Approximate Policy Iteration using Large-Margin Classifiers

Michail G. Lagoudakis and Ronald Parr
Duke University
Durham, NC 27708
{mgl,parr}@cs.duke.edu

Abstract

We present an approximate policy iteration algorithm that uses rollouts to estimate the value of each action under a given policy in a subset of states and a classifier to generalize and learn the improved policy over the entire state space. Using a multi-class support vector machine as the classifier, we obtained successful results on the inverted pendulum and the bicycle balancing and riding domains.

1 Introduction

Reinforcement learning provides an intuitively appealing framework for addressing a wide variety of planning and control problems. There has been significant success in tackling large-scale problems through the use of value function and/or policy approximation. The success of these methods, however, is contingent upon extensive and careful feature engineering, a common problem in most machine learning methods.

Modern classification methods have mitigated the feature engineering problems through the use of kernel methods, but very little has been done to exploit these recent advances for the purposes of reinforcement learning. Of course, we are not the first to note the potential benefits of modern classification methods to reinforcement learning. For example, Yoon *et al.* [2002] use inductive learning techniques, including boosting, to generalize across similar problems. Dietterich and Wang [2001] also use a kernel-based approximation method to generalize across similar problems. The novelty in our approach is the application of modern learning methods *within* a single, noisy problem at the inner loop of a policy iteration algorithm.[1] By using rollouts, we avoid the sometimes problematic step of value function approximation. Thus, our technique aims to address the critiques of value function methods raised by the proponents of direct policy search, while avoiding the confines of a parameterized policy space.

2 Definitions and Assumptions

A *Markov decision process* (MDP) is defined as a 6-tuple (S, A, P, R, γ, D) where: S is the state space of the process; A is a finite set of actions; P is a Markovian transition model, where $P(s,a,s')$ is the probability of making a transition to state s' when taking action a in state s; R is a reward (or cost) function, such that $R(s,a)$ is the expected reward for taking action a in state $\gamma \in [0,1)$ is the discount factor for future rewards; and, D is the initial state distribution from which states are drawn when the process is initialized.

In reinforcement learning, it is assumed that the learner can observe the state of the process and the immediate reward at every step, however P and R are completely unknown. In this paper, we also make the assumption that our learning algorithm has access to a generative model of the process which is a black box that takes a state s and an action a as inputs and outputs a next state s' drawn from P and a reward r. Note that this is not the same as having the model (P and R) itself.

A *policy* π for an MDP is a mapping $\pi : S \mapsto A$ from states to actions, where $\pi(s)$ is the action the agent takes at state s. The value $V_\pi(s)$ of a state s under a policy π is the expected, total, discounted reward when the process begins in state s and all decisions at all steps are made according to policy π:

$$V_\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, \pi(s_t)) \;\middle|\; s_0 = s\right]$$

The goal of the decision maker is to compute or learn an optimal policy π^* that maximizes the expected total discounted reward from the initial state distribution:

$$\pi^* = \arg\max_\pi \eta(\pi, D) = E_{s\sim D}\left[V_\pi(s)\right] \ .$$

It is known that for every MDP, there exists at least one optimal deterministic policy.

3 (Approximate) Policy Iteration

Policy iteration (PI) is a method of discovering such a policy by iterating through a sequence of monotonically improving policies. Improvement at each iteration i is typically achieved by computing or learning the state-action value function Q_{π_i} of the current policy π_i, defined as

$$Q_{\pi_i}(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s,a,s') V_{\pi_i}(s') \ ,$$

and then the improved policy as

$$\pi_{i+1}(s) = \arg\max_{a \in A} Q_{\pi_i}(s,a) \ .$$

[1]Fern, Yoon, and Givan are also pursuing a similar approach.

In practice, policy iteration terminates in a surprisingly small number of steps, however, it relies on exact representation of value functions and policies.

Approximate methods are frequently used when the state space of the underlying MDP is extremely large and exact (solution or learning) methods fail. A general framework of using approximation within policy iteration is known as *approximate policy iteration* (API). In its most general form, API assumes approximate representations of value functions *(Q)* and policies $(\widehat{\pi})$. As a result, API cannot guarantee monotonic improvement and convergence to the optimal policy. A performance bound on the outcome of API can be constructed in terms of L_∞ bounds on the approximation errors [Bertsekas and Tsitsiklis, 1996]. In practice, API often finds very good policies in a few iterations, since it normally makes big steps in the space of possible policies. This is in contrast to policy gradient methods which, despite acceleration methods, are often forced to take very small steps.

4 Practical API without Value Functions

The dependence of typical API algorithms on approximate value functions places continuous function approximation methods at their inner loop. These methods typically minimize L_2 error, which is a poor match with the L_∞ bounds for API. This problem is *not* just theoretical. Efforts to improve performance, such as adding new features, can sometimes lead to surprisingly worse performance, making feature engineering a somewhat tedious and counterintuitive task.

An important observation, also noted by Fern, Yoon and Givan 120031, is that a Monte-Carlo technique, called *rollouts,* can be used within API to avoid the problematic value function approximation step entirely. Rollouts estimate Q_{nt} *(s,a)* from a generative model by executing action *a* in state *s* and following policy 7r, thereafter, while recording the total discounted reward obtained during the entire trajectory. This simulation is repeated several times and the results are averaged over a large number of trajectories to obtain an accurate estimate $\widetilde{Q}_{\pi_i}(s,a)$ of $Q_{\pi_i}(s,a)$. Rollouts were first used by Tesauro and Galperin [1997] for online improvement of a backgammon player. For our purposes, we can choose a representative set of states S_p (with distribution ρ over the state space) and perform enough rollouts to determine which action maximizes $\widetilde{Q}_{\pi_i}(s,a)$ for the current policy. Rather than fitting a function approximator to the values obtained by the rollouts, we instead train a classifier on S_{py} where each state is labelled with the maximizing action for the state. The algorithm is summarized in Figure 1.

LEARN is a supervised learning algorithm that trains a classifier given a set of labeled training data. The termination condition is left somewhat open ended. It can be when the performance of the current policy does not exceed that of the previous one, when two subsequent policies are similar (the notion of similarity will depend upon the learner used), or when a cycle of policies is detected (also learner dependent). If we assume a fortuitous choice of S_p and a sufficiently powerful learner that can correctly generalize from S_p to the entire state space, the algorithm at each iteration learns the improved policy over the previous one, effectively implementing a full policy iteration algorithm, and terminating with the optimal policy. However, for large-scale problems, choosing S_ρ and dealing with imperfect classifiers poses challenges.

```
i ← 0
$\widehat{\pi}_0$ ← arbitrary policy (default: random)

repeat
    for each $s \in S_\rho$
        $\widetilde{Q}_{\widehat{\pi}_i}(s,a)$ ← ROLLOUT($s, a, \widehat{\pi}_i$), $\forall a \in \mathcal{A}$
        $\pi_{i+1}(s) = \arg\max_{a \in \mathcal{A}} \widetilde{Q}_{\pi_i}(s,a)$
    $\widehat{\pi}_{i+1}$ ← LEARN($\{(s, \pi_{i+1}(s)) : s \in S_\rho\}$)
    $i = i + 1$
until termination
```

Figure 1: API with rollouts and classification

We consider a number of alternative choices for the distribution *p.* A uniform distribution over the state space is a good choice for a low-dimensional state space, but it will result in poor coverage in high-dimensional spaces. A better choice would be a distribution that favors certain important parts of the state space over others. In classification, it is widely assumed that the classifier is trained on examples that are drawn from the same distribution it will be tested on. Therefore, representative states for learning policy π_{i+1} in iteration *i* should be drawn from the γ-**discounted** future state distribution [Jaakkola *et al,* 19951 of the yet unknown policy $7T_{i+1}$. Even though this policy is yet unknown, it is possible to closely approximate this distribution[2]. Starting in some state .s_0 drawn from Z), a trajectory under π_{i+1} can be simulated by using rollouts repeatedly in each visited state to determine the actions of π_{i+1}. If the trajectory terminates with probability $(1-\gamma)$, then states visited along such trajectories can be viewed as samples from the desired distribution.

The main contribution of this paper is a particular embodiment of the algorithm described in the previous section. For rollouts, we used basic statistical hypothesis testing (two-sample f.-test) to determine statistically significant differences between the rollout estimates of $Q_{\widehat{\pi}_i}$ for different actions. If action *a* is determined to be a clear winner for state s, then we treat *(s, a)* as a positive training example for state s, and all (s, a'), $a' \neq a$ as negative examples for *s.* Even if there is no clear winner action in some state s, negative examples can be still extracted for the clearly bad actions in s, allowing the classifier to choose freely any of the remaining (good) actions as the winner. The only case where no examples are generated is when all actions seem to be equally good.

The most significant contribution of effort is the use of support vector machines (SVMs) for LEARN. In our implementation we used the SVMTorch package [Collobert and Bengio, 2001] as the multi-class classifier. With the kernel trick, SVMs are able to implicitly and automatically consider classifiers with very complex feature spaces.

Special thanks to Alan Fern for sharing this observation.

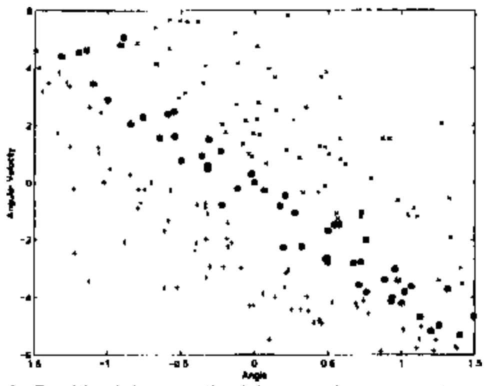

Figure 2: Positive(+), negativc(x) examples; support vcctors(o).

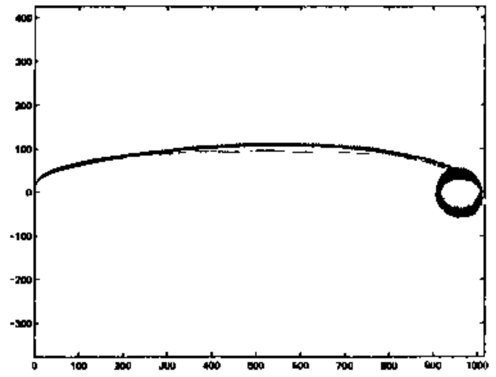

Figure 3: Successful trajectories of the bicycle.

5 Experimental Results

In the *inverted pendulum* problem the goal is to balance a pendulum by applying forces to the left (LF), or to the right (RF), or no force (NF) at all. All actions are noisy. The state space is continuous and consists of the vertical angle and the angular velocity of the pendulum. Transitions are governed by the nonlinear dynamics of the system and the time step is 0.01 seconds. The reward is -1 when the angle exceeds $\pi/2$ in absolute value (end of the episode) and 0 otherwise. The discount factor is 0.95. Using about 200 states from *p* to perform rollouts, the algorithm consistently learns balancing policies in one or two iterations, starting from the random policy. The choice of *p* or the kernel (polynomial/Gaussian) did not affect the results significantly. Figure 2 shows the training data for the LF action with *p* being the uniform distribution. The number of support vectors was normally smaller than the number of rollout states. The constant C, the trade-off between training error and margin, was set to 1.

In the bicycle balancing and riding problem [Randlov and Alstrom, 1998] the goal is to learn to balance and ride a bicycle to a target position located 1 km away from the starting location. The state description is a six-dimensional vector $(\theta, \dot\theta, \omega, \dot\omega, \ddot\omega, \psi)$, where θ is the angle of the handlebar, ω is the vertical angle of the bicycle, and ψ is the angle of the bicycle to the goal. The actions are the torque r applied to the handlebar {-2,0, +2} and the displacement of the rider *v* {-0.02,0,4-0.02}. Actions are restricted so that either r = 0 or *v* = 0 giving a total of 5 actions. The noise is a uniformly distributed term in [-0.02, + 0.02] added to the displacement. The dynamics of the bicycle are based on the model of Randlov and Alstrom [1998] and the time step is set to 0.02 seconds. As is typical with this problem, we used a shaping reward [Ng *et al.,* 1999]. The reward r_t given at each time step was r_t — $(d_{t-1} - \gamma d_t)$, where d_t is the distance of the back wheel of the bicycle to the goal position at time *t* and 7 is the discount factor which is set to 0.95.

Using a polynomial kernel of degree 5, we were able to solve the problem with uniform sampling of about 5000 rollout states. Sampling from the γ**-discounted** distribution of the target policy, the problem is solved with as few as 1000 rollout states, as shown in Figure 3 which shows ten sample trajectories of the bicycle in the two-dimensional plane from the initial position (0,0) (left side) to the goal position (1000,0) (right side). The input to the SVM was simply the six-dimensional state description and the value of *C* was 1. Performance was sensitive to the SVM parameters. We are currently doing further experimentation with larger numbers of rollout states and different kernel parameters.

This research was supported in part by NSF grant 0209088.

References

[Bcrtsekas and Tsitsiklis, 1996] D. Bertsekas and J. Tsitsiklis. *Neuro-Dynamic Programming.* Athena Scientific, Belmont, Massachusetts, 1996.

[Collobert and Bengio, 2001] Ronan Collobert and Samy Bcngio. SVMTorch: Support vector machines for large-scale regression problems. *Journal of Machine Learning Research (JMLR),* 1:143 160, 2001.

[Dietterich and Wang, 2001] T. G. Dietterich and X. Wang. Batch value funtion approximation via support vectors. In *Advances in Neural Information Processing Systems 14,* 2001. MIT Press.

I Fern *et al,* 2003] A. Fern, S. Yoon, and R. Givan. Approximately policy iteration with a policy language bias: Learning control policies in relational planning domains. In *Submitted to The Nineteenth International Conference on Machine Learning (ICML-2003),* July 2003.

[Jaakkola et al., 1995] Tommi Jaakkola, Satinder P. Singh, and Michael I. Jordan. Reinforcement learning algorithm for partially observable Markov decision problems. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems* 7, Cambridge, Massachusetts, 1995. MIT Press.

[Ng *et al,* 1999] Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: theory and application to reward shaping. In *Proc. 16th International Conf on Machine Learning,* pages 278-287. Morgan Kaufmann, San Francisco, CA, 1999.

[Randl0V and Alstram, 1998] J. Randlav and P. Alstrom. Learning to drive a bicycle using reinforcement learning and shaping. In *The Fifteenth International Conference on Machine Learning,* 1998. Morgan Kaufmann.

[Tesauro and Galperin, 1997] G. Tesauro and G. R. Galperin. Online policy improvement using monte-carlo search. In *Advances in Neural Information Processing Systems (NIPS) 9,* 1997.

[Yoon *et al,* 2002] S. W. Yoon, A. Fern, and B. Givan. Inductive policy selection for first-order MDPs. In *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence*, 2002. Morgan Kaufmann.