Practical Reinforcement Learning in Continuous Spaces

William D. Smart WDS@CS.BROWN.EDU
Computer Science Department, Box 1910, Brown University, Providence, RI 02912, USA

Leslie Pack Kaelbling LPK@AI.MIT.EDU
Artificial Intelligence Laboratory, MIT, 545 Technology Square, Cambridge, MA 02139, USA

Abstract

Dynamic control tasks are good candidates for the application of reinforcement learning techniques. However, many of these tasks inherently have continuous state or action variables. This can cause problems for traditional reinforcement learning algorithms which assume discrete states and actions. In this paper, we introduce an algorithm that safely approximates the value function for continuous state control tasks, and that learns quickly from a small amount of data. We give experimental results using this algorithm to learn policies for both a simulated task and also for a real robot, operating in an unaltered environment. The algorithm works well in a traditional learning setting, and demonstrates extremely good learning when bootstrapped with a small amount of human-provided data.

1. Introduction

Dynamic control tasks are good candidates for the application of reinforcement learning techniques. However, many of these tasks inherently have continuous state or action variables. Many existing reinforcement learning (RL) algorithms assume discrete sets of states and actions, which means that they are not directly applicable to these tasks. The continuous variables are often discretized, and the new discrete version of the problem is tackled with RL techniques. However, if we choose a bad discretization of the state or action space, we might introduce hidden state into the problem, making the learning of the optimal policy impossible. If we discretize too finely, we lose the ability to generalize and increase the amount of training data that we need. This is especially true when the task state is multi-dimensional, where the number of discrete states can be exponential in the state dimension.

It seems reasonable, then, to replace the discrete lookup tables of many RL algorithms with function approximators, capable of handling continuous variables with several dimensions and of generalizing across similar states. However, simply replacing the lookup table has been shown to cause learning to fail, even in benign cases (Boyan & Moore, 1995). The function approximation must be carried out carefully if the system is to succeed in learning a good control policy.

Although reinforcement learning techniques have been successfully applied to robots by several researchers, typically it would have been easier and faster to simply write a control program to achieve the task directly. Our goal in the work reported here is robust reinforcement learning augmented with information that is easy for a human expert to supply, resulting in a learning-system construction methodology that is of real practical value. This paper describes some initial steps in that direction.

This paper is mainly concerned with value function approximation and reinforcement learning on real robots. In addition to empirical studies of value function approximation (for example Boyan and Moore (1995) and Sutton (1996)), there is a growing body of theoretical work on this subject. Most relevant to this paper is the work by Thrun and Schwartz (1993) and by Gordon (1995) looking at some of the reasons why general function approximators fail when used for VFA. A variety of approaches based on gradient-descent methods (for example Williams (1992) and Baird and Moore (1999)) with guaranteed convergence and the ability to handle continuous spaces have also been proposed, but are of limited use to use because of our severe training data restrictions.

There have been a number of successes using reinforcement learning on real robots. Although many systems discretize the state and action spaces, there are several examples continuous spaces are used. Lin (1992) used recurrent neural networks for navigation tasks, in addition to teaching methods for accelerating learning that are similar to those reported in this paper. Mahadevan (1992) used real-valued abstracted sensor information, also to learn navigation tasks. There are also a growing number of robot-soccer learning systems (for example Asada et al. (1996) and later work from this group) using real-valued inputs for learning increasing complex control policies.

In the next section introduce HEDGER, an algorithm for safely approximating the value function used in Q-learning (Watkins & Dayan, 1992) for online learning of dynamic control policies. We also discuss and address some of the problems involved in using RL to learn such policies in an on-line setting.

2. Safely Approximating the Value Function

As we stated above, using value function approximation (VFA) with Q-learning is not simply a case of substituting in a supervised learning algorithm for the Q-value lookup table. Any supervised learning algorithm will suffer from a prediction error when trained on a real data set. However, when using such an approximator for VFA, these small errors can quickly accumulate and render the approximation useless.

In Q-learning, taking the action a from state s, resulting in a transition to state s' and a reward r, causes the (table-based) value function approximation to be updated according to

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

The value of the state-action pair (s,a) is updated according to the learning rate, α, the discount factor, γ, and the current approximation of the expected maximum value of the next state, s'. This is generally found by checking the values of every possible action from s' and selecting the largest one.

The key thing to note here is that the new value for $Q(s,a)$ is based both on the current (approximated) value for $Q(s,a)$ and the values of actions from state s'. This means that an error in any of these predictions will be incorporated into the update for the value of $Q(s,a)$. If this is the case, the new value learned by the approximation algorithm will be slightly incorrect. If $Q(s,a)$ is subsequently used to update the approximation for another state-action pair, the error can become larger still. This error can quickly dominate the approximation.

2.1 Reducing the Approximation Error

As both Thrun and Schwartz (1993) and Gordon (1995) note, one of the main sources of error when using function approximators to represent value functions is that they are prone to over-estimation. One major cause of this with many function approximators seems to relate to a problem known in the statistics literature as *hidden extrapolation*. The predictions of a function approximator are, in general, not valid for all queries. Unless we can make strong assumptions about the form of the function that we are learning, we can only make confident predictions in the area of the input space that is covered by the training data. Put another way, we can only use function approximators to *interpolate* training data, not *extrapolate* from it.

In a supervised learning setting, one could perform cross-validation experiments to determine the accuracy of the learned function in different areas of the input space. This would allow us to empirically determine where we should and should not allow queries. However, since we are iteratively estimating the value function, these tests do us no good. We can tell if the learned function models the training data, but we cannot tell if the training data itself, derived from the model, is actually correct.

One solution to this problem is to construct a convex hull around the training data and to only answer queries that lie within it. The problem then lies in how to efficiently construct this enclosing hull. Cook (1979) calls this structure the *independent variable hull* (IVH) and suggests that the best compromise of efficiency and safety is a hyper-elliptic hull, arguing that the expense of computing more complex hulls outweighs their predictive benefits. Determining if a point lies within this hull is straightforward. For a matrix, X, where the rows of X correspond to the training data points, we calculate the *hat matrix*

$$V = X(X'X)^{-1}X'.$$

An arbitrary point, $\vec{x}$, lies within the hull formed by the training points, X, if

$$\vec{x}'(X'X)^{-1}\vec{x} \leq \max_i v_{ii}$$

where v_{ii} are the diagonal elements of V. In the next section we show how we incorporate the IVH into our learning algorithm.

2.2 The HEDGER Prediction Algorithm

HEDGER is an instance-based learning algorithm, based on *locally weighted regression* (LWR). This is a variation of standard linear regression techniques, in which training points close to the query point have more influence over the fitted regression surface than those further away. A new regression is performed for every query point. This results in a globally non-linear model while retaining simple, locally linear models that can be estimated with well understood techniques (see Atkeson et al. (1997) for a comprehensive survey of LWR techniques and their use).

Training points in LWR are weighted according to a function of their distance from the query point. This function is typically a kernel function, such as a Gaussian, with a "width" parameter known as the bandwidth. Large bandwidths mean that points further away have more influence, resulting in a globally smoother approximated function. Small bandwidths allow more high-frequency variations in the learned model.

Algorithm 1 HEDGER prediction

```
Input:
    Query point, (s, a)
    LWR bandwidth, h
Output: Value function prediction, Q(s, a)
Concatenate s and a to form q⃗
Find set of points, K, closer to q⃗ than k_thresh
if |K| < k_min then
    Return "don't know" default value.
else
    Calculate the IVH, H, based on K.
    if q⃗ ∈ H then
        Calculate kernel weight, κ_i for each k_i ∈ K
            κ_i = exp(−(q⃗ − k⃗_i)^2 / h^2)
        Do local regression on K using weights κ_i
        Return fitted function f(s, a)
    else
        Return "don't know" default value.
```

We chose to base HEDGER on LWR techniques for a number of reasons. One of the most compelling is that LWR is an aggressive learning algorithm that is very fast to train. It can make reasonable predictions based on very little training data. This is important to us, since we are interesting in on-line learning from initially small amounts of training data. Since LWR retains all of its training data, we are also able to reevaluate our approximation based on the original training points. However, storage requirements can be large, and prediction times are longer than other, more compact, learning representations. However, with suitable optimizations, LWR seems to be a good choice for our purposes.

In order to use LWR for value function approximation, we must somehow combine the state and action vectors. This combined vector is then used as the input vector for the LWR system. Currently, we simply concatenate the state and action vectors.

Standard LWR techniques use all of the available training data for each query. However, for large data sets, this can become prohibitively expensive. Since most of the data points are likely to be far from the query point, they typically have little effect on the regression in any case. This allows us to make a computational optimization and apply the idea of an IVH to improve performance at the same time. Algorithm 1 sketches how HEDGER makes predictions. We set a distance threshold, k_{thresh}, based on the LWR bandwidth such that if a training point is further than this distance from the query point, it will have a negligible effect on the regression. All points closer that this threshold are included in K. The search for points to include in K is done relatively efficiently by storing points in a kd-tree structure (Atkeson et al., 1997). Typically, we need at least as many points in K as we have parameters to fit in the regression model. If we have fewer points than this, we return a default "don't know" Q-value. An IVH is constructed around K, and the query point is compared to this hull. If it is within the hull, we perform the LWR prediction as normal. If it is not inside the hull then, again, we return the default "don't know" Q-value. This results in behavior similar to initializing a table-based representation with an initial value before learning begins.

2.3 The HEDGER Training Algorithm

In this section, we describe how HEDGER uses experiences of the world to construct a value function approximation. Experiences are supplied as 4-tuples, (s, a, r, s'), representing action a taken from state s, resulting in a reward r and a transition to state s'. Algorithm 2 sketches how HEDGER uses an experience tuple to update its value function approximation. First, we obtain approximations for the current Q-value, and the maximum value from the resulting state, s'. We keep track of the points used in the LWR prediction of q for use later (line 3) and their associated weights (line 4). Line 5 calculates the new approximation of $Q(s, a)$, using the standard Q-learning update rule. This new value is then used as a normal supervised training point for the LWR subsystem. It is "learned" by simply storing it in memory. We then update each of the points in K, bringing its value closer to $Q(s, a)$ depending its proximity to (s, a), as measured its kernel weighting, κ_i. These updated values are changed directly in the points already stored in memory. They are not added as new, different points. Updating the approximation in this way allows us to keep the value function smooth, and to generalize the effects of updating one point to those around it. It also allows us to deal with the non-stationarity inherent in value function approximation.

Algorithm 2 HEDGER training

Input:

Experience, (s, a, r, s')
Learning rate, α
Discount factor, γ
LWR bandwidth, h

1: $q \leftarrow Q_{predict}(s, a)$ using Algorithm 1
2: $q_{next} \leftarrow \max_{a'} Q_{predict}(s', a')$
3: $K \leftarrow$ set used in calculation of q
4: $\kappa_i \leftarrow \exp(-(\vec{q} - \vec{k_i})^2 / h^2)$
5: $q_{new} \leftarrow q + \alpha\,(r + \gamma q_{next} - q)$
6: Learn $Q(s, a) \leftarrow q_{new}$
7: **for** each point, (s_i, a_i), in K **do**
8: $\quad Q(s_i, a_i) \leftarrow Q(s_i, a_i) + \kappa_i\,(q_{new} - Q(s_i, a_i))$

In line 2 we would like to find the best action (and its associated value) from the next state, s'. However, since we are in a continuous state space, we might never have visited that state before. Therefore, we must try to find the best action from the region of space close to the next state. The same optimization is also involved when executing the greedy policy defined by the learned value function. Such an optimization is difficult, especially when continuous actions are also involved. Currently this optimization implemented in HEDGER using a simple iterative algorithm, based on methods proposed by Brent (1973).

We begin by sampling n different actions from states close to the query state and predicting their Q-values. The algorithm then iterates, fitting a quadratic surface to the sampled points, and sampling a new point at the maximum of this fitted function. When two successive maxima are closer than some threshold, the algorithm terminates. This approach is similar to Newton's method, and consequently shares some of its problems. It is critically dependent on the initial sample points and can have problems with local maxima in the value function. However, we have found that initially sampling actions uniformly and applying this procedure works reasonably well in practice on the problems we have tried.

We perform one more computational optimization in HEDGER. We are interested in learning online, from small numbers of training runs and few data points. Thus, we must use any training data that we do get to its fullest. With this in mind, we use a technique proposed by Lin (1992), where we present experiences to the learner in reverse order. This allows immediate rewards to be propagated through the value function more efficient than presenting them in the order in which they are generated. It does mean, however, that no learning takes place during a training episode. All that we do during the episode is store the generated experiences, replaying them when it is over.

2.4 Supplying Initial Knowledge

Reinforcement learning systems often perform extremely poorly in the early stages of learning, being forced to act more-or-less at random until they acquire some experience of the world. This can present a serious problem in domains where the reward function is largely uniform, with informative rewards in only a few states. The problem is compounded if these states are difficult to reach by a random walk. The learning agent can spend a huge amount of time taking exploratory actions and learning nothing (since all rewards are the same) until it happens across an "unusual" state with a different reward value.

These problems are especially relevant in the robot control setting, especially when different actions have similar effects. Since robots are real mechanical devices, taking a succession of random actions might have no discernible effect due to mechanical slop in gear trains and the robot's inertia or momentum.

The solution that we propose is to provide the system with one or more example training runs from which to bootstrap the value function approximation. These runs can take the form of pre-recorded experience tuples, a piece of software to control the system or a human directly driving the robot. In each of these cases, HEDGER begins by learning passively, observing the experiences that the supplied initial policy generates. It uses these experiences to derive an approximation for the value function, as normal. After this initial training phase, the system switches to using the learned control policy. The initial knowledge bootstrapped into the value function approximation allows the agent to learn more effectively, and helps reduce the time spent acting randomly.

A key point to note is that the robot never tries to learn to replicate the training policy, which might be arbitrarily bad; it simply allows itself to be led through the world, while executing its own learning of the optimal policy. It follows that the initial example policies that are supplied to the robot do not have to be optimal. If we already knew the optimal policy, then learning would become pointless. The role of the supplied policy is not to show the agent what to do, it is to expose "interesting" areas of the state-action space. If we assume a mostly-uniform reward function, then we can define "interesting" as any area of the state-action space that generates an unusual reward.

Another approach commonly used to address the problem of value functions that are largely uniform is to set default Q-values that are overly optimistic. If the default Q-value for unknown states is higher than any actual Q-value, this will tend to drive the agent into areas of the state space that it has not yet seen. As soon as a state has been experienced, its value is lowered and it becomes less appealing than other, still unvisited states. However, since this approach encourages exploration in a somewhat random fashion it is not well-suited for robot control problems.

Supplying example trajectories through the state space in this manner leads us to a two-phase learning process. In the first phase, the example policy (either a human directly controlling the system, or a example piece of control code) is in control. The reinforcement learning system operates passively, bootstrapping information into the value function. After the value function has sufficient information, the second learning phase begins. The learned policy is in control of the system, and learning continues guided by this policy.

3. Experimental Results

In this section, we give experimental results of using HEDGER to learn policies for two control tasks. The first is a well-known simulation of an under-powered car trying to drive up a steep hill. The second is a corridor-following task involving a real robot.

3.1 The Mountain-Car Task

The mountain-car task involves trying to drive a car to the top of a steep hill, arriving with zero velocity. However, the car is not powerful enough to drive directly to the goal. Instead it must first reverse up the opposite slope in order to build up enough momentum to get to the top of the hill. The task is described by two continuous state variables (the position and velocity of the car), and and one action variable. We consider two formulations of this problem, with discrete and continuous actions. In the discrete case, there are three possible actions (backward, coast, and forward). We represent these actions as one variable that can take the values -1, 0 and 1, respectively. In the continuous case, the action is real-valued, lying between -1 and 1. The dynamics of the system correspond to those described by Singh and Sutton (1996). Reward is 0 everywhere, except at the top of the hill, where it is a linear function of velocity. Zero velocity yields a reward of 1, while maximum velocity results in a reward of 0. We used an ϵ-greedy exploration strategy, with ϵ set at 20%.

In all of the mountain-car experiments we begin a training episode from a randomly selected point in the state space. An episode runs for 200 time steps or until the goal is reached, whichever happens sooner. For all of the experiments, the learning rate, α, and the discount factor, γ, were both set to 0.8. If HEDGER returns a "don't know" value during greedy action selection, a random action is used.

Periodically, learning was disabled and evaluation runs were performed. Each evaluation consisted of 2500 episodes, starting from random points in the state space and following the current greedy policy. We evaluate the effectiveness of learning by looking at the average number of steps taken until the goal is reached, since this lets us compare our performance to other results reported in the literature. To provide a performance baseline, we ran standard tabular Q-learning on a finely discretized version of the problem until it converged. This resulted in a mean of 56 steps to the goal state. All comparisons between learning trials are significant at a 95% confidence level, unless otherwise stated.

Figure 1 shows results for the basic HEDGER system, trained on randomly-sampled states and actions. The performance of tabular Q-learning on a discretized state space with discrete actions is also shown for com-

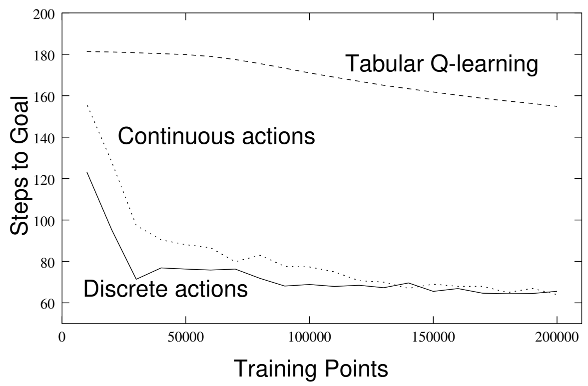

Figure 1. Basic HEDGER learning results for the mountain-car domain.

parison. Learning was evaluated after every 10000 training points. The main point to note from this figure is that HEDGER seems capable of dealing with continuous actions as well as with discrete ones. Initially, performance on the continuous-action system is worse than on the discrete-action one. However, as training progresses, the performances become indistinguishable. The final performance in Figure 1 is somewhat higher than the optimal of 56 steps. However, running the running the system for longer results in a slow convergence to a performance that is not significantly different from 56, after about 300,000 training points.

We can make learning faster by supplying some example trajectories to bootstrap the value function approximation. The examples were generated by users controlling a graphical simulation of the mountain-car system in real time. The example training runs were not limited to 200 steps, the users were constrained to use one of the three discrete actions and all runs started from the bottom of the hill. Eleven examples, generated by three different people were used. The average number of steps to reach the goal was 210, with the lowest being 106. The results of incorporating these example trajectories are shown in Figure 2. The top line again represents the system with continuous actions, while the bottom corresponds discrete actions. Again, the performance in the continuous-action case lags behind. However, after 200 phase-two training runs, the performance is not significantly different. The initial performance of the systems is already better than the average of the training trajectories. This underlines the point that we are not learning the actual policies used as examples, but simply using them to expose interesting parts of the state space.

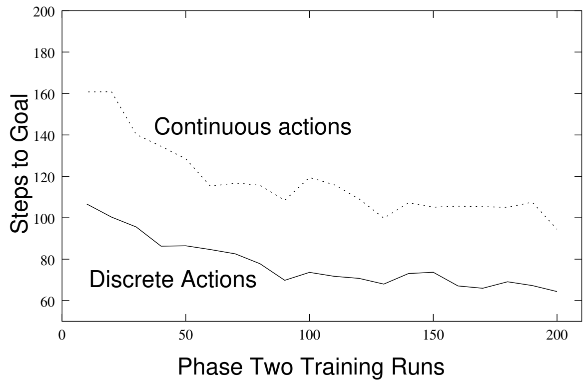

Figure 2. Basic HEDGER augmented with initial training examples.

It should also be noted that considerably less training data is being used in this experiment than in the previous one. A total of approximately 16,000 data points for the discrete case and 24,000 for the continuous case were used. These numbers compare with 200,000 training points for the results in Figure 1. The performance with this amount of data is better than in the previous experiment because it is sampled along trajectories in the state space. This, combined with presenting the training data in reverse order, allows the reward to propagate much more efficiently through the value function approximation.

Figure 3. Basic HEDGER without IVH calculations.

Figure 3 illustrates what happens when we disable the IVH checks on value function predictions. Both the discrete- and continuous-action systems perform significantly worse when this checking is removed. The final performance of both systems corresponds to a control policy that is only slightly better than one that selects actions at random. This demonstrates the crucial role of a knowledge of the area of learning coverage and refusal to propagate bad value estimates. An inspection of the actual values returned by HEDGER revealed that many of them were far too large, which is a symptom of hidden extrapolation, as described previously.

If we do not update neighboring points during training (lines 7 and 8 in Algorithm 2), performance on the continuous-action task decreases, as shown in Figure 4. However, the performance of the discrete-action system is not affected, and actually seems to be better initially. In the discrete case, we only sample actions with values of -1, 0 and 1. These are sufficiently far apart (according to our current distance metric), that altering the value of one of them is unlikely to have any effect on the others, even with region updating enabled. However, with continuous actions, ensuring the smoothness of the value function approximation seems to be important to ensure good value function prediction.

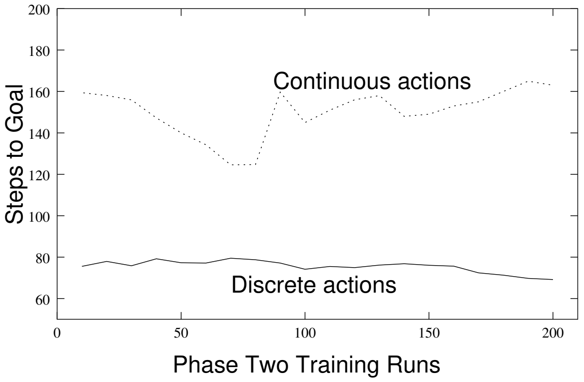

Figure 4. Basic HEDGER without local region updating during training.

3.2 The Corridor-Following Task

The corridor-following task involves learning to steer a real robot down a corridor towards a dead-end. We detect the corridor walls using data from a laser range-finder and use this to determine the angle of the corridor relative to the robot's current heading. We also calculate the robot's position with respect to the centerline of the corridor. These quantities, along with the distance to the end of the corridor, are used as the state input to HEDGER. The problem is to learn a steering policy that maps from these state variables to a rotation velocity. The robot's forward speed is controlled by a fixed policy. The reward is zero everywhere, except at the end of the corridor, where a reward of 10 is given. Our performance metric for this task is the number of time steps taken to traverse a given section of corridor, where each time step corresponds to roughly 0.3 seconds.

This delayed-reward formulation encourages the robot to get to the end of the corridor (where it receives reward) as quickly as possible. Thus, policies which spend less time zig-zagging from one side of the corridor to the other will be more successful.

The learning rate for these experiments was set to 0.2 and the discount factor to 0.99. Again, we used an ϵ-greedy exploration strategy with ϵ set to 20%. Instead of selecting a random exploratory action we added Gaussian noise to the greedy action. This was mainly to reduce the jerkiness of exploratory actions.

Twenty-five example trajectories were generated using a hand-coded corridor-following algorithm. This algorithm was developed quickly, with little attempt made at fine-tuning it. As a result, it took slightly over 106 steps to reach the goal, on average. The robot was also driven down the corridor under direct human control to provide an estimate on the best achievable performance. This yielded about 70 steps to the goal, on average. Learning was evaluated by placing the robot in a number of pre-specified starting positions and recording how long it took to reach the end of the corridor.

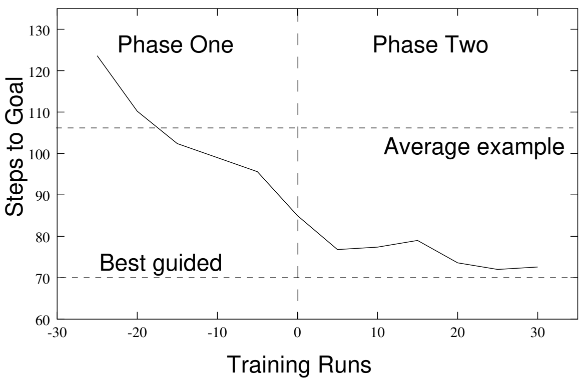

Figure 5. Performance on the real robot corridor-following task.

Figure 5 shows the performance of the robot during the first and second learning phases. During the first phase, while the supplied policy is in control, learning is particularly fast. Immediately after this initial training (training run 0 in the figure), the performance is significantly better than the supplied initial policy. As more and more training is done, the performance improves until it is not significantly different from the best achieved under human control.

At the end of the 30 phase-two training runs, the system had seen a total of approximately 5200 training points (including those generated by the example trajectories). The total time taken to write the initial control policy and perform the training was approximately two hours. in our experience this is, at worst, on a par with the time that would be needed to write and debug a policy that performs at a similar level.

4. Conclusions and Future Work

We presented HEDGER, an algorithm for safely approximating Q-learning value functions. We described the details of the algorithm and showed its effectiveness on two domains, the mountain-car task and corridor-following with a real robot. We also outlined a method for bootstrapping initial knowledge into the value function using supplied initial policies and looked at how a limited version of experience replay can help the value function approximation converge more quickly.

The algorithm works well in both domains, and achieves a performance that is competitive with the best published results. The key to this success seems to be the use of an IVH to enable safe value function predictions. Removing this feature causes the algorithm to fail to learn a good value function, illustrating the importance of guarding against propagating bad value estimates. Being conservative about predictions allows us to be more sure about the validity of the learned value function, but is also limits our generalization abilities. Even if the value function is well-behaved, we refuse to make predictions outside of the area that is supported by the training data. However, this does not seem to be a problem in the domains that we have looked at. Since we are following trajectories through the state space, we do not jump to areas in which we have no coverage in one step.

Several issues related to this work merit further attention. The concatenation of states and actions for use by the function approximator is unsatisfying. We use a standard Euclidean distance metric to compare these vectors, and we believe that some other metric might perform better. There are several improvements that we could, both to the representation and to the learning algorithm, including learning a dynamics model in tandem with the value function and a better greddy-action selection mechanism. This would allow us to use powerful algorithms from the reinforcement learning literature to make even more use of the limited data that we have available. Finally, the exploration/exploitation problem is one important issue that we have not currently addressed at all in this work. A logical continuation of the work presented here would be to use the learned value function model, along with its built-in knowledge of training data coverage, to generate more appropriate exploratory actions.

Acknowledgments

We would like to thank Cindy Grimm and Kee-Eung Kim for their help in providing example policies for the mountain-car task. We would also like to thank the reviewers for their helpful comments. This work was supported by DARPA contract #DABT63-99-1-0012.

References

Asada, M., Noda, S., Tawaratsumida, S., & Hosoda, K. (1996). Purposive behaviour acquisition for a real robot by vision-based reinforcement learning. *Machine Learning*, *23*, 279–303.

Atkeson, C. G., Moore, A. W., & Schaal, S. (1997). Locally weighted learning. *AI Review*, *11*, 11–73.

Baird, L., & Moore, A. (1999). Gradient descent for general reinforcement learning. *Advances in Neural Information Processing Systems: Proceedings of the 1998 Conference*. Cambridge: MIT Press.

Boyan, J. A., & Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. *Advances in Neural Information Processing Systems: Proceedings of the 1994 Conference* (pp. 369–376). Cambridge, MA: MIT Press.

Brent, R. P. (1973). *Algorithms for minimization without derivatives*. Englewood Cliffs, NJ: Prentice-Hall.

Cook, R. D. (1979). Influential observations in linear regression. *Journal of the American Statistical Association*, *74*, 169–174.

Gordon, G. J. (1995). Stable function approximation in dynamic programming. *Proceedings of the Twelfth International Conference on Machine Learning*. San Francisco: Morgan Kaufmann.

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

Mahadevan, S. (1992). Enhancing transfer in reinforcement learning by building stochastic models of robot actions. *Proceedings of the Ninth International Conference on Machine Learning* (pp. 290–299). San Francisco: Morgan Kaufmann.

Singh, S. P., & Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, *22*, 123–158.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference* (pp. 1038–1044). Cambridge, MA: MIT Press.

Thrun, S., & Schwartz, A. (1993). Issues in using function approximation for reinforcement learning. *Proceedings of the Fourth Connectionist Models Summer School*. Hillsdale, NJ: Lawrence Erlbaum.

Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292.

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