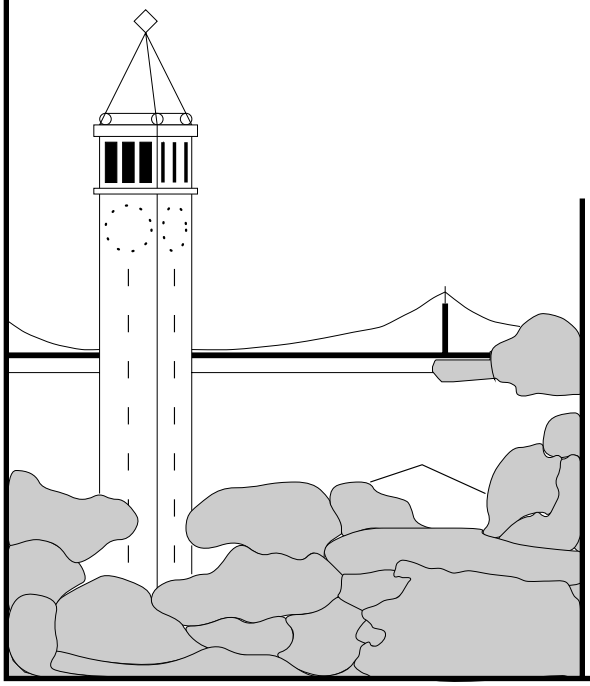
Practical reinforcement learning in continuous domains

Jeffrey Forbes and David Andre



Report No. UCB/CSD-00-1109

August 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Practical reinforcement learning in continuous domains *

Jeffrey Forbes and David Andre

August 2000

Abstract

Many real-world domains have continuous features and actions, whereas the majority of results in the reinforcement learning community are for finite Markov decision processes. Much of the work that addresses continuous domains either uses discretization or simple parametric function approximators. A drawback to some commonly-used parametric function approximation techniques, such as neural networks, is that parametric methods can "forget" and concentrate representational power on new examples. In this paper, we propose a practical architecture for model-based reinforcement learning in continuous state and action spaces that avoids the above difficulties by using an instance-based modeling technique. We present a method for learning and maintaining a value function estimate using instance-based learners, and show that our method compares favorably to other function approximation methods, such as neural networks. Furthermore, our reinforcement learning algorithm learns an explicit model of the environment simultaneously with a value function and policy. The use of a model is beneficial, first, because it allows the agent to make better use of its experiences through simulated planning steps. Second, the use of a model makes it straightforward to provide prior information to the system in the form of the structure of the environmental model. We extend a technique called generalized prioritized sweeping to the continuous case in order to focus the agent's planning steps on those states where the current value is most likely to be incorrect. We illustrate our algorithm's effectiveness with results on several control domains.

Keywords: *reinforcement learning, control,prioritized sweeping, instance-based learning*

*This work was supported by PATH MOU 83, PATH MOU 130, NSF PYI IRI-9058427, ONR N00014-98-1-0601, and ONR N00014-97-1-0941

1 Introduction

Imagine trying to learn how to safely drive a car using the standard methods of reinforcement learning. For many such methods, you would be asked to keep track of the value of performing an action for all of the possible configurations of your car, your hands, other cars, your tiredness, etc. In many reinforcement learning paradigms, you would be expected to crash the car many times before learning the proper way to drive. Of course, researchers in reinforcement learning have thought of and at least partially addressed these issues – but the car example serves as a tangible reminder of the difficulty of learning a skill in the real world. For example, it makes it clear that it is critical to use function approximators to handle the huge configuration space of driving. Given that real accidents are too costly a learning tool, it highlights the importance of having and using a model of the environment so that reasoning and planning can take place in the safety of simulation. Furthermore, because driving is a real-time enterprise, it stresses a balance between pondering and acting – waiting too long for a decision can easily cause accidents on a highway.

Motivated by the desire to apply reinforcement learning in continuous, real world situations, this paper presents a reinforcement learning system that utilizes an instance-based learner to represent the values of actions, a continuous dynamic probabilistic network to represent the model of the environment, and a technique to focus computation on planning episodes that are most likely to affect the success of later behaviors.

One approach to dealing with continuous environments is to simply discretize the world and use standard explicit state-space methods on the discretized model [McCallum, 1995]. Although useful for some domains, discretization doesn't scale well to high dimensional spaces, and, despite some work on adaptive discretization, choosing the discretization can be problematic. Coarse discretizations can yield unsatisfactory policies, while fine-grained representations can be intractable. Another approach is to use function approximators to represent the value of taking actions at different states in the world. Parametric models such as neural nets [Bertsekas and Tsitsiklis, 1989; Tesauro, 1989] or local averagers [Gordon, 1995] have often been used for this purpose. However, these parametric models have the problem of "forgetting", where all representational power can be co-opted by new examples. This is unfortunate for control problems, where dangerous situations may be seldom encountered once the behavior is reasonable. Of course, nonparametric regression methods are insensitive to nonstationary data distributions. Thus, we present a method for utilizing instance-based methods in a reinforcement learning context to store the expected value of taking actions in the environment.

As is reasonably well known in the reinforcement learning community, a model of the environment can be used to perform extra simulated steps in the environment, allowing extra planning steps to take place. In the extreme, the MDP in question could

be solved as well as possible given the current model at each step. Typically, however, this is intractable, and certainly undesirable when the model is not very accurate. Thus, most applications benefit from a middle ground of doing some extra planning steps after each actual step in the environment. The technique of Prioritized Sweeping [Andre *et al.*, 1997; Moore and Atkeson, 1993] chooses the states to update based on a priority metric that approximates the expected size of the update for each state. In this work, we apply the principles of generalized prioritized sweeping [Andre *et al.*, 1997] to a continuous state and action space using an instance-based value-function and a dynamic belief network (DBN) representation of the model. A potential disadvantage of using a model is that it must be learned, which can be complicated for a complex environment. However, by using a simple DBN for the model, we can take advantage of prior knowledge about the structure of the environment, and are only left with the problem of learning the parameters.

The structure of the paper is as follows. Section 2 first gives an overview of reinforcement learning and current value function approximation techniques. It then describes our instance-based representation, an algorithm for learning and maintaining the instance-based approximate value-function, and presents results of a simple comparison with neural networks. Section 3 briefly describes model based reinforcement learning, dynamic belief networks, and the representation that we use for the model. Section 4 describes methods for choosing which planning steps to take, including prioritized sweeping and how it applies to the continuous case when using DBNs for the model and instance-based methods for the value-function. Section 5 discusses the problems to which we have applied the system, and presents some results. Finally, Section 6 presents some conclusions and future work.

2 RL and Value Function Approximation

The reinforcement learning framework that we assume in this paper is the standard *Markov Decision Process* (MDP) setup for reinforcement learning [Kaelbling and 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. This is part of our assumption that the system is *Markovian* so 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 objective of the agent 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*, or expected reward to go, of every 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. In this paper, we will often refer to a state/action pair as a ψ, where a particular $\psi^i = \Psi(s, a)$ for some s and a. We will use $\Psi(s, a)$ to denote the state action pair composed of state s and action a. Furthermore, we say that $\psi^i_s = s$ and $\psi^i_a = a$.

When a transition from state s to state s' is observed under action a, the $Q(s, a) = Q(\Psi(s, a))$ estimate is updated according to the following rule:

$$Q(\Psi(s, a)) \leftarrow (1 - \alpha)Q(s, a) + \alpha[R + \gamma V(s')]$$

where $R(\Psi(s, a)$ is the immediate reward received, α is the learning rate, γ is a discount factor, and the value of state s', $V(s') = \max_{a'} Q(s', a')$. 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. The Q-function estimate is updated by first calculating the temporal difference error, the discrepancy between the value assigned to the current state and the value estimate for the next state: $E \leftarrow R + \gamma V(s') - Q(s, a)$. The weights in the function approximator are then updated according to the following rule: $\mathbf{w} \leftarrow \mathbf{w} + \alpha E \nabla_{\mathbf{w}} F(s, a, \mathbf{w})$.

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. Other neural network control algorithms control the distribution of examples explicitly to prevent such interference. 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 will increase.

An alternative approach is instance-based learning [Atkeson *et al.*, 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.

Memory-based techniques have four distinguishing parts: (1) the distance metric used to determine the similarity of two points, (2) the number of neighbors to consider

for a query, (3) the weighting of those neighbors, and (4) the way of using the local points to respond to a query.

A basic distance metric is Euclidean distance between a datapoint $\mathbf{x}$ and a query point $\mathbf{q}$ where $d_E(\mathbf{x}, \mathbf{q}) = \sqrt{\sum_j (x_j - q_j)^2}$. However, these methods still work with distance metrics which enable dimension scaling and skewing. In general, we will assume that all neighbors are being used. A typical weighting function is the Gaussian where the weight to the ith datapoint is assigned as $w_i = e^{\frac{-d(x_i,q)}{\tau_k}}$ where τ_k is the kernel width. Another function is the Epanechnikov kernel, $w_i^\epsilon = 3/4(1-d(x_i,q)^2)$, which has the optimal asymptotic mean integrated squared error for a one-dimensional kernel. In other words, the sum of the integrated squared bias and the integrated variance over the function is as low as possible.

Given a distance metric and a weighting function, the predictions in instance-based learning can be computed in a number of ways. In kernel regression, the fit is simply a weighted average of the outputs $y_i, i \in [1, n]$ of the neighbors: $\frac{\sum w_i y_i}{\sum w_i}$.

Locally weighted regression (LWR) is similar to kernel regression. If the data is distributed on a regular grid from any boundary, LWR and kernel regression are equivalent. For irregular data distributions, LWR tends to be more accurate [Atkeson *et al.*, 1997]. LWR fits a local model to nearby weighted data. The models are linear around the query point with respect to some unknown parameters. This unknown parameter vector, $\mathbf{b} \in \Re^k$, is found by minimizing the locally weighted sum of the squared residuals: $E = \frac{1}{2}\sum_{i=1}^n w_i(\mathbf{x}_i^T\mathbf{b} - y_i)^2$ where $\mathbf{x}_i \in \Re^k$ is the ith input vector, w_i is the corresponding weight, and $\mathbf{q}^T\mathbf{b}$ is the output for the query.

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 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 [Singh and Sutton, 1996]. 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.

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

For kernel regression,

$$\nabla Q(s_t, u_t) = \frac{w_i}{\sum_j w_j}$$

, and for locally weighted regression,

$$\nabla Q(s_t, u_t) \;=\; w_i \sum_{j=1}^{k} q_j G_{ji}$$

where G_{ij} is an element of the $G \;=\; (Z^T Z)^{-1} Z^T$ and q_i is the ith element of the query vector $\mathbf{q}$. For locally weighted regression, $\nabla Q(s_t, u_t) \;=\; w_i \sum_{j=1}^{k} q_j G_{ji}$ where G_{ij} is an element of the kxn matrix $G \;=\; ((WX)^T (WX))^{-1} (WX)^T$ and q_i is the ith element of the query vector $\mathbf{q}$. Instead of updating all elements, we can just update the nearest neighbors. We want to credit those cases in memory which may have had a significant effect on the $Q(s_t, u_t)$ that were within the kernel width (τ_k). Those cases make up the nearest neighbors $NN(s_t, u_t)$ of the query point. By updating only the neighbors, we can bound the overall update time per step.

There are some disadvantages to using these memory-based methods for value function approximation. First of all, memory and computation costs increase with more data. The space required to store the new examples increases linearly with the number of examples. We can restrict the number of examples somewhat by eliminating redundant examples. We should not enter any new examples which can be already be predicted by the database within some small range ϵ. Even so, most approaches using memory-based learning do not have a problem with running out of memory. The time costs are more limiting, since we can potentially have to iterate through all examples on each query. We can restrict the number of datapoints which are used in a query by only using the k nearest neighbors or by restricting the examples used to those within some function of the kernel width. Using data structures such as kd-trees, lookup of the nearest neighbors can be done in $O(\log n)$ average time, so that we look at the most relevant neighbors first [Moore, 1991]. Memory-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 learning methods can construct very accurate representations of the function where there are a number of examples. Very often, memory-based methods work very well for problems with a limited number (< 10) of input dimensions.

As discussed before, the optimal action in any state for a particular Q-function, $\pi(s) = \arg\sup_a Q(s, a)$. Finding the maximum value for an arbitrary function can be a hopelessly complex exercise by itself. In domains where the action is represented by a vector instead of a single scalar value, the maximum finding routine is accordingly much more complex. We generally assume a unimodal distribution for the Q-values with respect to a particular state for a particular state and scalar actions. We can then

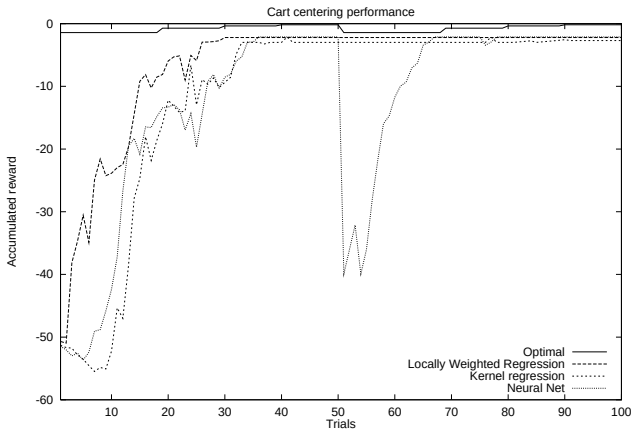


Figure 1: Q-learning with different types of function approximators versus the optimal policy on the cart-centering domain. The cart was started at various positions with 0 velocity. The first 20 trials at $+/-1$, the next 10 trials at $+/-0.5$, the next 10 trials at $+/-0.25$, and the final 10 trials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

use a gradient ascent strategy to find an approximation to the optimal action for our value function. Many algorithms discretize the action space, but for many control tasks, insufficiently quantized actions can cause oscillations and unstable policies [Kumar and Varaiya, 1986].

In order to test our instance-based Q-learning algorithm, we evaluated three function approximation algorithms in the cart centering domain. The domain is similar to the one described in Section 5, but it is a bit simpler (i.e. smaller world and smaller time step). 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.

3 Model-based Reinforcement Learning

The standard problem in model-based reinforcement learning is to learn a model of the environment simultaneously with an optimal policy. As the agent gains knowledge of the world's model, this information can be used to do various forms of planning, which can update the agent's value function without taking steps in the world. 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 computationally non-trivial. Fortunately, we can approximate this scheme, if we notice that the approximate model changes only slightly at each step. We can hope that the value function from the previous model can be easily "repaired" to reflect these changes. This approach was pursued in the DYNA [Sutton, 1990] framework, 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. Each value-propagation step in the standard model-based framework locally enforces the Bellman-equation by setting $\hat{V}(s) \leftarrow \sup_{a \in \mathcal{A}} \hat{Q}(s,a)$, where $\hat{Q}(s,a) = \hat{r}(s) + \gamma \int_{s'} p(s'|s,a)\hat{V}(s')$, where $p(s'|s,a)$ and $\hat{r}(s)$ are the agent's approximate model, and $\hat{V}$ is the agent's approximation of the value function. In the SARSA framework [Singh and Sutton, 1996] that we utilize in this paper, simulated value propagation steps are performed by taking simulated steps in the environment, using the learned model. To do a value propagation for a state/action pair ψ^i, we simulate the action ψ^i_a using the model, and calculate the resultant state ψ^j_s, where we then pick an action ψ^j_a, and calculate its value. Then, we can update the value of ψ^i as follows: $Q_{\psi^i} = (1-\alpha)Q_{\psi^i} + \alpha(R(\psi^i_s) + \gamma Q_{\psi^j})$.

In an explicit state-space, it is possible to store the model simply as a table of transition probabilities. Clearly, in a continuous domain, other forms of representation are required. Moore and Atkeson [Atkeson and Schaal, 1997] used a simple constrained parametric model of a robot arm in their learning from observation research. Other researchers in robotics have used Kalman-filter style models for navigation under uncertainty. In any complex reinforcement learning environment, however, there are often many state variables, and the transition matrices for a Kalman-filter might well get unwieldy. The curse of dimensionality can be somewhat sidestepped by using a model that takes advantage of local structure.

3.1 Dynamic Probabilistic Networks

We now examine a compact representation of $p(s'|s,a)$ that is based on *dynamic Probabilistic networks* (DPNs)[Binder *et al.*, 1997]. DPNs have gone by several other names as well, including Dynamic Bayesian Networks or Dynamic Belief Networks (DBNS). DPNs have been combined with reinforcement learning before in [Tadepalli and Ok, 1996], where they were used primarily as a means of reducing the effort required to learn an accurate model for a discrete world. [Andre *et al.*, 1997] also used DPNs in their work on extending prioritized sweeping.

We start by assuming that the environment state is described by a set of attributes. Let $X_1, \ldots, X_n$ be *random variables* that describe the values of these attributes. For now, we will assume that these attributes are continuous variables, with a predefined

range. An *assignment* of values $x_1, \ldots, x_n$ to these variables describes a particular environment state. Similarly, we assume that the agent's action is described by a random variables A_1. To model the system dynamics, we have to represent the probability of transitions $s \xrightarrow{a} t$, where s and t are two assignments to $X_1, \ldots, X_n$ and a is an assignment to A_1. To simplify the discussion, we denote by $Y_1, \ldots, Y_n$ the agent's state after the action is executed (eg the state t). Thus, $p(t|s,a)$ is represented as a conditional probability density function $p(Y_1, \ldots, Y_n \mid X_1, \ldots, X_n, A_1)$.

A DPN model for such a conditional distribution consists of two components. The first is a directed acyclic graph where each vertex is labeled by a random variable and in which the vertices labeled $X_1, \ldots, X_n$ and A_1 are roots. This graph specifies the *factorization* of the conditional distribution:
$P(Y_1, \ldots, Y_n \mid X_1, \ldots, X_n, A_1) = \prod_{i=1}^{n} P(Y_i \mid \varrho(i))$, where $\varrho(i)$ are the parents of Y_i in the graph. The second component of the DPN model is a description of the conditional probabilities $P(Y_i \mid \varrho(i))$. Together, these two components describe a unique conditional distribution. In the case of discrete variables, the simplest representation of $P(Y_i \mid \varrho(i))$ is a simple table. For continuous variables, we need a parameterized function to specify the conditional probabilities. In our case, we use a constrained form of the exponential family where the function $P(Y_i \mid \varrho(i))$ is represented by a Gaussian with a mean that is a linear combination over a set of functions on the inputs, and a variance over the same set of functions. That is, $(Y_i \mid \varrho(i) = \mathcal{N}(\theta^t \varphi(\varrho(i)), \sigma^2)$, where $\varphi(\varrho(i))$ is a set of functions over the parents of the given state attribute. For example, a node that was such a function of two parents X_1 and X_2 might have the form: $\mathcal{N}([\theta_0 + \theta_1 X_1 + \theta_2 X_2 + \theta_3 X_1 X_2], \sigma^2)$.

It is easy to see that the "density" of the DPN graph determines the number of parameters needed. In particular, a *complete* graph requires $O(N^2)$ parameters, whereas a sparse graph requires $O(N)$ parameters.

In this paper we assume that the learner is supplied with the DPN structure and the form of the sufficient statistics (the $\varphi(\varrho(i))$) for each node, and only has to learn the θ_{ij}, where i indexes the node Y_i and j the jth parameter of node Y_i. The structure of the model is often easy to assess from an expert. [1]

Learning the parameters for such a model is a relatively straightforward application of multi-variate linear regression. For the purposes of this paper, we assume a constant variance, although clearly this could be learned through experience using stochastic gradient or other related techniques. Without the need to learn the variance of the function at each node, we are left with simply determining the linear coefficients for each equation. By keeping track of the values of the sufficient statistics, we can learn the coefficients in an online manner at each node by solving a simple system of equa-

[1] [Friedman and Goldszmidt, 1998] describe a method for learning the structure of DPNs that could be used to provide this knowledge.

tions equal in size to the number of parameters (that can be derived simply from the derivative of the squared error function).

4 Prioritized Sweeping

As discussed above, one of the key advantages of having a model is that we can use it to do extra simulations (or planning steps). The general model-based algorithm that we use 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; let Δ_{Θ_M} be the change in the model
(4) perform value-propagation for $\Psi(s, a)$, let Δ_{Θ_V} be the change in the Q function
(5) while there is available computation time
(6) choose a state/action pair, ψ^i
(7) perform value-propagation for ψ^i, update Δ_{Θ_V}

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 possibility is to search forward from the current state/action pair ψ, 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 ψ 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 agent's 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 and Atkeson, 1993] and as Q-DYNA [Peng and Williams, 1993]. In [Andre *et al.*, 1997], the idea was generalized to non explicit-state based worlds through the principle that we should use the size of the Bellman error as the motivating factor. In the SARSA framework, we want to update those state/action pairs expected to have the highest changes in their value.

The motivation for this idea is fairly straightforward. Those q-values that are the most incorrect are exactly those that contribute the most to the policy loss. To implement prioritized sweeping we must have an efficient estimate of the amount that the value of a state/action pair will change given an other update. When will updating a state/action pair make a difference? First, when the value of the likely outcome states has changed. Second, when the transition model has changed, changing the likely output states. To calculate the priorities, we can follow [Andre *et al.*, 1997] and use the gradient of the expected update with respect to those parameters of our model

(Θ_m) and value function (Θ_v) that have recently changed. Let Θ represent the vector of all of our parameters (composed of $\Theta_m = \theta_{ij} \cup \theta_{ri}$ for the model, and Q_{ψ^i} for the Q values). Then, we estimate the expected change in value by the gradient of the expected update.

$$\begin{aligned} E[\Delta_{Q_{\psi^i}}] &\approx |\nabla E[\alpha(R(\psi_s^i) + \gamma \hat{V}[s'])] \cdot \Delta_\Theta| \\ &\approx |\nabla[\alpha(R(\psi_s^i) + \int_{s'} p(s'|\psi^i)[\gamma \hat{V}[s']]\, ds')] \cdot \Delta_\Theta|^2 \end{aligned}$$

Let us call $priority(\psi^{\mathrm{k}})$ the priority of a state/action pair ψ^k, and let $priority_{\mathrm{ri}}(\psi^{\mathrm{k}})$ be the priority contribution from the changes in reward parameter i, $priority_{\mathrm{ij}}(\psi^{\mathrm{k}})$ be the priority contribution from the changes in model parameter ij, and $priority_{\mathrm{Q}_{\psi^{\mathrm{j}}}}(\psi^{\mathrm{k}})$ be the priority contribution from the changes in the stored q-value ψ^j. Finally, let $\varphi_r(\psi^k)$ represent the set of sufficient statistics for the reward node in our DPN model calculated for state/action pair ψ^k, and let φ_i be the set of sufficient statistics for the node for state attribute i in the model. Then, by some straightforward algebra, we can derive the following expressions for the priority.

For the changes in the q function, we have:

$$priority_{\mathrm{Q}_{\psi^{\mathrm{j}}}}(\psi^{\mathrm{k}}) = \gamma \int_{\mathrm{s}'} \mathrm{p}(\mathrm{s}'|\psi^{\mathrm{k}}) \left[\frac{\mathrm{w}(\Psi(\mathrm{s}', \hat{\mathrm{a}}), \psi^{\mathrm{j}}) \Delta_{\psi^{\mathrm{j}}}}{\sum_{\psi^{\mathrm{i}}} \mathrm{w}(\Psi(\mathrm{s}', \hat{\mathrm{a}}), \psi^{\mathrm{i}})} \right] \mathrm{ds}'$$

Where s' ranges over the set of possible outcomes from state/action pair ψ^k, and $\hat{a}$ is the best action from state s'. For the changes in the model, we have

$$priority_{\mathrm{ri}}(\psi^{\mathrm{k}}) = \varphi_{\mathrm{r}}(\psi^{\mathrm{k}})_{\mathrm{i}} \Delta_{\theta_{\mathrm{ri}}}$$

$$priority_{\mathrm{ij}}(\psi^{\mathrm{k}}) = \gamma \int_{\mathrm{s}'} \mathrm{p}(\mathrm{s}'|\psi^{\mathrm{k}}) \mathrm{v}[\mathrm{s}'] \left[[\mathrm{s}'_{\mathrm{i}} - \theta^{\mathrm{t}}_{\mathrm{i}} \varphi(\psi^{\mathrm{k}}_{\mathrm{s}})] \varphi(\psi^{\mathrm{k}}_{\mathrm{s}})_{\mathrm{j}} \Delta_{\theta_{\mathrm{ij}}} \right] \mathrm{ds}'$$

Now there is, of course, a problem with the above equations. We have to evaluate the integrals in practice, which we can do by sampling. We must be careful, however, to insure that the calculation of priorities remains a cheap operation in comparison with the cost of doing a value update – otherwise it would be potentially advisable to

[2] In the above, note that we are working with the expected change in value of $Q(\psi^i)$. To calculate the likelihood that this change in the Q value affects the value of a state and thus the future policy, we have to take into account the likelihood that the action ψ^i_a is the chosen action. This is simply a multiplicative factor on the priority, and the equations above, without this factor, remain an upper bound on the size of the true priority. Furthermore, note that the final above equation is essentially the gradient of the $Q(s, a)$ part of the standard Bellman equation.

spend the time on extra random updates, rather than on calculating priorities. To get around this problem, we use the following approximation: $E[\hat{V}[s']] = \hat{V}[E[s']]$. This approximation essentially uses the value of the expected outcome of the probability distribution as the expected value. This will be a good approximation only in domains where the value distribution is roughly symmetric about the expected outcome. For example, if the value function is roughly linear about the mean, and the probability distribution is roughly Gaussian, then this approximation will be correct. In more complex or hybrid domains, doing more thorough sampling is probably required.

The result of this approximation is that we can use the mean outcome as the value for s', and the overall expression for the priority can be written as:

$$\begin{aligned} priority(\psi^{\mathrm{k}}) &= \varphi_r(\psi^k)^t \Delta_{\theta_r} + \gamma \sum_{\psi^j} \left[\frac{w(\Psi(s', \hat{a}), \psi^j) \Delta_{\psi^j}}{\sum_{\psi^i} w(\Psi(s', \hat{a}), \psi^i)} \right] + \\ & \gamma \hat{V}[s'] \sum_{i=1}^{N} \left[[s'_i - \theta_i^t \varphi(\psi_s^k)] \sum_{j \in parents(i)} \varphi(\psi_s^k)_j \Delta_{\theta_{ij}} \right] \end{aligned}$$

where N is the number of state attributes, and $w(\Psi(s', \hat{a}), \psi^j)$ is the kernel function from one stored q value to another.

There are several remaining algorithmic issues. First, we have to specify how we choose an action $\hat{a}$. We want to avoid the full optimization that takes place when choosing the optimal action at a state when we are computing an action $\hat{a}$. Instead, we use a weighted scaled combination of the actions of the neighbooring stored Q_{ψ^i}, such that the neighboring q values affect the action proportionally with respect to both their distance from the state s' and the relative goodness of their q-value. Second, we must explain how we choose states for which to calculate the priority. In previous work on prioritized sweeping [Moore and Atkeson, 1993; Andre *et al.*, 1997],it was assumed that priorities were calculated only for a small number of states. The predecessors of a state were known exactly, and it was only those states where a priority calculation was performed. In our domain, however, not only are there too many predecessor states to enumerate, but they are difficult to compute given the possible complexity of our model.

The fact that we have a priority estimate allows us to choose a large set of states for which to evaluate the priority, and then to only perform updates on a reasonably small number of these. We utilize a priority queue to implement this idea. We choose states on which to evaluate the priority in two ways: (1) by randomly creating a state vector according to the independent mean and variance of each attribute, and (2) by using stochastic inference to deduce likely predecessor states using the model. The second step would be computable for many simple conditional probability distributions, but because of the fact that we allow arbitrary linear combinations of functions over the

input, the exact backwards inference problem is difficult. For example, the functional form of the conditional probability of a node can be an arbitrary polynomial, and thus an exact solution is difficult or impossible. Thus, we use a simple MCMC technique to find state/action pairs that, with high probability, result in the given state.

For a given state s, we generate an initial sample ψ^k nearby s with Gaussian noise. We then use the Metropolis-Hastings algorithm [Metropolis *et al.*, 1953] to find state action pairs ψ^i approximately sampled from the distribution $p(\psi^i|s)$, where s is the resultant state of the transition from ψ^i. By Bayes rule, we have that $p(\psi^i|s) = \frac{p(s|\psi^i)p(\psi^i)}{Z}$, where Z is a normalizing constant. When applied to our situation, the Metropolis-Hastings algorithm can be expressed as follows: (1) choose a candidate next state/action pair ψ^c using a simple symmetric piecewise Gaussian proposal distribution. (2) Accept the candidate new state/action pair with probability $min\left[1.0, \frac{p(s|\psi^c)p(\psi^c)}{p(s|\psi^k)p(\psi^k)}\right]$. The proposal distribution does not show up in our acceptance probability because it is a symmetric distribution. This process gives us state/action pairs approximately sampled from the desired distribution for which we can then calculate the priority.

The algorithm for prioritized sweeping is very similar to the algorithm presented above for the general model-based case 4. The key difference is that priorities are used to determine which simulations are performed. After each change to the model or the value function, we call the update-priorities routine. The state/action pair with the highest priority is chosen to be simulated at each iteration of the "while time is available" loop.

procedure update-priorities ($\Delta_\Theta, state, pQueue$)
  W = PickRandomPsi(NumStates) $\cup$ UseMCMCForPsi(state)
  for all $\psi^i \in W$
    priority = RemoveFromQueueIfThere(ψ^i, pQueue)
    priority += CalculatePriorityForValue(Δ_{Θ_v},ψ^i)
    priority += CalculatePriorityForModel(Δ_{Θ_m},ψ^i)
    PlaceOnQueueWithPriority(psi^i,*priority*, pQueue)

5 Results

We tested our system on four continuous control domains: (1) cart-centering, (2) cart-pole balancing, (3) lane following, and (4) tailgating. For the cart-centering problem, we show the structure of the DPN, the function forms of the conditional density functions, and comparative results against Q-Learning.

In the cart-centering problem [Santamaria *et al.*, 1998], the task is to get a cart centered and stopped in a one-dimensional space as quickly as possible, but without extreme accelerations or velocities. The control action specifies a force that acts on the cart. The problem is fairly simple, but is mildly interesting because the reward function

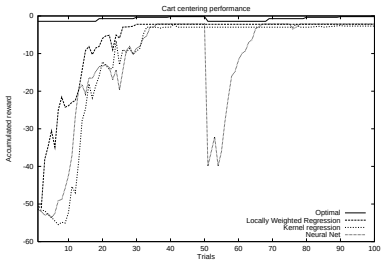


Figure 2: DPN for the cart centering problem.

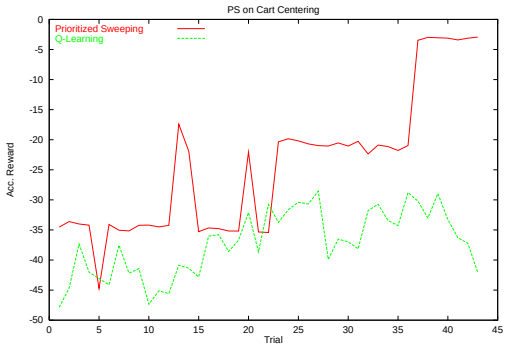


Figure 3: Average reward per trial for the second cart centering problem.

restricts the set of optimal solutions to the problem. Note that this problem is the same as the cart problem described in section 2, but is actually slightly more difficult because of a difference in the time scale. The DPN for this problem is shown in figure 2. The variables of this problem are x, the position of the cart; v, the velocity of the cart; $inWorld$, a variable indicating whether the agent is still within the confines of the world; a, the action specified by the agent; and R, the reward provided by the world. The sets of sufficient statistics, φ, for each variable, are as follows: $\varphi_x = \{x, v, a\}$, $\varphi_v = \{v, a\}$, $\varphi_{inW} = \{x\}$, $\varphi_a =$ specified by user, $\varphi_R = \{inW, x^2, a^2\}$.

In the cart-pole balancing problem, the task is to keep a pole attached to the top of a cart balanced and as vertical as possible. The goal state is to get the pole and cart to have zero velocity and have the pole be perfectly vertical. The action is force exerted on the cart, and the state variables are the position of the cart, x, the velocity of the cart, v, the angle of the pole, $angle$, and the angular velocity of the pole, $angledot$. The problem fully exercises the DPN representation that we use, as the functions $\sin(angle)$ and $\cos(angle)$ are members of the set of sufficient statistics for several of the state variables.

Lane following is the task of keeping a car centered within a lane on a curved highway within the BAT simulator [Forbes *et al.*, 1997].

The tailgating problem is the task of maintaining a minimal safe following distance from another car, again within the BAT simulator. Essentially, the task is never to tailgate, but to still maximize speed.

We have applied our system to these four problems, and the preliminary results are encouraging, indicating that the prioritized sweeping method has promise. Figure Figure 3 shows results comparing the prioritized sweeping approach with instance-based Q-Learning on the cart-centering problem. The graph shows average reward per trial over ten runs.

6 Conclusions and Future Work

We have presented an approach to reinforcement learning in continuous state spaces. We demonstrated a mechanism for maintaining the Q-function in instance-based function approximators, and presented an extension of the prioritized sweeping principle for continuous domains using instance-based function approximators for the Q-function and continuous dynamic probabilistic networks to represent the model. Our system is able to work with complex conditional density functions in the DPN because we utilize a stochastic sampling method (MCMC) to perform backward inference. In sum, the method shows promise for problems with continuous states and actions, but as always, there are are many active research directions that can build upon this work. For example, we are working on extending the system to deal with arbitrary Generalized Linear Models (GLiMs) in the DPN, and to partially observable worlds.

References

[Andre *et al.*, 1997] David Andre, Nir Friedman, and Ronald Parr. Generalized prioritized sweeping. In *Advances in Neural Information Processing Systems*, volume 10, 1997.

[Atkeson and Schaal, 1997] C. G. Atkeson and S. Schaal. Robot learning from demonstration. In *Proceedings of the Fourteenth International Conference on Machine Learning*, Nashville, Tennessee, July 1997. Morgan Kaufmann.

[Atkeson *et al.*, 1997] C. G. Atkeson, S. A. Schaal, and Andrew W. Moore. Locally weighted learning. *AI Review*, 11:11–73, 1997.

[Bertsekas and Tsitsiklis, 1989] D. C. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[Binder *et al.*, 1997] John Binder, Daphne Koller, Stuart Russell, and Keiji Kanazawa. Adaptive probabilistic networks with hidden variables. *Machine Learning*, 29:213–244, 1997.

[Forbes *et al.*, 1997] Jeffrey Forbes, Nikunj Oza, Ronald Parr, and Stuart Russell. Feasibility study of fully automated vehicles using decision-theoretic control. Technical Report UCB-ITS-PRR-97-18, PATH/UC Berkeley, 1997.

[Friedman and Goldszmidt, 1998] N. Friedman and M. Goldszmidt. Learning Bayesian networks with local structure. In Michael I. Jordan, editor, *Learning and Inference in Graphical Models*, Cambridge, Massachusetts, 1998. MIT Press.

[Gordon, 1995] Geoffrey J. Gordon. Stable function approximation in dynamic programming. In *Proceedings of the Twelfth International Conference on Machine Learning*, Tahoe City, CA, July 1995. Morgan Kaufmann.

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

[Kumar and Varaiya, 1986] P.R. Kumar and Pravin Varaiya. *Stochastic systems: Estimation, identification, and adaptive control*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[McCallum, 1995] Andrew Kachites McCallum. *Reinforcement Learning with Selective Perception and Hidden State*. PhD thesis, University of Rochester, December 1995.

[Metropolis *et al.*, 1953] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1091, 1953.

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

[Moore, 1991] Andrew W. Moore. An introductory tutorial on kd-trees. Technical Report 209, Computer Laboratory, University of Cambridge, 1991. Extract from A. W. Moore's Phd. thesis: Efficient Memory-based Learning for Robot Control.

[Peng and Williams, 1993] J. Peng and R. J. Williams. Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, 2:437–454, 1993.

[Santamaria *et al.*, 1998] Juan C. Santamaria, Richard C. Sutton, and Ashwin Ram. Experiments with reinforcement learning in problems with continuous state and action spaces. *Adaptive Behavior*, 6(2), 1998.

[Singh and Sutton, 1996] Satinder Singh and R.S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158, 1996.

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

[Tadepalli and Ok, 1996] Prasad Tadepalli and DoKyeong Ok. Scaling up average reward reinforcmeent learning by approximating the domain models and the value function. In *Proceedings of the Thirteenth International Conference on Machine Learning*, Bari, Italy, July 1996. Morgan Kaufmann.

[Tesauro, 1989] G. Tesauro. Neurogammon wins computer olympiad. *Neural Computation*, 1(3):321–323, 1989.