Knowledge-Based Support-Vector Regression for Reinforcement Learning

Richard Maclin[†], Jude Shavlik[‡], Trevor Walker[‡], Lisa Torrey[‡]

University of Minnesota – Duluth[†]   University of Wisconsin – Madison[‡]
Computer Science Department   Computer Sciences Department
1114 Kirby Dr, Duluth, MN 55812   1210 W Dayton St, Madison, WI 53706
rmaclin@d.umn.edu   {shavlik,torrey,twalker}@cs.wisc.edu

Abstract

Reinforcement learning (RL) methods have difficulty scaling to large, complex problems. One approach that has proven effective for scaling RL is to make use of advice provided by a human. We extend a recent advice-giving technique, called Knowledge-Based Kernel Regression (KBKR), to RL and evaluate our approach on the *KeepAway* subtask of the RoboCup soccer simulator. We present empirical results that show our approach can make effective use of advice. Our work not only demonstrates the potential of advice-giving techniques such as KBKR for RL, but also offers insight into some of the design decisions involved in employing support-vector regression in RL.

1 Introduction

Reinforcement learning (RL) techniques such as *Q*-learning and SARSA [Sutton and Barto, 1998] are effective learning techniques, but often have difficulty scaling to challenging, large-scale problems. One method for addressing the complexity of such problems is to incorporate advice provided by a human teacher. The approach of using advice has proven effective in a number of domains [Lin, 1992; Gordon and Subramanian, 1994; Maclin and Shavlik, 1994; Andre and Russell, 2001; Kuhlmann et al., 2004].

Recently, Mangasarian et al., [2004] introduced a method called Knowledge-Based Kernel Regression (KBKR) that allows a kernel method to incorporate advice given in the form of simple IF-THEN rules into a support vector method for learning regression (i.e., real-valued) problems. Their technique proved effective on the simple regression problems on which they tested it. In this article we extend the general KBKR approach to RL and test it on a complex game from the RoboCup soccer simulator [Noda et al., 1998] – *KeepAway* [Stone and Sutton, 2001].

In applying the KBKR approach to *KeepAway* we found that we had to make a number of adjustments and extensions, both to the KBKR method and to our representation of the problem. These adjustments and extensions proved critical to effectively learning in the *KeepAway* task.

In the next section of the paper we present the basic KBKR method. In Section 3 we present the RoboCup simu-

lator and *KeepAway* in particular, and discuss some of the difficulties for this game. In Section 4 we discuss how a support-vector regressor can be used in *Q*-learning and then present our reformulation of the KBKR method and some of the issues we had to address in order to get good results. Following that we present results of experiments using our new approach on *KeepAway*. The final sections discuss related research, future directions, and conclusions.

2 Knowledge-Based Support Vector Regression

In this section we present the basics of Knowledge-Based Kernel Regression [Mangasarian et al., 2004].

2.1 Support Vector Regression

A linear regression problem involves trying to find a set of weights (w) and a offset (b) to learn a function of the form $f(x) = w^T x + be$, where T indicates the transpose of a vector, x is a vector of numeric features describing a particular instance (e.g., values describing the soccer field as seen from the player's point of view), $f(x)$ is the value that instance is labeled with (e.g., the Q value of taking action *HoldBall*), and e denotes a vector of ones. From now on, for clarity we will omit e, with the understanding that b is a scalar.

For a particular set of observed input vectors (a set of states observed during learning) and a corresponding set of $f(x)$ values (the current Q estimates for each of those states), we find a solution to the equation:
$$Aw + b = y$$
where A is the set of states, one row for each input vector, one column for each feature, and y is the set of expected $f(x)$ values, one for each input vector. Since an exact solution is often infeasible, this equation is generalized to:
$$Aw + b \approx y \qquad (1)$$
Solutions to this problem are ranked by how well they meet some performance criterion (such as minimum error with respect to the y values).

In a kernel approach, the weight vector w is replaced with its dual form $A^T\alpha$, which converts Eq. 1 to:
$$AA^T\alpha + b \approx y$$
This formulation is then generalized by replacing the AA^T term with a kernel, $K(A,A^T)$, to produce:
$$K(A,A^T)\alpha + b \approx y \qquad (2)$$
However, in this article we simply use Eq. 1 above, since linear models are more understandable and scale better to

large numbers of training examples. We use *tile coding* (an example is shown below) to produce the non-linearity in our models. The reader should note that using Eq. 1 is not identical to simply using a linear kernel (K) in Eq. 2.

To use a linear programming (LP) method to learn a model we simply have to indicate the expression to be minimized when producing a solution. One common formulation for linear regression models is the following, which we call LP1 so we can refer to it again later:

$$\min_{s \geq 0} \| w \|_1 + \| b \|_1 + C \| s \|_1$$
$$s.t. \quad -s \leq Aw + b - y \leq s$$

In this formulation we use *slack* variables s to allow the solution to be inaccurate on some training examples, and we penalize these inaccuracies in the objective function that is to be minimized. We then minimize a weighted sum of the s slack terms and the absolute value of weights and the b term (the *one-norm*, $\|\cdot\|_1$, computes the sum of absolute values). This penalty on weights (and $b)$ penalizes the solution for being more complex. C is a parameter for trading off how inaccurate the solution is (the s terms) with how complex the solution is (the weights and b). The resulting minimization problem is then presented to a linear program solver, which produces an optimal set of w and b values.

2.2 Knowledge-Based Kernel Regression

In KBKR, a piece of advice or domain knowledge is represented in the notation:

$$Bx \leq d \quad \Rightarrow \quad f(x) \geq h^T x + \beta \qquad (3)$$

This can be read as:

If certain conditions hold ($Bx \leq d$), the output, $f(x)$, should equal or exceed some linear combination of the inputs (h^Tx) plus a threshold term (β).

The term $Bx \leq d$ allows the user to specify the region of input space where the advice applies. Each row of matrix B and its corresponding d values represents a constraint in the advice. For example, a user might give the rule:

IF (*distanceA* + 2 *distanceB*) ≤ 10
THEN $f(x) \geq 0.5$ *distanceA* + 0.25 *distanceB* + 5

For this IF-THEN rule, matrix B would have a single row with a 1 in the column for feature *distanceA* and a 2 in the column for *distanceB* (the entry for all other features would be 0), and the d vector would be a scalar with the value 10.

In general, the rows of B and the corresponding d values specify a set of linear constraints that are treated as a conjunction and define the polyhedral region of the input space to which the right-hand side of the advice applies. The vector h and the scalar β then define a linear combination of input features that the predicted value $f(x)$ should match or exceed. For the above rule, the user advises that when the left-hand side condition holds, the value of $f(x)$ should be greater than ½ *distanceA* plus ¼ *distanceB* plus 5. This would be captured by creating an h vector with coefficients of 0.5 and 0.25 for the features *distanceA* and *distanceB* (0 otherwise)*,* and setting β to 5.

In this advice format, a user in a reinforcement-learning task can define a set of states in which the Q value for a specific action should be high (or low). We later discuss how we numerically represent "high Q."

Mangasarian et al. prove that the advice implication in Eq. 3 is equivalent to the following set of equations having a solution (we have converted to non-kernel form):

$$B^T u + w - h = 0, \quad -d^T u + b - \beta \geq 0, \quad u \geq 0 \qquad (4)$$

"Softening" the first two of these leads to the following optimization problem in the case of linear models (LP2):

$$\min_{s \geq 0, u \geq 0, z \geq 0, \zeta \geq 0} \| w \|_1 + \| b \|_1 + C \| s \|_1 + \mu_1 \| z \|_1 + \mu_2 \zeta$$
$$s.t. \quad -s \leq Aw + b - y \leq s$$
$$-z \leq B^T u + w - h \leq z$$
$$-d^T u + \zeta \geq \beta - b$$

The z and ζ are slack terms associated with the advice; they allow Eq. 4 to be only approximately satisfied. The μ_1 and μ_2 parameters specify how much to penalize these slacks. In other words, these slacks allow the advice to be only partially followed by the learner.

Mangasarian et al., [2004] tested their method on some simple regression problems and demonstrated that the resulting solution would incorporate the knowledge. However, the testing was done on small feature spaces and the tested advice placed constraints on *all* of the input features. In this article we apply this methodology to a more complex learning problem based on RL and the RoboCup simulator.

3 RoboCup Soccer: The Game *KeepAway*

We experimented on the game *KeepAway* in simulated RoboCup soccer [Stone and Sutton, 2001]. In this game, the goal of the N "keepers" is to keep the ball away from $N-1$ "takers" as long as possible, receiving a reward of 1 for each time step they hold the ball (the keepers learn, while the takers follow a hand-coded policy). Figure 1 gives an example of *KeepAway* involving three keepers and two takers.

To simplify the learning task, Stone and Sutton chose to have learning occur only by the keeper who currently holds the ball. When no player has the ball, the nearest keeper pursues the ball and the others perform hand-coded moves to "get open" (be available for a pass). If a keeper is holding the ball, the other keepers perform the "get open" move.

The learnable action choice then is whether to hold the ball or to pass it to another keeper. Note that passing requires multiple actions in the simulation (orienting the body, then performing multiple steps of kicking), but these low-level actions are managed by the simulator and are not addressed in Stone and Sutton's, nor our, experiments.

The policy of the takers is simple; if there are only two takers they pursue the ball. When there are more than two takers, two pursue the ball and the others "cover" a keeper.

For our work we employ the feature representation used by Stone and Sutton. They measure 13 values that define the state of the world from the perspective of the keeper that currently has the ball. These 13 features record geometric properties such as the pair-wise distances between players and the angles formed by sets of three players.

The task is made more complex because the simulator incorporates noise into the information describing the state. In addition, the actions of the agents contain noise. For example, there is a chance the keeper passing the ball to another keeper will misdirect the ball, possibly sending it out
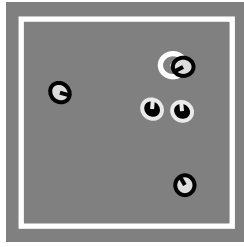
Figure 1. A sample KeepAway game where there are three keepers (light gray with black outlines), two takers (black with gray outlines), and the ball (currently held by the keeper in the upper right). A game continues until one of the takers holds the ball for at least 5 time steps (0.5 sec) or if the ball goes out of bounds (beyond the white lines).

of bounds or towards one of the takers. The overall score of a keeper team is measured in terms of how long they are able to hold onto the ball.

Stone and Sutton [2001] demonstrated that this task can be learned with RL. They employed SARSA learning with replacing eligibility traces, and used CMAC's as their function approximator. They used a tile encoding of the state space, where each feature is discretized several times into a set of overlapping bins. For example, one could divide a feature that ranges from 0 to 5 into four overlapping bins of width 2: one bin covering values [0,2], one [1,3], one [2,4] and one [3,5]. This representation proved very effective in their experiments and we use it also.

4 Using KBKR for KeepAway

In order to use regression for RL we must formulate the problem as a regression problem. We represent the real-valued Q function as a set of learned models, one for each action. The input to each model is the state and each model makes a prediction of the Q value for the action. We use one-step SARSA to estimate the Q value.

Since incremental training algorithms are not well developed for support vector machines, we employ batch training. We save the series of states, actions, and reinforcements experienced over each set of 100 games, we then stop to train our models, and then use the resulting models in the next chunk of 100 games. When we create training examples from old data, we use the *current* model to compute the Q values for the next state in one-step *SARSA* learning since these estimates are likely to be more accurate than those obtained when these old states were actually encountered.

This batch approach is effective but leads to a potential problem. As the game continues data accumulates, and eventually the sets of constraints in LP1 and LP2 become intractably large for even commercial LP solvers. In addition, because the learner controls its experiences, older data is less valuable than newer data.

Hence, we need a mechanism to choose which training examples to use. We do this by taking a stochastic sample of the data. We set a limit on the number of training examples (we currently set this limit to 1500 and have not experimented with this value). If we have no more examples than this limit, we use them all. When we need to discard some examples, we keep a (uniformly) randomly selected 750 (i.e., half our limit) and discard others according to their age. The probability we select an as yet unselected example is ρ^{age} (ρ raised to the power *age*) and we set ρ to a value in [0,1] to produce a data set of our specified maximum size.

In our initial experiments on *KeepAway* we employed kernel-based regression directly using the 13 numeric features used by Stone and Sutton without tile encoding. In these experiments, we found that both Gaussian and linear kernels applied to just these 13 features performed only slightly better than a random policy (the results stay right around 5 seconds – compare to Figure 3's results).

Using Stone and Sutton's tile coding led to substantially improved performance, and we use that technique for our experiments. We provide to our learning algorithms *both* the 13 "raw" numeric features as well as binary features that result from (separately) tiling each of the 13 features. We create 32 binary features per raw feature. We keep the numeric features to allow our methods to explore a wide range of possible features and also since the numeric features are more easily expressed in advice.

One *critical* adjustment we found necessary to add to the KBKR approach (LP2) was to append additional constraints to the constraints defined by the *B* matrix and *d* vector of Eq. 3. In our new approach we added for each feature not mentioned in advice constraints of the form:

$$\min(feature_i) \leq feature_i \leq \max(feature_i)$$

For example, if *distanceC* ranges from 0 to 20, we add the constraint: $0 \leq distanceC \leq 20$.

This addresses a severe limitation in the original KBKR method. In the original KBKR approach the advice, when unmodified with slack variables, implies that the right-hand side must be true in *all* cases (no matter what the values of the other features are). For example, if we advise "when *distanceA* is less than 10 we want the output to be at least 100," but do not place any restrictions on the other features' values, the KBKR algorithm cannot include any other features in its linear model, since such a feature could hypothetically have a value anywhere from $-\infty$ to $+\infty$, and one of these extremes would violate the THEN part of advice. By specifying the legal ranges for all features (including the Booleans that result from tiling), we limit the range of the input space that KBKR has to consider to satisfy the advice.

For this reason, we also automatically generate constraints for any of the binary features constrained to be true or false by advice about numeric feature. For example, assume the advice says *distanceA>10*. We then add constraints that capture how this information impacts various tiles. If we had two tiles, one checking if *distanceA* is in [0,10], and a second checking if *distanceA* is in [10,20], we would add constraints that the first tile must be false for this advice and the second tile must be true.

If we tile a feature into several bins where more than one tile might match the constraint – imagine that *distanceA* was divided into tiles [0,5], [5,10], [10,15] and [15,20] – we would add constraints indicating that each of the first two must be false for the constraint (*distanceA>10*) and *one* of the last two must be true. This last constraint equation (that one of the last two tiles must be true) would be:

$$distanceA[10,15] + distanceA[15,20] = 1$$

where *distanceA[X,Y]* denotes the feature value (0 or 1) for the tile of *distanceA* covering the range *[X,Y]*.

In cases where a tile only partially lines up with a constraint included in advice – for example if *distanceA* was covered by tiles [0,4], [4,8], [8,12], [12,16] and [16,20] and the advice included *distanceA>10,* we would still add constraints to indicate that the first two tiles ([0,4] and [4,8]) must be false and then add a constraint that one of the other three tiles must be true (as in the equation shown above). These cases will often occur, since we cannot count on advice lining up with our tiling scheme. Since we conjoin the constraints on the tiles with the original constraint on the numeric feature, it is safe to include tiles that span beyond the original advice's constraint on the numeric feature.

We also needed to extend the mechanism for specifying advice in KBKR in order to apply it to RL. In the original KBKR work the output values are constrained by a linear combination of constants produced by the user (see Eq. 3). However, in RL advice is used to say when the *Q* for some action should be *high* or *low*. So we need some mechanism to convert these terms to numbers. We could simply define *high* to be, say, ≥ 100 and *low* to be ≤ 50, but instead we decided to let the training data itself specify these numbers. More specifically, we allow the term *averageQ* to be used in advice and this value is computed over the examples in the training set. Having this term in our advice language makes it easier to specify advice like "in these states, this action is 10 units better than in the typical state."

As briefly mentioned earlier, our linear programs penalize the *b* term in our models. In our initial experiments, however, we found the *b* term led to underfitting of the training data. Recall that our training data for action *A*'s model is a collection of states where *A* was applied. As training progresses, more and more of the training examples come from states where executing action *A* is a good idea, (i.e., action *A* has a high *Q* value in these states). If the *b* term is used in the learned models to account for the high *Q* values, then when this model is applied to a state where action *A* is a bad choice, the predicted *Q* may still be high.

For instance, imagine a training set contains 1000 examples where the *Q* is approximately 100 and 10 examples where the *Q* is 0. Then the constant model *Q* = 100 might be the optimal fit to this training set, yet is unlikely to lead to good performance when deployed in the environment.

One way to address this weakness is to include in the training set more states where the *Q* for action *A* is low, and we partly do this by keeping some early examples in our training set. In addition we address this weakness by highly penalizing non-zero *b* terms in our linear programs (for clarity we did not explicitly show a scaling factor on the *b* term earlier in our linear programs). The hypothesis behind this penalization of *b* is that doing so will encourage the learner to instead use weighted feature values to model the *Q* function, and since our objective function penalizes having too many weights in models, the weights used to model the set of high *Q* values will have captured something essential about these states that have high *Q*'s, thereby generalizing better to future states. Our improved empirical evidence after strongly penalizing *b* supports our hypothesis. In general, one needs to carefully consider how to choose training examples when using a non-incremental learner in RL.

5 Experimental Results

We performed experiments on the *KeepAway* task using our approach for incorporating advice into a learner via the KBKR method. As an experimental control, we also consider the same support-vector regressor but without advice. In other words, LP2 described in Section 2 is our main algorithm and LP1 is our experimental control, with both being modified for RL as explained in Section 4. We measure our results in terms of the length of time the keepers hold the ball. Our results show that, on average, a learner employing advice will outperform a learner not using advice.

5.1 Methodology

Our experiments were performed on 3 versus 2 *KeepAway* (3 keepers and 2 takers). The takers employed a fixed policy as described in Section 3. The keepers were all learning agents and pooled their experience to learn a single model which is shared by all of the keepers.

The reinforcement signals the learners receive are 0.1 for each step in the game and a 0 when the game ends (when the takers control the ball or the ball goes out of bounds). Our discount rate is set to 1, the same value used by Stone and Sutton [2001]. For our action-selection process we used a policy where we performed an exploitation action (i.e., chose the action with the highest value) 99% of the time and randomly chose an action (exploration) the remaining time, again following Stone and Sutton. We report the average total reinforcement for the learners (the average time the keepers held the ball) over the previous 1000 games.

We set the values of *C*, *μ1*, and *μ2* in LP1 and LP2 to be 100/#examples, 10, and 100 respectively. By scaling *C* by the number of examples, we are penalizing the *average* error on the training examples, rather than the *total* error over a varying number of examples. Since the number of weights is fixed in LP1 and LP2, we do not want the penalty due to data mismatch to grow as the number of training examples increases. We tried a small number of settings for *C* for our non-advice approach (i.e., our experimental control) and found this value worked best. We use this same value
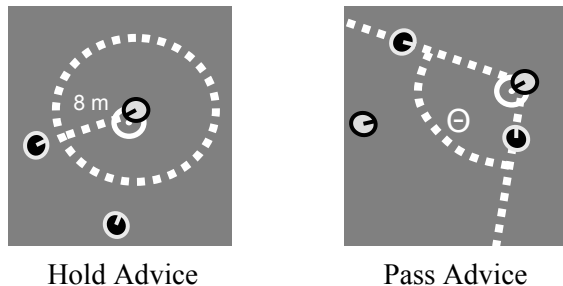

Hold Advice   Pass Advice

Figure 2. The two pieces of advice involve a suggestion when to hold the ball (if the nearest taker is at least 8m away) , and when to pass the ball (if a taker is closing in, the teammate is further away than the takers and there is a large passing lane - the value of Θ is $\geq 45°$).
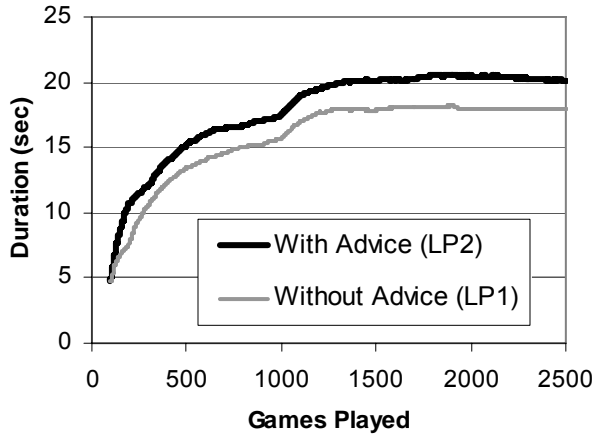
Figure 3. Results of standard support vector linear regression versus a learner that receives at the start of learning the advice described in the text.

for our KBKR approach. We simply chose the $\mu 1$ and $\mu 2$ values and have not experimented with different settings.

Each point in our results graphs is averaged over ten runs. The results are reported as a function of the number of games played, although since games are of different length, the amount of experience differs. This result is somewhat mitigated in that we provide at most 1500 state and action pairs to our learners, as discussed above.

5.2 Advice We Used

We employed two pieces of advice. The advice is based on advice used in Kuhlman et al., [2004]. The first rule suggests the keeper with the ball should hold it when the nearest taker is at least 8m away (see Fig. 2 left). When this advice applies, it suggests the Q for holding should exceed the average for holding by 1 second for each meter the closest taker is beyond 8 meters. The advice in our notation is:

IF   *distanceNearestTaker* ≥ 8
THEN *Q(hold)* $\geq$ *averageQ + distanceNearestTaker - 8*

The second piece of advice indicates when to pass the ball (see Figure 2 right). This advice tests whether there is a taker closing in, whether there is a teammate that is further away than either of the takers and whether there is a passing lane (a value of Θ that is at least 45 degrees) to that teammate. When this advice applies, it suggests that the Q for passing to the nearest teammate exceeds the average by 0.1 seconds for each degree (up to 60 degrees, and by 6 seconds for angles larger than 60 degrees).

5.3 Results and Discussion

Fig. 3 presents the results of our experiments. These results show that a learner with advice obtains gains in performance due to that advice and retains a sizable advantage in performance over a large number of training examples. Figure 3 indicates that advice-taking RL based on KBKR can produce significant improvements for a reinforcement learner ($p < 0.01$ for an unpaired t-test on the performance at 2500 games played). Although other research has demonstrated the value of advice previously (see next section), we believe

that the advantages of using a support-vector based regression method make this a novel and promising approach.

Our results are not directly comparable to that in Stone and Sutton [2001] because we implemented our own RoboCup players, and their behavior, especially when they do not have the ball, differs slightly. We also tile features differently than they do. Stone and Sutton's learning curves start at about 6 seconds per game and end at about 12 after about 6000 games (our results are initially similar but our keepers games last longer – possibly due to somewhat different takers). We have not implemented Stone and Sutton's method, but our LP1 is a good proxy for what they do. Our focus here is on the relative impact of advice, rather than a better non-advice solution.

6 Related Work

A number of researchers have explored methods for providing advice to reinforcement learners. These include methods such as replaying teaching sequences [Lin, 1992], extracting information by watching a teacher play a game [Price and Boutilier, 1999], and using advice to create reinforcement signals to "shape" the performance of the learner [Laud and DeJong, 2002]. Though these methods have a similar goal of shortening the learning time, they differ significantly in the kind of advice provided by the human.

Work that is more closely related to the work we present here includes various techniques that have been developed to incorporate advice in the form of textual instructions (often as programming language constructs). Gordon and Subramanian [1994] developed a method that used advice in the form IF *condition* THEN *achieve goals* that adjusts the advice using genetic algorithms. Our work is similar in the form of advice, but we use a significantly different approach (optimization by linear programming) to incorporate advice.

In our previous work [Maclin and Shavlik, 1994], we developed a language for providing advice that included simple IF-THEN rules and more complex rules involving multiple steps. These rules were incorporated into a neural network, which learned from future observations. In this earlier work new hidden units are added to the neural network that represent the advice. In this article, a piece of advice represents constraints on an acceptable solution.

Andre and Russell [2001] developed a language for creating RL agents. Their language allows a user to specify partial knowledge about a task using programming constructs to create a solver, but also includes "choice" points where the user specifies *possible* actions. The learner then acquires a policy to choose from amongst the possibilities. Our work differs from theirs in that we do not assume the advice is correct.

In Kuhlmann et al., [2004], advice is in the form of rules that specify in which states a given action is good (or bad). When advice is matched, the predicted value of an action in that state is increased by some fixed amount. Our work differs from this work in that our advice provides constraints on the Q values rather than simply adding to the Q value. Thus our learner is better able to make use of the advice when the advice is already well represented by the data. We

tested the Kuhlmann et al. method with our no-advice algorithm (LP1), but found it did not improve performance.

A second area of related research is work done to employ support vector regression methods in RL agents. Both Dietterich and Wang [2001] and Lagoudakis and Parr [2003] have explored methods for using support vector methods to perform RL. The main limitation of these approaches is that these methods assume a model of the environment is available (or at least has been learned) and this model is used for simulation and Q-value inference. Our approach is a more traditional "model-free" approach and does not need to know the state-transition function.

7 Conclusions and Future Directions

We presented and evaluated an approach for applying Mangasarian et al.'s [2004] Knowledge-Based Kernel Regression (KBKR) technique to RL tasks. In our work we have investigated the strengths and weaknesses of KBKR and have developed adjustments and extensions to that technique to allow it to be successfully applied to a complex RL task. Our experiments demonstrate that the resulting technique for employing advice shows promise for reducing the amount of experience necessary for learning in such complex tasks, making it easier to scale RL to larger problems. The key findings and contributions of this paper are:

1. We demonstrated on the challenging game *KeepAway* that a variant of the KBKR approach (LP2) could be successfully deployed in a reinforcement-learning setting. We also demonstrated that "batch" support-vector regression (LP1) can learn in a challenging reinforcement-learning environment without needing to have a model of the impact of actions on its environment.

2. We found that in order for the advice to be used effectively by the KBKR algorithm, we had to specify the legal ranges for *all* input features. Otherwise advice had to be either absolutely followed or "discarded" (via the slack variables of LP2) since, when the model includes any input feature not mentioned in the advice, the THEN part of advice (Eq. 3) can not be guaranteed to be met whenever the current world state matches the IF part. We also augment advice about numeric features by making explicit those constraints on the associated binary features that results from tiling the numeric features.

3. We found that it was critical that our optimization not only penalize the size of the weights in the solution, but that a sizable penalty term should also be used for the "b" term (of the "$y = wx + b$" solution) so that the learner does not simply predict the mean Q value.

4. Because little work has been done on incremental support vector machines, we chose to learn our Q models in a batch fashion. For a complex problem, this large set of states quickly results in more constraints than can be efficiently solved by a linear-programming system, so we had to develop a method for selecting a subset of the available information with which to train our models.

5. We found that without tile coding, we were unable to learn in *KeepAway*. One advantage of using tile coding is that we did not need to use non-linear kernels; the non-linearity of tile coding sufficed.

6. Finally, we looked at simple ways to extend the mechanism used for specifying advice in KBKR. We found it especially helpful to be able to refer to certain "dynamic" properties in the advice, such as the average Q value, as a method of giving advice in a natural manner.

Our future research directions include the testing of our reformulated version of KBKR on additional complex tasks, the addition of more complex features for the advice language (such as multi-step plans) and the use of additional constraints on the optimization problem (such as directly including the Bellman constraints in the optimization formulation and the ability to give advice of the form "in these world states, action A is better than action B"). We believe that the combination of support-vector techniques and advice taking is a promising approach for RL problems.

Acknowledgements

References

[Andre and Russell, 2001] D. Andre and S. Russell, Programmable reinforcement learning agents, *NIPS '02*.

[Dietterich and Wang, 2001] T. Dietterich and X. Wang, Support vectors for reinforcement learning, *ECML '01*.

[Gordon and Subramanian, 1994] D. Gordon and D. Subramanian, A multistrategy learning scheme for agent knowledge acquisition, *Informatica* 17: 331-346.

[Kuhlmann et al., 2004] G. Kuhlmann, P. Stone, R. Mooney and J. Shavlik, Guiding a reinforcement learner with natural language advice: Initial results in RoboCup soccer, *AAAI '04 Workshop on Supervisory Control of Learning and Adaptive Systems.*.

[Lagoudakis and Parr, 2003] M. Lagoudakis and R. Parr, Reinforcement learning as classification: Leveraging modern classifiers, *ICML '03*.

[Laud and DeJong, 2002] A. Laud and G. DeJong, Reinforcement learning and shaping: Encouraging intended behaviors, *ICML '02*.

[Lin, 1992] L-J. Lin, Self-improving reactive agents based on reinforcement learning, planning, and teaching, *Machine Learning*, 8:293-321.

[Maclin and Shavlik, 1994] R. Maclin and J. Shavlik, Incorporating advice into agents that learn from reinforcements, *AAAI '94*.

[Mangasarian et al., 2004] O. Mangasarian, J. Shavlik and E. Wild, Knowledge-based kernel approximation. *Journal of Machine Learning Research*, 5, pp. 1127-1141.

[Noda et al., 1998] I. Noda, H. Matsubara, K. Hiraki and I. Frank, Soccer server: A tool for research on multiagent systems, *Applied Artificial Intelligence* 12:233-250.

[Price and Boutilier, 1999] B. Price and C. Boutilier, Implicit imitation in multiagent reinforcement learning, *ICML '99*.

[Stone and Sutton, 2001] P. Stone and R. Sutton, Scaling reinforcement learning toward RoboCup Soccer, *ICML '01*.

[Sutton and Barto, 1998] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press.