Reinforcement Planning: RL for Optimal Planners

Matt Zucker
Department of Engineering
Swarthmore College
Swarthmore PA, USA
mzucker1@swarthmore.edu

J. Andrew Bagnell
The Robotics Institute
Carnegie Mellon University
Pittsburgh PA, USA
dbagnell@ri.cmu.edu

***Abstract*— Search based planners such as A* and Dijkstra's algorithm are proven methods for guiding today's robotic systems. Although such planners are typically based upon a coarse approximation of reality, they are nonetheless valuable due to their ability to reason about the future, and to generalize to previously unseen scenarios. However, encoding the desired behavior of a system into the underlying cost function used by the planner can be a tedious and error-prone task. We introduce *Reinforcement Planning*, which extends gradient based reinforcement learning algorithms to automatically learn useful surrogate cost functions for optimal planners. Reinforcement Planning presents several advantages over other learning approaches to planning in that it is not limited by the expertise of a human demonstrator, and that it acknowledges the domain of the planner is a simplified model of the world. We demonstrate the effectiveness of our method in learning to solve a noisy physical simulation of the well-known "marble maze" toy.**

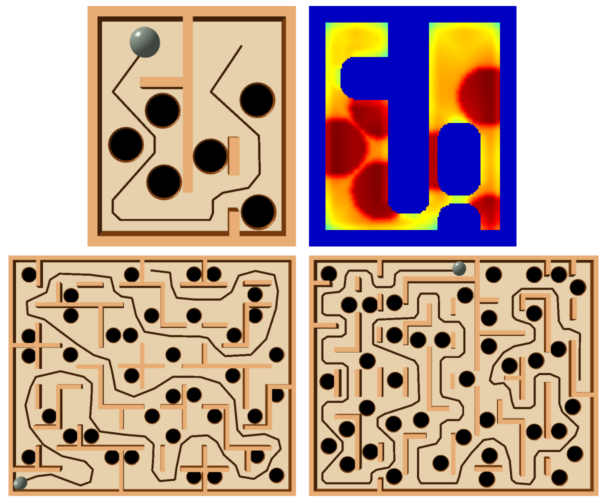


Fig. 1. Marble maze used for training (*top left*) and corresponding cost function learned by our method (*top right*). The system learns to assign high costs near holes, and low costs near walls and corners. Despite being trained on a such a simple maze, the system is able to successfully generalize to much more difficult problem instances (*bottom row*).

I. Introduction

State-of-the-art robotic systems [1], [2], [3] increasingly rely on search-based planning or optimal control methods to guide decision making. Such methods are nearly always extremely crude approximations to the reality encountered by the robot: they consider a simplified model of the robot (as a point, or a "flying brick"), they often model the world deterministically, and they nearly always optimize a *surrogate* cost function chosen to induce the correct behavior rather than the "true" reward function corresponding to a declarative task description. Despite this crudeness, optimal control methods have proven quite valuable because of their efficiency, and also due to their ability to transfer knowledge to new domains; given a way to map features of the world to a continuous cost, we can compute a plan that navigates a robot in a never-before-visited part of the world. While value-function methods and reactive policies are popular in the reinforcement learning (RL) community, it often proves remarkably difficult to transfer the ability to solve a particular problem to related ones using such methods [4]. Planning methods, by contrast, consider a sequence of decisions in the future, and rely on the principle underlying optimal control that cost functions are more parsimonious and generalizable than plans or values.

However, planners are only successful to the extent that they can transfer domain knowledge to novel situations. Most of the human effort involved in getting systems to work with planners stems from the tedious and error-prone task of adjusting surrogate cost functions, which has until recently been a black art. Imitation learning by Inverse Optimal Control, using, *e.g.* the Learning to Search approach [3], has proven to be an effective method for automating this adjustment; however, it is limited by human expertise. Our approach, Reinforcement Planning (RP), is straightforward: we demonstrate that crude planning algorithms can be learned as part of a direct policy search or value function approximator, thereby allowing a system to experiment on its own and outperform human expert demonstration.

A. Background and Related Work

Central to this work is the distinction between the "true" reward function for a robotic system, and the surrogate reward (denoted here as cost) function used by an optimal planner. Consider an unmanned ground vehicle (UGV) system similar to that discussed in [5]. Modern UGV software architectures are typically hierarchical planning and control schemes, as depicted in Figure 2: a coarse global planner computes a path from the robot's current position to a distant goal, and a "local planner" or control policy runs at a much higher frequency in order to guide the vehicle along the planned path. The coarse planner for our hypothetical vehicle may neglect to model the kinematics and dynamics of the underlying robotic platform, reducing the planning problem to motion on an 8-way-connected, regularly sampled grid.

The true reward function for the UGV control system may be simple to write down: it encompasses some assessment of

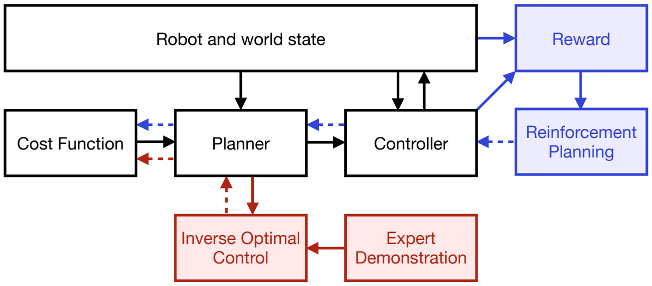


Fig. 2. A schematic comparison of Reinforcement Planning (RP) and Inverse Optimal Control (IOC) in a typical hierarchical planning system. Solid arrows denote information flow and causal relationships; dotted arrows indicate feedback from learning algorithms. Note that unlike the proposed method, IOC receives no feedback about the quality of the plans executed by the robot.

the current robot state x, evaluated at every control cycle until termination. For example, the system could incur a reward of \$100 for success, -\$100,000 for unrecoverable failure, and -\$0.01 at every timestep otherwise. However, because the planner reasons over a crude approximation of reality, it is unable to optimize the true reward function directly. Instead, the planner reasons about a surrogate cost function, which might consist of a linear combination of scalar features associated with a discrete grid state s, such as numerical estimates of slope, roughness, visibility, traversability, etc. This disconnect between true reward and surrogate cost is what makes tuning cost functions difficult for system designers; their task is to identify a cost function which implicitly causes the system to behave favorably in terms of the actual reward.

The Learning to Search (LEARCH) approach [3] removes the burden of manually parameterizing cost functions by allowing a system designer to specify the desired output of the coarse planner. Whereas the goal of optimal control is to identify a path or trajectory that minimizes a cost function, *Inverse Optimal Control* (IOC) schemes such as LEARCH attempt to find a cost function under which demonstrated paths or trajectories appear optimal. Although LEARCH is powerful due to its convex formulation and its ability to reduce development time for planning systems, the approach has limitations. First and foremost, it is limited by the performance of the expert demonstration. A human may be unskilled at the target task, or worse, unable to demonstrate it at all. Our Reinforcement Planning method, on the other hand, propagates the reward signal back through the controller and planner to the parameters of the underlying cost function, and is not limited by any expert demonstration.

We also note that planning algorithms have been used extensively in previous RL work (*e.g.* Dyna [6]); our work contrasts with these in embracing the reality that real-world planners operate on a coarse approximation and must be trained to have a cost function that induces the correct behavior; hence, they do not directly optimize the "true" reward function. Moreover, approaches such as Dyna attempt to (approximately) solve the full Markov Decision Problem (MDP) associated with the system. For the types of complex high-dimensional systems that are considered by hierarchical planning approaches, the curse of dimensionality makes an explicit representation of a full value function or policy corresponding to the solution of such an MDP computationally intractable. The most crucial distinction is that RP generalizes to novel problem instances without the need for further learning.

II. Value Function Subgradient

The central observation underlying RP is that the value function computed by an optimal planner possesses a subgradient with respect to the parameters of the cost function. [1] If a policy for the system (as defined in Section III) is computed in terms of the planner's value function, the subgradient of the value function will appear in the policy gradient wherever the value function appears in the policy. Subsequently, any gradient based reinforcement learning algorithm can be straightforwardly extended to consider optimal planners.

The task for a discrete optimal planner is to find the minimum-cost sequence of actions to transition from a starting state to a goal state. Let $s \in \mathcal{S}$ denote a discrete state, and let $a \in \mathcal{A}$ denote an action. The successor state given an action is specified by a state transition model $s' = \mathrm{succ}(s, a) : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$. Denote by $\xi(s_0)$ a sequence of state-action pairs (s_k, a_k) which starts at s_0 and reaches a goal state in $\mathcal{S}_{goal} \subset \mathcal{S}$. Given a one-step cost function $c(s, a) : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$, the formal task for the planner is to compute $\xi^*(s_0)$, a minimum-cost path from s_0 to the goal. The *value* $\tilde{V}(s_0)$ of the state s_0 is defined as the sum of costs along that path. The *value function* $\tilde{V}(s)$ is then the function which maps each state s to the corresponding minimum cost: $\tilde{V}(s) = \min_{\xi(s)} \sum_{(s_k, a_k) \in \xi(s)} c(s_k, a_k)$.

Suppose that the cost function is defined as a linear combination of features defined over state-action pairs, based on a weighting θ: $c(s, a, \theta) = \theta^T f(s, a)$. Then the subgradient of $\tilde{V}(s, \theta)$ with respect to θ is simply the sum of the one-step-cost gradients along the optimal path $\xi^*(s)$, namely the features themselves: $\nabla \tilde{V}(s, \theta) = \sum_{(s_k, a_k) \in \xi^*(s)} \nabla c(s_k, a_k, \theta) = \sum_{(s_k, a_k) \in \xi^*(s)} f(s_k, a_k)$. This "feature counting" interpretation of differential value has been described before in the derivation of Maximum Margin Planning and other LEARCH algorithms [3]; however, instead of directly matching feature counts with those of an expert as in LEARCH, our goal in Reinforcement Planning is to provide a way for a learning system to modify planner values in order to maximize expected reward. [2]

III. Reinforcement Planning

Let $x \in \mathcal{X}$ be the full state (or observation) space for a robotic system, and let $u \in \mathcal{U}$ be the space of

[1] A *subgradient* is a linear lower bound, the analog to the familiar gradient operator for functions which are convex but not necessarily differentiable.

[2] We note that in the case of a non-deterministic MDP, the gradient $\nabla \tilde{V}$ contains *expected* feature counts encountered on the way to the goal state as opposed to a deterministic sum of features. Such expected feature counts can be computed or sampled after solving the coarse MDP using value or policy iteration (as opposed to deterministic graph search algorithms such as Dijkstra's or A*).

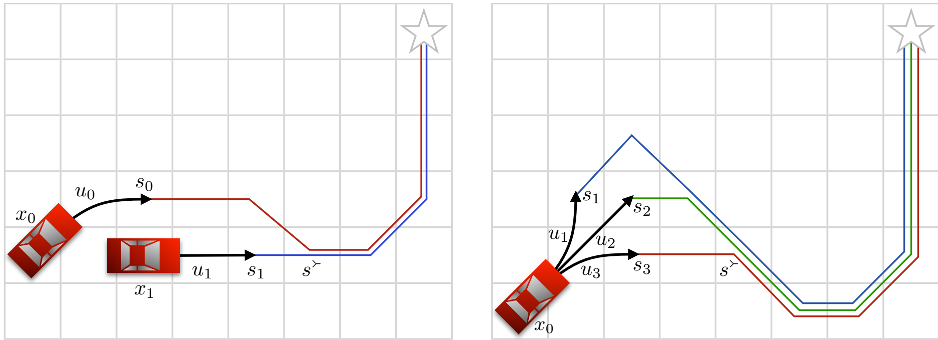


Fig. 3. Illustration of SARSA(λ)-style (*left*) and REINFORCE-style (*right*) gradients under RP. In both cases, gradient terms involve a sum of features along optimal paths (see Section III-A and Section III-B).

controls. We assume the existence of a coarse-graining function $\mathrm{proj}(x) : \mathcal{X} \mapsto \mathcal{S}$ that maps a full state x to the corresponding "nearby" state s in the domain of the planner. An admissible *Q-function* for Reinforcement Planning is a function $Q(x, u, \theta) \mapsto \mathbb{R}$ which computes the relative quality of applying action u at state x in terms of the optimal cost-to-go $\tilde{V}(s', \theta)$ for some coarse planner state(s) s' that depends on x and u, along with an optional cost associated with x and u themselves. For example, assume there is a deterministic state transition function $x' = \mathrm{succ}(x, u)$ for the system. Then one admissible Q-function for RP would be to compute the optimal cost-to-go of the projected successor state, along with a term penalizing large actions: $Q(x, u, \theta) = \tilde{V}\left(\mathrm{proj}\left(\mathrm{succ}(x, u)\right), \theta\right) + \lambda \|u\|^2$. If, as in a traditional MDP, there is no deterministic state transition function, we would instead consider the expected cost-to-go under the distribution $p(x'|x, u)$: $Q(x, u, \theta) = \int \tilde{V}\left(\mathrm{proj}(x'), \theta\right) p(x'|x, u) + \lambda \|u\|^2 \, dx'$.

We now proceed by constructing a policy (in the RL sense) around Q, and adapting existing gradient based RL algorithms to optimize θ based on measured rewards. Any such gradient based RL algorithm will examine ∇Q, the gradient of Q with respect to the cost function parameters. We note that anywhere $\tilde{V}$ appears in the definition of Q, so too will $\nabla \tilde{V}$ appear in the expression of ∇Q. Broadly speaking, the taxonomy of gradient based RL algorithms can be broken down into those which use value function approximators, and those which learn policies directly. The following subsections illustrate how to apply RP in both cases.

A. Value function approximation

The Q-function defined above can be considered as a parametric value function approximator. By constructing an ε-greedy policy around Q, we can learn the cost function parameters θ via steepest descent on Bellman Error with an eligibility trace. Let $U(x) \subseteq \mathcal{U}$ be the set of actions available at state x. Then the ε-greedy policy $\pi(x, \theta)$ is

$$\pi(x, \theta) = \begin{cases} \arg\min_{u \in U(x)} Q(x, u, \theta), & \text{with } p = (1 - \varepsilon) \\ \text{random } u \in U(x), & \text{with } p = \varepsilon \end{cases}$$

where p denotes the probability of taking an action. We then execute the policy and accumulate one-sample estimates of the gradient of the Bellman Error

$$\delta \;=\; r(x, u) + Q(x', u', \theta) - Q(x, u, \theta)$$

by combining the residual elimination method of Baird [7] along with an eligibility trace similar to SARSA(λ) [8]. We call the resulting convergent algorithm RP-VFA, for Reinforcement Planning via Value Function Approximation, listed in Algorithm 1.

Algorithm 1: RP-VFA algorithm for steepest descent on Bellman Error with eligibility trace.

```
initialize e(x,u) ← 0 for all x, u;
initialize Δ ← 0;
pick initial x, u;
while x is not terminal do
    take action u, observe reward r(x,u) and successor
    state x';
    choose u' from π(x', θ);
    δ ← r(x,u) + Q(x', u', θ) − Q(x, u, θ) ;
    e(x,u) ← e(x,u) + ∇Q(x', u', θ) − ∇Q(x, u, θ) ;
    foreach previous x, u do
        Δ ← Δ + δ e(x,u);
        e(x,u) ← λ e(x,u);
    end
    x ← x'; u ← u';
end
θ ← θ + αΔ;
```

The left hand side of Figure 3 illustrates the effect of the gradient term: initially, the system is at state x_0 and chooses action u_0 based on the value of the underlying coarse planner state s_0. After executing action u_0, the system ends up in state x_1, incurring reward $r_0 = r(x_0, u_0)$. (Note that due to, i.e. modeling inaccuracy, the state x_1 does not project down to s_0; this is exactly the type of coarse approximation which RP is designed to work with.) The next action chosen is u_1, based on the value of the projected successor state s_1. The gradient of δ above is equal to the difference of the features encountered along the two optimal paths in the figure. Note that both optimal paths to the goal merge at the state $s^{\curlywedge}$; hence any features encountered along the shared

Algorithm 2: REINFORCE for direct policy learning with RP

initialize $z_0 \leftarrow 0$, $\Delta_0 \leftarrow 0$;
initialize $t \leftarrow 0$;
initialize x_0;
while x_t is not terminal **do**
  sample u_t from $p(u_t|x_t, \theta)$;
  execute u_t, observe reward $r_t \leftarrow r(x_t, u_t)$ and successor state x_{t+1};
  $z_{t+1} \leftarrow \beta z_t + \frac{\nabla p(u_t|x_t,\theta)}{p(u_t|x_t,\theta)}$;
  $\Delta_{t+1} \leftarrow \Delta_t + \frac{1}{t+1}\,(r_t\, z_t - \Delta_t)$;
  $t \leftarrow t+1$;
end
$\theta \leftarrow \theta + \alpha \Delta_t$;

subpath from $s^{\succ}$ to the goal cancel out and do not appear in the resulting value of $\nabla\delta$. See Algorithm 2 for a listing.

B. Direct policy learning

Instead of considering Q as a value function approximator, we can use it as the basis for a stochastic policy, and subsequently draw upon direct policy learning algorithms such as REINFORCE [9] to learn θ. We begin by constructing a stochastic policy $p(u|x, \theta)$ around Q:

$$p(u_i|x, \theta) = \frac{1}{Z} \exp\big(- \nu\, Q(x, u_i, \theta)\big)$$

where Z is the normalizer that causes $p(u|x, \theta)$ to sum to one over all $u \in U(x)$, and ν is positive scalar weight. REINFORCE requires us to compute the *score ratio* $\nabla p_i/p_i$. For the Boltzmann-style distribution defined above, the score ratio can be expressed as

$$\frac{\nabla p(u_i|x, \theta)}{p(u_i|x, \theta)} = -\nu \left(\nabla Q(x, u_i, \theta) - E_p \left[\nabla Q(x, u, \theta)\right]\right)$$

At this point, the REINFORCE algorithm (*e.g.* as derived in [10]) can be applied straightforwardly. We note that alternative policy gradient algorithms besides REINFORCE may be used; however, generally they will all require computing the score ratio $\nabla p/p$.

The score ratio for RP is illustrated on the right hand side of Figure 3: consider the robot at state x_0. The respective probabilities p_i of the actions u_i are based on the optimal cost-to-go of the projected successor states s_i, for $i \in 1, 2, 3$. Say that we sample action u_1. Then the score ratio $\nabla p_1/p_1$ is defined to be the difference between the sum of features along the blue path from s_1, and the expected features over all three paths. As with the value function approximation example, all optimal paths merge at a future planner state $s^{\succ}$, and therefore the features along the common subpaths cancel each other out when computing $\nabla p/p$.

C. Discussion of RP implementations

As illustrated in Figure 3, the gradient update rules for both approaches modify the parameters θ based upon observed rewards, and a set of features along optimal paths to the goal. One interpretation of the RP gradient terms is to consider them as a comparison of features which differ among various optimal paths found by the low-level planner. As a gradient based method, Reinforcement Planning is by no means guaranteed to converge to a global optimum of θ for either type of approach. Therefore, initializing θ to a principled estimate is necessary in practice. If *a priori* estimates of θ are not available, using LEARCH or a similar IOC approach is a suitable initialization method.

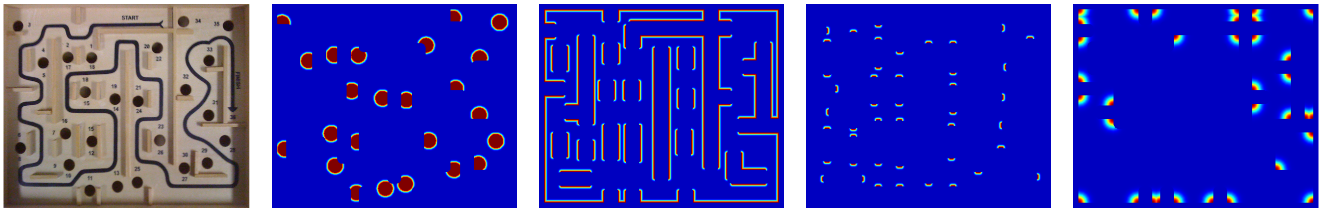
IV. Example Domain: Marble Maze

The familiar "marble maze" or labyrinth toy (shown in Figure 4) has been used in the past as a benchmark application for learning algorithms, particularly imitation learning [4], [11]. However, previous approaches learned policies specifically targeted at a specific maze, and failed to generalize to novel unseen mazes without significant additional learning [12]. Just as with more complex dynamical systems such as legged robots, the marble maze task exhibits significant momentum effects, and requires reasoning about the consequences of future actions. We chose to demonstrate Reinforcement Planning on the marble maze task in order to show that a very simple planner can produce policies that not only perform well in the face of dynamics and uncertainty, but also generalize to novel scenarios.

A. Marble maze dynamics and reward

The marble maze consists of a flat board with raised walls. An operator may apply forces to the marble by turning knobs which rotate the board along its x and y axes. The goal of the game is to successfully navigate the marble from a starting position to a goal position, avoiding any number of holes drilled into the board. Falling into a hole ends the game. We define the state of the marble maze as $x = (p_x, p_y, \dot{p}_x, \dot{p}_y, r_x, r_y)$, which combines the 2D position p_x, p_y and velocity $\dot{p}_x, \dot{p}_y$ of the ball with the current tilt angles r_x, r_y of the board. The command for the system $u = (u_x, u_y)$ corresponds to desired tilt angles for the board. The reward for the system depends only upon the state. Evaluated once per 10 Hz control update until the game has ended, it is defined as \$100 if the goal is reached, -\$100 if the ball falls in a hole, and -\$0.01 otherwise.

We model the marble maze in a physical simulation. The acceleration imparted on the ball by board tilt is given by: $\ddot{p}_x = g \sin(r_y)$, and $\ddot{p}_y = -g \cos(r_y) \sin(r_x)$, where $g = 9.8m/s^2$. There is additional acceleration due to rolling friction, which is computed to be proportional to the velocity of the ball. The coefficient of rolling friction is $\mu_{roll} = 0.05$. Collisions between the ball and a wall result in an instantaneous reflection of the velocity vector about the normal of the wall, and damping by a coefficient of restitution $c_r = 0.85$. After the acceleration and impacts are computed, the position and velocity of the ball is updated via the midpoint method. Second-order dynamics of the rotation of the board are ignored. We assume that the board tilt is highly position controlled, with no significant dynamics beyond a maximum velocity constraint.



example maze | holes | walls | wall ends | corners

Fig. 4. An example marble maze and features used for planning. All features are computed by convolving, or "blurring", discretized board geometry with two different kernel sizes (only one is shown here for each feature).

We also model a variety of adverse effects expected to be encountered in a true physical system such as control noise, local variation in μ_{roll} and c_r, and board warp (local variation in board tilt). Control noise is drawn from a normal distribution with mean zero and $\sigma_u = 0.02\text{rad}$. Local variations in the physical parameters are drawn from a spatially coherent noise distribution [13]. Finally, we also model differences between the idealized board considered by the planner and the underlying board used for simulation by slightly displacing walls and holes in the two models. Our goal is to learn a policy mapping states to actions using RP. Again, as expected in a physical system, we present our policy with a noisy observation of state instead of the true simulated state of the system.

B. Marble maze planner and policy

As in Figure 2, we create a hierarchical planning architecture for the noisy marble maze simulation. We construct a coarse planner that operates over a 2D grid corresponding to the 2D workspace $\mathcal{S}$ of the maze board. Each grid cell $s \in \mathcal{S}$ is assigned a set of features $f(s)$ representing aspects of the board geometry. Features, shown in Figure 4, are computed as blurred representations of holes, walls, wall ends, and corners. There are two different kernel radii used for the blurring. All feature values lie in the interval $[0, 1]$. We augment the feature vector with the logical complement of each feature $1 - f$, and add a constant bias feature of 1.0, for a total of 17 features in all.

We use Dijkstra's algorithm [14] to compose a value function representing the optimal cost-to-go from each grid cell s to the goal. The one-step-cost is computed as $c(s) = \theta^T f(s)$ with θ a positive vector of weights. Grid cells lying inside holes and walls are treated specially by our value function algorithm: the value function in such location always points back to the nearest grid cell accessible to the ball. The policy defined for the marble maze considers a discrete set of candidate actions. For each action u at state x, we define the function $Q(x, u, \theta) = \frac{1}{2}k_u \|u - u^*(x, \theta)\|^2 + k_v \tilde{V}(s', \theta)$, where k_u and k_v are scalar weights, $u^*(x, \theta)$ is a commanded tilt that attempts to align the velocity of the ball with the local direction of the planner value function, and s' is the projected successor state $\text{proj}(\text{succ}(x, u))$. Projection is defined as finding the grid cell nearest to p_x, p_y. See the appendix a derivation of u^* and its gradient. To apply RP, we simply take the gradient of Q with respect to θ:

$$\nabla Q(x, u, \theta) = -k_u \nabla u^*(x, \theta)\big(u - u^*(x, \theta)\big) + k_v \nabla \tilde{V}(s', \theta)$$

Here, $\nabla u^*(x, \theta)$ is a 17-by-2 matrix obtained by differentiating the velocity controller with respect to θ, which itself contains terms based on the subgradient of $\tilde{V}$.

V. Results

We used RP with both the RP-VFA algorithm of Section III-A as well as the REINFORCE algorithm derived in Section III-B to learn a cost function for the coarse planner to solve the noisy marble maze simulation. The initial value for the weight vector θ was zero for all features, except for a small positive bias term which encouraged goal-seeking behavior. Reinforcement Planning was run on a very small training board for 100 trial episodes with both learning methods, and the performance was evaluated on the training board as well as two significantly larger test boards (shown in Figure 1).

We compared RP to a simple regression algorithm that attempts to match the one-step-costs of the planner to the measured rewards in the simulator. The regression algorithm failed to learn any significant structure of the problem, presumably due to the fact that it was unable to ascribe any of the current reward to features previously encountered by the system. This "smearing" back in time of the correspondence between rewards and features is handled well by both the REINFORCE and the RP-VFA algorithms.

Reinforcement Planning, however, was able to successfully learn how to solve not only the training board far better than the initial weights, but it also vastly outperformed the initial weights and the regression algorithm on the much larger, and previously unseen, test boards. The results are summarized in Figure 5. Learning converges after approximately 100 trials; training for many more trials did not significantly alter performance.

The cost function learned by RP is shown in Figure 1. Reinforcement Planning confirms human experts' own intuition about solving the marble maze problem: not only is cost high near holes, but it is lower next to walls and corners, presumably because their effect of reducing uncertainty of the rolling marble dynamics. Furthermore, the system has discovered the correct weighting of features on its own, obviating the need for hand-tuning cost function parameters.

VI. Conclusion

We have introduced Reinforcement Planning, an extension to existing reinforcement learning algorithms which allows them to reason with optimal planners. The key of RP is

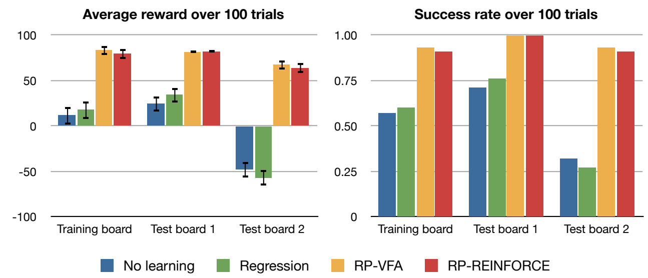


Fig. 5. *Top:* Results of experiments on noisy marble maze simulation after training for 100 episodes.

to define a policy for a learning system in terms of the value computed by a coarse planner. Computing the gradient of the policy evaluates the subgradient of the planner's value function. Reinforcement Planning can be used in both value function approximation, and direct policy gradient settings. We experimentally verified the effectiveness of RP in learning to control a noisy physical simulation of the marble maze game.

In future work, we will investigate a number of extensions to the methods proposed here and implement them on physical robotic systems as opposed to computer simulations. We believe it will be straightforward to derive the functional analogues of the algorithms put forward in Section III-A and Section III-B: just as MMPBoost [15] is a boosted version of Maximum Margin Planning [3], so as to derive boosted versions of RP for automatic feature discovery or non-linear cost functions. Our derivations of Reinforcement Planning in this paper have focused on discrete search as the underlying implementation of optimal planners. However, other optimal planning and control schemes exist, such as variational methods which use Pontryagin's minimum principle or Differential Dynamic Programming [16], which operates on continuous state and action domains. We note that the approach extends straightforwardly by integrating feature counts along continuous trajectories. Work on a physical implementation of the marble maze solver has already begun (see Figure 6); video of the the system operating using a cost-function based policy is available at http://www.youtube.com/user/reinforceplan.

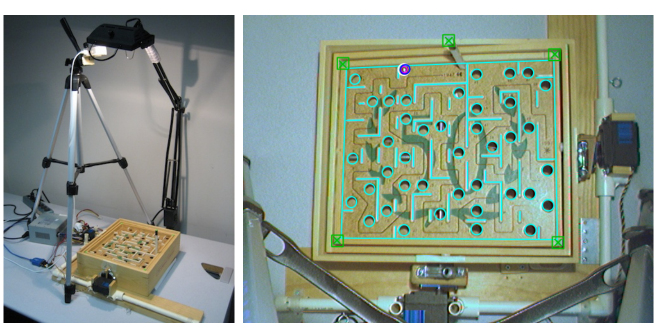


Fig. 6. Physical robotic platform for playing the marble maze. *Left:* the robot consists of two hobby servos, and an overhead camera. *Right:* a computer vision system recognizes the position of the ball and markers on the board to reconstruct the 3D pose of the system.

Appendix: Velocity controller for marble maze

We define $u^*(x, \theta)$ as a proportional controller on ball velocity with gain k_p. The desired velocity v_d comes from the normalized directional gradient of the planner value function $n(x, \theta)$ with respect to the board x- and y-axes, scaled by a gain ℓ. The error $\epsilon(x, \theta)$ between desired and current velocity is converted to a commanded tilt by a skew-symmetric matrix M:

$$u^*(x, \theta) = k_p \, M \, \epsilon(x, \theta)$$

where

$$\begin{aligned}
n(x, \theta) &= \begin{bmatrix} \frac{\partial}{\partial p_x} \\ \frac{\partial}{\partial p_y} \end{bmatrix} V\big(\mathrm{proj}(x), \theta\big) \\
v_d(x, \theta) &= \ell \frac{n(x, \theta)}{\|n(x, \theta)\|} \\
\epsilon(x, \theta) &= v_d(x, \theta) - (\dot{p}_x, \dot{p}_y)^T \\
M &= \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}
\end{aligned}$$

To apply RP, we need to compute $\nabla u^*(x, \theta)$, the 17-by-2 matrix

$$\nabla u^*(x, \theta) = k_p \ell \left(M \left(I - \hat{n}\hat{n}^T \right) \begin{bmatrix} \frac{\partial}{\partial p_x} \\ \frac{\partial}{\partial p_y} \end{bmatrix} \nabla V\big(\mathrm{proj}(x), \theta\big)^T \right)^T$$

Above, I is the 2-by-2 identity matrix, and $\hat{n} = \frac{n(x,\theta)}{\|n(x,\theta)\|}$.

References

[1] Joel Chestnutt. *Navigation Planning for Legged Robots*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, December 2007.
[2] C. Urmson et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 2008.
[3] N. Ratliff, D. Silver, and J. A. Bagnell. Learning to search: Functional gradient techniques for imitation learning. *Autonomous Robots*, 2009.
[4] Martin Stolle. *Finding and Transferring Policies Using Stored Behaviors*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 2008.
[5] D. Silver, J. A. Bagnell, and A. Stentz. High performance outdoor navigation from overhead data using imitation learning. In *Proc. Robotics: Science and Systems Conference*, June 2008.
[6] R. S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *ICML*, 1990.
[7] L. Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proc. International Conference on Machine Learning*, 1995.
[8] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introdution*. MIT Press, 1998.
[9] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992.
[10] J. Baxter and P.L. Bartlett. Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 2001.
[11] Darrin Bentivegna. *Learning from Observation Using Primitives*. PhD thesis, Georgia Institute of Technology, 2004.
[12] Martin Stolle and Chris Atkeson. Knowledge transfer using local features. In *Proc. IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning*, 2007.
[13] K. Perlin. Improving noise. *ACM Transactions on Graphics (TOG)*, 2002.
[14] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
[15] Nathan Ratliff, David Bradley, J. Andrew Bagnell, and Joel Chestnutt. Boosting structured prediction for imitation learning. In *NIPS*, Vancouver, B.C., December 2006.
[16] D.H. Jacobson and D.Q. Mayne. *Differential dynamic programming*. Elsevier Publishing Company, 1970.