Real-time reinforcement learning in continuous domains

Jeffrey Forbes and **David Andre**
Computer Science Division, University of California, Berkeley
387 Soda Hall # 1776, Berkeley, CA 94720-1776
{jforbes,dandre}@cs.berkeley.edu

Introduction

In domains such as driving, there is rarely a known optimal trajectory. Instead the goal is to maximize general performance according to a given set of factors. Reinforcement learning (RL) is one method whereby the agent successively improves its control strategy through experience and feedback (reward) from the system. Reinforcement learning techniques have shown some promise in solving complex control problems. However, RL algorithms often do not scale well to nonuniform problems with large or infinite state and action spaces. We propose an architecture for making RL tractable on these realistic control problems with real time constraints. Our work has two main contributions. First, we present a method for maintaining an accurate continually updated estimate of our value function while keeping the update and query time bounded. Second, we show how to make the most of our time in the world by extending model-based methods with prioritized sweeping for continuous domains.

RL in continuous domains

The reinforcement learning framework that we assume in this paper is the standard *Markov Decision Process* (MDP) setup for reinforcement learning (Kaelbling & Moore 1996). We assume that at each point in time the environment is in some state s. At each step, the agent selects an action a, which causes the agent to transition to some new state t. Furthermore, the agent can receive some reward $r(s)$ that depends only on the state s and not on the past. We assume that the system is *Markovian*; i.e. that the probability $p(s'|s,a)$ of reaching state s' from state s by executing a does not depend on how the system arrived at state s. In this setting, the agent's objective is to maximize its *expected discounted accumulated reward*.

In reinforcement learning, the optimal mapping from states to optimal behavior (policy) is determined entirely by the expected long-term return from each state, which is called its *value*. Optimal decisions can be made in RL by learning the value of each state. The Q-function, $Q(s,a)$, is defined as the estimate of the expected long-term return of taking action a in state s, and taking the best known possible actions thereafter. Given a continuous state space, some sort of function approximation is necessary, since it would be impossible to represent the value function using a table. Generally, a parametric function approximator, such as a neural network, is used: $Q(s,a) \approx Q_w(s,a) = F(s,a,w)$ where w is a parameter vector with k elements.

In our preliminary experiments in applying this method to the task of lane following with the BAT (Forbes *et al.* 1997) simulator, we found that the technique had reasonable success for the task of lane following, quickly learning to stay centered in the lane with a simple reward function. Unfortunately, the RL controlled car was never able to stay entirely centered within the lane and after driving near the center of the lane for a period of time, it would "unlearn" and exhibit bad behavior. We suspect that the unlearning occurred because the distribution of states tends to focus more and more in the states just around the center of the lane, so there is forgetting of the other states. The intuition behind the forgetting problem is as follows. For a particular distribution of examples, the network weights will converge to a particular value in the steady state. If that input distribution shifts, the parameters will once again shift again. It can be shown that after this shift, the error on the previous examples increases.

An alternative approach is instance-based learning (Atkeson, Schaal, & Moore 1997) (also known as memory-based or lazy learning) that does not have this problem of forgetting because all examples are kept in memory. For every experience, the example is recorded and predictions and generalizations are generated in real-time in response to query. Unlike parametric models such as neural networks, lazy modeling techniques are insensitive to nonstationary data distributions. Generally, instance-based learning techniques are used in *supervised learning* where the true inputs and outputs are given throughout the training process. In reinforcement learning, the examples are only estimates of the value that may be very inaccurate initially. So while we improve our Q-estimates, we must update the out- of-date values in the database. Locally

Copyright © 2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

weighted regression can then can be used to represent the Q-function. Here, we perform what is known as a SARSA (state, action, reward, state, action) backup. We are in state s_t, perform action u_t, and we are given a immediate reward of r_{t+1} as we arrive in state s_{t+1}, and from there we will choose, according to our policy, action u_{t+1}. For every action, we can add a new example into the database: $Q_t = r_{t+1} + \gamma Q(s_{t+1}, u_{t+1})$. We update each Q-value Q_i in our database according to a temporal-difference update.

$$Q_i \leftarrow Q_i + \alpha[r_{t+1} + \gamma Q(s_{t+1}, u_{t+1}) - Q(s_t, u_t)] \nabla_{Q_i} Q(s_t, u_t)$$

As the agent spends more time in the world, the number of potential stored examples increases. The space required will increase linearly with the number of examples. The computation costs are even more limiting as we could potentially have to iterate through all stored examples on a query. We can speed up lookup time by using kd-trees. Lookup of the nearest neighbors can then be done in average-case $O(\log n)$ time where n is the number of examples. In order to operate continually in bounded time, the number of stored examples must be bounded.

Given a bound of K on stored examples, we can bound the amount of time for a query or an update. For problems with nonlinear, high-dimensional Q-functions, the performance will suffer if K is not set sufficiently high. Instance-based methods tend to break down in domains with high dimensions because of an exponential dependence of needed training data on the number of input dimensions: the curse of dimensionality. Luckily, most tasks only require high accuracy in small slices of the input space. For a robot with more than 8 degrees of freedom, it would be impossible for the robot to experience all significantly different configurations in a lifetime. The instance averaging or deletion scheme itself also needs to be efficient. We will need to reduce our example database regularly if not on every step.

We have two schemes for keeping the number of stored examples bounded. First, we do incremental instance-deletion by not adding any point whose value can already be predicted within some small ϵ. The second technique is instance-averaging – where we scan and reduce two neighbors into one exemplar. We chose the pair that is classified the best by their neighbors and where the neighbors can still be classified well or possibly better without the pair. The new exemplar has double the weight of the previous two examples. The score for each exemplar, Q_i can be computed as follows:

$$\text{score}_i = |Q_i - Q^{-i}(s_i, a_i)| + \frac{1}{K} \sum_{j \neq i} |Q(s_j, a_j) - Q^{-i}(s_j, a_j)|$$

A similar approach was proposed in (Salzberg 1991) where instead of averaging instances, he generalizes instances into nested hyperrectangles. While partitioning the space into hyperrectangles can work well for discrete classification tasks, it has flaws for regression. Most importantly, all of the points within a region almost never have the exact same value. In general, we only care about the highest Q-value for a particular state and thus we can give up precision for the values of suboptimal actions. An interesting future direction would be to design a heuristic for instance-averaging which takes into account the reduced utility of suboptimal Q values for a particular state.

Prioritized sweeping

For many real-time autonomous agents, although computation is cheaper than action, the time costs of computation must be taken into account. However, we do want to take advantage of any time available for computation, as actions in the environment may be expensive or even dangerous. By using a model of its actions, an agent can make the most of its actual experiences in the environment by doing simulated planning steps to determine the effects of a perceived change in the value of a state on the policy. For example, if the agent learns through experience that some state is painful or dangerous, this information can be computationally propagated back to the states leading to the painful state so that future painful episodes are avoided.

In the ideal case, the agent would compute the optimal value function for its model of the environment each time it updates it. This scheme is unrealistic since finding the optimal policy for a given model is intractable. Fortunately, we can approximate this scheme, if the approximate model changes only slightly at each step. This approach was pursued in DYNA (Sutton 1990), where after the execution of an action, the agent updates its model of the environment, and then performs some bounded number of value propagation steps to update its approximation of the value function.

In a small state-space, it is possible to store the model simply as a table of transition probabilities. Clearly, in a large or continuous domain, other forms of representation are required. In order to keep the amount of required learning as low as possible, we use a factored probabilistic model that takes advantage of local structure and is based on *dynamic Bayes networks* (DBNs)(Binder *et al.* 1997). In the full paper, we will give a more detailed treatment of the models.

Given that we can only do so much computation(planning) after each step in the world, the general model-based algorithm used by the model-based agent is as follows:

```
procedure DoModelBasedRL ()
(1) loop
  (2) perform an action a in the environment from
      state s, end up in state t
  (3) update the model
  (4) perform value-propagation for (s, a)
  (5) while there is available computation time
    (6) choose a state/action pair, (s', a')
    (7) perform value-propagation for (s', a'),
```

There are several different methods of choosing the state and action pairs for which to perform simulations.

One possibility is to take actions from randomly selected states. This was the approach pursued in the DYNA (Sutton 1990) framework. Another is to search forward from the current state/action pair (s, a), doing simulations of the N next steps in the world. Values can then be backed up along the trajectory, with those q-values furthest from (s, a) being backed up first. This form of lookahead search is potentially quite useful as it focuses attention on those states of the world that are likely to be encountered in the very near future.

However, there is another possibility. We can attempt to update those states where an update will cause the largest change in the value function. This idea has been expressed previously as prioritized sweeping (Moore & Atkeson 1993). In (Andre, Friedman, & Parr 1997), the idea was motivated and generalized by noting that the expected size of the update is well approximated by the gradient of the update rule for Q values. In our system we want to update those state/action pairs expected to have the highest changes in their value. We do this by calculating the gradient of the update rule error at a large sample of states (themselves chosen to be likely to be relevant either by lookahead search, backward search, or by choosing states likely to be affect by changes in the transition model). Then, we rank the states, and only do value-propagations for those states at the top of the queue – those with the highest expected change in value. By doing this, we take maximum advantage of possibly limited time for planning after each action in the real world. Also, note that because the number of examples we are storing is limited, the prioritized sweeping phase of learning has a bounded time as well, regardless of the length of time the agent is running in the world. In the longer version of the paper, we explain in detail both the calculation of the gradient of the update rule for our function approximator and parametric model and a novel algorithm for sampling states on which to calculate the priority.

Preliminary results

Our preliminary results have been promising. The instance-based RL methods have been able to learn good policies without forgetting. In order to test our instance-based Q-learning algorithm, we evaluated three function approximation algorithms in the cart centering domain. In order to simulate the effects of a changing task, we moved the start point closer to the goal as the agents completed the trials. We show the performance of the controllers in comparison the the optimal policy derived with a Linear Quadratic Regulator in Figure 1. The neural network performs relatively well, but when the start state is moved back to the initial starting position, the neural network controller has to relearn the value function for the outer states. The instance-based methods, locally weighted and kernel regression, had very little drop off in performance. LWR was slightly closer to optimal. Nevertheless, we generally use kernel regression in our subsequent problems because it is somewhat faster and more straightforward.

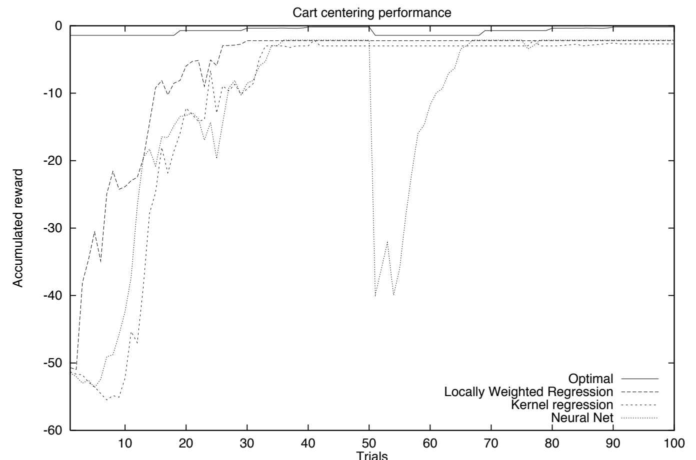

Figure 1: Q-learning with different types of function approximators versus the optimal policy on the cart-centering domain. The cart was started at various positions with 0 velocity. The first 20 trials at $+/-1$, the next 10 trials at $+/-0.5$, the next 10 trials at $+/-0.25$, and the final 10 trials at $+/-0.125$. At that point, the sequence was repeated again.

In the full paper, all of our new algorithms will be evaluated in the common domains of cart centering and pole balancing. We will also further test the methods' ability to learn acceptable policies for driving tasks such as lane following, lane changing, and car following all within the Bayesian Automated Taxi project simulator (Forbes *et al.* 1997).

References

Andre, D.; Friedman, N.; and Parr, R. 1997. Generalized prioritized sweeping. In *Advances in Neural Information Processing Systems*, volume 10.

Atkeson, C. G.; Schaal, S. A.; and Moore, A. W. 1997. Locally weighted learning. *AI Review* 11:11–73.

Binder, J.; Koller, D.; Russell, S.; and Kanazawa, K. 1997. Adaptive probabilistic networks with hidden variables. *Machine Learning* 29:213–244.

Forbes, J.; Oza, N.; Parr, R.; and Russell, S. 1997. Feasibility study of fully automated vehicles using decision-theoretic control. Technical Report UCB-ITS-PRR-97-18, PATH/UC Berkeley.

Kaelbling, Leslie P. an Littman, M. L., and Moore, A. W. 1996. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research* 4:237–285.

Moore, A. W., and Atkeson, C. G. 1993. Prioritized sweeping–reinforcement learning with less data and less time. *Machine Learning* 13:103–130.

Salzberg, S. 1991. A nearest hyperrectangle learning method. *Machine Learning* 6(3):251–276.

Sutton, R. S. 1990. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine Learning: Proceedings of the Seventh International Conference*. Austin, Texas: Morgan Kaufmann.