Reinforcement Learning as Classification: Leveraging Modern Classifiers

Michail G. Lagoudakis MGL@CS.DUKE.EDU
Ronald Parr PARR@CS.DUKE.EDU
Department of Computer Science, Duke University, Durham, NC 27708 USA

Abstract

The basic tools of machine learning appear in the inner loop of most reinforcement learning algorithms, typically in the form of Monte Carlo methods or function approximation techniques. To a large extent, however, current reinforcement learning algorithms draw upon machine learning techniques that are at least ten years old and, with a few exceptions, very little has been done to exploit recent advances in classification learning for the purposes of reinforcement learning. We use a variant of approximate policy iteration based on rollouts that allows us to use a pure classification learner, such as a support vector machine (SVM), in the inner loop of the algorithm. We argue that the use of SVMs, particularly in combination with the kernel trick, can make it easier to apply reinforcement learning as an "out-of-the-box" technique, without extensive feature engineering. Our approach opens the door to modern classification methods, but does not preclude the use of classical methods. We present experimental results in the pendulum balancing and bicycle riding domains using both SVMs and neural networks for classifiers.

1. Introduction

Reinforcement learning provides an intuitively appealing framework for addressing a wide variety of planning and control problems(Sutton & Barto, 1998). Compelling convergence results exist for small state spaces(Jaakkola et al., 1994) and there has been some success in tackling large state spaces through the use of value function approximation and/or search through a space of parameterized policies (Williams, 1992).

Despite the successes of reinforcement learning, some frustration remains about the extent and difficulty of feature engineering required to achieve success. This is true both of value function methods, which require a rich set of features to represent accurately the value of a policy, and policy search methods, which require a parameterized policy function that is both rich enough to express interesting policies, yet smooth enough to ensure that good policies can be discovered. The validity of such criticisms is debatable (since nearly all practical applications of machine learning methods require *some* feature engineering), but there can be no question that recent advances in classifier learning have raised the bar. For example, it is not uncommon to hear anecdotal reports of the naive application of support vector machines matching or exceeding the performance of classical approaches with careful feature engineering.

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 bagging, 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 its orientation towards the application of modern classification methods *within* a single, noisy problem at the inner loop of a policy iteration algorithm. By using rollouts and a classifier to represent policies, we avoid the sometimes problematic step of value function approximation. Thus we aim 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.

We note that in recent work, Fern, Yoon and Givan(2003) also examine policy iteration with rollouts and an inductive learner at the inner loop. However, their emphasis is different. They focus on policy space bias as a means of searching a rich space of policies, while we emphasize modern classifiers as a method of recovering high dimensional structure in policies.

2. Basic definitions and algorithms

In this section we review the basic definitions for Markov Decision Processes (MDPs), policy iteration, approximate policy iteration and rollouts. This is intended primarily as a review and to familiarize the reader with our notation. More extensive discussions of these topics are widely available (Bertsekas & Tsitsiklis, 1996).

Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003), Washington DC, 2003.

2.1. MDPs

An MDP is defined as a 6-tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma, D)$ where: $\mathcal{S}$ is the state space of the process; $\mathcal{A}$ 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 s; $\gamma \in [0, 1)$ 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 *deterministic policy* π for an MDP is a mapping $\pi : \mathcal{S} \mapsto \mathcal{A}$ 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:

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

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

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

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

2.2. Policy Iteration

Policy iteration is a method of discovering such a policy by iterating through a sequence π_1, π_2, ..., π_k of improving policies. The iteration terminates when there is no change in the policy ($\pi_k = \pi_{k-1}$) and π_k is the optimal policy. This is typically achieved by computing V_{π_i}, which can be done iteratively or by solving a system of linear equations and then determining a set of Q-values:

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

and the improved policy as

$$\pi_{i+1}(s) = \arg\max_a Q_{\pi_i}(s, a) \quad .$$

In practice, policy iteration terminates in a surprisingly small number of steps. However, it relies on the availability of the full model of the MDP, exact evaluation of each policy, and exact representation of each policy.

2.3. Approximate methods

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 a policy $\widehat{\pi}_i$ at each step, which may not necessarily be an exact implementation of the improved policy from the previous time step. Typically, the error introduced is bounded by the maximum norm (L_∞) error in what is presumed to be an approximate $\widehat{Q}_{\pi_{i-1}}$ that was used to generate π_{i+1} (Bertsekas & Tsitsiklis, 1996). API cannot guarantee monotonic improvement and convergence to the optimal policy. However, in practice it 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.

LSPI (Lagoudakis & Parr, 2001) is an example of an API algorithm. It uses temporal differences of sample transitions between state-action pairs to approximate each Q_π. It is also possible to use a technique closer to pure Monte Carlo evaluation called *rollouts*. Rollouts estimate $Q_\pi(s, a)$ by executing action a in state s and following policy π thereafter, while recording the total discounted reward obtained during the run. This processes is repeated several times and it requires a simulator with resets to arbitrary states (or a generative model) since a large number of trajectories is required to obtain an accurate estimate of $Q_\pi(s, a)$. Rollouts were first used by Tesauro and Galperin (1997) to improve online performance of a backgammon player. (A supercomputer did rollouts before selecting each move.) However, they can also be used to specify a set of target values for a function approximator used within API to estimate Q_π. The rollout algorithm is summarized in Figure 1.

3. API without Value Functions

The dependence of typical API algorithms on value functions places continuous function approximation methods at the inner loop of most approximate policy iteration algorithms. 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 to a neural network or new basis functions to LSPI don't always have the expected effect. Increasing expressive power can sometimes lead to

Rollout ($\mathcal{M},\ s,\ a,\ \gamma,\ \pi,\ K,\ T$)

// $\mathcal{M}$: Generative model
// (s, a) : State-action pair whose value is sought
// γ : Discount factor
// π : Policy
// K : Number of trajectories
// T : Length of each trajectory

for $k = 1$ **to** K
  $(s', r) \leftarrow \text{SIMULATE}(\mathcal{M}, s, a)$
  $\widetilde{Q}_k \leftarrow r$
  $s \leftarrow s'$
  for $t = 1$ **to** $T - 1$
    $(s', r) \leftarrow \text{SIMULATE}(\mathcal{M}, s, \pi(s))$
    $\widetilde{Q}_k \leftarrow \widetilde{Q}_k + \gamma^t r$
    $s \leftarrow s'$

$$\widetilde{Q} \leftarrow \frac{1}{K}\sum_{k=1}^{K} \widetilde{Q}_k$$

return $\widetilde{Q}$

Figure 1. Estimation of state-action values using rollouts.

surprisingly worse performance, which can make feature engineering a somewhat tedious and counterintuitive task.

If a model is available and the structure of the value function approximation is compatible with the model, value function approximation minimizing L_∞ error is possible (Guestrin et al., 2001). There are arguments that such approximations are better suited to MDPs, suggesting the use of something like support vector regression, which tries to limit the worst case approximation error. Unfortunately, the approach does not generalize easily to reinforcement learning in the presence of noise. To see this, consider that in minimizing L_2 error, we serve two objectives: averaging within states and smoothing across states. Thus, the mean state value (based upon trajectories or temporal differences) is also the one that minimizes L_2 error at that state. In contrast, minimizing L_∞ error is ill suited to raw data of the type encountered in reinforcement learning since the mean state value can differ sharply from the L_∞ error minimizing estimate. (Suppose we draw 100 next states from state s, of which 99 have value 1.0 and 1 has value 0.0. The mean value for $V(s)$ is 0.99, which is also the value that minimizes the L_2 error in the temporal differences. However, $V(s) = 0.5$ minimizes the L_∞ error.)

An important observation, also noted by Fern, Yoon and Givan (2003), is that rollouts can be used within API to avoid the problematic value function approximation step entirely. We choose some representative set of states S_ρ and assume that we can perform enough rollouts to determine which action maximizes $Q_\pi(s, a)$ for the current policy. Rather than fitting a function approximator to the values obtained by the rollouts, we instead train a classification learner where the maximizing action is the label

// $\mathcal{M}$: Generative model
// D_ρ: Source of rollout states
// γ : Discount factor
// π_0: Initial policy (default: uniformly random)
// K: Number of trajectories
// T : Length of each trajectory

$\pi' = \pi_0$

repeat
  $\pi = \pi'$
  $\mathsf{TS} = \varnothing$
  for each $s \in D_\rho$
    for each $a \in \mathcal{A}$
      $\widetilde{Q}^\pi(s, a) \leftarrow$ **Rollout**($\mathcal{M},\ s,\ a,\ \gamma,\ \pi,\ K,\ T$)
    $a^* = \arg\max_{a \in \mathcal{A}} \widetilde{Q}^\pi(s, a)$
    if $\forall\ a \in \mathcal{A},\ a \neq a^* :\ \widetilde{Q}^\pi(s, a) \lesssim \widetilde{Q}^\pi(s, a^*)$
      $\mathsf{TS} \leftarrow \mathsf{TS} \cup \{(s, a^*)^+\}$
    for each $a \in \mathcal{A} :\ \widetilde{Q}^\pi(s, a) \lesssim \widetilde{Q}^\pi(s, a^*)$
      $\mathsf{TS} \leftarrow \mathsf{TS} \cup \{(s, a)^-\}$
  π' = **Learn**(TS)
until ($\pi \approx \pi'$)

return π

Figure 2. Approximate Policy Iteration with Rollouts.

for the state. This approximate policy iteration algorithm is described in Figure 2.

We use the notation $(s, a)^+$ to indicate a positive training example, and $(s, a)^-$ to indicate a negative example. **Learn** is a supervised learning algorithm that trains a classifier given a set of labeled training data. The structure of the policy iteration algorithm naturally suggests a *batch* implementation since the policy is updated in distinct phases. 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_ρ and a sufficiently powerful learner that can correctly generalize from S_ρ to the entire state space, the i^{th} iteration of this algorithm will learn the improved policy of π_i, effectively implementing a full policy iteration algorithm, and terminating with the optimal policy. For large-scale problems, choosing S_ρ and dealing with imperfect classifiers will pose some challenges.

4. Choosing S_ρ

For the choice of S_ρ, we have a number of alternatives. The simplest is to try to dense, uniform covering of the state space. For low-dimensional state spaces, this will be practical, but it scales poorly. A similar option would be to randomly select S_ρ from some uniform distribution over the state space. This is again problematic due to poor cov-

erage for high-dimensional spaces.

A natural choice of S_ρ would be the distribution of states induced by the current policy, π_i. While intuitively appealing, this distribution may differ dramatically from the distribution of the subsequent policy, π_{i+1}, for which we must train our classifier. (To see this, consider a π_i that directs the system towards one "side" of the state space and a π_{i+1} that directs the system towards another side. If we train our classifier on states drawn from π_i, when we try to use our classifier to execute π_{i+1}, it may be asked to classify states that are disjoint from the ones it has been trained on.) This mismatch between training and testing can be dealt with by using a step size α, $0 < \alpha \leq 1$, to keep the policy for the next iteration sufficiently close to the current policy so that performance does not degrade:

$$\tilde{\pi}_{i+1} = \alpha \arg\max_a \widehat{Q}_{\pi_i}(s,a) + (1-\alpha)\widehat{\pi}_i \ ,$$

Note that $\widehat{\pi}_{i+1}$ is now a stochastic policy that chooses from the improved policy with probability α and from the old policy with probability $(1-\alpha)$. A positive α that improves performance is guaranteed to exist (Jaakkola et al., 1995). Recently, Kakade and Langford (2002) demonstrated a method for picking α near optimally, although the largest "safe" value α may be quite small.

Fortunately, our assumption of a generative model gives us the luxury of drawing states from the policy we wish to learn before we have completely discovered or learned it. While this may sound paradoxical at first, it is actually quite simple thanks to an observation by Fern (personal communication). If we begin in some state s_0, we can use rollouts to determine π_{i+1} for s_0. We can then sample s_1, by executing action $\pi_{i+1}(s_0)$ in s_0. We continue by using rollouts to determine $\pi_{i+1}(s_1)$ and executing this action to obtain s_2. We continue in this fashion until we have sampled states and actions along an entire trajectory of π_{i+1} starting from s_0. Trajectories produced during rollouts are discarded, the only training kept are from π_{i+1}.

Fern's observation mostly solves the S_ρ problem, but it leaves open the question of how the initial state s_0 is selected. The initial distribution D may seem like a natural distribution from which to draw s_0. In practice, however, this can cause API to get stuck in local optima: Suppose π_i visits only a small region of the state space. To improve upon π_i, rollouts must discover better alternatives at the fringe of the states reachable by π_i. However, our classifier for π_i was never trained on states that aren't reachable by π_i, making it unlikely that rollouts from the frontier of π_i will produce a better alternative to staying within the region normally circumscribed by π_i. The choice of a "restart distribution" which differs from the problems natural starting distribution is also explored by Kakade and Langford(2002), who show that a poor choice of a restart distribution can lead to arbitrarily bad performance (policy loss that grows with the size of the state space). Of course, the ideal restart distribution would be that of the optimal policy, but this begs the question.

A related practical problem is what to do in states where rollouts cannot provide sufficient information to select π_{i+1}. We discuss this issue in Section 6.

5. Imperfect Learners

Suppose that our learner fails to learn $\pi_i(s)$ perfectly when presented with $\tilde{\pi}_i(s)$ for all s in S_ρ. To quantify the extent of this failure, we must first define the test distribution. Following Kakade and Langford (2002), we express the natural test distribution for this problem as the set of states encountered when starting from states drawn from the initial distribution D, and following $\widehat{\pi}_i$. The probability of reaching future states is discounted by γ and the infinite sum is normalized by $(1-\gamma)$, resulting in:

$$d_{\pi_i,D}(s) = (1-\gamma)\sum_{t=0}^{\infty}\gamma^t (P_{\pi_i}^{t+1}D)_s$$

where the s subscript indicates that we are selecting component s of the matrix-vector product.

If we have an *a priori* guarantee that our learner will choose the wrong action with probability at most δ on states drawn from this distribution, then we can bound the expected shortfall from following π_i instead of $\widehat{\pi}_i$ as follows:

$$V_{\widehat{\pi}_i}(s) - V_{\pi_i}(s) \leq \frac{\delta R_{\max}}{(1-\gamma)^2},$$

where $R_{\max}$ is the maximum reward value. This pessimistic bound arises from the assumption that all mistakes are made in the initial states, which occur with probability $(1-\gamma)$, incurring penalty $R_{\max}/(1-\gamma)$. Since we cannot guarantee that such learners exist, this is not meant to be a serious bound, but reassurance that in principle good learners can produce good policies. In practice, the actual loss is best measured empirically by Monte Carlo evaluation, or estimated by the error rate on the training set. From a practical standpoint, high observed errors (or low performance) will suggest a change in representation, or a change in the learning mechanism, such as a change of kernel.

6. A Practical Algorithm

The main contribution of our paper is a particular embodiment of the approximate policy iteration algorithm described in Section 3. Training examples can be formed for any given state $s \in S$ assuming some underlying policy $\widehat{\pi}$. The estimated values $\widetilde{Q}^{\widehat{\pi}}(s,a)$ are computed by rollouts for all possible actions in state s. If the values $\widetilde{Q}^{\widehat{\pi}}(s,a)$

were exact, then the maximizing action a^* would yield one positive example $(s, a^*)^+$ and the rest of the actions would yield a number of negative examples $(s, a)^-$ for all $a \neq a^*$. Unfortunately, the estimates $\widetilde{Q}^{\widehat{\pi}}(s, a)$ are noisy and could yield incorrect examples if treated as exact. Thus, we used a simple two-sample t-test to compare rollout values. To generate examples in any state s using the rollout values $\widetilde{Q}^{\pi}(s, a)$, we did the following:

1. Use a fixed budget of k samples to determine $\widetilde{Q}^{\pi}$ in state s and a^*:

$$a^* = \arg\max_{a \in \mathcal{A}} \widetilde{Q}^{\pi}(s, a) \ .$$

2. Generate a positive example $(s, a^*)^+$ if the value of action a^* is statistically significantly bigger than the value of every other action $a \in \mathcal{A}$:

$$\forall\ a \in \mathcal{A},\ a \neq a^* :\ \widetilde{Q}^{\pi}(s, a) \lesssim \widetilde{Q}^{\pi}(s, a^*) \ .$$

3. Generate a negative example $(s, a)^-$ for each action a whose value is statistically significantly smaller than the value of action a^*:

$$\forall a \in \mathcal{A} :\ \widetilde{Q}^{\pi}(s, a) \lesssim \widetilde{Q}^{\pi}(s, a^*) \ .$$

A positive example is generated only if there is a clearly best action in which case all other actions generate negative examples. If there is no best action, negative examples can still be generated for the actions that are clearly inferior. Notice that in this case the remaining actions appear to be equally good and, by not generating a positive example, the classifier is essentially given the freedom to choose any of them. The only case where no training examples are generated is when *all* actions appear to be equally good. We expect this approach would benefit from more sophisticated approaches to managing the number of samples used (Kaelbling, 1993; Kearns et al., 1999).

One peculiarity of rollout based policy iteration is that if the current policy is very good, i.e. able to recover from small mistakes, there will be no statistically significant differences between many of the actions. This can make it difficult to acquire sufficient training data for the next policy. We mitigate this problem by treating the demonstrably bad actions as negative training examples even if we cannot determine a single, clearly superior action. Note that randomly selecting an action among the equivalent ones and marking it as positive will create a lot of noise for our learner since subsequent visits to the same state may pollute the training set with multiple "optimal" actions for the same state. A simple lexicographic ordering can also have unexpected side effects at execution time by introducing strong preferences for particular actions and heavily biasing the training data with examples of just one class.

The most significant contribution of effort is that it opens reinforcement learning to the full array of modern classification methods through the **learn** function. SVMs are a particularly appealing choice to the reinforcement learning practitioner vexed by the feature selection problem. We offer a brief sketch of how SVMs work to justify this appeal: With the kernel trick, SVMs are able to implicitly and automatically consider classifiers with very complex feature spaces. Nevertheless, the optimization performed by SVMs can be interpreted as a search through a space of classifiers to find one that is both a good fit and has low VC dimension. In the most optimistic interpretation, this dodges the feature selection problem while simultaneously demonstrating resistance to overfitting. In practice there are, of course, complications but if SVMs come close to this dramatic and optimistic description, we should be able to feed the raw state variables used by our simulators into our SVM classifier with little regard for the feature engineering required to obtain success in these problems using value function methods.

While SVMs are a particularly appealing choice for **learn**, they are not the only option and may not be the most desirable option in many cases. The theoretical motivations for using SVMs are not as crisp for multiclass problems. For problems with many actions, other classification methods may be more natural: neural nets, Bayes nets, decision trees, etc. For these reasons, and for the sake of comparison, we also implemented **learn** using a neural network. We designed the neural network with a number of outputs equals to the number of actions and trained the network to activate output i (and not others) output on positive examples $(s, a_i)^+$. Our neural network classifier did not take advantage of negative examples.

7. Experimental Results

We implemented the SVM version of our API algorithm using SVMTorch (Collobert & Bengio, 2001), a publicly available implementation of support vector machines. The SVMTorch package provides a simple multiclass capability (one versus all), but is not necessarily representative of the best that can be done on multiclass problems using SVM technology. We also implemented a version of our algorithm using a simple feedforward, multi-layer neural network as the multiclass classifier. In this section, we present experimental results on the *inverted pendulum* problem and the *bicycle balancing and riding* problem. Our goal in these preliminary experiments is not necessarily to demonstrate the superiority of our rollout approach in terms of CPU cycles or sample complexity, but rather its viability as an alternate approach to the reinforcement learning control problem.

In our experiments we ran approximate policy iteration un-

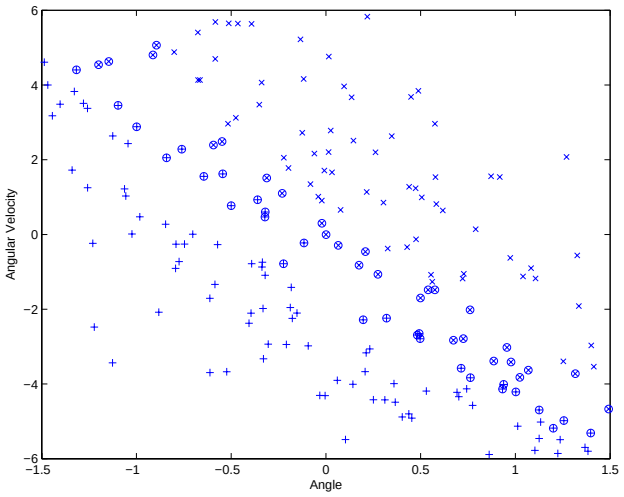

Figure 3. Training data (+ : positive, x : negative) and support vectors (o) for the LF action.

til the observed performance of the policy, as measured with experiments with the simulator, decreased. Since approximate policy iteration does not ensure monotonically improving policies, it is possible that continuing to run policy iteration beyond an initial setback could still result in better policies, but we did not explore this possibility.

7.1. Inverted pendulum

In the inverted pendulum domain, the task is to balance a pendulum of unknown length and mass at the upright position by applying forces to the cart to which it is attached. Three actions are allowed: left force LF (-50 Newtons), right force RF ($+50$ Newtons), or no force (NF) at all (0 Newtons). All three actions are noisy; uniform noise in $[-10, 10]$ is added to the chosen action. The state space of the problem is continuous and consists of the vertical angle θ and the angular velocity $\dot{\theta}$ of the pendulum. The transitions are governed by the nonlinear dynamics of the system (Wang & Griffin, 1996). and depend on the current state and the current (noisy) control u:

$$\ddot{\theta} = \frac{g\sin(\theta) - \alpha ml(\dot{\theta})^2\sin(2\theta)/2 - \alpha\cos(\theta)u}{4l/3 - \alpha ml\cos^2(\theta)} \quad ,$$

where g is the gravity constant ($g = 9.8m/s^2$), m is the mass of the pendulum ($m = 2.0$ kg), M is the mass of the cart ($M = 8.0$ kg), l is the length of the pendulum ($l = 0.5$ m), and $\alpha = 1/(m+M)$. A reward of 1 is given as long as the angle of the pendulum does not exceed $\pi/2$ in absolute value (the pendulum is above the horizontal line). An angle greater than $\pi/2$ signals the end of the episode and a reward (penalty) of 0. The discount factor of the process is set to 0.95.

Using about 200 rollout states, the algorithm consistently learns excellent balancing policies in one or two iterations with both neural nets and SVMs, starting with an initial policy that selects actions randomly with uniform probability. Such "excellent" policies balance the pendulum for more then 3 simulated minutes (in practice, we found that such policies could balance essentially indefinitely). The choice of the sampling distribution did not affect the results significantly. For illustration, we used uniform sampling for rollout states. Figure 3 shows the training data obtained for the LF action. A positive example indicates a state where LF was found to be the best action and a negative example is a state where LF was found to be a bad choice. It is easy to see that positive and negative examples are easily separated. The same figure also shows the resulting support vectors for the LF classifier using a polynomial kernel of degree 2.

Figure 4 shows the entire learned policies (blue/dark-gray for LF, red/medium-gray for RF, and green/light-gray for NF) for all three classifiers: SVM with a polynomial kernel, SVM with a Gaussian kernel, and a neural network classifier with 5 hidden units. Interestingly, in the case of the polynomial kernel, the policy does not use the NF action at all, whereas the other policies do. This is due to the limited abilities of the polynomial degree-2 kernel. All policies are excellent in the sense that they can all balance the pendulum for a long time, perhaps indefinitely. In all cases, the input to the SVM or the neural network was just the 2-dimensional state description. For SVMs, 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.

We note that pendulum balancing is a relatively simple problem. The classes are nearly linearly separable, so good classification performance here should not be surprising to those familiar with modern classification methods. Noteworthy features from the reinforcement learning perspective are the small number of iterations of policy iteration required and the non-parametric representation of the policy. Figure 3 shows the ability of the SVM to adapt the representation to match the training data since only the support vectors are used to represent the policy.

7.2. Bicycle riding

In the bicycle balancing and riding problem (Randløv & Alstrøm, 1998) the goal is to learn to balance and ride a bicycle to a target position located 1 km away from the starting location. Initially, the bicycle's orientation is at an angle of 90° to the goal. The state description is a six-dimensional real-valued 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.

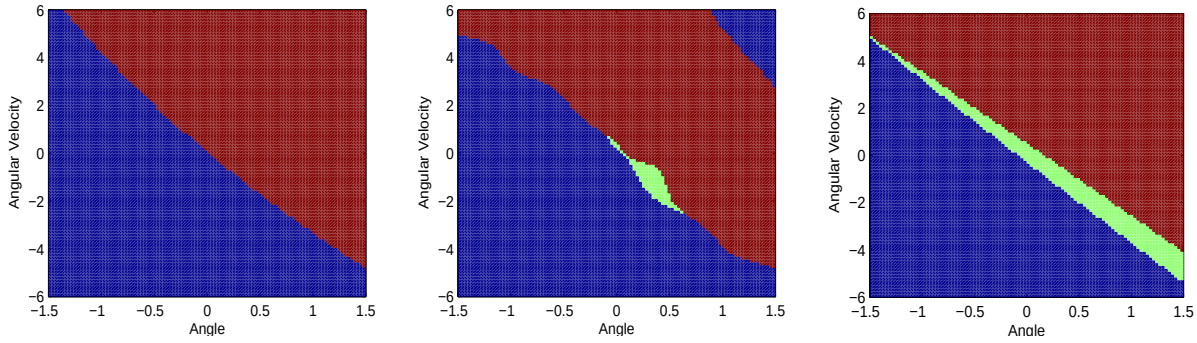

Figure 4. Pendulum: policies learned with the polynomial kernel SVM, the Gaussian kernel SVM, and the neural network classifier.

The actions are the torque τ applied to the handlebar (discretized to $\{-2, 0, +2\}$) and the displacement of the rider υ (discretized to $\{-0.02, 0, +0.02\}$). In our experiments, actions are restricted so that either $\tau = 0$ or $\upsilon = 0$ giving a total of 5 actions. The noise in the system is a uniformly distributed term in $[-0.02, +0.02]$ added to the displacement component of the action. The dynamics of the bicycle are based on the model of Randlov and Alstrom (1998) and the time step of the simulation is set to 0.02 seconds. As is typical with this problem, we used a shaping reward (Ng et al., 1999).

Our experiments with the bicycle did show some sensitivity to the parameters of the problem as well as the parameters of our learner. This made it difficult for us to find parameters that consistently produced good performance. Some of this may simply be reflective of our inexperience in tuning the parameters of SVMs. It is also possible that we did not consider enough samples.

For our SVM experiments, we used a shaping reward of r_t given at each time step, where $r_t = (d_{t-1} - \gamma d_t)$ as long as $|\omega| < \pi/15$, and $r = 0$ otherwise. d_t is the distance of the back wheel of the bicycle to the goal position at time t. The discount factor was set to 0.95.

In our preliminary experiments with this domain, we were able to solve the problem with uniform sampling and polynomial kernels of low degree. However it required a large number of rollout states (about $5,000$). With sampling from the distribution of the next policy, we were able to solve the problem with fewer rollout states and both RBF and polynomial kernels. However we did not find kernels that consistently produced good policies with reasonable sample sizes. (The balancing problem is solved easily using any of the classification methods, but riding to the goal proved more difficult.)

Figure 5 shows a sample trajectory from the final policy of one of our better policy iteration runs using SVMs. The bicycle moves in the 2-dimensional plane from the initial position $(0, 0)$ (left side) to the goal position $(1000, 0)$ (right side). This policy was produced with a polynomial kernel of degree 3 and 4000 rollout states. In the final policy, the bicycle rides to the goal, then turns around toward the goal in a very tight radius. This policy was obtained in just two API iterations, starting with a uniformly random action selection policy. Similarly to the pendulum, the input to the SVM was the raw 6-dimensional state description and $C = 1$.

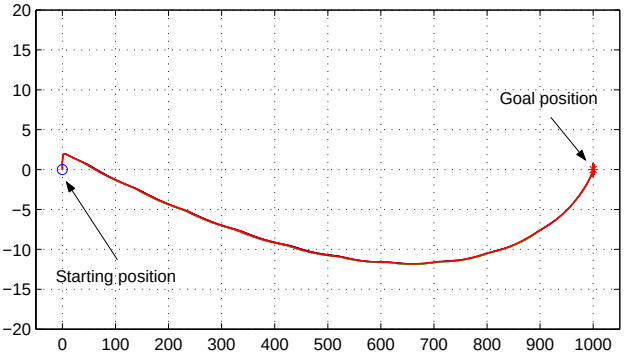

Figure 5. Bicycle: Trajectory of an SVM policy.

For our neural network experiments, we used a shaping reward of r_t given at each time step, where $r_t = 1 + (d_{t-1} - \gamma d_t)$ as long as $|\omega| < \pi/15$, and $r = 0$ otherwise. d_t is the distance of the back wheel of the bicycle to the goal position at time t. The discount factor was set to 0.99. Since our neural network learner only uses positive examples and not all states successfully produce positive training instances, we used 8000 rollout states.

Figure 5 shows sample trajectories of one of our better neural network policy iteration runs using 30 hidden units. After the first iteration, the learned policy can only balance the bicycle for a few steps and it crashes. The policy at the second iteration reaches the goal, but fails to return to it. Finally, the policy at the third iteration, reaches the goal faster and stays there. The best neural network policy is not

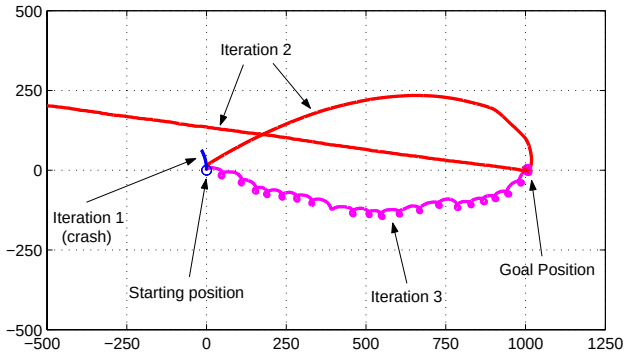

Figure 6. Bicycle: policies at successive iterations (NN classifier).

as good as the best SVM policy, but it is more illustrative of the progress of policy iteration because it takes an extra iteration.

8. Discussion

We have presented a case for an approach to RL that combines policy iteration and pure classification learning with rollouts. The emphasis of the approach in this paper is the ability to use of modern classification techniques such as SVMs to alleviate some of the burden of feature engineering from the practitioner of reinforcement learning. However, our empirical results also suggest that more traditional methods such as neural networks can be used successfully.

We believe that these initial successes will help open the door to greater exploitation of modern classification methods on more challenging reinforcement learning domains. Of course, many questions remain. More thorough investigation of issues relating to sample complexity and restart distributions are important areas for future work.

Acknowledgments

This research was supported in part by NSF grant 0209088. We also thank Alan Fern, Bob Givan, Carlos Guestrin and Ryan Deering for helpful discussions.

References

Bertsekas, D., & Tsitsiklis, J. (1996). *Neuro-dynamic programming*. Belmont, Massachusetts: Athena Scientific.

Collobert, R., & Bengio, S. (2001). SVMTorch: Support vector machines for large-scale regression problems. *Journal of Machine Learning Research (JMLR), 1*, 143–160.

Dietterich, T. G., & Wang, X. (2001). Batch value funtion approximation via support vectors. *Advances in Neural Information Processing Systems 14: Proceedings of the 2001 Conference*. Vancouver, British Columbia: MIT Press.

Fern, A., Yoon, S., & Givan, R. (2003). *Approximate policy iteration with a policy language bias: Learning control knowledge in planning domains*Technical report TR-ECE-03-11). Purdue University School of Electrical and Computer Engineering.

Guestrin, C. E., Koller, D., & Parr, R. (2001). Max-norm projections for factored MDPs. *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)* (pp. 673 – 680). Seattle, Washington: Morgan Kaufmann.

Jaakkola, T., Jordan, M., & Singh, S. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation, 6*, 1185–1201.

Jaakkola, T., Singh, S. P., & Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable Markov decision problems. *Advances in Neural Information Processing Systems 7* (pp. 345–352). Cambridge, Massachusetts: MIT Press.

Kaelbling, L. P. (1993). *Learning in embedded systems*. Cambridge, Massachusetts: MIT Press.

Kakade, S., & Langford, J. (2002). Approximately optimal approximate reinforcement learning. *The Nineteenth International Conference on Machine Learning (ICML-2002)*. Sydney, Australia.

Kearns, M., Mansour, Y., & Ng, A. Y. (1999). A sparse sampling algorithm for near-optimal planning large markov decision processes. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)* (pp. 1324–1331). Stockholm, Sweden: Morgan Kaufmann.

Lagoudakis, M., & Parr, R. (2001). Model free least squares policy iteration. *To appear in 14th Neural Information Processing Systems (NIPS-14)*. Vancouver, Canada.

Ng, A. Y., Harada, D., & Russell, S. (1999). Policy invariance under reward transformations: theory and application to reward shaping. *Proc. 16th International Conf. on Machine Learning* (pp. 278–287). Morgan Kaufmann, San Francisco, CA.

Randløv, J., & Alstrøm, P. (1998). Learning to drive a bicycle using reinforcement learning and shaping. *The Fifteenth International Conference on Machine Learning*. Madison, Wisconsin: Morgan Kaufmann.

Sutton, R., & Barto, A. (1998). *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press.

Tesauro, G., & Tesauro, G. (1997). On-line policy improvement using monte-carlo search. *9th Neural Information Processing Systems (NIPS-9)*. Denver, Colorado.

Wang, H. Tanaka, K., & Griffin, M. (1996). An approach to fuzzy control of nonlinear systems: Stability and design issues. *IEEE Transactions on Fuzzy Systems, 4*, 14–23.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning, 8*, 229–256.

Yoon, S. W., Fern, A., & Givan, B. (2002). Inductive policy selection for first-order MDPs. *Proceedings of the Eighteenth Conference on Uncertainty in Artificial Intelligence (UAI-02)*. Edmonton, Canada: Morgan Kaufmann.