Proceedings of the
47th IEEE Conference on Decision and Control
Cancun, Mexico, Dec. 9-11, 2008

Approximate Dynamic Programming Using Support Vector Regression

Brett Bethke, Jonathan P. How, and Asuman Ozdaglar

Abstract— **This paper presents a new approximate policy iteration algorithm based on support vector regression (SVR). It provides an overview of commonly used cost approximation architectures in approximate dynamic programming problems, explains some difficulties encountered by these architectures, and argues that SVR-based architectures can avoid some of these difficulties. A key contribution of this paper is to present an extension of the SVR problem to carry out approximate policy iteration by forcing the Bellman error to zero at selected states. The algorithm does not require trajectory simulations to be performed and is able to utilize a rich set of basis functions in a computationally efficient way. Computational results for an example problem are shown.**

I. Introduction

Dynamic programming is a framework for addressing problems involving sequential decision making under uncertainty [1]. Such problems occur frequently in a number of fields, including engineering, finance, and operations research. This paper considers the general class of infinite horizon, discounted, finite state Markov Decision Processes (MDPs). The MDP is specified by $(\mathscr{S},\mathscr{A},P,g)$, where $\mathscr{S}$ is the state space, $\mathscr{A}$ is the action space, $P_{ij}(u)$ gives the transition probability from state i to state j under action u, and $g(i,u)$ gives the cost of taking action u in state i. We assume that the MDP specification is fully known. Future costs are discounted by a factor $0<\alpha<1$. A policy of the MDP is denoted by $\mu:\mathscr{S}\rightarrow\mathscr{A}$. Given the MDP specification, the problem is to minimize the so-called cost-to-go function J_μ over the set of admissible policies Π:

$$\min_{\mu\in\Pi} J_\mu(i_0) = \min_{\mu\in\Pi} \mathbb{E}[\sum_{k=0}^{\infty} \alpha^k g(i_k,\mu(i_k))].$$

While MDPs are a powerful and general framework, they suffer from the well-known "curse of dimensionality", which states that as the problem size increases, the amount of computation necessary to find the solution increases exponentially rapidly. Indeed, for most MDPs of interest in the real world, this difficulty renders them impossible to solve exactly. To overcome the curse of dimensionality, researchers have investigated a number of methods for generating approximate solutions to large dynamic programs, giving rise to fields such as *approximate dynamic programming*, *reinforcement learning*, and *neuro-dynamic programming*. An important technique employed in many of these methods is the use of a parametric function approximation architecture to approximate the cost-to-go function J_μ of a given policy μ. Once the cost-to-go function (or a suitable approximation thereof) is known, an improved policy can usually be computed. This process of policy evaluation followed by policy improvement is then repeated, yielding a method known as *approximate policy iteration* which produces a sequence of potentially improving policies [2].

A important problem in these approximate policy iteration methods is the choice of the cost function approximation architecture employed. The approximation is denoted by $\widetilde{J}_\mu(i;\underline{\theta})$, where $i\in\mathscr{S}$ is a chosen state and $\underline{\theta}$ is a vector of tunable parameters. Numerous approximation architectures have been investigated. Neural networks have received much attention as a useful approximation architecture. For example, Tesauro used a neural network approach in his famous TD-Gammon computer backgammon player, which was able to achieve world-class play [3]. The *linear combination of basis functions* approach [1], [2], [4]–[7] has also been investigated extensively. In this approach, the designer picks a set of r basis functions $\phi_k(i), k\in\{1\ldots r\}$. The approximation $\widetilde{J}_\mu(i;\underline{\theta})$ is then given by a linear combination of the basis functions,

$$\widetilde{J}_\mu(i;\underline{\theta}) = \sum_{k=1}^{r} \theta_k\phi_k(i) = \underline{\theta}^T\underline{\phi}(i). \qquad (1)$$

Once an approximation architecture is selected, the values of the tunable parameters $\underline{\theta}$ can be chosen in many ways [1, Vol II, Chapter 6]. So-called *direct* or *simulation-based* methods use simulation of state trajectories and the resulting trajectory costs to update the parameters. Another approach, known as *Bellman error* methods [2, Chapter 6, Section 10], attempt to choose the parameters by minimizing the Bellman error over a set of sample states $\mathscr{S}_s$:

$$\min_{\underline{\theta}} \sum_{i\in\mathscr{S}_s} \left(\widetilde{J}_\mu(i;\underline{\theta}) - \left(g(i,\mu(i)) + \alpha\sum_{j\in\mathscr{S}} P_{ij}(\mu(i))\widetilde{J}_\mu(j;\underline{\theta})\right)\right)^2.$$

A. Motivation for Support Vector Techniques

Unfortunately, there are several difficulties associated with both the choice of approximation architecture and the method used to select the parameters. In both the neural network and basis function approximation architectures, the designer must trade off the expressiveness of the architecture (which may be thought of as the set of functions it can reproduce perfectly) with the need to maintain computational tractability. The former consideration pushes the design toward architectures with a large number of parameters. Indeed, a lookup table approximation architecture with $n=|\mathscr{S}|$ parameters could

B. Bethke is a PhD Candidate, Dept. of Aeronautics and Astronautics, MIT, Cambridge, MA 02139, USA, bbethke@mit.edu

J. How is a Professor in the Dept. of Aeronautics and Astronautics, Massachusetts Institute of Technology, jhow@mit.edu

A. Ozdaglar is an Associate Professor in the Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, asuman@mit.edu

represent any arbitrary function $J_\mu(.)$ perfectly by simply storing the value of the function at each of the n states. Computational considerations require the use of fewer parameters, since implementing any architecture with a large number of parameters renders the resulting computations difficult to perform.

In addition to the issue regarding the number of parameters, each of the approximation architectures discussed so far presents its own unique difficulties. In the case of neural networks, the optimization problem that must be solved in the training process is nonconvex, which may lead to implementation difficulties. In addition, there can be ambiguity in the proper choice of the topology of the network (e.g., how many hidden layers to choose, how many nodes). Typically, experimentation is necessary to properly "tune" a neural network to a particular problem [8], [9]. In a basis function architecture, the set of basis functions themselves must be chosen, which may be difficult to do unless the designer has some prior knowledge about the structure of the true cost-to-go $J_\mu(.)$ [10]. If a poor set of basis functions is chosen, the architecture may be unable to approximate $J_\mu(.)$ well even if the number of functions is large.

Furthermore, simulation-based methods for updating the parameters of the chosen approximation architecture suffer from the *simulation noise* problem [2, Chapter 6]. This problem arises because of the need to sample state trajectories from the Markov chain associated with a given policy μ. Simulation noise refers to the fact that there is randomness in the states and associated costs that will be observed in a given trajectory, which may ultimately lead to inaccurate estimates of the cost-to-go function. In practice, it may be necessary to simulate a very large number of trajectories, leading to increased computation time, in order to gain confidence that the resulting cost estimates are accurate. In contrast, Bellman error methods avoid the use of simulations by working directly with the Bellman equation. However, like simulation-based methods, they still suffer from the difficulties associated with the choice of approximation architecture discussed above.

To address these issues, we propose a different approximation architecture based on the idea of support vector regression (SVR) [8], [11]. SVR is similar in form to the basis function approach in that the approximation is given by a linear combination of basis functions (also called *features* in this context) as in Eq. (1). However, we will demonstrate that SVR has a number of advantages over both neural network and basis function architectures in the approximate dynamic programming problem, including:

- Being a kernel-based method, SVR can handle very large, possibly infinite, numbers of basis functions in a computationally tractable way. This makes the SVR architecture very expressive, yet practical to apply from a computational standpoint.
- The SVR solution is calculated via a convex quadratic program which has a unique optimal solution (contrast this with the nonconvex problem that arises in training a neural network).
- The difficulties of choosing network topologies, activation functions, and basis function sets are eliminated. Instead, the designer needs to select an appropriately powerful kernel (which *implicitly* specifies the basic function set). Many powerful yet easily computable kernels are known [12, Chapter 4].

The simplest use of the SVR architecture employs a simulation-based method to choose the parameters of the architecture. In this paper, we show that it is also possible to develop an SVR-based method that is similar in spirit to traditional Bellman error methods. This SVR-based method retains the advantage possessed by traditional Bellman error methods of not requiring trajectory simulations, while eliminating the difficulties associated with the choice of approximation architecture.

This paper presents this SVR-based method for approximate policy evaluation and proves its correctness in the limit of sampling the entire state space. The method is shown to be computationally efficient, and a full approximate policy iteration algorithm based on it is given. Finally, computational results are provided that show that the algorithm converges quickly to a near-optimal policy in two test problems.

II. Support Vector Regression Basics

This section provides a basic overview of support vector regression; for more details, see [8]. The objective of the SVR problem is to learn a function

$$f(x) = \sum_{k=1}^{r} \theta_k \phi_k(x) = \underline{\theta}^T \underline{\phi}(x)$$

that gives a good approximation of a given set of training data $\{(x_1,y_1),\ldots,(x_n,y_n)\}$ where $x_i \in \mathbb{R}^m$ is the input data and $y_i \in \mathbb{R}$ is the observed output. Note that the functional form assumed for $f(x)$ is identical to Eq. (1). The training problem is posed as the following quadratic optimization problem:

$$\min_{\underline{\theta},\underline{\xi}} \quad \frac{1}{2}||\underline{\theta}||^2 + c\sum_{i=1}^{n}(\xi_i+\xi_i^\star) \tag{2}$$

$$s.t. \quad y_i - \underline{\theta}^T\underline{\phi}(x_i) \leq \varepsilon+\xi_i \tag{3}$$

$$-y_i + \underline{\theta}^T\underline{\phi}(x_i) \leq \varepsilon+\xi_i^\star \tag{4}$$

$$\xi_i, \xi_i^\star \geq 0 \qquad \forall i \in \{1,\ldots,n\}.$$

Here, the regularization term $\frac{1}{2}||\underline{\theta}||^2$ penalizes model complexity, and the $\xi_i, \xi_i^\star$ are slack variables which are active whenever a training point y_i lies farther than a distance ε from the approximating function $f(x_i)$. The parameter c trades off model complexity with accuracy of fitting the observed training data. As c increases, any data points for which the slack variables are active incur higher cost, so the optimization problem tends to fit the data more closely (note that fitting too closely may not be desired if the training data is noisy).

The minimization problem [Eq. (2)] is difficult to solve when the number of basis functions r is large, for two reasons. First, it is computationally demanding to compute the values of all r basis functions for each of the data points. Second, the number of decision variables in the problem is

r (since there is one θ_i for each basis function $\phi_i(.)$), so the minimization must be carried out in an r-dimensional space. To address these issues, one can solve the primal problem through its dual, which can be formulated by computing the Lagrangian and minimizing with respect to the primal variables $\underline{\theta}$ and $\xi_i, \xi_i^\star$ (again, for more details, see [8]). The dual problem is

$$\begin{aligned} \max_{\underline{\lambda},\underline{\lambda}^\star} \quad & -\frac{1}{2}\sum_{i,i'=1}^{n}(\lambda_i^\star-\lambda_i)(\lambda_{i'}^\star-\lambda_{i'})\underline{\phi}(x_i)^T\underline{\phi}(x_{i'}) \\ & -\varepsilon\sum_{i=1}^{n}(\lambda_i^\star+\lambda_i)+\sum_{i=1}^{n}y_i(\lambda_i^\star-\lambda_i) \qquad (5) \\ s.t. \quad & 0\le\lambda_i,\lambda_i^\star\le c \quad \forall i\in\{1,\ldots,n\}. \end{aligned}$$

Note that the basis function vectors $\underline{\phi}(x_i)$ now appear only as inner products. This is important, because in many cases a *kernel function* $K(x_i,x_{i'})=\underline{\phi}(x_i)^T\underline{\phi}(x_{i'})$ can be defined whose evaluation avoids the need to explicitly calculate the vectors $\underline{\phi}(x_i)$, resulting in significant computational savings. Also, the dimensionality of the dual problem is reduced to only $2n$ decision variables, since there is one λ_i and one $\lambda_i^\star$ for each of the data points. Again, when the number of basis functions is large, this results in significant computational savings. Furthermore, it is well known that the dual problem can be solved efficiently using techniques such as Sequential Minimal Optimization (SMO) [13], [14]. Many libraries such as libSVM [15] implement SMO to solve the SVR problem (i.e. find the values of the dual variables $\underline{\lambda},\underline{\lambda}^\star$). Once the dual variables are known, the function $f(x)$ can be computed using the so-called *support vector expansion*:

$$f(x)=\sum_{i=1}^{n}(\lambda_i-\lambda_i^\star)\underline{\phi}(x_i)^T\underline{\phi}(x)=\sum_{i=1}^{n}(\lambda_i-\lambda_i^\star)K(x_i,x).$$

III. SVR-Based Policy Evaluation

Given the advantages outlined above of an SVR-based cost approximation architecture, this section now presents a progression of ideas on how to incorporate SVR into the approximate policy evaluation problem.

A. Formulation 1: Simulation-based Approach

The first formulation is similar in spirit to simulation-based methods such as TD(λ) [16] and LSPE(λ) [6]. In particular, a number of sample states $\mathscr{S}_s$ are chosen, and many trajectories are simulated starting at these states. The costs for each state i are averaged over the trajectories to give an approximation $\hat{J}_i$ to the cost-to-go, the training data $\{(i,\hat{J}_i)|i\in\mathscr{S}_s\}$ are taken as the input to the SVR training problem, and an approximating function $\widetilde{J}_\mu(i;\underline{\theta})$ is calculated by solving the basic SVR problem.

B. Formulation 2: Simulation-Free Approach Using Bellman Error

The approach of Formulation 1 leverages the power of the SVR architecture and allows the use of high-dimensional feature vectors in a computationally efficient way, which is an attractive advantage. However, it still requires simulation of multiple trajectories per state, which may be time-consuming and also introduces undesirable simulation noise into the problem.

Fortunately, the SVR-based approach can be reformulated to eliminate the need for trajectory simulation. To do this, first recall the standard Bellman equation for evaluating a given policy μ exactly. Writing the exact solution for the cost-to-go as a vector $\underline{J}_\mu\in\mathbb{R}^n$, where $n=|\mathscr{S}|$ is the size of the state space, the Bellman equation is

$$\underline{J}_\mu=T_\mu\underline{J}_\mu=\underline{g}+\alpha P^\mu\underline{J}_\mu, \qquad (6)$$

where T_μ is the fixed-policy dynamic programming operator [1],

$$P_{ij}^\mu\equiv P_{ij}(\mu(i))$$

is the probability transition matrix associated with μ, and

$$g_i\equiv\sum_{j\in\mathscr{S}}P_{ij}^\mu g(i,\mu(i),j) \qquad (7)$$

is the expected single stage cost of the policy μ starting from state i. The Bellman equation given by Eq. (6) is a linear system in $\underline{J}_\mu$, and can be solved exactly:

$$\underline{J}_\mu=(I-\alpha P^\mu)^{-1}\underline{g}. \qquad (8)$$

In practice, of course, the size of the state space is much too large to admit solving the Bellman equation exactly, which is why approximation methods must be used. Note, however, that the Bellman equation still provides useful information even in the approximate setting. In particular, a candidate approximation function $\widetilde{J}_\mu(.)$ should be "close" to satisfying the Bellman equation if it is truly a good approximation. To quantify this notion, define the Bellman error $BE(i)$ as

$$BE(i)\equiv\widetilde{J}_\mu(i)-(g_i+\alpha\sum_{j\in\mathscr{S}}P_{ij}^\mu\widetilde{J}_\mu(j)), \qquad (9)$$

which is a measure of how well the approximation function $\widetilde{J}_\mu(i)$ solves Eq. (6), and therefore, how well the approximation reflects the true cost. Note that $BE(i)$ is a function of the state i, and ideally, $|BE(i)|$ should be as small as possible for as many states as possible. In particular, note that if $BE(i)=0$ for *every* state, then by definition $\widetilde{J}_\mu(i)=J_\mu(i)$ at every state; that is, $\widetilde{J}_\mu(i)$ is exact.

Recall that the SVR problem seeks to minimize the absolute value of some error function (mathematically, this is stated as the constraints Eqs. (3) and (4) in the optimization problem). In Formulation 1, the error function is $(\hat{J}_i-\widetilde{J}_\mu(i;\underline{\theta}))$; that is, the difference between the simulated cost values and the approximation. Note that this formulation is an indirect approach which generates an approximation function $\widetilde{J}_\mu(.)$, which in turn hopefully keeps the Bellman error small at many states. The key idea behind Formulation 2 is that *the SVR optimization problem can be modified to minimize the Bellman error directly, without the need to simulate trajectories*. The change to the optimization problem is simple: the Bellman error is substituted [Eq. (9)] as the error

function in the normal SVR problem:

$$\begin{array}{rrcll} \min\limits_{\underline{\theta},\underline{\xi}} & \frac{1}{2}||\underline{\theta}||^2 & + & c\sum\limits_{i\in\mathscr{S}_s}(\xi_i+\xi_i^\star) & \\ s.t. & BE(i) & \leq & \varepsilon+\xi_i & \\ & -BE(i) & \leq & \varepsilon+\xi_i^\star & \\ & \xi_i,\xi_i^\star & \geq & 0 & \forall i\in\mathscr{S}_s. \end{array}$$

This optimization problem is referred to as the *Bellman error optimization problem*. Note that by substituting the functional form of $\widetilde{J}_\mu$ [Eq. (1)] into Eq. (9), the Bellman error can be written

$$\begin{aligned} BE(i) &= \underline{\theta}^T\underline{\phi}(i)-(g_i+\alpha\sum_{j\in\mathscr{S}}P_{ij}^\mu(\underline{\theta}^T\underline{\phi}(j))) \\ &= -g_i+\underline{\theta}^T\underline{\psi}(i), \end{aligned}$$

where

$$\underline{\psi}(i)\equiv\underline{\phi}(i)-\alpha\sum_{j\in\mathscr{S}}P_{ij}^\mu\underline{\phi}(j). \tag{10}$$

Therefore, the Bellman error optimization problem can be rewritten as

$$\begin{array}{rrcll} \min\limits_{\underline{\theta},\underline{\xi}} & \frac{1}{2}||\underline{\theta}||^2 & + & c\sum\limits_{i\in\mathscr{S}_s}(\xi_i+\xi_i^\star) & (11) \\ s.t. & -g_i+\underline{\theta}^T\underline{\psi}(i) & \leq & \varepsilon+\xi_i & (12) \\ & g_i-\underline{\theta}^T\underline{\psi}(i) & \leq & \varepsilon+\xi_i^\star & (13) \\ & \xi_i,\xi_i^\star & \geq & 0 \qquad \forall i\in\mathscr{S}_s. & \end{array}$$

This problem is identical in form to the basic support vector regression problem [Eq. (2)]. The only difference is that the feature vector $\underline{\phi}(i)$ has been replaced by a new feature vector $\underline{\psi}(i)$; this amounts to simply using a different set of basis functions and does not alter the structure of the basic problem in any way. Therefore, the dual problem is given by Eq. (5) with the substitution $\underline{\phi}(i)\rightarrow\underline{\psi}(i)$:

$$\begin{aligned} \max_{\underline{\lambda},\underline{\lambda}^\star} \quad & -\frac{1}{2}\sum_{i,i'=1}^{n}(\lambda_i^\star-\lambda_i)(\lambda_{i'}^\star-\lambda_{i'})\underline{\psi}(i)^T\underline{\psi}(i') \\ & -\varepsilon\sum_{i=1}^{n}(\lambda_i^\star+\lambda_i)+\sum_{i=1}^{n}g_i(\lambda_i^\star-\lambda_i) \\ & s.t. \quad 0\leq\lambda_i,\lambda_i^\star\leq c \quad \forall i\in\mathscr{S}_s. \end{aligned}$$

In order to solve the dual problem, the g_i values and the kernel values $\mathscr{K}(i,i')=\underline{\psi}(i)^T\underline{\psi}(i')$ must be computed. The g_i values are given by Eq. (7), and a simple calculation using Eq. (10) yields a formula for $\mathscr{K}(i,i')$:

$$\begin{aligned} \mathscr{K}(i,i') = K(i,i') &- \alpha\sum_{j\in\mathscr{S}}\left(P_{i'j}^\mu K(i,j)+P_{ij}^\mu K(i',j)\right) \\ &+\alpha^2\sum_{j,j'\in\mathscr{S}}P_{ij}^\mu P_{i'j'}^\mu K(j,j'). \end{aligned} \tag{14}$$

Here, $K(i,i')=\underline{\phi}(i)^T\underline{\phi}(i')$ is the kernel function corresponding to the feature vectors $\underline{\phi}(i)$. Note that, similar to the basic SVR problem, significant computational savings can be achieved by using a closed-form kernel function which avoids the need to explicitly evaluate the feature vectors $\underline{\phi}(i)$.

Once the kernel values $\mathscr{K}(i,i')$ and the cost values g_i are computed, the dual problem is completely specified and can be solved (e.g., using a standard SVM solving package such as libSVM [15]), yielding the dual solution variables $\underline{\lambda}$. Again, in direct analogy with the normal SV regression problem, the primal variables $\underline{\theta}$ are given by the relation

$$\underline{\theta}=\sum_{i\in\mathscr{S}_s}(\lambda_i-\lambda_i^\star)\underline{\psi}(i).$$

Substituting this expression for $\underline{\theta}$ into Eq. (1) yields

$$\begin{aligned} \widetilde{J}_\mu(k) &= \underline{\theta}^T\underline{\phi}(k) \\ &= \sum_{i\in\mathscr{S}_s}(\lambda_i-\lambda_i^\star)\left(K(i,k)-\alpha\sum_{j\in\mathscr{S}}P_{ij}^\mu K(j,k)\right). \end{aligned} \tag{15}$$

Thus, once the dual variables $\underline{\lambda}$ are known, Eq. (15) can be used to calculate $\widetilde{J}_\mu(k)$.

C. Formulation 3: Forcing the Bellman Error to Zero

Formulation 2 performs approximate policy evaluation without having to resort to trajectory simulation, thereby eliminating simulation noise. A final reformulation is possible by considering the settings of the SVR parameters c and ε. Recall that, in the standard SVR problem, the choice of c and ε are related to how noisy the input data is. In a simulation-based method, simulation noise introduced into the problem must be considered. The situation in Formulation 2, however, is fundamentally different: *there is no noise in the problem*. That is, instead of observing noisy simulation data generated by sampling the underlying random Markov chain associated with the policy μ, we are working directly with the Bellman equation, which is an exact description of a mathematical relation between the cost-to-go values $J_\mu(i)$ for all states $i\in\mathscr{S}$. Therefore, it is reasonable to explicitly require that the Bellman error be exactly zero at the sampled states, which can be accomplished by setting $\varepsilon=0$ and $c=\infty$.

Having fixed c and ε, the optimization problem of Formulation 2 [Eq. (11)] can be recast in a simpler form. With $c=\infty$, the optimal solution must have $\xi,\xi^\star=0$ for all i, since otherwise the objective function will be unbounded. Therefore, any feasible solution must have $\xi,\xi^\star=0$. Furthermore, if ε is also zero, then the constraints [Eqs. (12) and (13)] become equalities. With these modifications, the primal problem reduces to

$$\begin{array}{rll} \min\limits_{\underline{\theta}} & \frac{1}{2}||\underline{\theta}||^2 & (16) \\ s.t. & g_i-\underline{\theta}^T\underline{\psi}(i)=0 \qquad \forall i\in\mathscr{S}_s, & \end{array}$$

where $\underline{\psi}(i)$ is given by Eq. (10). The Lagrange dual of this optimization problem is easily calculated. The Lagrangian is

$$\mathscr{L}(\underline{\theta},\underline{\lambda})=\frac{1}{2}||\underline{\theta}||^2+\sum_{i\in\mathscr{S}_s}\lambda_i\left(g_i-\underline{\theta}^T\underline{\psi}(i)\right). \tag{17}$$

Maximizing $\mathscr{L}(\underline{\theta},\underline{\lambda})$ with respect to $\underline{\theta}$ can be accomplished by setting the corresponding partial derivative to zero:

$$\frac{\partial\mathscr{L}}{\partial\underline{\theta}}=\underline{\theta}-\sum_i\lambda_i\underline{\psi}(i)=0,$$

and therefore

$$\underline{\theta} = \sum_i \lambda_i \underline{\psi}(i). \tag{18}$$

Thus, the approximation $\widetilde{J}_\mu(i)$ is given by

$$\begin{aligned} \widetilde{J}_\mu(k) &= \underline{\theta}^T \underline{\phi}(k) \\ &= \sum_{i \in \mathscr{S}_s} \lambda_i \left(K(i,k) - \alpha \sum_{j \in \mathscr{S}} P_{ij}^\mu K(j,k) \right). \end{aligned} \tag{19}$$

Finally, substituting Eq. (18) into Eq. (17) and maximizing with respect to all λ_i variables gives the dual problem:

$$\max_{\underline{\lambda} \in \mathbb{R}^n} \quad -\frac{1}{2} \sum_{i,i' \in \mathscr{S}_s} \lambda_i \lambda_{i'} \underline{\psi}(i)^T \underline{\psi}(i') + \sum_i \lambda_i g_i.$$

This can also be written in vector form as

$$\max_{\underline{\lambda} \in \mathbb{R}^n} \quad -\frac{1}{2} \underline{\lambda}^T \mathbb{K} \underline{\lambda} + \underline{\lambda}^T \underline{g}, \tag{20}$$

where $\mathbb{K}_{ii'} = \underline{\psi}(i)^T \underline{\psi}(i') = \mathscr{K}(i,i')$ is the Gram matrix of the kernel $\mathscr{K}(.,.)$ (which is calculated using Eq. (14)). Note that the problem is a simple, unconstrained maximization of a quadratic form which has a unique maximum since $\mathbb{K}$ is positive definite. The solution is found analytically by setting the derivative of the objective with respect to $\underline{\lambda}$ to zero, which yields

$$\mathbb{K} \underline{\lambda} = \underline{g}. \tag{21}$$

Note the important fact that the dimension of this linear system is $n_s = |\mathscr{S}_s|$, the number of sampled states. Recall that the original problem of calculating the exact solution $\underline{J}_\mu$ [Eq. (8)] involved solving a linear system in n variables, where n is the size of the entire state space (which, by assumption, is very large and makes the solution of the original system difficult to find). After developing a support vector regression-based method for approximating the true cost-to-go and reformulating it based on parameter selection observations, the approximation problem has been reduced to a problem of solving another linear system [Eq. (21)]. However, this time the system is in $n_s = |\mathscr{S}_s|$ variables, where $n_s \ll n$. Furthermore, since the designer is in control of $\mathscr{S}_s$, they can select the number of sample states based on the computational resources available.

IV. Support Vector Policy Iteration Algorithm

The preceding section showed how to construct a cost-to-go approximation $\widetilde{J}_\mu(.)$ of a fixed policy μ. This section now presents a full support vector policy iteration algorithm.

Step 1 (Preliminary) Choose a kernel function $K(i,i')$ defined on $\mathscr{S} \times \mathscr{S}$.

Step 2 (Preliminary) Select a subset of states $\mathscr{S}_s \subset \mathscr{S}$ to sample. The cardinality of $\mathscr{S}_s$ should be based on the computational resources available but will certainly be much smaller than $|\mathscr{S}|$ for a large problem.

Step 3 (Preliminary) Select an initial policy μ_0.

Step 4 (Policy evaluation) Given the current policy μ_k, calculate the kernel Gram matrix $\mathbb{K}$ for all $i,i' \in \mathscr{S}_s$ using Eq. (14). Also calculate the cost values g_i using Eq. (7).

Step 5 (Policy evaluation) Using the values calculated in Step 4, solve the linear system [Eq. (21)] for $\underline{\lambda}$.

Step 6 (Policy evaluation) Using the dual solution variables $\underline{\lambda}$, construct the cost-to-go approximation $\widetilde{J}_{\mu_k}(i)$ using Eq. (19).

Step 7 (Policy improvement) Using the cost-to-go $\widetilde{J}_{\mu_k}(i)$ found in Step 6, calculate the one-step policy improvement $T_{\mu_{k+1}} \widetilde{J}_{\mu_k} = T \widetilde{J}_{\mu_k}$:

$$\mu_{k+1}(i) = \arg\min_u \sum_{j \in \mathscr{S}} P_{ij}(u) \left(g(i,u) + \alpha \widetilde{J}_{\mu_k}(j) \right)$$

Step 8 Set the current policy $\mu = \mu_{k+1}$ and go back to Step 4.

The computational complexity of the algorithm is dominated by Step 5 (solving the linear system), which takes $\mathscr{O}(|\mathscr{S}_s|^3)$ operations. In comparison, finding the exact cost-to-go using Eq. (8) takes $\mathscr{O}(|\mathscr{S}|^3)$ operations. Since $|\mathscr{S}_s| \ll |\mathscr{S}|$, the algorithm can dramatically reduce the time required to compute a solution to the MDP.

V. Computational Results

The support vector policy iteration algorithm was implemented on the well-known "mountain car problem" [17], [18] to evaluate its performance. In this problem, a unit mass, frictionless car moves along a hilly landscape whose height $H(x)$ is described by

$$H(x) = \begin{cases} x^2 + x & \text{if } x < 0 \\ \frac{x}{\sqrt{1+5x^2}} & \text{if } x \geq 0 \end{cases}$$

The system state is given by $(x, \dot{x})$ (the position and speed of the car). A horizontal control force $-4 \leq u \leq 4$ can be applied to the car, and the goal is to drive the car from its starting location $x = -0.5$ to the "parking area" $0.5 \leq x \leq 0.7$ as quickly as possible. The problem is challenging because the car is underpowered: it cannot simply drive up the steep slope. Rather, it must use the features of the landscape to build momentum and eventually escape the steep valley centered at $x = -0.5$. The system response under the optimal policy (computed using value iteration) is shown as the dashed line in Figure 1; notice that the car initially moves *away* from the parking area before reaching it at time $t = 14$.

In order to apply the support vector policy iteration algorithm, an evenly spaced 9x9 grid of sample states,

$$\begin{aligned} \mathscr{S}_s = \{(x,\dot{x}) \mid\ & x = -1.0, -0.75, \ldots, 0.75, 1.0 \\ & \dot{x} = -2.0, -1.5, \ldots, 1.5, 2.0\} \end{aligned}$$

was chosen. Furthermore, a squared exponential kernel

$$K((x_1,\dot{x}_1),(x_2,\dot{x}_2)) = \exp\left(-(x_1 - x_2)^2 - (\dot{x}_1 - \dot{x}_2)^2\right)$$

was used. The algorithm was executed, resulting in a sequence of policies (and associated cost functions) that converged after three iterations. The sequence of cost functions is shown in Figure 2 along with the optimal cost function

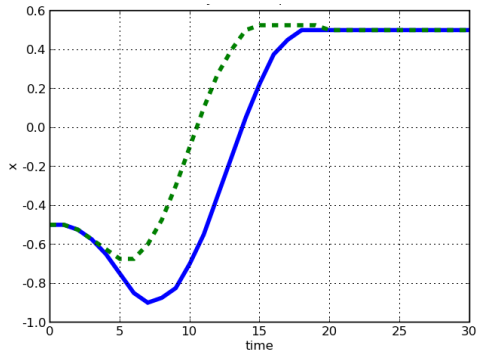

Fig. 1. System response under the optimal policy (dashed line) and the policy learned by the support vector policy iteration algorithm (solid line).

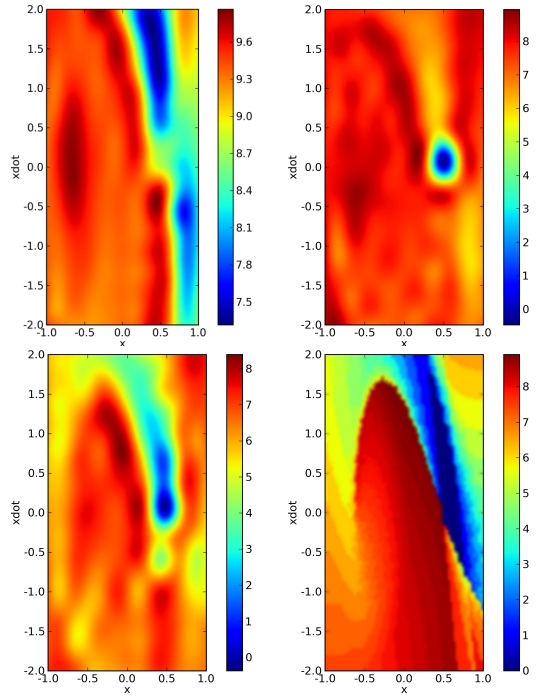

Fig. 2. Approximate cost-to-go for iterations 1 (upper left), 2 (upper right), and 3 (lower left); exact cost-to-go (lower right).

for comparison. Of course, the main objective is to learn a policy that is similar to the optimal one. The solid line in Figure 1 shows the system response under the approximate policy generated by the algorithm after 3 iterations. Notice that the qualitative behavior is the same as the optimal policy; that is, the car first accelerates away from the parking area to gain momentum. The approximate policy arrives at the parking area at $t = 17$, only 3 time steps slower than the optimal policy.

VI. Conclusion

This paper has presented an approximate policy iteration algorithm based on support vector regression that avoids some of the difficulties encountered in other approximate dynamic programming methods. In particular, the approximation architecture used in this algorithm is kernel-based, allowing it to implicitly work with a very large set of basis functions. This is a significant advantage over other methods, which encounter computational difficulties as the number of basis functions increases. In addition, the architecture is "trained" by solving a simple, convex optimization problem (unlike, for example, the neural network training process). Furthermore, the support vector policy iteration algorithm avoids simulations and the associated simulation noise problem by minimizing the Bellman error of the approximation directly. Although not discussed here, it is possible to prove that the algorithm has the attractive theoretical property of reducing to exact policy iteration in the limit of sampling the entire state space.

Computational results of implementing the algorithm on a classic reinforcement learning problem indicate that it yields a high-quality policy after a small number of iterations.

Acknowledgments

Research supported in part by the Boeing Company under the guidance of Dr. John Vian at the Boeing Phantom Works, Seattle and by AFOSR grant FA9550-04-1-0458. The first author is also supported by the Hertz Foundation and the American Society for Engineering Education.

References

[1] D. Bertsekas, *Dynamic Programming and Optimal Control*. Belmont, MA: Athena Scientific, 2007.

[2] D. Bertsekas, J. Tsitsiklis, *Neuro-Dynamic Programming*. Belmont, MA: Athena Scientific, 1996.

[3] G. Tesauro, "Temporal difference learning and TD-Gammon," *Commun. ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[4] M. Lagoudakis and R. Parr, "Least-squares policy iteration," *Journal of Machine Learning Research*, vol. 4, pp. 1107–1149, 2003.

[5] A. B. W. P. J. Si and D. Wunsch, *Learning and Approximate Dynamic Programming*. NY: IEEE Press, 2004. [Online]. Available: http://citeseer.ist.psu.edu/651143.html

[6] D. Bertsekas and S. Ioffe, "Temporal Differences-Based Policy Iteration and Applications in Neuro-Dynamic Programming," http://web.mit.edu/people/dimitrib/Tempdif.pdf, 1996.

[7] M. Valenti, "Approximate Dynamic Programming with Applications in Multi-Agent Systems," Ph.D. dissertation, Massachusetts Institute of Technology, 2007.

[8] A. Smola, B. Scholkopf, "A Tutorial on Support Vector Regression," *Statistics and Computing*, vol. 14, pp. 199–222, 2004.

[9] B. Curry and P. Morgan, "Model selection in neural networks: Some difficulties," *European Journal of Operational Research*, vol. Volume 170, Issue 2, pp. p.567–577, 2006.

[10] R. Patrascu, "Linear approximations for factored markov decision processes," PhD Dissertation, University of Waterloo, Department of Computer Science, February 2004.

[11] A. S. B. Scholkopf, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, 2002.

[12] C. Rasmussen and C. Williams, *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, MA, 2006.

[13] J. Platt, "Using sparseness and analytic QP to speed training of support vector machines," in *Advances in Neural Information Processing Systems*, 1999, pp. 557–563.

[14] S. Keerthi, S. Shevade, C. Bhattacharyya, and K. Murthy, "Improvements to Platt's SMO algorithm for SVM classifier design," http://citeseer.ist.psu.edu/244558.html, 1999.

[15] C.-C. Chang and C.-J. Lin, *LIBSVM: a library for support vector machines*, 2001, Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

[16] R.S. Sutton, "Learning to Predict by the Methods of Temporal Differences," *Machine Learning*, vol. 3, pp. 9–44, 1988.

[17] R. Sutton, A. Barto, *Reinforcement learning: An introduction*. MIT Press, 1998.

[18] C. Rasmussen and M. Kuss, "Gaussian processes in reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 16, pp. 751–759, 2004.