, pp. 52–60.

[21] M. Fink, Online learning of search heuristics, Journal of Machine Learning Research - Proceedings Track 2 (2007) 114–122.

[22] L. Jaillet, J. Cortés, T. Siméon, Sampling-based path planning on configuration-space costmaps, IEEE Transactions on Robotics 26 (4) (2010) 635–646.

[23] R. Korf, Real-Time Heuristic Search, Artificial Intelligence 27 (1985) 189–211.

[24] A. G. Barto, S. J. Bradtke, S. P. Singh, Learning to act using real-time dynamic programming, Artificial Intelligence 72 (1995) 81–138.

[25] A. Kolobov, M. Daniel, D. S. Weld, ReTrASE: Integrating paradigms for approximate probabilistic planning, in: Proceedings 21st International Joint Conference on Artificial Intelligence (IJCAI-09), Pasadena, USA, 2009, pp. 1746–1753.

[26] S. Gelly, D. Silver, Combining online and offline knowledge in UCT, in: Proceedings 24th International Conference on Machine Learning (ICML), 2007, pp. 273–280.

[27] L. Buşoniu, R. Postoyan, J. Daafouz, Near-optimal strategies for nonlinear networked control systems using optimistic planning, in: Proceedings American Control Conference 2013 (ACC-13), Washington, DC, 2013.

[28] R. Fonteneau, L. Buşoniu, R. Munos, Optimistic planning for belief-augmented markov decision processes, in: 2013 IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL-13), Singapore, 2013.

[29] R. Munos, Cs. Szepesvári, Finite time bounds for fitted value iteration, Journal of Machine Learning Research 9 (2008) 815–857.

[30] C. G. Atkeson, A. W. Moore, S. Schaal, Locally weighted learning, Artificial Intelligence Review 11 (1–5) (1997) 11–73.

[31] J. Suykens, J. Vandewalle, B. D. Moor, Optimal control by least squares support vector machines, Neural Networks 14 (1) (2001) 23 – 35.

[32] J. Suykens, T. van Gestel, J. de Brabanter, Least Squares Support Vector Machines, World Scientific, 2002.

[33] F. M. Ham, I. Kostanic, Principles of Neurocomputing for Science & Engineering, McGraw-Hill, 2001.

[34] K. Hinderer, Lipschitz continuity of value functions in Markovian decision processes, Mathematical Methods of Operations Research 62 (1) 3–22.

[35] E. Alpaydin, Introduction to Machine Learning, MIT Press, 2010.

[36] M. Plooij, M. Wisse, A novel spring mechanism to reduce energy consumation of robotic arms, in: Proceedings 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-12), 2012, pp. 2901–2908.

[37] J.-F. Hren, Planification optimiste pour systèmes déterministes, Ph.D. thesis, Université de Lille (2012).