Escaping Groundhog Day

Abstract

The dominant approaches to reinforcement learning rely on a fixed state-action space and reward function that the agent is trying to maximize. During training, the agent is repeatedly reset to a predefined initial state or set of initial states. For example, in the classic RL Mountain Car domain, the agent starts at some point in the valley, continues until it reaches the top of the valley and then resets to somewhere else in the same valley. Learning in this regime is akin to the learning problem faced by Bill Murray in the 1993 movie *Groundhog Day* in which he repeatedly relives the same day, until he discovers the optimal policy and escapes to the next day. In a more realistic formulation for an RL agent, every day is a new day that may have similarities to the previous day, but the agent never encounters the same state twice. This formulation is a natural fit for robotics problems in which a robot is placed in a room in which it has never previously been, but has seen similar rooms with similar objects in the past. We formalize this problem as optimizing a learning or planning algorithm for a set of environments drawn from a distribution and present two sets of results for learning under these settings. First, we present *goal-based action priors* for learning how to accelerate planning in environments drawn from the distribution from a training set of environments drawn from the same distribution. Second, we present *sample-optimized Rademacher complexity*, which is a formal mechanism for assessing the risk in choosing a learning algorithm tuned on a training set drawn from the distribution for use on the entire distribution.

Keywords: Meta-learning, transfer learning, learning to plan,

1 Introduction

The dominant approaches to reinforcement learning rely on a fixed state-action space and reward function that the agent is trying to maximize. During training, the agent is repeatedly reset to a predefined initial state or set of initial states. For example, in the classic RL Mountain Car domain, the agent starts at some point in the valley, continues until it reaches the top of the valley and then resets to somewhere else in the same valley. Learning in this regime is akin to the learning problem faced by Bill Murray in the 1993 movie *Groundhog Day* in which he repeatedly relives the same day, until he discovers the optimal policy and escapes to the next day. In a more realistic formulation for an RL agent, every day is a new day that may have similarities to the previous day, but the agent never encounters the same state twice. This formulation is a natural fit for robotics problems in which a robot is placed in a room in which it has never previously been, but has seen similar rooms with similar objects in the past. We formalize this problem as optimizing a learning or planning algorithm for a set of environments drawn from a distribution and present two sets of results for learning under this setting.

In both cases, we assume environments in the distribution are defined by a Markov decision process (MDP). An MDP is define by the tuple (S, A, T, R, s_0), where S is the state space, a set of states in which the agent can make decisions; A is the set of actions the agent can take; $T(s'|s, a)$ is the transition dynamics, which specifies the probability of transitioning to state s' after taking action a in state s; $R(s, a, s')$ is the reward function, which specifies the reward received by the agent for taking action a in state s and then transitioning to state s'; and s_0 is an initial state of the MDP.

The goal of planning or learning in an MDP is to find a (near-)optimal policy $\pi : S \rightarrow A$ that maps states to actions. A policy is typically considered optimal if following it from the initial state maximizes the expected discounted future reward $E\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0, \pi\right]$, where $\gamma \in [0, 1]$ is a discount factor specifying how much the agent prefers immediate rewards over more distant rewards and r_t is the reward received at time t. For a planning algorithm, the agent has complete access to all elements the MDP and can search for a policy before ever acting in the world. In reinforcement learning (RL), the agent does not have access to either the transition dynamics, reward function, or both and must learn how to act in the world by interacting with it.

In our learning setting, an agent is given a set of sample training MDPs on which to optimize its performance. In the case of planning, the agent wants to learn information that allows it to find solutions to other MDPs drawn from the distribution in less CPU time than it would have taken without learning. In the case of learning, the agent wants to find a learning algorithm that will allow it to perform well and and learn more quickly in new MDPs.

We present results for the learning to plan and learning to learn setting. In the learning to plan setting we introduce learnable *goal-based action priors* that accelerate planning on related MDPs. In the learning to learn setting, we present *sample-optimized Rademacher complexity*, which is a formal mechanism for assessing the risk in choosing a learning algorithm tuned on a training set drawn from the distribution for use on the entire distribution.

2 Learning to Plan

Robots operating in unstructured, stochastic environments such as a factory floor or a kitchen face a difficult planning problem due to the large state space and the very large set of possible tasks [5, 8]. A powerful and flexible robot such as a mobile manipulator in the home has a very large set of possible actions, any of which may be relevant depending on the current goal (for example, robots assembling furinture [8] or baking cookies [5].) When a robot is manipulating objects in an environment, an object can be placed anywhere in a large set of locations. The size of the state space increases exponentially with the number of objects, which bounds the placement problems that the robot is able to expediently solve. Depending on the reward function (which is unknown before runtime), any of these states and actions may be relevant to the solution, but for any specific reward function, most of them are irrelevant. For instance, when making brownies, the oven and flour are important, while the soy sauce and sauté pan are not. For a different task, such as stir-frying broccoli, the robot must use a different set of objects and actions.

To confront this state-action space explosion, prior work has explored adding knowledge to the planner, such as options [12] and macro-actions [6, 10]. However, while these methods can allow the agent to search more deeply in the state space, they add non-primitive actions to the planner which *increase* the branching factor of the state-action space. The resulting augmented space is even larger, which can have the paradoxical effect of increasing the search time for a good policy [7]. Deterministic forward-search algorithms like hierarchical task networks (HTNs) [9], and temporal logical planning (TLPlan) [1, 2], add knowledge to the planner that greatly increases planning speed, but do not generalize to stochastic domains. Additionally, the knowledge provided to the planner by these methods is quite extensive, reducing the agent's autonomy and must be manually supplied by the designer.

To address these issues, we augment an Object Oriented Markov Decision Process (OO-MDP) with a specific type of action prior conditioned on the current state and an abstract goal description. Because we condition on both the state and goal description, we refer to this goal-based action prior as a knowlege base of *affordances*. We rigorously formal-

Figure 1: Two different problems from the same domain, where the agent's goal is to smelt the gold in the furnace while avoiding the lava. Our agent is unable to solve the problem on the right before learning because the state/action space is too large (since it can place the gold block anywhere). After learning, it can quickly solve the larger problem.

ize the notion of affordances as a prior on actions conditioned on features of the current state as well as the robot's goal. Affordances enable the robot to prune irrelevant actions on a state-by-state basis based on the agent's current goal and focus on the most promising parts of the state space. Affordances can be specified by hand or alternatively learned through experience in related problems, making them a concise, transferable, and learnable means of representing useful planning knowledge. Our experiments demonstrate that affordances provide dramatic improvements for a variety of planning tasks compared to baselines in simulation, and are applicable across different state spaces. Moreover, while manually provided affordances outperform baselines, affordances learned through experience yield even greater improvements.We conduct experiments in the game Minecraft, which has a very large state-action space, and on a real-world robotic cooking assistant. Figure 1 shows an example of two problems from the same domain in the game Minecraft; the agent learns on randomly generated problems and tests on new problems from the same domain that it has never previously encountered.

To learn affordances, we provide a set of training worlds from the domain (W), for which the optimal policy, π, may be tractably computed using existing planning methods. Then, we compute the maximum likelihood estimate of the parameter vector θ_i for each action using the policy. During the learning phase, the agent learns which actions are useful under different conditions. At test time, the agent will see different, randomly generated worlds from the same domain, and use the learned affordances to increase its speed at inferring a plan. For simplicity, our learning process uses a strict separation between training and test; after learning is complete our model parameters remain fixed.

We evaluate our approach using the game Minecraft. Minecraft is a 3-D blocks game in which the user can place, craft, and destroy blocks of different types. Minecraft's physics and action space allow users to create complex systems, including logic gates and functional scientific graphing calculators[1]. Minecraft serves as a model for robotic tasks such as cooking assistance, assembling items in a factory, object retrieval, and complex terrain traversal. Figure 1 shows two scenes from Minecraft. Our experiments consisted of five common tasks in Minecraft, including constructing bridges over trenches, smelting gold, tunneling through walls, basic path planning, and digging to find an object. We tested on randomized worlds of varying size and difficulty. The generated test worlds varied in size from tens of thousands of states to hundreds of thousands of states. The agent learned affordances from a training set consisting of 25 simple state spaces of each map type (100 total maps), each approximately a 1,000-10,000 state world. We conducted all tests with a single knowledge base. Learning this knowledge base took approximately one hour run in parallel on a computing grid.

We use Real-Time Dynamic Programming (RTDP) [3] as our baseline planner, a sampling-based algorithm that does not require the planner to visit all states. We compare RTDP with learned affordance-aware RTDP (LA-RTDP), and expert-defined affordance-aware RTDP (EA-RTDP). We terminated each planner when the maximum change in the value function was less than 0.01 for 100 consecutive policy rollouts, or the planner failed to converge after 1000 rollouts. The reward function was -1 for all transitions, except transitions to states in which the agent was in lava, where we set the reward to -10. The goal was set to be terminal and the discount factor was $\gamma = 0.99$. To introduce non-determinism into our problem, movement actions (move, rotate, jump) in all experiments had a small probability (0.05) of incorrectly applying a different movement action. This noise factor approximates noise faced by a physical robot that attempts to execute actions in a real-world domain and can affect the optimal policy due to the existence of lava pits that the agent can fall into.

[1]https://www.youtube.com/watch?v=wgJfVRhotlQ

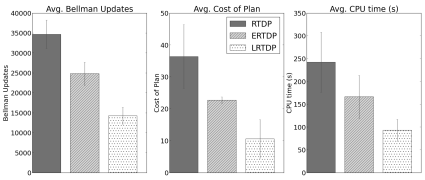

Figure 2: Average results from all maps.

3 Learning to Learn

For the case of optimizing a learning algorithm for a distribution of MDPs, we seek the answer the question: How do you know if your reinforcement-learning (RL) algorithm is the right one for your problem? How can you tell if you are at risk for overfitting? Or underfitting?

A core concept used in mitigating the effects of overfitting is to apply more powerful hypothesis classes only when copious data is available to fit their free parameters, restricting learners to weaker hypothesis classes when data is more sparse. In some well-studied cases, bounds relating the hypothesis class to the amount of data required to fit it accurately are known [4]. In this paper, we provide a formal procedure for making estimates even in the context of considerably more complex hypothesis classes for which analytical results are unknown and use it in the context of RL.

As an example, consider the 5-state chain [11]: a well known Markov decision process (MDP). This problem (see Figure **??**) consists of a set of 5 states arranged in a linear order. Action a_1 causes a transition to the next state in the ordering. Action a_2 resets the state to the first in the chain. Action a_1 from the last state in the chain results in remaining in the state with a high reward ($+10$) and zero reward otherwise. Action a_2 always has a reward of $+2$. Rewards are discounted, $\gamma = 0.95$. With probability 0.2 (the slip probability), however, the selected action has the effect of the non-selected action. Though tiny, this MDP is challenging for some learning algorithms because action a_2 acts as a temptation that keeps the agent from discovering the optimal policy (always take action a_1). We measure the performance of a learning algorithm in the environment by its probability of finding an optimal policy after 1000 steps of experience.

Consider two different variations of the Q-learning algorithm [13] that we might want to apply to this problem. In both, the learning rate (α) is set to a value between 0.0 and 0.5 and the exploration rate (ϵ) is set to a value between 0.0 and 0.4. In the *constant initialization* algorithm, all Q values are initially set to 45 (something in the vicinity of the likely final value function). In the *variable initialization* algorithm, each state–action pair is initialized independently to some value between 0 and 200. Note that the variable initialization algorithm subsumes the constant initialization algorithm since it can be configured to initially set all Q values to 45.

If we tune parameters to the 5-state chain MDP, the variable initialization algorithm will perform better. However, that does not mean the tuned variable initialization algorithm will perform better on a distribution of varying 5-state chain problems, depending on the properties of the distribution. For example, consider two different MDP distributions. In distribution 1, all MDPs are 5-state chains with states ordered consistently but slip probability varying between 0.19 and 0.21. In distribution 2, all MDPs are 5-state chains, with the order of states in the chain varying from MDP to MDP and slip probability varying between 0.00 and 0.50. By exhaustive testing shown in Figure 3, we find that for distribution 1, tuning variable initialization on a single MDP results in a training performance comparable to performance across the entire distribution. However, for distribution 2, the training performance of variable initialization on a single MDP is deceptive and constant initialization performs better on the full distribution. That is, variable initialization overfits to the single training example. As more training samples for distribution 2 are provided, variable initialization's training performance becomes more reflective of its performance on the full distribution and is the better choice.

Since performance on the training set of MDP samples can be deceptive, we would like a way to bound the *generalization error*. The generalization error is the difference in the performance of a selected algorithm on a training set and its performance on the full distribution. More formally, let $\mathcal{A}$ be the space of parameterized algorithms and let $\mathcal{F}$ be the space of evaluation functions of each algorithm $a \in \mathcal{A}$ such that for all $f_a \in \mathcal{F}$, $f_a : \mathcal{D} \to [0, 1]$, where $\mathcal{D}$ is the space of MDPs our MDP distribution spans. Let S be a sample $z_1, \ldots, z_m$, chosen from a distribution D on $\mathcal{D}$. Then the generalization error of algorithm a is $|\hat{E}_S[f_a] - E[f_a]|$, where $\hat{E}_S[f_a] = \frac{1}{m}\sum_{i=1}^{m} f_a(z_i)$ is the expected performance of a on the training data, and $E[f_a] = \int_{\mathcal{D}} f_a(z)D(z)dz$ is the expected performance of a on the full distribution.

References

[1] Fahiem Bacchus and Froduald Kabanza. Using temporal logic to control search in a forward chaining planner. In *In Proceedings of the 3rd European Workshop on Planning*, pages 141–153. Press, 1995.

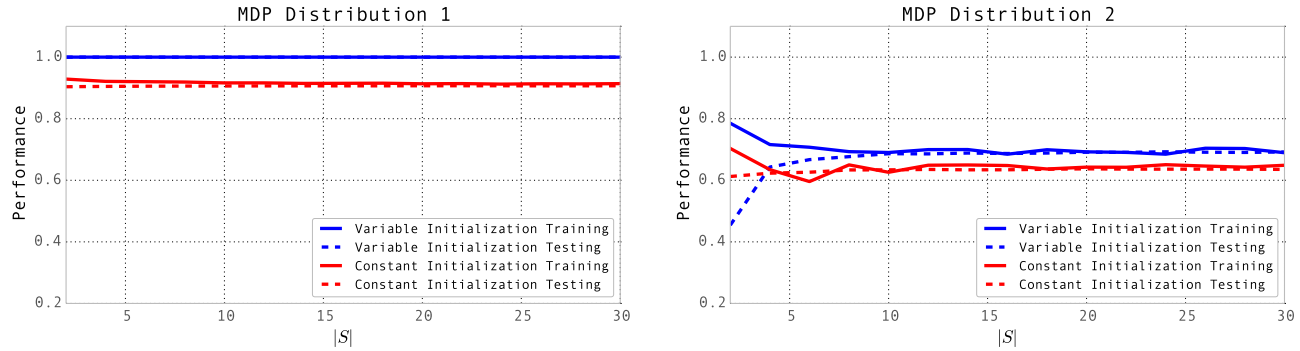

Figure 3: Training and testing performance for two distributions in the 5-state chain environment. Performance is measured as the probability of converging to the optimal policy.

[2] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:2000, 1999.

[3] Andrew G Barto, Steven J Bradtke, and Satinder P Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1):81–138, 1995.

[4] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM*, 36(4):929–965, 1989.

[5] Mario Bollini, Stefanie Tellex, Tyler Thompson, Nicholas Roy, and Daniela Rus. Interpreting and executing recipes with a cooking robot. In *Proceedings of International Symposium on Experimental Robotics (ISER)*, 2012.

[6] Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer. Macro-ff: Improving ai planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005.

[7] Nicholas K. Jong. The utility of temporal abstraction in reinforcement learning. In *Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multiagent Systems*, 2008.

[8] Ross A. Knepper, Stefanie Tellex, Adrian Li, Nicholas Roy, and Daniela Rus. Single assembly robot in search of human partner: Versatile grounded language generation. In *Proceedings of the HRI 2013 Workshop on Collaborative Manipulation*, 2013.

[9] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'99, pages 968–973, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[10] M Newton, John Levine, and Maria Fox. Genetically evolved macro-actions in ai planning problems. *Proceedings of the 24th UK Planning and Scheduling SIG*, pages 163–172, 2005.

[11] Malcolm Strens. A Bayesian framework for reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, 2000.

[12] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.

[13] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1 edition, 1998.