Report Number: WUCSE-2005-19

2005-05-01

Manifold Representations for Continuous-State Reinforcement Learning

Robert Glaubius and William D. Smart

Reinforcement learning (RL) has shown itself to be an effective paradigm for solving optimal control problems with a finite number of states. Generalizing RL techniques to problems with a continuous state space has proven a difficult task. We present an approach to modeling the RL value function using a manifold representation. By explicitly modeling the topology of the value function domain, traditional problems with discontinuities and resolution can be addressed without resorting to complex function approximators. We describe how manifold techniques can be applied to value-function approximation, and present methods for constructing manifold representations in both batch and online... **Read complete abstract on page 2.**

Manifold Representations for Continuous-State Reinforcement Learning

Robert Glaubius and William D. Smart

Complete Abstract:

Reinforcement learning (RL) has shown itself to be an effective paradigm for solving optimal control problems with a finite number of states. Generalizing RL techniques to problems with a continuous state space has proven a difficult task. We present an approach to modeling the RL value function using a manifold representation. By explicitly modeling the topology of the value function domain, traditional problems with discontinuities and resolution can be addressed without resorting to complex function approximators. We describe how manifold techniques can be applied to value-function approximation, and present methods for constructing manifold representations in both batch and online settings. We present empirical results demonstrating the effectiveness of our approach.

2005-19

Manifold Representations for Continuous-State Reinforcement Learning

Authors: Glaubius, Robert; Smart, William D.

May 1, 2005

Abstract: Reinforcement learning (RL) has shown itself to be an effective paradigm for solving optimal control problems with a finite number of states. Generalizing RL techniques to problems with a continuous state space has proven a difficult task. We present an approach to modeling the RL value function using a manifold representation. By explicitly modeling the topology of the value function domain, traditional problems with discontinuities and resolution can be addressed without resorting to complex function approximators. We describe how manifold techniques can be applied to value-function approximation, and present methods for constructing manifold representations in both batch and online settings. We present empirical results demonstrating the effectiveness of our approach.

Manifold Representations for Value-Function Approximation in Reinforcement Learning

Robert Glaubius and William D. Smart
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO 63130
United States
{rlg1,wds}@cse.wustl.edu

1 Introduction

Reinforcement learning (RL) has shown itself to be a successful paradigm for solving optimal control problems. However, that success has been mostly limited to problems with a finite set of states and actions. The problem of extending reinforcement learning techniques to the continuous state case has received quite a bit of attention in the last few years.

One approach to solving reinforcement learning problems relies on approximating the value function of an underlying Markov decision process. This solution technique is referred to as value-based reinforcement learning, and will be discussed in more detail in Section 2. In this paper, we propose a finite support method for value-based reinforcement learning based on the theory of manifolds. By constructing a representation of the underlying state space topology, we can avoid some of the pitfalls that other value-based techniques experience.

Consider the world in figure 1 as a two-dimensional navigation domain. Typically this state space is represented as the product space of two real intervals, which is homeomorphic to a disk. What such a representation doesn't capture is the fact that the state space is actually topologically a cylinder, due to the obstacle in the center of the environment. A failure to account for this may lead function approximators on this domain to generalize through the obstacle, resulting in a poor approximation. Figure 2 illustrates our approach to modelling the state space of the world in Figure 1. By breaking the world up into a set of simpler (rectangular) models, we can preserve the appropriate connectivity of the state space. Note that the rectangles on opposite sides of the obstacle do not overlap. The intuition behind our approach is that representing the world by the union of many simple models we can capture the underlying topology of the state space. We can then use this model to prevent the problem of inappropriate generalization.

This paper is organized as follows. In Section 2, we go into more detail regarding value-based Reinforcement Learning and the continuous state space problem. Section 3 discusses related work. In sections 4 and 5 we cover our approach in depth, including several approaches to constructing manifold representations and methods for using these representations for value function approximation. We provide empirical results of our approach on the Mountain car benchmark domain [4] in section 6.

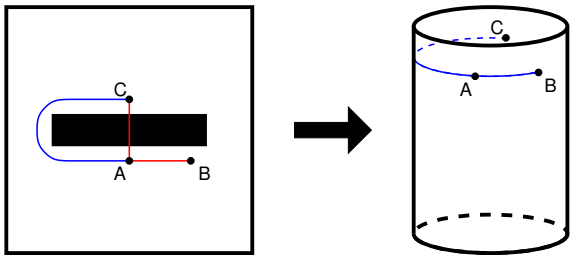

Figure 1: A simple 2-dimensional navigation domain with a center obstacle [6]. The distance between A and B is the same as that between A and C (shown in red), but is innacurate because of the obstacle. The blue line corresponds to the distance on the surface of the cylinder (Right) that more accurately represents distances in this world.

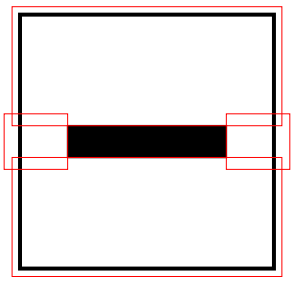

Figure 2: Rectangle-based decomposition of the obstacle world.

2 Background

Reinforcement learning problems are most often modeled as Markov decision processes. A Markov decision process consists of a set of states S, a set of actions A, a reward function R, and a transition function T. We make the simplifying assumption that every action is applicable in every state. An MDP is said to be finite whenever S and A are finite.

The reward function $R : S \times A \times S \to \mathbb{R}$ maps a state, action, state triple (s, a, s') to the immediate utility of taking action a from state s and ending up in s'. For notational compactness, we will denote $R(s, a, s')$ as $R^a_{ss'}$, following Sutton and Barto [17]. The transition function $T(s, a)$ maps each state-action pair to a probability distribution over possible next states. We denote the probability of arriving in state s' immediately after taking action a from state s as $P^a_{ss'}$. An MDP is deterministic if $P^a_{ss'}$ is always 0 or 1 for every pair of states s and s' and action a.

A policy π recommends actions from each state of an MDP. For any policy, there exists a value function $V^\pi(s)$, defined as follows, where $\pi(s_t) = a_t$:

$$V^\pi(s) = E\left\{\sum_{t=0}^{\infty} \gamma^t \sum_{s' \in S} P^{a_t}_{s_t s'} R^{a_t}_{s_t s'} \mid s_t = s\right\} \tag{1}$$

The value function then is the expected sum of discounted future rewards obtained when following the policy π. The optimal policy is that which maximizes the value function; such a policy is guaranteed to exist.

The discount factor $\gamma \in [0, 1]$ adjusts how myopic the agent is in weighing future rewards; one interpre-

tation of γ is that it is the prior probability of surviving until the next time step. The optimal policy π^* is such that $V^{\pi^*}(s)$ is maximal; such a policy is guaranteed to exist for any MDP. Finding this optimal policy is the objective in a reinforcement learning problem.

Dynamic programming solutions exist for computing the value function of a finite MDP. However, directly computing the value function requires knowing the transition function, which we often do not have access to. For instance, if the agent is learning to fly a helicopter, factors such as wind make it difficult to model the transition function. In this paper we assume that we have no model of the transition function. One popular solution to this problem is to compute the state-action value function $Q(s, a)$, which is defined for a fixed policy π as follows:

$$Q^\pi(s, a) = \sum_{s' \in S} R^a_{ss'} + \gamma \sum_{s' \in S} P^a_{ss'} V^\pi(s') \tag{2}$$

The optimal Q function can be computed by sampling the transition function using Watkins's Q-learning [20], or the Sarsa learning rule [15]. Given the optimal Q^* function, the optimal policy $\pi^*(s) = \underset{a \in A}{\text{argmax}}\, Q^*(s, a)$.

Many real-world problems are most intuitively modeled with real-valued state. For example, in the case of a robot moving about in the real world, we are likely to have at least three real-valued state features corresponding to position and orientation. There are a couple of detriments to applying algorithms for the finite MDP case to the continuous case. The methods for determining the optimal value function discussed above rely on making an arbitrarily large number of iterative updates to a table of values. It is infeasible both to keep a one-to-one table of values in the continuous case; it is also highly unlikely that the agent will ever be in the same state twice. In this case, we must resort to value-function approximation.

Value-function approximation techniques rely on a relatively compact representation of the value function. The most straight-forward approach is to aggregate states into a finite collection of subspaces, and treat these subspaces as the problem state space. One problem with this approach is that the problem is no longer Markov; discretization methods also tend to scale poorly as the number of dimensions (state features) increases. These methods do have the benefit that the learning algorithms used for the finite case can be applied, although the loss of the Markov property does remove convergence guarantees.

A second class of value-function approximation techniques use parameterized function approximators as drop-in replacements for the value table. For example, we might use one artificial neural network with one input for each state feature, and a single real-valued output. We could then train one network for each discrete action, and use these to represent the Q function. Unfortunately, well-behaved function approximators may diverge even for simple environments [4].

One reason that parameterized function approximators may fail in general is that they rely on Euclidean neighborhoods in order to drive generalization. This can be a problem if the true value function varies too much in this neighborhood; this can occur when the neighborhood is divided by an obstacle, for example.

In this paper we present a method for applying value-function approximation to problems with continuous, bounded, infinite state. We assume that actions are discrete and finite. This method for VFA is based on constructing a topological model of the state space: a manifold representation. Our goal is to approximate the connectivity of the state space with respect to the transition function by covering the reachable portion of the state space with a set of charts, and then placing simple function approximators on each chart. In the next section, we describe the construction of these representations and how these can be used for value-function approximation.

3 Related Work

Reinforcement learning in continuous state spaces has received a great deal of attention recently. Much of this effort has been focused on approximating the value function.

Variable resolution discretization [11, 19, 12] partition the state space into a finite set of states. Munos's method requires that the transition and reward functions are known in advance; both Munos's VRD and Continuous U-Tree may perform poorly in case of a poor discretization. Moore and Atkeson's parti-game adaptively partitions the space, and so avoids these problems, but requires that the user provides a greedy controller as an input.

CMACs [1], or Tile Coding [16] have also been applied to value function approximation in the continuous state case. This approach uses several coarse, exhaustive partitions of the state space as a discrete, locally linear representation of the value function. This allows a greater effective resolution with a relatively small number of predictive features, but both this and the variable resolution discretization methods scale poorly as the dimension of the state space increases.

Memory-based approximators [2, 3] generalize from a set of stored training instances. Usually, a stored training instance influences the value of nearby states as a function of distance from the training instance. This generalization based on a distance metric can lead to poor approximation when, say, generalizing between two close together states that are seperated by a wall.

Recently, Ratitch and Precup [14] introduced a finite-support value-based reinforcement learning strategy based on sparse distributed memories. This approach is similar to our online chart allocation, discussed in Section 5.1. Their framework dynamically allocates predictive features, and allows tuning of the number and position of features.

Boyan and Moore [4] demonstrated empirically that value-function approximation may diverge even for simple reinforcement learning problems, while Gordon [7] proved the non-divergence of SARSA(λ) for averaging approximators. Szepesvári and Smart [18] proved the convergence of approximators obeying a particular interpolative property under a fixed exploration policy.

4 Manifolds

We propose a value function approximation strategy based on the theory of manifolds. Loosely speaking, our approach constructs a manifold representation of the state space that is composed of local subspaces. The manifold representation is defined by these subspaces and their overlaps; each of these subspaces is paired with a function approximator that is trained only on training instances that originate in the paired subspace.

In order to help build an intuition for what a manifold representation does, consider an atlas of the earth's surface. The atlas contains a set of maps, each showing some small portion of the surface. It is possible to plot a course between two cities on the same map directly; to plot a course between cities on different maps that intersect you can choose a point in the overlap, and navigate from the first city to that point, and from that point to the goal city. Essentially, the atlas is a manifold; Euclidean distance is valid within a single map, but not between maps due to differences of scale.

More formally, each point on x on a manifold M has a Euclidean neighborhood, denoted U_x. These U_xs are the local subspaces mentioned above, or the portions of the world represented by a map in our atlas. For U_x to be Euclidean, there must be a homeomorphism $\varphi_x : U_x \to \mathbb{R}^n$. φ_x is called a chart, and can be thought of as a coordinate transform from manifold to the n-dimensional reals. This chart is analogous to the maps in the examle above; the collection of these charts $\Phi = \{\varphi_x : x \in M\}$ is called an atlas.

We are interested in the subclass of manifolds that are differential. This means that it is possible to perform analysis (calculus) on the manifold surface. Formally, a differential manifold is a manifold such that if $U_x \cap U_y \neq \emptyset$, then $\varphi_y \circ \varphi_x^{-1}$ is a C^∞ function from $\mathbb{R}^n$ to $\mathbb{R}^n$. Essentially, this means that there is a well-defined tranformation between the coordinate systems of U_x and U_y. For instance, the earlier atlas example is an example of a differential manifold. Suppose two maps overlap, and that a centimeter on one map corresponds to d_1 kilometers, and d_2 kilometers in the other. Then the composite map is just a scale by a factor of $\frac{d_1}{d_2}$ to go from the first to the second map.

5 Constructing and Using Manifolds

The two significant aspects of our manifold representation are the value functon approximation and the chart allocation. Section 5.1 describes in detail several approaches to actually constructing our manifold representations. Section 5.2 describes our approach to embedding function function approximators on a manifold representation. This covers both using an embedded approximator for prediction as well as training an embedded approximator.

Our manifold representation consists of a set of chart domains covering the state space S of a reinforcement learning problem. For reasons of computational tractability we restrict the class of domains we consider to hypercubes. The chart φ for a chart domain then is just a translation and scale from the state space to the open unit cube $(-1, 1)^n$.

5.1 Chart Allocation

The first problem that we need to address is how we go about constructing a manifold representation. We approach this chart allocation problem in both a batch setting and an online setting. In the batch setting the manifold is allocated prior to training, while in the online setting chart allocation is interleved with training.

5.1.1 Batch Allocation

All of our chart allocation schemes assume that we have access to the system dynamics and reward only through sampling, which is the case in most realistic settings. We will discuss three methods for batch allocation, two of which assume a large available sample set, or alternatively, the ability to arbitrarily sample the environment.

The simplest batch allocation does not require any samples, but only that we know some bounded subspace $U \subseteq \mathbb{R}^n$ such that $S \subseteq U$. We can then cover U with fixed-width charts placed uniformly at random. We refer to this simple strategy as "random allocation". Of course, this strategy does not take into account any properties of S beyond the requirement that S is bounded.

A second strategy is walk-based allocation. In this allocation strategy charts are allocated until the space is covered; each chart is allocated by first choosing an uncovered state in the state space, and then taking a series of random walks from that state. The smallest axis-aligned bounding box containing all observed states along each of these walks becomes the domain of a new chart. Each walk is terminated after either a minimum number of steps or a minimum path length has been met; the path length of a trajectory is just the sum of Euclidean distances between the start and endpoint of each observed transition along the walk.

This allocation strategy requires quite a bit of access to the state space. We must be able to find uncovered points in the state space, and have the ability to arbitrarily sample the state space. Unfortunately, this is not a reasonable requirement in most realistic environments; however, this strategy will later prove useful in the discussion of online allocation methods.

The third batch chart allocation strategy actually describes a family of strategies. Given a sample set Ξ, where each sample ξ is a tuple consisting of (s, a, r, s'), the chart allocation algorithm uses Ξ to iteratively grow charts based on some measure of sample-set similarity between an initial chart domain and adjacent portions of the state space.

For example, one such sample set similarity measure is the expectation that the step size in each region exceeds some threshold θ. Step size $\sigma(\xi) = \sqrt{\sum_{i=1}^{d}(s_i - s'_i)^2}$ is just the Euclidean distance between the start and end states of the sample $\xi = (s, a, r, s')$. The expectation that step size exceeds θ for some $U \subseteq S$ is defined as follows, where Ξ_U is the set of samples in Ξ with initial state in U:

$$E[\sigma(\xi) > \theta | \Xi_U] = \frac{|\{\xi \in \Xi_U : \sigma(\xi) > \theta\}|}{|\Xi_U|} \tag{3}$$

In other words, this expectation is just the number of samples with start state in U with step size exceeding θ normalized by the total number of samples with start state in U. This statistic is useful for measuring the locality of the transition function in the sense that it is a measure of how likely taking an action in U will move the agent a "long" distance in terms of θ.

Given such a sample set statistic, the idea behind this growth-based chart allocation is that while we can find an uncovered state s, we place a small initial chart domain U centered about s. We then try to extend each face of the hyperrectangle U; if the distribution of our selected statistic is not significantly different between U and this region that we try to include, then U is extended in that direction. If the distributions are significantly different, then U will not be extended, and we will not try to extend that face again. The process of extending faces of U continues until every face has failed exactly once, at which point we allocate a chart with domain U. Chart allocation continues until the state space is covered.

When used with this algorithm, the step size statistic above allows the extension of a possible chart domain only when it is either similarly likely or unlikely that the agent will take a large step from both the chart domain and the region that chart would contain if extended. Significant changes in this statistic are likely when there are obstacles in an environment for example. Significant differences in this statistic also occur in regions of transition from short to long steps; in this case the θ parameter acts as a threshold for chart boundary placement along the transition from short to long steps.

This final method is useful in that it often places charts where it seems intuitive to do so. However, it has the unfortunate drawback of being extremely sample intensive, and is likely to be feasible only for cases in which a simulator of the actual environment is available to the agent.

Aside from issues of sample complexity, one of the problems that these methods share is an exponential dependence on state space dimension. Any method that covers all of S is likely to suffer greatly the curse of dimensionality. For example, any covering with fixed-width charts of width δ requires at least $\frac{1}{\delta}^n$ charts to cover the unit hypercube $(0, 1)^n$.

Online allocation is one solution to this potential problem. One characteristic of high-dimensional control problems is that often the actual state space lies on a low-dimensional manifold. This means that there may be large portions of seemingly valid state vectors that will never be observed in practice, so covering these states with charts is a waste. By only covering the observed states, we are likely to only allocate charts that are necessary. This is the idea behind online allocation.

5.1.2 Online Allocation

In online allocation setting, the samples collected by an agent interacting with the environment are used to construct the manifold representation incrementally on-the-fly. Whenever the agent encounters a state that

is not on its current manifold representation, it will begin allocating a new chart that will be added to the manifold; samples observed off of the manifold can be used first to update the representation, and then later to train the approximators embedded on the manifold.

We have two schemes for performing chart allocation in the event that the agent walks off of its manifold representation. These are a fixed-size allocation and a walk-based allocation analogous to the random and walk-based batch allocations discussed earlier. Both of these methods are inexpensive to compute in terms of both time and samples.

In the fixed-size allocation the agent simply covers any uncovered state with a fixed-size chart domain centered at that state. This strategy is parameterized on the chart extent in each dimension; in practice we use hypercube charts, e.g., the extent in each dimension is the same.

The second online allocation scheme is also fairly straightforward. Suppose the agent follows a trajectory $s_0, a_0, s_1, a_1, \ldots s_t, a_t \ldots$ through the environment. If the state s_t is the earliest state along the trajectory not on the current manifold, we then allocate a chart domain that is the smallest bounding box containing $\{s_{t-1}, s_t, \ldots, s_{t+k}\}$, where k is a user-defined parameter. This is then repeated for the remaining trajectory until the entire trajectory is covered.

It would be a straightforward modification to allocate charts that bound walks of length θ rather than walks of k steps. The choice between these methods most likely depends on the environment; if there are regions in which step size is small, bounding the number of steps is likely to result in a large number of small charts, while bounding the path length is likely to result in charts of more uniform volume.

5.2 Manifold-Based Value-Function Approximation

In the previous section we described several methods for allocating a set of chart domains covering the state space S. In this section we describe how to use this representation for value-function approximation.

The idea of our approach is that, for every chart φ in our atlas Φ, we will define two functions: a blend function b_φ and a local embedded approximator f_φ. The global function approximation at a point will then just be a blended combination of the predictions of these local approximators. A bit more formally, given a chart $\varphi : U_\varphi \to \mathbb{R}^n$, we have a C^∞ blend function $b_\varphi : S \to \mathbb{R}$ that is defined to be non-zero only on states in U_φ, with all derivatives vanishing at the boundary of the chart domain. The local embedded approximator $f_\varphi : \mathbb{R}^n \times A \to \mathbb{R}$ predicts the value of states in U_i after the change of coordinates defined by φ has been applied. See the appendix for a definition of the blend function that we employ in our implementation.

Given the functions described above, the value-function approximated at a point is defined as follows.

$$Q(s,a) = \frac{\sum_{\varphi \in \Phi} b_\varphi(s) f_\varphi(\varphi(s), a)}{\sum_{\varphi \in \Phi} b_\varphi(s)} \tag{4}$$

In other words, the prediction at a point is just a linear combination of the contribution of each local approximator at that point. Since each blend function is defined to be zero everywhere outside of its associated chart domain, only local approximators embedded on charts containing s contribute to the prediction of $Q(s,a)$.

That covers the method for value-function prediction given our manifold representation, but leaves the question of updating the prediction given transition samples. We adapt both the Q-learning and SARSA update equations to update manifold-based approximators.

Given a transition sample (s, a, r, s'), the Q-learning update equation is defined as follows.

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma \max_{a' \in A} Q(s', a')] \tag{5}$$

This update rule moves the value of $Q(s,a)$ towards $\delta = r + \gamma \max_{a' \in A} Q(s',a')$ at a rate governed by the gain α. The straightforward application of this rule in manifold-based approximation is to supply the training instance (s,δ) to each approximator f_φ such that $s \in U_\varphi$. The details of this update are dependent upon the approximator f_φ.

Given a transition sample (s,a,r,s',a'), where a' is the action taken from state s', the SARSA update equation is defined as follows.

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha[r + \gamma Q(s',a')] \tag{6}$$

Again, we adapt this training rule to the manifold-based case in the straightforward way, defining $\delta = r + \gamma Q(s',a')$.

In order to illustrate how these updates work, suppose that for each action a, $f_\varphi(\cdot,a)$ is an unthresholded linear unit with a vector of weights w of dimension n. The perceptron training rule for updating these weights is

$$w \leftarrow w + \alpha(\delta - f_\varphi(s,a))s \tag{7}$$

6 Experiments

6.1 Experiment Design

We ran a detailed series of experiments to evaluate the performance of manifold-based value-function approximation on the Mountain car domain [4]. The emphasis of these tests was to examine the effect of the various batch and online chart allocation strategies on the quality of learned policies. We compare the performance of the manifold-based methods with a tile-coding approximator used by Sutton [16]. We expect our performance to be comparable to that of the tile-coding approximator, but with substantially less hand-tuning of the approximator representation. The tile-coding approximator consists of 10 9x9 state space partitions.

We use a dense reward function description of the mountain car problem, with an action penalty of -1 on each time step except at the goal. The agent receives a reward of 0 at the goal, which is to reach the top of the rightmost hill. The position and velocity ranges are normalized to (0,1).

In both batch and online settings we experimented with fitting constants on charts, as well as unthresholded linear units as the local embedded functions. The former method of fitting a constant on a chart uses the usual table-based update equations; the only real distinction between fitting a constant and table-based approximation is that we allow overlapping chart domains, while table-based approximation's predictive features are cells in an exhaustive partition.

All of our experiments train the agent over a series of 10,000 trials. In each trial, the agent is placed at the environment rest state – the valley between left and right hills. The agent then follows an ϵ-greedy exploration policy, where $\epsilon = \frac{1}{trial}$. Each trial is terminated when the agent reaches the goal or after 4000 steps have been taken, whichever occurs first. Experiences observed during the trial are stored until the trial is terminated, at which time the entire trajectory is "replayed" from end to beginning, using the SARSA update equation.

We present results for batch allocation with random, walk-based, and growth-based allocations. Examples of each of these allocations are shown in figure 3.

In our experiments with growth-based chart allocation we used the step size statistic discussed in Section 5.1. We varied the step-size threshold θ, using settings 0.01, 0.05, and 0.1. The initial charts have a width of 0.01, and candidate expansion regions are defined by extending a chart by 0.01 along either the

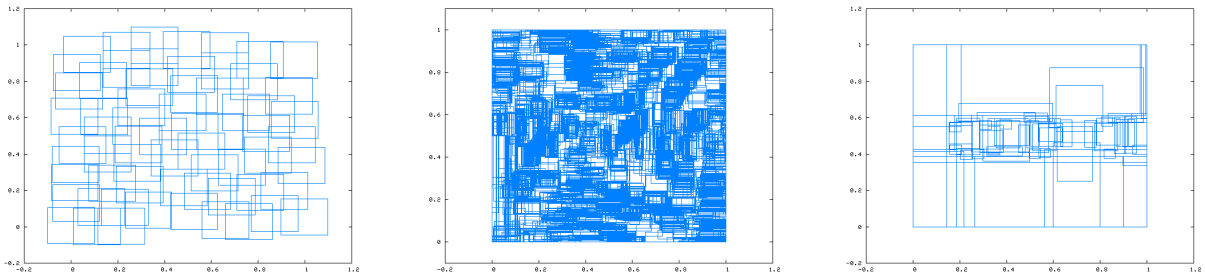

Figure 3: Random (left), walk-based (center), and growth-based (right) allocations for the mountain car domain. The random allocation uses charts with side-length 0.1. A path length threshold of 0.1 and a single walk was used to generate the walk-based allocation. The growth-based allocation was generated using $\theta = 0.1$.

Random	239
Walk	3491
Growth	251

Table 1: Number of charts allocated by three batch allocation schemes

position or velocity dimension on each growth iteration. We assumed the ability to arbitrarily sample the environment in order to obtain these allocations, as was also the case for walk-based allocation.

We experimented with both the fixed-size and walk-based online allocations as well. In the tests with fixed-size online allocation we use chart half-widths of 0.05 and 0.1. In the walk-based online allocation tests we varied the number of steps that would be used to construct a chart from 5 to 10.

6.2 Batch Allocation Results

Table 1 shows the number of charts allocated by each of the three batch allocation schemes. The first, random, uses 0.1×0.1 charts. The walk-based numbers are for charts constructed around one walk threshholded at length 0.1. The growth allocation uses the expectation measure 3 with $\theta = 0.05$.

We show here the results for linear approximation on batch-allocated manifolds. Results for constant approximation are not shown for batch allocations, as the global approximation was not stable except for very small charts. As can be seen in Figure 4, the performance of our manifold-based methods is competitive with the tile coding approximator.

The tile coding approximator uses 810 tiles with no hashing, therefore it is significant that the random allocation is competitive with only 239 charts. The fact that the tile coding learner finds a good policy significantly faster than the walk-based manifold learner can be explained by the larger number of predictive features in the walk-based allocation.

Growth-based allocation results are not shown; neither constant nor linear approximation are sufficiently stable to obtain meaningful results. The true value function for the mountain car problem is well-represented with linear approximation on many of the larger charts, resulting in unstable, poor performance.

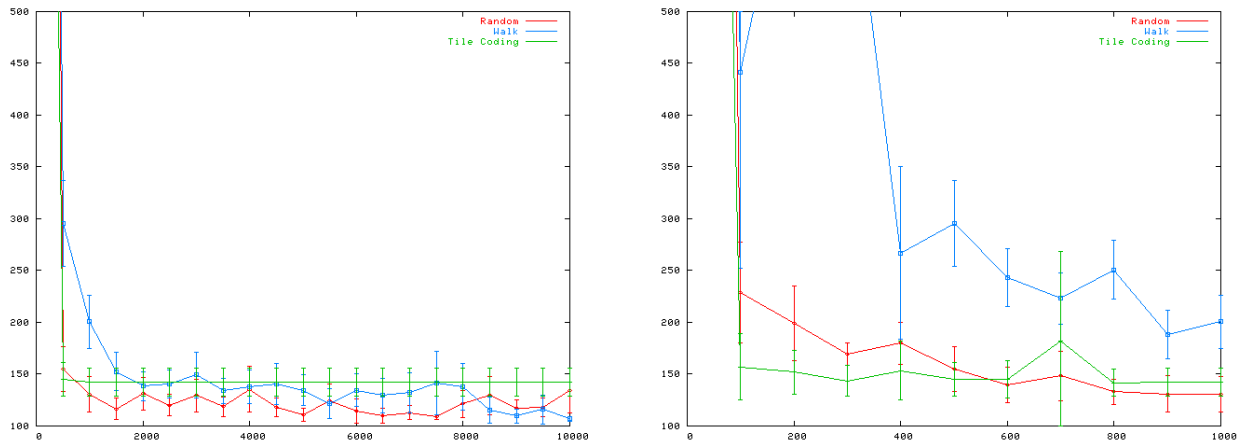

Figure 4: Average step-to-goal over 10 runs for random and walk-based batch allocations. The random allocation shown uses a chart half-width of 0.05; the walk-based allocation uses 1 walk threshholded at path length 0.1. Tile-coding approximator used for baseline comparison. Left: Results over 10,000 trials. Right: detail of the first 1000 trials.

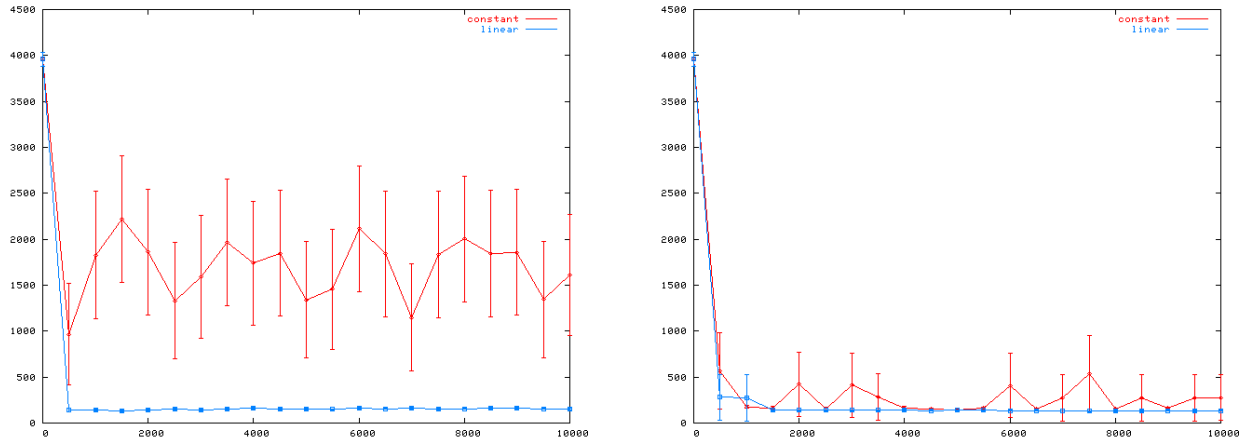

Figure 5: Average step-to-goal over 30 runs for online allocation with fixed-width charts. Left: constant vs. linear approximation with charts of half-width 0.1. Right: constant vs. linear approximation with charts of half-width 0.05. Confidence intervals bound the mean at the 95% level.

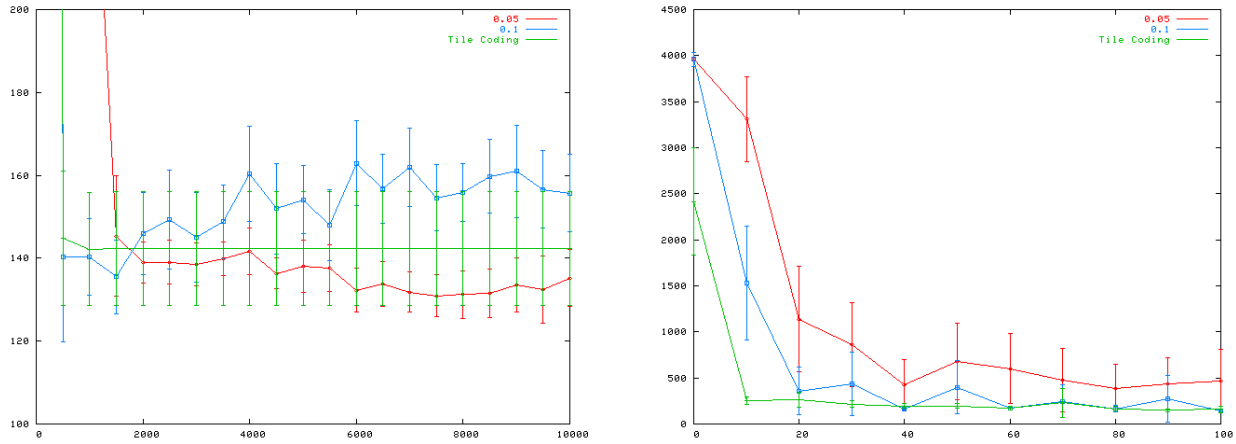

Figure 6: Average step-to-goal for online allocation with fixed-width charts compared to a tile coding learner. Left: small vs. large charts with linear approximation. Right: detail from the first 100 trials. Confidence intervals bound the mean at the 95% level.

6.3 Online Allocation Results

The results for online chart allocation with fixed-width charts are shown in Figures 5 and 6. The low representative power of fitting a constant value on a chart is well-illustrated in the case of the larger charts, while linear approximation quickly converges to a good policy for either chart size. Fitting a constant on each chart is too unstable to be practical for large charts.

Figure 6 compares the performance of linear local approximation against the tile coding learner; the tile coding learning quickly converges to a reasonable policy, while the online learners converge significantly more slowly, which is to be expected, since the tile coding learner is provided with a domain representation *a priori*. The autonomous manifold representation construction allows the manifold-based learners performance to catch up to the tile coding learner performance within the first 20 trials. As expected, when comparing the two manifold-based learners, the smaller charts allow for a significant performance improvement over the larger charts. Figure 7 compares linear and constant approximation on walk-based online allocation. Here linear approximation is significantly better than fitting a single weight on each chart. This is as expected; if a chart bounds k steps it will often be the case that values may differ by γ^k for any two states on a chart that does not contain a reward-bearing in its domain. We again use the tile coding approximator as a baseline for comparison; it converges much more quickly to a successful policy; however, the linear approximator performs significantly better with more experience.

Figure 8 simply illustrates the performance for constant and linear approximation. These results are for varying the number of steps provided as a parameter to the walk-based allocation. This serves to illustrate the relative stability and consistency of linear approximation as compared to fitting a single constant on a chart.

Figure 9 compares the performance of online allocation with linear approximation. The chart shows the best performing fixed-width and walk-based allocation parameter settings, each with linear approximation. Fixed-width allocation is significantly better early on; this is likely due to a larger chart size, resulting in faster generalization early on. Later the walk-based allocation dominates as more experiences have been observed; the change in dominant strategies is again due to the difference in chart size, as well as the

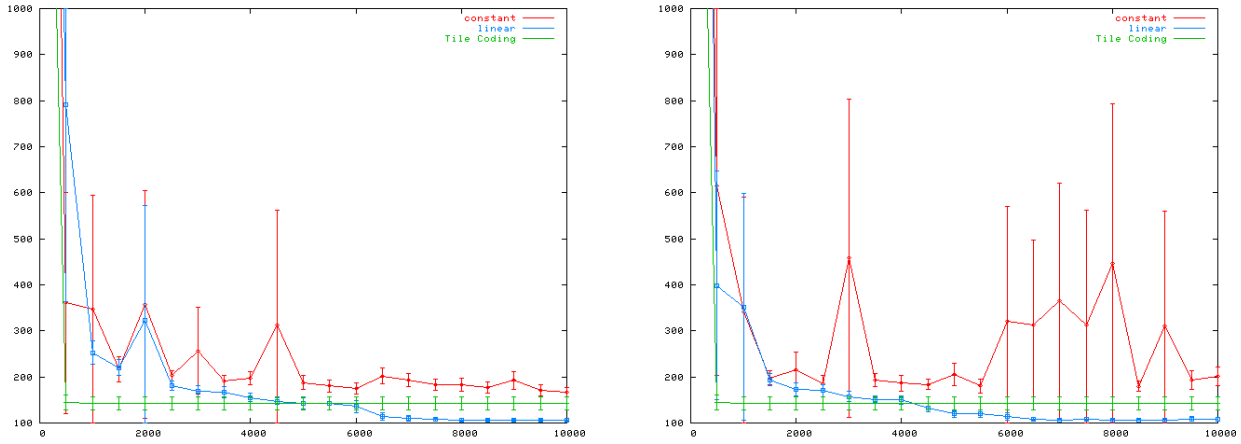

Figure 7: Average step-to-goal over 30 runs for online allocation with online walk-based allocation. Left: Constant vs. linear approximation for 6-step allocation. Right: Constant vs. linear approximation for 7-step allocation. 95% confidence intervals are shown.

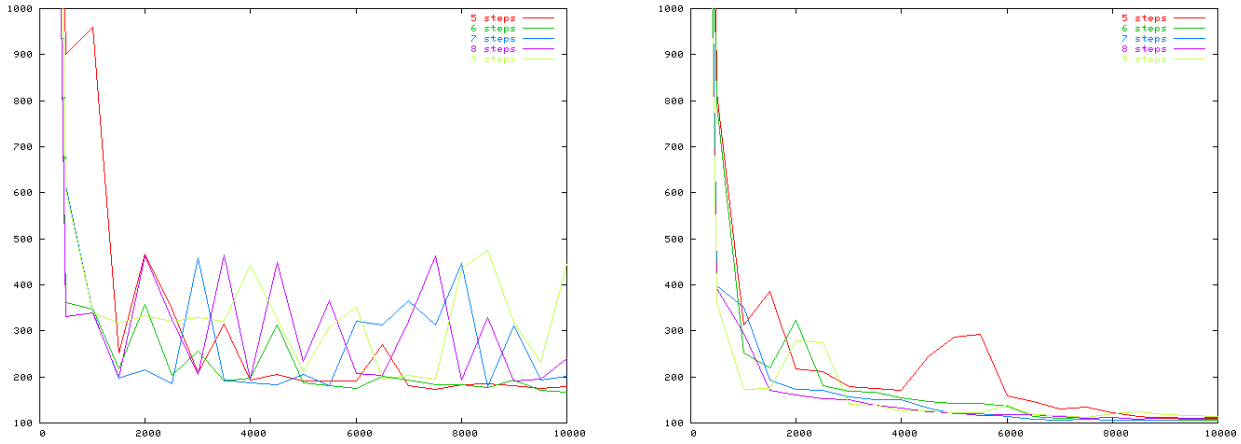

Figure 8: Average step-to-goal over 30 runs for online walk-based allocation. Left: Fitting a constant. Right: Linear local approximation.

Walk-based online allocation					
Steps	5	6	7	8	9
Charts	1050	776	618	503	409

Fixed-width online allocation		
Half-width	0.05	0.1
Charts	251	70

Table 2: Maximum charts allocated for each online allocation scheme.

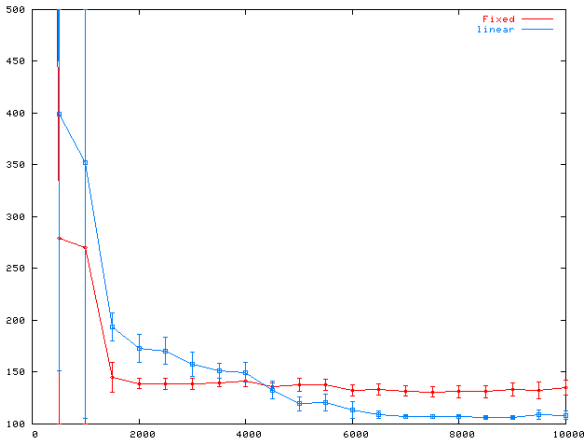

Figure 9: Comparision of results for the best observed online allocation performance. Both use linear approximation; The fixed-size allocation uses charts with half-width of 0.05, while the walk-based allocation uses 7-step allocation.

difference in chart count (see Table 2). The larger number of smaller charts gives the walk-based learner a higher effective resolution which increases the representational power of manifold-based approximation.

7 Conclusions

Our results for batch allocation show that the manifold method is competitive with a good hand-tuned approximation strategy for the mountain car domain. However, batch allocation methods will suffer the curse of dimensionality, since batch allocation methods cover the state space. This difficulty is compounded by the sample complexity of methods other than random allocation.

Both the batch and the online allocation results support the idea that more complex approximation can be tied to a local representation to improve performance. Using just a simple unthresholded linear unit on each chart gives us a substantially more stable approximation than a single weight on each chart. The results also support the hypothesis that a tradeoff exists between chart size and approximator complexity, as a single weight approximation was unstable except for small charts.

One of the most compelling results is the quality of policies found by online allocation. One of the weakness of value-function approximation strategies is that often a great deal of low-level "tweaking" of various is necessary in order to get good performance. However, particularly in the case of fixed-width online allocation, we were able to obtain strong results by varying only a single parameter, the chart size.

7.1 Future Work

One weakness of our current online allocation approach is that we have no way to bound the number of charts that are allocated; pruning methods such as those in [14] may be beneficial in this regard. Another direction for improvement may be adjustment of the chart placement after allocation, although results from using radial basis function networks for value-based RL have shown that this may actually harm performance [9]. However, Kretchmar's work only used minimization of temporal difference error as the metric for improving basis function placement; methods based on utile distinction [10, 19] may be more productive.

The manifold representation is useful for more than just value-function approximation, however. One use for the representation is as a practical means of implementing reward shaping via potential functions [13]. By using the connectivity of the manifold representation, one a goal state has been identified it is possible to embed a potential function on the manifold, which can be used to speed up learning.

Another use for the manifold representation is as a vehicle for experience reuse. Assuming that we can identify when two charts are "the same" in terms of the reward and system dyanmics, it may be possible to define a mapping that carries experiences gained in one chart to the other. In this way, we can effectively synthesize experience, which is particularly useful in real-world domains where experiences are expensive to generate.

8 Appendix

- Charts: The blend function we use was derived by Bruno and Kunyansky [5]; it is defined as follows:

$$h(s) = e^{\frac{2e^{-1/s}}{1-s}} \tag{8}$$

$$b(t) = \begin{cases} 1 & 0 \leq t \leq \delta \\ \frac{h((t-\delta)/a)}{h((t-\delta)/a)+h(1-(t-\delta)/a)} & \delta < t < 1-\delta \\ 0 & t \leq \delta \end{cases} \tag{9}$$

  where $a = 1 - 2\delta$. In our implementation we use $\delta = 0$.

- Manifolds: In our implementation a manifold is a chart container that supports point membership operations. Currently charts are stored in a list, so membership queries take $O(dn)$ time, where n is the number of charts and d is the manifold's dimension. This can be sped up using techniques from spatial indexing, particularly the R-tree [8].

Representational Equivalence Since we define our manifold representations in terms of subspaces of the state space along with an approximator on each state space, it is straightforward to represent other value function approxmation approaches with a manifold representation.

For instance, consider a CMAC. There are a couple of ways that we could construct a manifold representation that is analogous to the CMAC. The most trivial is to simply embed a CMAC on a single chart with domain S, the RL state space. The second method would be to define a chart domain for each tile in the CMAC, and embed a single real-valued weight on each. Ignoring chart boundary conditions, the manifold update equation for fitting a single weight reduces to the CMAC update equation.

The only real concern in showing that the manifold representation encompasses the representational power of the CMAC is that of dealing with the boundary conditions. In order to adhere to the definition of a manifold, chart domains must be open or closed, while tiles in a CMAC are half-open. Therefore, an exact equivalence is not possible; however, we can approximate the CMAC arbitrarily closely by dilating charts by some ϵ.

In a similar fashion we can construct manifold representations that are analogous to variable resolution discretizations of the state space, including the naïve uniform discretization. There is also a non-trivial means of representing a radial basis function network with a manifold representation. In this case, our chart domains are unbounded, allowing us to use infinite support. By constructing one domain for each basis function, and embedding one basis function on each chart, we obtain the RBFN representation.

References

[1] James S. Albus. A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *Journal of Dynamic Systems, Measurement and Control*, pages 220–227, 1975.

[2] Chris Atkeson, Andrew Moore, and Stefan Schaal. Locally weighted learning. *AI Review*, 11:11–73, April 1997.

[3] Christopher G. Atkeson, Andrew W. Moore, and Stefan Schaal. Locally weighted learning for control. *Artif. Intell. Rev.*, 11(1-5):75–113, 1997.

[4] Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. S. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 369–376. MIT Press, 1995.

[5] Oscar P. Bruno and Leonid A. Kunyansky. A fast, high-order algorithm for the solution of surface scattering problems: Basic implementation, tests, and applications. *Journal of Computational Physics*, 169(1):80–110, May 2001.

[6] Robert Glaubius and William D. Smart. Manifold representations for value-function approximation. In *Working Notes of the Workshop on Markov Decision Processes, AAAI 2004*, San Jose, California, USA, 2004.

[7] Geoffry J. Gordon. Reinforcement learning with function approximation converges to a region. In *Neural Information Processing Systems*, pages 1040–1046, 2001.

[8] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD'84, Proceeds of Annual Meeting*, pages 47–57. ACM Press, 1984.

[9] R. Matthew Kretchmar and Charles W. Anderson. Comparison of cmacs and radial basis functions for local function approximation in reinforcement learning. In *International Conference on Neural Networks*, 1997.

[10] Andrew K. McCallum. Learning to use selective attention and short-term memory in sequential tasks. In P. Maes, M. Matari'c, J.-A. Meyer, J. Pollack, and S. Wilson, editors, *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 315–324. The MIT Press, 1996.

[11] Andrew W. Moore and Christopher G. Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21:199–234, 1995.

[12] Rémi Munos and Andrew W. Moore. Variable resolution discretization in optimal control. *Machine Learning*, 1999.

[13] Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the 16th International Conference on Machine Learning*, pages 278–287, 1999.

[14] Bohdana Ratitch and Doina Precup. Sparse distributed memories for on-line value-based reinforcement learning. In *Proceedings of the 15th European Conference on Machine Learning*, 2004.

[15] G. A. Rummery and M. Niranjan. On-line q-learning using connectionist systems, 1994.

[16] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044. The MIT Press, 1996.

[17] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computations and Machine Learning. The MIT Press, Cambridge, MA, 1998.

[18] Csaba Szepesva'ri and William D. Smart. Interpolation-based q-learning. In *Proceedings of the Twenty-first International Conference on Machine Learning*, 2004.

[19] William T. B. Uther and Manuela M. Veloso. Tree based discretization for continuous state space reinforcement learning. In *AAAI/IAAI*, pages 769–774, 1998.

[20] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.