Relational Reinforcement Learning via Sampling the Space of First-Order Conjunctive Features

Trevor Walker   TWALKER@CS.WISC.EDU

Jude Shavlik   SHAVLIK@CS.WISC.EDU

Richard Maclin   RMACLIN@D.UMN.EDU

Computer Sciences Department, University of Wisconsin-Madison, 1300 University Avenue, Madison, WI 53706 USA

Abstract

We propose a novel method for reinforcement learning in domains that are best described using relational ("first-order") features. Our approach is to rapidly sample a large space of such features, selecting a good subset to use as the basis for our Q-function. Our Q-function is created via a regression model that combines the collection of first-order features into a single prediction. To control the effect of the random predictions we use an ensemble approach for our predictions, generating multiple Q-function models and then combining the results of these models into a single prediction. Experiments with our technique on an interesting reinforcement learning problem, the Keep-Away subtask of RoboCup, suggest that our method can learn to effectively predict Q-values for a challenging reinforcement-learning task.

1. Introduction

Many reinforcement-learning tasks involve domains that are inherently relational, often involving arbitrary numbers of interacting objects. These range from the number of blocks in a classical blocks-world problem to the number of players in a robotic soccer match to the number of interacting proteins in a regulatory network of a cell. A promising approach to reinforcement learning in such domains is to represent situations using an arbitrarily sized set of predicate-calculus facts, rather than engineering the world description into the more traditional fixed-length feature vector. We are investigating learning algorithms that can approximate the real-valued *Q*-functions used in reinforcement learning from these world-state descriptions expressed in predicate calculus.

———————

One previously investigated approach (Dzeroski, De Raedt, & Blockeel, 1998) for accomplishing this task is to use *relational regression trees* (Kramer, 1996; Blockeel & De Raedt, 1998). This approach partitions the space of possible state descriptions into discrete regions and assigns a constant to each region; for all future states falling into a region, their Q-value is estimated by this constant value. Another proposed approach is built on instance-based learning (Driessens & Ramon, 2003). In our new project, we are investigating the alternate approach of stochastically sampling the extremely large space of possible conjunctions of the predicate-calculus facts (e.g., on(X, Y) ∧ color(Y, blue) ∧ above(Y, Z) ∧ color(Z, red) - "*some block is on a blue block that is above a red block*").

Our approach is to rapidly sample a large space of such compound features, selecting a good subset to use as the basis for our Q-function. Our Q-function will then be a weighted combination of these conjunctive features; we choose these weights by using optimization methods that perform regularized linear regression (Shanno & Rocke, 1986) or regularized kernel regression (Vapnik, Golowich, & Smola, 1997) using a Gaussian kernel.

In Section 2 we present our approach for rapidly creating relational features and then creating a Q-function using these first-order features. In Section 3 we present our test domain, the Keep-Away subtask of RoboCup (Stone & Sutton, 2001). Section 4 presents our initial results demonstrating that our proposed approach can indeed learn a Q function. We then present conclusions and future directions for this work.

2. Our Approach

Table 1 shows our algorithm for generating a Q-function as an ensemble of predictors using stochastically generated first-order-logic features. Our approach is based on Srinivasan's (2000) method for stochastically generating random predicates for a search space. Srinivasan's method defines the set of possible predicates that could be created up to a given length *L*, and he presents a simple method for instantiating a random

member of that set of predicates. In our approach we generate an ensemble of **P** predictors of the Q-value for each action, where each predictor is made up of a weighted set of **M** randomly generated first-order logic features.

Table 1. **Algorithm for creating a Q-function as a weighted ensemble of predictors where each predictor is made up of a weighted set of randomly generated first order logic features. Parameters of the algorithm include *P*, the number of members in the ensemble, *M*, the number of features used in each predictor, *L*, the maximum number of predicates in a feature, *K*, the minimum number of differences between feature truth values, and the low and high percentages of examples covered to accept a feature.**

Given: A training set of *state/action/Q-value* triples of the Q-function to be learned

For each distinct action **a**, create an ensemble of **P** predictors by repeating the following **P** times:

- Select **M** features for this predictor by repeating the following **M** times:

  1. Stochastically create a feature consisting of at most **L** conjuncts of predicates, using a technique introduced by Srinivasan (2000).

  2. Reject the current feature and return to step 1 if the feature covers too small a percentage or too large a percentage of the states from the training set (e.g., if the feature is true in less than 25% of the training cases or true in more than 75% of the training cases).

  3. Reject the current feature and return to step 1 if the feature is not significantly different from any of the previous features for this predictor. This is determined by requiring that the feature, when compared to each of the features collected so far, have a different truth value in at least **K** of the states in the training set. That is, if we apply each feature to all of the states we will get a vector of truth values (one element for each state) for that feature. A new feature's vector must differ from each of the previous feature's vectors in at least **K** places.

  4. Accept the current feature and add it to the current predictor.

- Build a model using the **M** input features using an appropriate regression method (such as regularized linear regression or kernel regression using a Gaussian kernel).

To predict the Q-value of each action for a new state, we calculate a Q-value for each of the **P** predictors in the ensemble for that action and then take the average of the four middle predictions (e.g., if there are ten predictions, 1, 3, 40, 50, 70, 80, 80, 95, 356 and 1012 we would predict a Q value of 70, the average of 50, 70, 80 and 80). This approach allows us to avoid outliers that could easily skew an averaged prediction. The resulting predicted Q-value for each action would then be compared and the action with the highest Q-value chosen (during "exploitation" – when "exploring" reinforcement learners can choose suboptimal actions).

Our hypothesis is that, compared to relational tree regression, we can more rapidly create *Q*-functions that are more accurate. Randomly sampling the space of predicate-calculus conjuncts previously has been shown useful on classification tasks (Srinivasan, 2000), as has using weighted combinations of complex features generated from relational features (Popescul, et al., 2003).

Another advantage of our approach is that it is relatively easy to convert it to a semi-incremental version, since the model created in the last step can be incrementally adjusted using, say, gradient descent as each new training example arrives. Periodically a complete "batch" (i.e., non-incremental) learning run will be performed, for example, when the error over the last several predictions exceeds a threshold. Alternatively, this batch learning can be continually done in a parallel process that runs in the background.

Note that the algorithm in Table 1 could be varied in a number of ways. In the step where features are generated, we could use a local search based on how well the feature matches the data to look for a more effective feature, an approach similar to that explored by Zelezny, Srinivasan, and Page (2003). We could also explore other methods for building the regression model using the **M** input features. We expect that the family of methods based on our algorithm could produce fast and effective learning methods.

3. The Task: *Keep-Away*

We evaluate our approach on the *Keep-Away* subtask of simulated RoboCup soccer (Stone & Sutton, 2001). In this task, 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 keep the ball (the keepers learn, while the takers follow a hand-coded policy). Figure 1 gives an example of a *Keep-Away* game snapshot involving three keepers and two takers.

To simplify the learning task, currently learning occurs only in the specific situation where a keeper holds the ball. When no player has the ball, the nearest keeper pursues the ball and the others perform moves to "get open" (to be available for a pass). If a keeper is holding the ball, the other keepers perform the "get open" move.
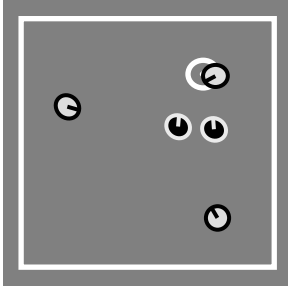
Figure 1. **A sample *Keep-Away* game in progress where there are three keepers (light gray with black outlines), two takers (black with gray outlines), and the ball (currently being held by the keeper in the upper right). A game continues until one of the takers holds the ball for at least 5 time steps or if the ball goes out of bounds (beyond the white lines).**

The learnable action choice then is whether to hold the ball or to pass it to one of the other keepers. Note that the passing steps require multiple actions in the simulation (orienting the player body then performing multiple steps of actual kicking), but these low-level actions are managed by the simulator and are not addressed in our experiments.

The policy of the takers is also fairly simple; if there are only two takers they both pursue the ball. When there are more than two takers, two pursue the ball and the others "cover" one of the keepers.

The *Keep-Away* task has been explored in other reinforcement-learning research (Stone & Sutton, 2001; Kuhlmann, Stone, Mooney, & Shavlik, 2004). One interesting question we intend to explore is whether the relational approach will make it easy for keeper behavior to scale from simpler problems (e.g., a "3 keepers vs. 2 takers" game) to a more complex game (e.g., 5 keepers and 4 takers).

4. Testing

The goal of our initial testing is to demonstrate that a Q function can be accurately learned using the method discussed in the previous section. To do this, we focus on predicting the expected TD(1) values of a set of hand-coded players playing "3 keeper, 2 taker" Keep-Away. In our experiments we demonstrate that our method learns a predictive model that is more accurate than a naïve predictor of the Q values.

4.1 Methodology

To generate training data for evaluation we record the states, actions taken, and Q values from a single agent, which we call the *learner*. To simulate exploration during initial reinforcement learning, the learner uses a random policy and plays the game with two "smart" keepers. The policy of the "smart" keepers is discussed below. The

two takers playing the game follow the policy discussed in Section 3.

The learner receives as input the state described using the predicates described in Table 2. We also provide as background knowledge a set of derived predicates based on the facts of a state, shown in Table 3.

The hand-coded policy of the two "smart" keepers attempts to choose an appropriate action to perform using the following rules:

1. If no taker is within a fixed distance, hold the ball.

2. Otherwise, calculate the widest passing lane to each of the other keepers and pass to the one with the widest passing lane.

3. If all passing lanes are zero width, randomly choose an action to perform.

Table 2. **Each state is described by a set of predicates capturing what the player can see about the world.**

---

keeper(P1) – the player is a keeper

taker(P1) – the player is a taker

dist_to(P1,P2,Dist,Game,Step) – the distance from player 1 to player 2 during a step of a game. There is one of these for each pair of players (takers and keepers).

dist_to_ball(P1,Dist,Game,Step) – distance from a player to the ball. One for each player.

controls_ball(P1) - whether a player controls the ball. One for each player.

x_position(P1,XPos,Game,Step) – the X position of the player on the field. One for each player.

y_position(P1,YPos,Game,Step) – the Y position of the player on the field. One for each player.

dist_to_top(P1,Dist,Game,Step) – distance of the player to the top border. One for each player.

dist_to_bottom(P1,Dist,Game,Step) – distance of player to bottom border. One for each player.

dist_to_left(P1,Dist,Game,Step) – distance of player to left border. One for each player.

dist_to_right(P1,Dist,Game,Step) – distance of player to right border. One for each player.

angle_between(P1,P2,Ang,Game,Step) – the angle between players 1 and 2 from the point of view of the player viewing the world. There is one of these for each pair of players other than the player whose view this describes.

---

teammate_dist_less_than(P1,P2,Dist,Game,Step)
true if the distance from player *P1* to *P2* is less than the fixed value *Dist*. We calculate a similar predicate to test than the distance is greater than the fixed value *Dist*, and do the same thing for the opponents.

teammate_dist_in_range(P1,P2,MinD,MaxD,Game,Step)
true if the distance from *P1* to *P2* is in the range *MinD* to *MaxD*. We also calculate a similar predicate for opponents.

some_teammate_dist_less_than(P1,Dist,Game,Step)
determines if there is some teammate closer than *Dist*. We also calculate a predicate that there is some teammate farther than the *Dist* and predicates that check the same thing for opponents. We further calculate predicates that test whether these conditions are true for *all* teammates or *all* opponents.

all_players_dist_greater_than(P1,Dist,Game,Step)
the player is at least *Dist* far from all players.

border_dist_less_than*(P1,Border,Dist,Game,Step)*
true if the given border is less than *Dist* away. We calculate a similar predicate that is true if the border is greater than that distance away.

some_border_dist_less_than(P1,Dist,Game,Step)
true if some border is less than *Dist* away. We calculate a similar predicate for *greater than*, and similar predicates to check if *all* the borders meet these distance requirements.

angle_btwn_teammates_less_than(P1,P2,Ang,Game,Step)
true if the angle between teammates is less than some angle *Ang*. We calculate a similar predicate for the greater than check, and these predicates for opponents and for any pair of players.

angle_btwn_tmtes_range(P1,P2,MinA,MaxA,Game,Step)
true if the angle between teammates is in the range between *MinA* and *MaxA*. We calculate a similar predicate for the greater than check, and these predicates for opponents and for any pair of players.

Distances are chosen from {1, 2, 3, 5, 7, 10, 15} and angles from {15, 30, 45, 60, 90, 120, 150 degrees}. The field size is 20x20.

We define the passing lane's width to be the base of an isosceles triangle with the apex at the position of the passing player, the midpoint of the base at the position of the receiver, and no takers contained in the area of the triangle. The widest passing lane is the lane with the widest possible base.

Using the random learner agent, two "smart" agents, and the standard takers, we generated several hundred games to be used as a training, tuning, and testing sets (as further explained below). In each game we record the state using the predicates described in Table 2 and focus on predicting the log of the TD(1) (Monte Carlo) estimate of the Q-value. We focus on the logarithm of the Q value because we noticed that the distribution of the log of the Q values closely approximates a normal distribution.

In order to measure the effect of having different amounts of data for training, we divided the training data into 3 groups of 100 games and then performed training using the first 100 games, the first 200, and the first 300 games. For this work, we focus on learning for the two *pass* actions rather than the *hold* action, since the passes are more interesting in most cases (a *hold* action generally does not change the state situation much since the player simply keeps the ball where it is). On average, there are about 3 passes in a game; for 300 games there are therefore about 1200 training examples.

Since we have several parameters that must be determined as part of the learning, we employ a tuning-set methodology. We collected the results from 200 additional games to (a) tune parameters and (b) estimate predictive accuracy. In our experiments, we divide this additional data into two groups of 100 games and then in one run used one group of 100 games as the tuning set and the other 100 games as the testing set. In the next run we use the second group of 100 games as the tuning set and the first as the testing set. Note that our use of tuning sets means that the total size of our training sets during our experimentation is actually 200, 300, and 400 (the training set plus the tuning set).

In our experiments we arbitrarily set the number of features (*M*) to be 50, the number of members of the ensemble (*P*) to 6, the maximum number of predicates in a feature (*L*) to 3, the minimum number of differences a new feature should have from a previously accepted feature (*K*) to 5, and the low and high percentage of examples that a feature must cover to 40% and 80%, respectively. We use the tuning set to choose the method to be used for combining our first-order features into a Q-value prediction.

In order to select the weights used to combine the features for a predictor, we use a regularized kernel regression method from Mangasarian and Musicant (2002). In their method, a regression problem is phrased as a support vector problem. The main parameters of their method are *C*, the value used to penalize the support-vector's "slack variables" (the larger this value, the higher the cost of mis-predicting training points), and μ, a term that is used to determine how much leeway is allowed in precisely fitting each point (the μ term is a global parameter that specifies a "tube" of leeway around the learned solution; training points that fall within this tube are not penalized in the expression being optimized). Since the problem is

phrased as a kernel problem, we can investigate different kernels for producing a final model.

In our experiments we use both a linear kernel and a Gaussian kernel. Since the Gaussian kernel has a parameter (the **σ** term indicating the width of the Gaussian), we also investigated different values of this parameter. In our experiments we look at four kernels: a linear kernel and three Gaussian kernels with **σ** values of 0.05, 0.5, and 5, respectively. For each of these four kernels, we built models using **μ** values of 0 or 0.5 and **C** values of 0.1, 1, or 10. The **μ** values are suggested by Mangasarian and Musicant (2002), while we chose the other values in an ad-hoc manner. The combination of parameter values and kernels leads to 24 different possible combinations. The tuning set of 100 games is then used to select the best model and the test set of 100 games is used to test generalization.

For testing, we measure how closely the learned model is able to approximate the actual Q values measured in our sample games. To determine this we are interested in the mean error of the overall test set and how it relates to the overall prediction. For this reason we calculate the mean of the absolute value of the errors and normalize it by dividing by the average value of the measurement we are predicting. For example, if we are predicting an item with values of 10 and 20, and predict 15 both times, the average error is 5, and the percentage difference obtained is 33% (5 as compared to an average predicted value of 15). For our experiments below, we calculate the average percentage difference and plot this as a function of the amount of training data.

As a baseline for comparison, we also compare to the simple "baseline" algorithm that merely always predicts, for each action, the average Q value in the training data for that action.

4.2 Results

Figure 2 shows the results of our initial experiments. Although not much better than our simple baseline algorithm, our proposed method does show a small but steady decrease in the error as the size of the training set grows. It takes less than a minute on average to select a set of relational features (using Prolog) and learn one regularized regression model (using Matlab).

5. Future Directions

There are several directions related to our approach that we plan to pursue in the future. First we plan to pursue methods to increase the accuracy of the Q-function estimates. To accomplish this, we plan to look at developing an enhanced feature-selection algorithm.
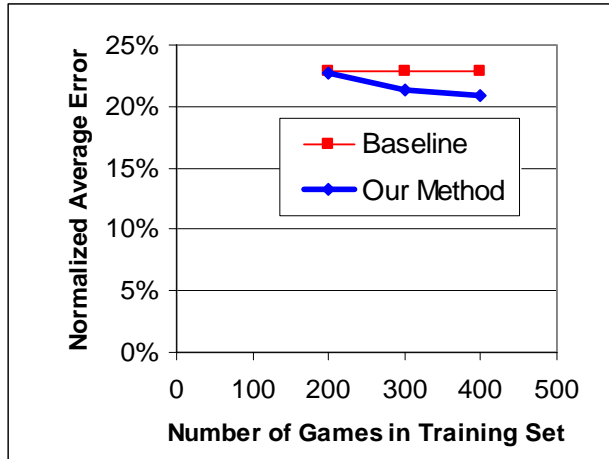

Figure 2. **Normalized error results for training our proposed Q-function learning method and the baseline method (see text). Results are shown for training set sizes of 200, 300, and 400 (where 100 of the training games are used for tuning).**

Enhanced feature selection could be accomplished in a many ways. Currently we use all the features that we generate, if they are non-redundant and cover between 40% and 80% of the training examples. This could be improved by generating more features and performing a local greedy search over those features to continually improve the accuracy of the Q-function estimate.

Once we have improved the accuracy of the Q-function estimate, we plan to perform traditional Q-learning using our relational algorithms. We plan to test the learning speed of the algorithm (as a function of the number of games played) and the rewards received. It is our hope that our relational method outperforms traditional non-relational methods, in terms of both training time and received rewards, especially as the size of the task (i.e., number of players on each team) increases.

After establishing the viability of our Q-learning, we will determine how well our algorithm scales to more complex problems. Due to the relational nature of our algorithm, we expect it to scale readily to larger *Keep-Away* field sizes with an increased number of keepers and takers with little or no retraining. We also hope that we can move from large fields to smaller, with little retraining.

We also plan to empirically compare to propositional learners, as well as other approaches for relational reinforcement learning (e.g., decision trees and instance-based methods), for on this test bed.

6. Conclusions

We presented a novel method for reinforcement learning in domains that constructs first-order features to describe the state of the environment. This approach has the advantage that learning can be scaled to larger problems and that generalization can occur across tasks with structural similarities. Our approach involves rapidly sampling a large space of first-order features and then selecting a good subset to use as the basis for our Q-function. Our Q-function is created by producing a regression model that combines the collection of first-order features into a single Q-value prediction. To prevent extreme predictions resulting from the possibility of generating skewed sets of random features, we generate an ensemble of models and then combine the predictions of the resulting models by taking the median prediction of the middle four models. Experimenting with our technique on an interesting reinforcement learning problem, the *Keep-Away* subtask of RoboCup, shows a small gain in accuracy in predicting Q values compared to a simple model, indicating that our proposed method could lead to effectively reinforcement learning.

Acknowledgments

References

Blockeel, H. and De Raedt, L. (1998). Top-down induction of first-order logical decision trees. *Artificial Intelligence:* 101, 285-297.

Driessens, K. and Ramon, R. (2003). Relational instance-based regression for relational reinforcement learning. *Proceedings of the 20th International Conference on Machine Learning (ICML '03),* Washington, DC.

Dzeroski, S., and De Raedt, L. and Blockeel, H. (1998). Relational reinforcement learning. *Proceedings of the 15th International Conference on Machine Learning (ICML'98)*, Madison, WI.

Kramer, S. (1996). Structural regression trees. *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, Portland, Oregon. G.

Kuhlmann, P. Stone, R. Mooney, and J. Shavlik (2004). Guiding a reinforcement learner with natural language advice: Initial results in RoboCup soccer. *Proceedings of the AAAI'04 Workshop on Supervisory Control of Learning and Adaptive Systems*, San Jose, CA.

Mangasarian, O., and Musicant, D. (2002). Large scale kernel regression via linear programming. *Machine Learning*, 46: 255-269.

Popescul, A., Unger, L., Lawrence, S., and Pennock, D. (2003). Statistical relational learning for document mining. *Proceedings of the 3rd International Conference on Data Mining*, Melbourne, FL.

Shanno, D., and Rocke, D. (1986). Numerical methods for robust regression: Linear models. *SIAM J. Sci. Stat. Comput.* 7: 86-97.

Srinivasan, A. (2000). A study of two probabilistic methods for searching large spaces with ILP. *Technical Report PRG-TR-16-00.* Oxford Univ. Computing Lab.

Stone, P. and Sutton, R. (2001). Scaling reinforcement learning toward RoboCup Soccer. *Proceedings of the 18th International Conference on Machine Learning*, Williams, Massachusetts.

Vapnik, V., Golowich, S. and Smola, A. (1997). Support vector method for function approximation, regression estimation, and signal processing. In M. Mozer, M. Jordan, and T. Petsche, (eds.), A*dvances in Neural InformationProcessing Systems 9*, pages 281-287.

Zelezny, F., Srinivasan, A. and Page, D. (2003). Lattice-search runtime distributions may be heavy-tailed. *Proceedings of the 12th International Conference on Inductive Logic Programming (ILP'02),* Sydney, Australia.