Experiments in Value Function Approximation with Sparse Support Vector Regression

Tobias Jung and Thomas Uthmann

Institut für Informatik
Johannes Gutenberg-Universität
55099 Mainz, Germany
{tjung,uthmann}@informatik.uni-mainz.de

Abstract. We present first experiments using Support Vector Regression as function approximator for an on-line, *sarsa*-like reinforcement learner. To overcome the batch nature of SVR two ideas are employed. The first is sparse greedy approximation: the data is projected onto the subspace spanned by only a small subset of the original data (in feature space). This subset can be built up in an on-line fashion. Second, we use the sparsified data to solve a reduced quadratic problem, where the number of variables is independent of the total number of training samples seen. The feasability of this approach is demonstrated on two common toy-problems.

1 Introduction

One central approach to solve reinforcement learning (RL) problems relies on estimating value functions either from simulation-based experience or model-based search (e.g. [10]). For problems with large or continuous state spaces it becomes necessary to represent the value function by some form of function approximator in order to generalize estimated values from previously visited states to similar but never seen before ones. Although many different approximation architectures are possible – both parametric (e.g. Neural Networks, RBF-Networks, tile coding) and non-parametric (e.g. locally weighted regression) have been tried to a varying degree of success – value function approximation remains one of the key obstacles in scaling RL to high-dimensional control tasks.

Support Vector Machines (SVM) could represent a powerful alternative: in principle unharmed by the dimensionality, trained by solving a well defined optimization problem, and with generalization capabilities presumably superior to local instance-based methods. The goal of our ongoing work reported herein is to utilize SV-Regression to approximate the value function in common RL algorithms. Related results are limited to off-line learning: in [3] SVR has been applied along the lines of approximate value iteration. Inputs to the SVR were the real targets, solved by value iteration on a subset of states, and presented in batches to the approximator. However, for the on-line case this has not been attempted before, as far as we are aware.

J.-F. Boulicaut et al. (Eds.): ECML 2004, LNAI 3201, pp. 180–191, 2004.

A reason could be that value function approximation is plagued by some peculiar characteristics that do not agree very well with the instance-based nature of SVR. We are facing the following two problems: First, RL proceeds iteratively. Thus training samples arrive one at a time from an endless stream which is generated while the learning agent interacts with its environment. Second, many RL algorithms usually rely on bootstrapped samples that are continually updated with new estimates. Therefore we have to track a highly non-stationary target function and need to access and modify the stored training samples. Both of these points pose no problems for parametric function approximators, since their gradient based training deals naturally with incremental learning. Non-parametric function approximators, however, are memory-based and must explicitly remember all training data. Either adding a new example or updating an old one both amount to a change to the stored training samples. In this case, however, learning cannot be done incrementally so that we have to re-train the SVR every time-step anew. Moreover, a SVR is usually solved by a constrained quadratic optimization problem where the number of variables equals the number of training examples. Clearly, a constantly increasing number would sooner or later render the computational cost completely prohibitive. In order to keep this problem tractable it is necessary to keep this number from growing linearly in time.

Fortunately, we do not have to remember every training sample. As was pointed out in [4] one can eliminate those from the final expansion that are linear dependant (in feature space) without changing the obtained solution. Thus, it is reasonable to store only a few selected examples in a reduced dictionary. Examples of how to arrive at sparse dictionaries can be found in [9, 12, 7], along with applications of how to benefit from it. In particular, in [6, 5] a sparse algorithm is presented that greedily adds examples to the reduced dictionary and is especially suited to on-line applications. Together with this sparsification procedure the authors propose in [6] an incremental SVR-algorithm where the solution is obtained by only considering the samples in the reduced dictionary. This way, the per-time-step complexity is limited by the intrinsic dimensionality of the data in the feature space instead of the ever increasing number of arriving training examples.

Here we will pursue a similar approach and apply their greedy sparsification method to obtain a function approximator suitable for RL. The remainder of the paper is structured as follows: Sect. 2 briefly reviews RL, Sect. 3 and Sect. 4 describe the modified sparsification method. In Sect. 5 everything is wrapped up in some experiments that demonstrate the feasability of this approach, and Sect. 6 concludes.

2 A Brief Reminder of Reinforcement Learning

Reinforcement learning belongs to a class of optimal control methods that are based on estimating value functions either from simulation-based experiences (Monte-Carlo or Temporal Difference branch) or model-based search (Dynamic

Programming branch), e.g. see [1, 10]. In this work we make the following assumptions: we consider the case of time-discrete control tasks with continuous states and discrete action set. Once the value function is known we assume that the learner has access to a model to derive the best course of actions.

The goal is to (approximately) compute the value function which is here defined to be the infinite horizon, discounted sum of rewards

$$V^{\pi}(s) = E^{\pi}\Big(\sum_{t=0}^{\infty} \gamma^t r_t \;\Big|\; s_0 = s \Big) \tag{1}$$

when starting in state s and choosing actions according to policy π. Here $\gamma \in [0, 1)$ denotes the discount factor and r_i the reward obtained while traversing from s_i to s_{i+1}. For a given policy π we can compute V^{π} by iteratively applying the TD(0) rule

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)] \tag{2}$$

where s' and r is the state and reward following s while executing π, and $\alpha \in [0, 1]$ is the step size. Applying (2) infinitely often to every state (assuming a finite state set) in arbitrary order is guaranteed to yield (1). Once the value function V^{π_k} is known for a fixed policy π_k (policy evaluation step) we can derive the greedy policy w.r.t. V^{π_k} to otain a new policy π_{k+1}

$$\pi_{k+1}(s) = \operatorname*{argmax}_a \sum_{s'} P^a(s, s')[R^a(s, s') + \gamma V^{\pi_k}(s')] \tag{3}$$

in the so called policy improvement step. Many popular RL algorithms adhere to the framework of alternately carrying out policy evaluation and policy improvement. Roughly speaking, they only vary the extent to which policy evaluation is executed before improving the current policy. Throughout this paper we will use an on-line variant thereof, see algorithm 1 below.

Parameter: ϵ, α, γ
Initialize: function approximator F
loop
  $s \leftarrow$ start state
  while s is not terminal **do**
    $v \leftarrow$ predict(F, s)
    $a \leftarrow$ policy-action(F, s) from (3) {ϵ-greedy}
    execute a, observe r, and s'
    if s' is terminal **then**
      $v \leftarrow v + \alpha[r - v]$
    else
      $v \leftarrow v + \alpha[r + \gamma$ predict$(F, s') - v]$
    learn(F, s, v)
    $s \leftarrow s'$

Algorithm 1. Model-based *sarsa* with instance-based function approximator

3 Sparse Support Vector Regression

3.1 The SVR Problem

We start by restating the standard formulation[1] of the SVR problem with ε-insensitive cost function [8]. Consider some training data $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_\ell, y_\ell)\}$, where $\mathbf{x}_i \in \mathbb{R}^n$ are the input-patterns and $y_i \in \mathbb{R}$ are the target outputs. The overall idea is to map the input-patterns into some high-dimensional feature space, and to perform linear regression in that feature space. Let Φ be the non-linear function that maps the input-patterns to this feature space. Considering functions $f(\cdot) = \langle \mathbf{w}, \Phi(\cdot) \rangle + b$ that are linearly parametrized in the feature space the objective is to find one that minimizes the regularized risk

$$R_{reg}(\mathbf{w}) = C\frac{1}{\ell}\sum_{i=1}^{\ell} |y_i - (\langle \mathbf{w}, \Phi(\mathbf{x}_i) \rangle + b)|_\varepsilon + \frac{1}{2}\|\mathbf{w}\|^2 \tag{4}$$

where C is a constant penalizing complexity. The ε-insensitive cost function $|\cdot|_\varepsilon \stackrel{\text{def}}{=} \max(0, |\cdot| - \varepsilon)$ only takes into account deviations larger than ε and enforces sparsity in the final regressor. It is known that the function minimizing (4) can be expressed solely in terms of the input-patterns (Representer Theorem) and has the form $f(\cdot) = \sum_{i=1}^{\ell}(\alpha_i^* - \alpha_i)\langle \Phi(\mathbf{x}_i), \Phi(\cdot) \rangle + b$. The coefficients α_i^*, α_i of the solution satisfy $\alpha_i^*, \alpha_i \geq 0$ and $\alpha_i^* \alpha_i = 0$. They are obtained by solving the constrained quadratic optimization problem (dual)

$$\begin{aligned} \max_{\boldsymbol{\alpha}^*, \boldsymbol{\alpha} \in \mathbb{R}^\ell} \; & -\tfrac{1}{2}\sum_{i,j=1}^{\ell}(\alpha_i^* - \alpha_i)(\alpha_j^* - \alpha_j)\langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle \\ & -\varepsilon \sum_{i=1}^{\ell}(\alpha_i^* + \alpha_i) + \sum_{i=1}^{\ell} y_i(\alpha_i^* - \alpha_i) \\ \text{s.t.} \quad & 0 \leq \alpha_i^*, \alpha_i \leq C \quad i = 1 \ldots \ell \\ & \sum_{i=1}^{\ell}(\alpha_i^* - \alpha_i) = 0 \end{aligned} \tag{5}$$

What is commonly referred to as 'kernel-trick' allows us to replace the inner product in feature space $\langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle$ by a Mercer kernel function $k(\mathbf{x}_i, \mathbf{x}_j)$. Thus once the solution to (5) has been worked out we obtain the final regressor

$$f(\mathbf{x}) = \sum_{i=1}^{\ell}(\alpha_i^* - \alpha_i)k(\mathbf{x}_i, \mathbf{x})$$

where usually many of the coefficients α_i^*, α_i vanish.

[1] Note that we stick to the standard notation of denoting the input with $\mathbf{x}_i$ and the output with y_i. Since SVR is used to approximate the value function, we are actually dealing with input-output pairs of the form $(\mathbf{s}, \tilde{V}(\mathbf{s}))$, with $\tilde{V}(\mathbf{s})$ being the new estimated target for the (vector-valued) state $\mathbf{s}$

3.2 Sparse Greedy Matrix Approximation

Unfortunately SVM methods were developed primarily with batch function approximation in mind. The setting of on-line learning differs considerable: here we assume that we have a (possible infinite) stream of incoming data. Recall that we wish to apply SVR as function approximator for *sarsa*-like on-line reinforcement learning (cf. Algorithm 1). At every step t we add the current state and its (re-) estimated value to the stored instances and thus end up with a training set consisting of *all* updates made to this point. At every time step we also need to make a new prediction that incorporates the previous changes. Hence we need to re-train the SVR once every single step. As there is generally no efficient way to get this done incrementally this task would be computationally completely infeasable: it would require that we solve the ever-increasing optimization problem (5) in t variables once every step[2]. Likewise the time to perform the prediction would also increase linearly in t.

Clearly the per-time-step complexity should not depend on the iteration count t. One way to deal with this problem are sparse greedy approximation techniques [6, 9]. The underlying idea is based on the observation that usually the mapped data $\Phi(\mathbf{x}_i)$ spans a subspace whose effective dimension is surprisingly low when compared to the dimension of the feature space. Therefore we can easily remove many of the input-patterns with negligible impact in the precision. This can be done by picking a subset (called 'dictionary' in [6]) of the input-patterns and representing the remaining patterns as linear combination in feature space. Thus after having chosen a dictionary of m inputs $\{\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_m\}$ we can approximate the remaining features $\{\Phi(\mathbf{x}_1), \ldots, \Phi(\mathbf{x}_\ell)\} \setminus \{\Phi(\tilde{\mathbf{x}}_1), \ldots, \Phi(\tilde{\mathbf{x}}_m)\}$ by

$$\Phi(\mathbf{x}_i) \approx \tilde{\Phi}(\mathbf{x}_i) := \sum_{j=1}^{m} a_{ij} \Phi(\tilde{\mathbf{x}}_j)$$

Since the approximation is carried out in feature space, we need to find coefficients a_{ij} such that the distance $d_i := \|\Phi(\mathbf{x}_i) - \tilde{\Phi}(\mathbf{x}_i)\|^2$ is minimized. Remembering to replace the inner products with kernels, this is equivalent to

$$\min_{\mathbf{a}_i \in \mathbf{R}^m} k(\mathbf{x}_i, \mathbf{x}_i) - 2 \sum_{j=1}^{m} a_{ij} k(\tilde{\mathbf{x}}_j, \mathbf{x}_i) + \sum_{j,l=1}^{m} a_{ij} a_{il} k(\tilde{\mathbf{x}}_j, \tilde{\mathbf{x}}_l) \tag{6}$$

or written in more convenient matrix notation $\min_{\mathbf{a}_i} \{k_{ii} - 2\mathbf{a}_i^T \tilde{\mathbf{k}}_\mathbf{i} + \mathbf{a}_i^T \tilde{K} \mathbf{a}_i\}$ where $[\tilde{K}]_{ij} = k(\tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_j)$, $k_{ii} = k(\mathbf{x}_i, \mathbf{x}_i)$, and $\tilde{\mathbf{k}}_i = (k(\tilde{\mathbf{x}}_1, \mathbf{x}_i), \ldots, k(\tilde{\mathbf{x}}_m, \mathbf{x}_i))^T$. Setting the derivates to zero we find that the minimizer of (6) is given by

$$\mathbf{a}_i = \tilde{K}^{-1} \tilde{\mathbf{k}}_i \tag{7}$$

Once we have determined $\mathbf{a}_i$ for input-pattern $\mathbf{x}_i$ we simultaneously obtain

$$d_i = k_{ii} - \tilde{\mathbf{k}}_i^T \mathbf{a}_i \tag{8}$$

[2] In everything but the most trivial problem the step count easily ranges in the 100,000s and more

as the distance to the span of the dictionary. Using this byproduct as a criterion to judge whether or not the current dictionary approximates a given input well enough, the authors in [6] turn this procedure into an on-line algorithm. Imagine running sequentially through the full training set, one can add the current $\mathbf{x}_i$ to the dictionary whenever the corresponding d_i is worse than some tolerance `TOL2`.

3.3 Solving a Reduced Problem

Let $[A]_{ij} \in \mathbb{R}^{\ell \times m}$ be the matrix consisting of rows $\mathbf{a}_i$ from (7) for the full training set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_\ell, y_\ell)\}$ and the fixed dictionary $\{\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_m\}$. Note that if $\mathbf{x_k}$ belongs to the dictionary then $\mathbf{a}_k$ is the kth unit vector $(0, \ldots, 1, 0 \ldots)^T$ since $\Phi(\mathbf{x}_k)$ is exactly representable by itself. This way we can use the reduced kernel matrix $\tilde{K} \in \mathbb{R}^{m \times m}$ and the transformation A to obtain a (low-rank) approximation to the full kernel matrix $K \in \mathbb{R}^{\ell \times \ell}$ with $K \approx A\tilde{K}A^T$.

Now we come to the all-decisive step: this approximation can be used to derive a new reduced QP in $m \ll \ell$ variables, which can be solved instead of the original problem to yield approximately the same regressor but requring only a fraction of the time. First, writing the objective function of our constrained QP (5) in matrix notation

$$\min_{\boldsymbol{\alpha}, \boldsymbol{\alpha}^* \in \mathbb{R}^\ell} -\frac{1}{2}(\boldsymbol{\alpha}^* - \boldsymbol{\alpha})^T K (\boldsymbol{\alpha}^* - \boldsymbol{\alpha}) - \varepsilon(\boldsymbol{\alpha}^* + \boldsymbol{\alpha})^T \mathbf{e} + (\boldsymbol{\alpha}^* - \boldsymbol{\alpha})^T \mathbf{y} \qquad (9)$$

we can replace in (9) kernel matrix K with its approximation $A\tilde{K}A^T$. Next, we define the $2m$ new variables $\tilde{\boldsymbol{\alpha}} = (\tilde{\alpha}_1, \ldots, \tilde{\alpha}_m)^T$ through $\tilde{\boldsymbol{\alpha}} = A^T\boldsymbol{\alpha}$, and similarly $\tilde{\boldsymbol{\alpha}}^* = A^T\boldsymbol{\alpha}^*$. Before we can complete the change of variables we must compensate the transformation for the remaining summands in (9) using the (pseudo-)inverse $A^\dagger = (A^TA)^{-1}A^T$. Thus define the new targets $\tilde{\mathbf{y}} = A^\dagger \mathbf{y}$. Now we arrive at a reduced QP with objective function

$$\min_{\tilde{\boldsymbol{\alpha}}, \tilde{\boldsymbol{\alpha}}^* \in \mathbb{R}^m} -\frac{1}{2}(\tilde{\boldsymbol{\alpha}}^* - \tilde{\boldsymbol{\alpha}})^T \tilde{K} (\tilde{\boldsymbol{\alpha}}^* - \tilde{\boldsymbol{\alpha}}) - \varepsilon(\tilde{\boldsymbol{\alpha}}^* + \tilde{\boldsymbol{\alpha}})^T A^\dagger \mathbf{e} + (\tilde{\boldsymbol{\alpha}}^* - \tilde{\boldsymbol{\alpha}})^T \tilde{\mathbf{y}} \qquad (10)$$

in the $2m$ variables $\tilde{\boldsymbol{\alpha}}, \tilde{\boldsymbol{\alpha}}^*$ instead of (9). Solving the reduced problem (10) we obtain the regressor $\tilde{f}(\cdot) = \sum_{j=1}^m (\tilde{\alpha}_j^* - \tilde{\alpha}_j) k(\tilde{\mathbf{x}}_j, \cdot)$ which is approximately the one we would have obtained solving the full problem:

$$\begin{aligned} \tilde{f}(\cdot) &= \sum_{j=1}^{\ell} (\alpha_j^* - \alpha_j) \sum_{l=1}^{m} a_{jl} k(\tilde{\mathbf{x}}_l, \cdot) \\ &\approx \sum_{j=1}^{\ell} (\alpha_j^* - \alpha_j) k(\mathbf{x}_j, \cdot) = f(\cdot) \end{aligned}$$

To illustrate the approximation quality we ran some tests on the synthetic problem $\sin \|\mathbf{x}\|/\|\mathbf{x}\|$ where $\mathbf{x} \in [10, 10]^2$ was scaled to lie inside the unit cube. Table 1 shows that for various different settings the final test-error from the regressor obtained by solving (10) closely resembles that of the full problem, at significantly lower training costs.

Table 1. Performance of sparse greedy approximation. Results for solving the 2-dimensional *sinc* function are shown for the full problem (9) and the reduced problem (10) for Gaussian kernels $k(x,y) = \exp(-\|x-y\|^2/\sigma)$. The training set consisted of 500 randomly drawn examples. Each regressor was tested on a 40×40 grid. With $\|\mathbf{y} - AA^\dagger\mathbf{y}\|^2$ we measure the error induced by projecting the data. All error-terms are given as sum of squared errors

Kernel width σ	Test-error full	TOL2	m	$\|\mathbf{y} - AA^\dagger\mathbf{y}\|^2$	Test Error Reduced
$\sigma = 0.2$		10^{-1}	15 (03.0%)	9.9087	42.6080
$c_{\text{full}} = 2500$	8.4440	10^{-3}	36 (07.2%)	7.2672	31.2892
$c_{\text{red}} = 5000$		10^{-6}	73 (14.6%)	0.2134	20.2643
$\sigma = 0.1$		10^{-1}	25 (05.0%)	8.4494	35.7908
$c_{\text{full}} = 50$	0.3589	10^{-3}	60 (12.0%)	0.5141	3.4052
$c_{\text{red}} = 5000$		10^{-6}	117 (23.4%)	$< 10^{-4}$	0.0049
$\sigma = 0.04$		10^{-1}	58 (11.6%)	0.4152	1.1733
$c_{\text{full}} = 1$	0.0037	10^{-3}	125 (25.0%)	0.0002	0.0043
$c_{\text{red}} = 500$		10^{-6}	220 (44.0%)	$< 10^{-7}$	0.0028

4 The Complete Algorithm

Let us now walk through a complete iteration of our SVR-RL approximator. Let t be the current iteration count and $(\mathbf{x}_t, y_t)$ the current sample from algorithm 1. Assume we have assembled the following

- $X_{\text{old}} \in \mathbb{R}^{\ell \times m}$ and $\mathbf{y}_{\text{old}} \in \mathbb{R}^\ell$ the training set of all ℓ sufficiently 'different' instances (see Step 1 below) seen up to time t where usually $\ell < t$.
- $D_{\text{dict}} = \{\tilde{\mathbf{x}}_1, \ldots, \tilde{\mathbf{x}}_m\}$ the dictionary consisting of m linear independent inputs where $m \ll t$.
- $\tilde{K}^{-1}_{\text{old}} \in \mathbb{R}^{m \times m}$ the inverse of $[\tilde{K}]_{ij} = k(\tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_j)$ (which is not stored).
- $A_{\text{old}} \in \mathbb{R}^{\ell \times m}$ consisting of rows $\mathbf{a}_i$ from (7).
- $(A^T\mathbf{y})_{\text{old}} \in \mathbb{R}^m$
- $(A^TA)^{-1}_{\text{old}} \in \mathbb{R}^{m \times m}$
- $\tilde{\mathbf{y}}_{\text{old}} \in \mathbb{R}^m$ the targets $\tilde{\mathbf{y}}_{\text{old}} := A^\dagger_{\text{old}}\mathbf{y}_{\text{old}} = (A^TA)^{-1}_{\text{old}}(A^T\mathbf{y})_{\text{old}}$ for the reduced problem (10).

Step 1: Tribute to the non-stationarity of RL. First we need to check whether $\mathbf{x}_t$ is a state for which we already have stored a value. In that case we will interpret y_t as an update (a new estimated value). Otherwise we need to add it to the training set, GOTO STEP 2. The criterion we use is proximity in feature space, that is if

$$\min_{i=1\ldots\ell} \|\Phi(\mathbf{x}_t) - \Phi(X_{\text{old}}[i,:])\|^2 < \texttt{TOL1} \tag{11}$$

is below some chosen tolerance TOL1 we update that component from $\mathbf{y}_{\text{old}}$ for which the index in (11) attains its minimum, i.e. the 'nearest' state. Once $\mathbf{y}_{\text{old}}$ is modified to yield $\mathbf{y}_{\text{new}}$ we need to compute $\tilde{\mathbf{y}}_{\text{new}}$ which, unfortunately, cannot be done incrementally in this case. Thus we first need to re-compute $(A^T\mathbf{y})_{\text{new}} = A^T_{\text{old}}\mathbf{y}_{\text{new}}$ and then $\tilde{\mathbf{y}}_{\text{new}} = (A^TA)^{-1}_{\text{old}}(A^T\mathbf{y})_{\text{new}}$. Finally we need to re-train the regressor, proceed with STEP 3.

Step 2: Sparse Approximation. The current state is sufficiently new and thus added to the full training set: append a row in $X_{\text{new}} = \begin{bmatrix} X_{\text{old}}^T & \mathbf{x}_t \end{bmatrix}^T$ and $\mathbf{y}_{\text{new}} = \begin{pmatrix} \mathbf{y}_{\text{old}}^T & y_t \end{pmatrix}^T$. Increment the number of training samples ℓ. Now we test if $\mathbf{x}_t$ can be approximated by D_{dict}: compute $\mathbf{a}_t$ from (7) using $\tilde{K}_{\text{old}}^{-1}$ and obtain d_t from (8). If d_t is below some chosen tolerance `TOL2` then $\mathbf{x}_t$ is approximately linearly dependent on the dictionary and thus does not contribute to the number of variables in the reduced problem, GOTO CASE 2.2. Otherwise, we need to add $\mathbf{x}_t$ to the dictionary, GOTO CASE 2.1.

Case 2.1: Add $\mathbf{x}_t$ to the dictionary and increment its size m. Since $\mathbf{x}_t$ was linearly independent the approximation coefficient $\mathbf{a}_t$ becomes $\begin{pmatrix} 0, \ldots, 0, 1 \end{pmatrix}^T$ and so every expression involving A_{old} is very cheaply updated:

$$A_{\text{new}} = \begin{bmatrix} A_{\text{old}} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix}, \qquad (A^T A)_{\text{new}}^{-1} = \begin{bmatrix} (A^T A)_{\text{old}}^{-1} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{12}$$

$$(A^T \mathbf{y})_{\text{new}} = \begin{bmatrix} (A^T \mathbf{y})_{\text{old}} \\ y_t \end{bmatrix}, \tilde{\mathbf{y}}_{\text{new}} = \begin{bmatrix} (A^T A)_{\text{old}}^{-1} & \mathbf{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} (A^T \mathbf{y})_{\text{old}} \\ y_t \end{bmatrix} = \begin{bmatrix} \tilde{\mathbf{y}}_{\text{old}} \\ y_t \end{bmatrix} \tag{13}$$

However, augmenting the dictionary means updating $\tilde{K}_{\text{old}}^{-1}$. Fortunately, this can be done efficiently since $\tilde{K}_{\text{new}}$ and $\tilde{K}_{\text{old}}$ differ only by a rank-1 update. Again denote by $\tilde{\mathbf{k}}_t = \begin{pmatrix} k(\tilde{\mathbf{x}}_1, \mathbf{x}_t), \ldots, k(\tilde{\mathbf{x}}_m, \mathbf{x}_t) \end{pmatrix}^T$ and $k_{tt} = k(\mathbf{x}_t, \mathbf{x}_t)$. Then [8]

$$\tilde{K}_{\text{new}}^{-1} = \begin{bmatrix} \tilde{K}_{\text{old}} & \tilde{\mathbf{k}}_\mathbf{t} \\ \tilde{\mathbf{k}}_\mathbf{t}^T & k_{tt} \end{bmatrix} = \begin{bmatrix} \tilde{K}_{\text{old}}^{-1} + \eta^{-1} \mathbf{a}_t \mathbf{a}_t^T & -\eta^{-1} \mathbf{a}_t \\ -\eta^{-1} \mathbf{a}_t^T & \eta^{-1} \end{bmatrix} \tag{14}$$

where $\eta = (k_{tt} - \tilde{\mathbf{k}}_t^T \tilde{K}_{\text{old}}^{-1} \tilde{\mathbf{k}}_t)$ and $\mathbf{a}_t$ from (7). GOTO STEP 3.

Case 2.2 The current input $\mathbf{x}_t$ is linearly dependent on the dictionary, thus D_{dict}, its size m, and $\tilde{K}_{\text{new}}^{-1} = \tilde{K}_{\text{old}}^{-1}$ remain unchanged. Append the approximation coefficients $\mathbf{a}_t$ from (7) as row in $A_{\text{new}} = \begin{bmatrix} A_{\text{old}}^T & \mathbf{a}_t \end{bmatrix}^T$. To update $(A^T A)_{\text{old}}^{-1}$ note that $(A^T A)_{\text{new}} = (A^T A)_{\text{old}} + \mathbf{a}_t \mathbf{a}_t^T$ and so, using the matrix inversion lemma, we obtain the recursive update formula

$$(A^T A)_{\text{new}}^{-1} = (A^T A)_{\text{old}}^{-1} - \frac{(A^T A)_{\text{old}}^{-1} \mathbf{a}_t \mathbf{a}_t^T (A^T A)_{\text{old}}^{-1}}{1 + \mathbf{a}_t^T (A^T A)_{\text{old}}^{-1} \mathbf{a}_t} \tag{15}$$

Now we take care of $(A^T \mathbf{y})_{\text{new}}$, and $\tilde{\mathbf{y}}_{\text{new}}$. While the first part is obtained easily, the second part has no recursive update and must be computed anew

$$(A^T \mathbf{y})_{\text{new}} = (A^T \mathbf{y})_{\text{old}} + y_t \mathbf{a}_t, \qquad \tilde{\mathbf{y}}_{\text{new}} = (A^T A)_{\text{new}}^{-1} (A^T \mathbf{y})_{\text{new}} \tag{16}$$

Again, GOTO STEP 3.

Step 3: Solve the reduced problem. Finally, we use the sparsified training set $\{(\tilde{\mathbf{x}}_1, \tilde{y}_1), \ldots, (\tilde{\mathbf{x}}_m, \tilde{y}_m)\}$ with $\tilde{\mathbf{y}}_{\text{old}} = (\tilde{y}_1, \ldots, \tilde{y}_m)^T$ to solve the reduced QP (10).

Paramater: $\texttt{TOL1}, \texttt{TOL2}, C, \varepsilon$ and kernel
Initialize: $X_{\text{old}} = [\mathbf{x}_1], \mathbf{y}_{\text{old}} = (y_1), \tilde{\mathbf{y}}_{\text{old}} = (y_1), D_{\text{dict}} = \{\mathbf{x}_1\}, \tilde{K}_{\text{old}}^{-1} = [k_{tt}^{-1}], A_{\text{old}} = [1], (A^T\mathbf{y})_{\text{old}} = [y_1], (A^TA)_{\text{old}}^{-1} = [1], m = 1, \ell = 1$
for $t = 1, 2, \ldots$ **do**
  Get new sample $(\mathbf{x}_t, y_t)$ from algorithm 1
  if $\min_{i=1\ldots\ell} \|\Phi(\mathbf{x}_t) - \Phi(X_{\text{old}}[i,:])\|^2 < \texttt{TOL1}$ **then**
    $\mathbf{y}_{\text{old}}[k] \leftarrow y_t$ where k is argmin from (11)
    Compute $(A^T\mathbf{y})_{\text{new}}$ and $\tilde{\mathbf{y}}_{\text{new}} = (A^TA)_{\text{old}}^{-1}(A^T\mathbf{y})_{\text{new}}$
  else
    $X_{\text{new}} = [X_{\text{old}}^T \quad \mathbf{x}_t]^T, \mathbf{y}_{\text{new}} = (\mathbf{y}_{\text{old}}^T \quad y_t)^T, \ell = \ell + 1$
    Compute $\tilde{\mathbf{k}}_t$
    Compute $\mathbf{a}_t$ from (7) and d_t from (8)
    if $d_t > \texttt{TOL2}$ **then**
      Add $\mathbf{x}_t$ to D_{dict}, $m = m + 1$
      Compute $\tilde{K}_{\text{new}}^{-1}$ from (14)
      Compute A_{new} and $(A^TA)_{\text{new}}^{-1}$ from (12)
      Compute $(A^T\mathbf{y})_{\text{new}}$ and $\tilde{\mathbf{y}}_{\text{new}}$ from (13)
    else
      $A_{\text{new}} = [A_{\text{old}}^T \quad a_t]^T$
      Compute $(A^TA)_{\text{new}}^{-1}$ from (15)
      Compute $(A^T\mathbf{y})_{\text{new}}$ and $\tilde{\mathbf{y}}_{\text{new}}$ from (16)
  Solve the reduced QP (10) for $\{(\tilde{\mathbf{x}}_1, \tilde{y}_1), \ldots, (\tilde{\mathbf{x}}_m, \tilde{y}_m)\}$

Algorithm 2. Sparse on-line SVR for *sarsa* function approximation

5 Experiments

We applied the *sarsa* and SVR combination to two control tasks with continuous states. The training procedure consisted of a series of trials each starting in a state chosen randomly and proceeding thereafter until the goal was reached.

The first domain is a 21×21 gridworld embedded inside the unit cube. Starting in the bottom row the goal was to get to the upper right corner $(1, 1)$ in minimum time. Reaching the goal ends the episode, every other step entails a cost of -1. Actions were the four compass directions. The corresponding value function is made discontinuous by introducing a vertical barrier. Parameters for the *sarsa* component were $\gamma = 1, \alpha = 0.7$, and $\epsilon = 0.1$, the last choice has been made as to ensure sufficient exploration. For the SVR component we chose $C = 10, \varepsilon = 0.1, \texttt{TOL1} = 0.005$, and $\texttt{TOL2} = 0.1$. Kernels were the Gaussian $k(x, y) = \exp(-\|x - y\|^2/\sigma)$. The QP was solved using the C++ package *libSVM* [2]. Figure 1 shows the resulting approximation for various kernel widths σ after 5000 trials. As could be expected, wide kernels entail a large approximation error near the barrier. Yet in all cases a nearly optimal path to the goal was learned.

As second task we tried the well-known mountain car problem. The objective is trying to drive a car to the top of a steep hill. However, the car is not powerful enough to directly drive to the goal. Instead it must first go up the opposite direction to build up enough momentum. The two dimensional state space consists of the position and speed, and again was scaled to lie inside the

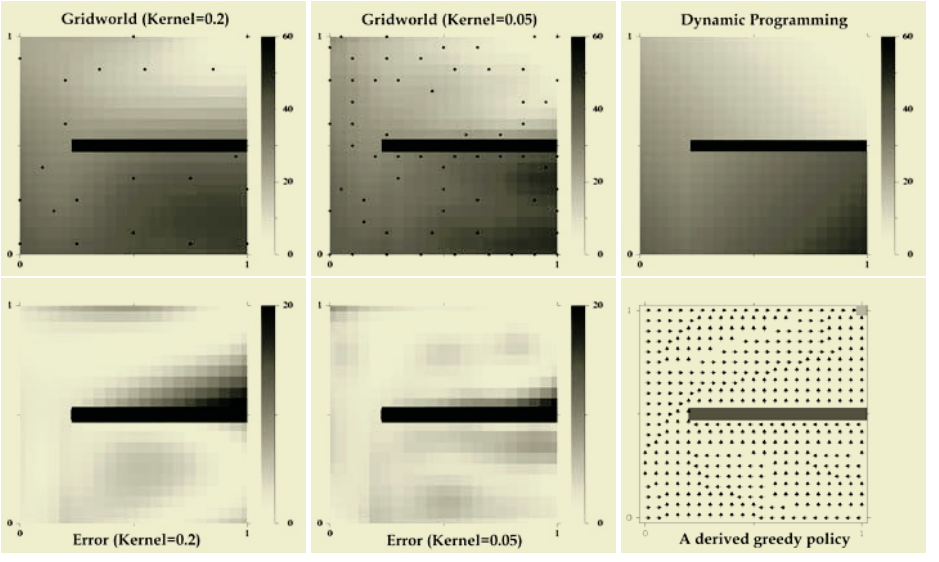

Fig. 1. Approximated value function for the gridworld task after 5000 trials for wide and narrow kernels. Black dots mark the states that were admitted to the dictionary. From left to right: $\sigma = 0.2$ with $m = 21$, $\sigma = 0.05$ with $m = 63$, and the optimal value function. The bottom row shows the corresponding approximation error and a derived greedy policy

unit cube. Three actions are possible: forward, backward, and coast. Failing to reach the goal within 500 steps terminates the trial, but no additional penalty is incurred. The remaining setup and simulated dynamics are identical to those used in [10]. The parameters for SVR and *sarsa* were the same as before.

First, we were interested in the on-line performance. In Fig. 2 we compare the results of our SVR method with the very popular tile coding function approximator [11]. We used 10 overlapping tilings, each consisting of 10×10 grids. As baseline we also included the optimal number of steps computed on a 200×200 grid. To make the results more comparable, each trial started at the bottom of the valley with zero velocity. The performance measure we considered were the steps-to-goal during a trial and the return of the corresponding greedy policy, which was evaluated independently. Each curve shows the average of 10 completely different runs. As can be seen from the plots the overall performance of SVR is roughly on par with tile coding, and even exceeds it later on. The total number of parameters (i.e. weights) that have to be adjusted in the SVR expansion ($m = 37$ for $\sigma = 0.05$) is only a fraction of the number of weights in tile coding ($10 \times 10 \times 10$). However, we should also mention that in return the overall computional costs are considerably higher for SVR.

Second, we plotted the approximation quality for various kernel widths after 10000 trials, where as before each trial started in a randomly chosen state, see Fig. 3. Although we could evidently derive a fairly optimal policy previously,

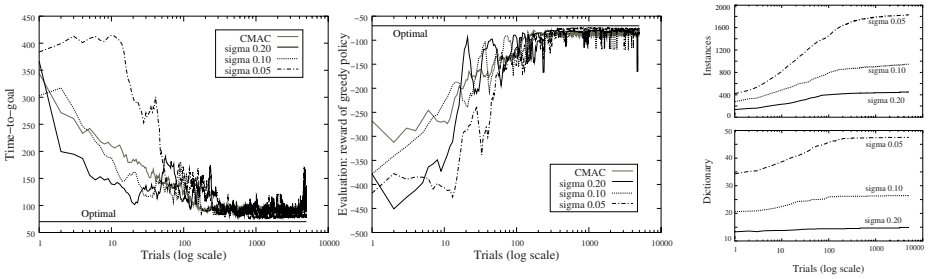

Fig. 2. Comparing SVR to tile coding in the mountain car task. Each curve shows the average of 10 different runs. Left: time-to-goal per trial. Middle: return of the respective greedy policy. The last plot compares the number of instances stored with the size of the dictionary

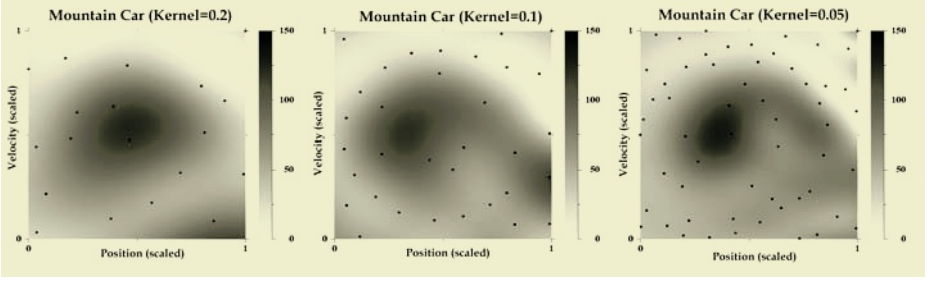

Fig. 3. Approximated value function for the mountain car task after 10000 trials for various kernel widths σ. Black dots mark the states that were admitted to the dictionary. From left to right: $\sigma = 0.2$ with $m = 19$, $\sigma = 0.1$ with $m = 33$, and $\sigma = 0.05$ with $m = 57$. For wide kernels the discontinuities are smeared out. With narrow kernels the boundaries are sharpened and more pronounced

the quality of the approximation near the discontinuities (in particular for wide kernels) is far from good. Note the way in which the states in the dictionary are distributed. The circular, spiraling pattern corresponds to the initial trajectories of the learner, oscillating back and forth between the two hill tops.

6 Conclusion and Future Research

We demonstrated experimentally that on-line RL in conjunction with SVR function approximation is possible. The SVR method described in this paper relied on solving a reduced QP obtained from projecting the data on a small subset of the original training samples. This way the main workload during training only depends on the effective dimensionality of the data which can be considered as being asymptotically independent of the total number of training pairs encountered in the RL algorithm. Another consequence is that one obtains a very sparse regressor which, when used as predictor in subsequent operations such as the policy improvement step, again helps to reduce the computational

complexity. The resulting SVR-RL framework was shown to solve two common toy-problems and to approximate the optimal value function.

The algorithm and simulations that were presented can only be considered as a first step to demonstrate the overall feasability of this approach. Additional conceptual and algorithmic refinements and more experimentation are needed. The framework very naturally carries over to other RL methods: currently we are extending it to include the model-free case and to perform TD(λ) policy-evaluation. In particular, we expect it to work well when coupled with off-line control methods like partially optimistic policy-iteration, since here the updates to the value function can be carried out in a batch manner.

References

1. D. Bertsekas and J. Tsitsiklis. *Neuro-dynamic programming.* Athena Scientific, 1996.
2. Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.
3. T. Dietterich and X. Wang. Batch value function approximation via support vectors. In *NIPS 14*, pages 1491–1498, 2002.
4. T. Downs, K. E. Gates, and A. Masters. Exact simplification of support vector solutions. *Journal of Machine Learning Research*, 2:293–297, 2001.
5. Y. Engel, S. Mannor, and R. Meir. Kernel recursive least squares. In *Proc. of 13th European Conference on Machine Learning*. Springer, 2002.
6. Y. Engel, S. Mannor, and R. Meir. Sparse online greedy support vector regression. In *Proc. of 13th European Conference on Machine Learning*. Springer, 2002.
7. S. Fine and K. Scheinberg. Efficient SVM training using low-rank kernel representation. *Journal of Machine Learning Research*, 2:243–264, 2001.
8. B. Schölkopf and A. Smola. *Learning with Kernels.* Cambridge, MA: MIT Press, 2002.
9. A. J. Smola and B. Schölkopf. Sparse greedy matrix approximation for machine learning. In *Proc. of 17th ICML*, 2000.
10. R. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 1998.
11. R. S. Sutton. Generalization in reinforcement learning: successful examples using sparse coarse coding. In *NIPS 7.*, pages 1038–1044, 1996.
12. C. Williams and M. Seeger. Using the nyström method to speed up kernel machines. In *NIPS 13*, pages 682–688, 2001.