Approximating value functions in classifier systems

Lashon B. Booker

The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102-7508, USA
booker@mitre.org

1 Introduction

While there has been some attention given recently to the issues of function approximation using learning classifier systems (e.g. [13, 3]), few studies have looked at the quality of the value function approximation computed by a learning classifier system when it solves a reinforcement learning problem [1, 8]. By contrast, considerable attention has been paid to this issue in the reinforcement learning literature [12]. One of the fundamental assumptions underlying algorithms for solving reinforcement learning problems is that states and state-action pairs have well-defined values that can be computed and used to help determine an optimal policy. The quality of those approximations is a critical factor in determining the success of many algorithms in solving reinforcement learning problems.

In most classifier systems, the information about the value function is stored and computed by individual rules. Each rule maintains an independent estimate of the value of taking its designated action in the states that match its condition. From this standpoint, each rule is treated as a separate function approximator. The quality of the approximations that can be achieved by simple estimates like this is not very good. Even when those estimates are pooled together to compute a more reliable collective estimate, it is still questionable how good the overall approximation will be. It is also not clear what the best way is to improve the quality of those approximations.

One approach to improving approximation quality is to increase the computational abilities of individual rules so that they become more capable function approximators [13]. Another idea is to look back to the original concepts underlying the classifier system framework and seek to take advantage of the properties of distributed representations in classifier systems [2]. This paper follows the latter approach. We describe a new way to tap the distributed representational power present in a collection of rules to improve the quality of value function approximations. The basic idea is to treat rules as features that collectively specify a linear gradient-descent function approximator.

The paper begins with a brief overview of the role of value functions and approximations in reinforcement learning. Then we examine the corresponding issues in classifier systems and make an empirical comparison with a widely used technique from the reinforcement learning community. This comparison points out some weaknesses in the typical classifier system methods. Finally, a new approach to value function approximation — called hyperplane coding — is introduced along with empirical results showing how effective it is.

2 Value Function Approximation and Reinforcement Learning

We begin with a formal characterization of value functions in the context of reinforcement learning problems. Assume that the problem environment can be characterized as a discrete time, stochastic dynamic system with a finite set of states. This setting is well studied in the theory of reinforcement learning, as it provides the starting point for the analysis of finite Markov decision processes. A Markov decision process satisfies the Markov property and therefore can be characterized by the one-step dynamics of the environment. This means that in addition to specifying all possible states and actions, a problem definition includes two other things: transition probabilities $p_{ij}(u)$, which give the probability that the next state is j if action u is taken in the current state i; and, scalar rewards $r_i(u)$ which indicate the immediate feedback available after applying action u in state i. An agent trying to solve such a problem uses a decision policy π to specify which action is selected as a function of the current observed state. A decision policy is a function of states and actions that computes the probability of taking action u when in state i. Given a fixed policy π, the value function is a mapping that computes, for each state, the long term expected reward the agent will accrue by using the policy π to make decisions. If a discounted reward criterion is used to compute the long term reward, the formal recursive definition of the value function is given by

$$V_\pi(i) = \sum_u \pi(i,u) \sum_j p_{ij}(u)[r_i(u) + \gamma V_\pi(j)]$$

where $\gamma \in [0,1]$ is a discount factor that determines the influence of future rewards on current decisions.

Most approaches to solving reinforcement learning problems explicitly compute and store some representation of V_π. For very simple problems, a lookup table is an adequate way to represent the value function. In most cases of interest, however, the input space is too large to represent V_π exhaustively in tabular form so the function must be represented more compactly. Efficient storage is not the only important issue though. In a large state space the learning agent will only directly experience a relatively small number of inputs. The agent nevertheless needs to leverage that experience to determine how to behave when it encounters inputs that have not been seen before. This implies that generalization is a key issue for reinforcement learning problems with large state spaces. The most common approach to addressing these issues is to use function approximation techniques to compute a compact representation of V_π that generalizes well.

The approach to approximating V_π used in learning classifier systems belongs to a class of techniques known as soft state aggregation [10]. In the simplest forms of state aggregation, the states are partitioned into a set of disjoint groups or clusters. A reinforcement learning problem can be solved at the cluster level to compute a value function for the clusters. The value of a cluster is then used as the value for each of the states in that cluster. Soft state aggregation techniques allow a single state to belong to more than one cluster, providing for cluster overlap. This is accomplished by defining cluster probabilities $P(x|i)$ that specify the degree to which state i is associated with cluster x. The value for a state is given by a weighted average of the values of the clusters the state is associated with; that is,

$$V_\pi(i) = \sum_x P(x|i) V_\pi(x)$$

Rule input conditions designate the clusters of states used by learning classifier systems. Each condition represents a set of states whose value is summarized in various ways by the rule's utility measure. In XCS, for example, a cluster's value is represented by the prediction parameter of the corresponding rule. The cluster probabilities are given by the rule's fitness divided by the sum of the fitnesses of all the rules matching state i.

While state aggregation approaches to function approximation can be useful in some settings, they are known to have serious shortcomings [12]. First, they tend to scale poorly as the number of dimensions of the state space increases. Second, large numbers of clusters may be needed to represent smooth functions accurately[1]. The most widely used approaches to function approximation for reinforcement learning avoid these problems by relying on linear gradient-descent methods.

The remainder of this paper takes a brief look at linear gradient-descent methods and one important special case that uses binary features. We then propose a new approach to using linear gradient descent in a classifier system setting and present empirical results showing that the idea has merit.

3 Linear Approximations and Coarse Coded Features

Linear gradient-descent methods for value function approximation begin with a linearly parameterized representation of the value function given by

$$V(x_t) = \sum_i w_i(t) \phi_i(x_t)$$

where the ϕ_i are features defined on the state space and the w_i are real-valued adjustable weight parameters. The weights are adjusted to try to reduce the error

[1] This limitation will become more important to the classifier system community as classifier systems are applied to function approximation problems [13].

on the observed sample points x, and to generalize from that data to provide good approximations for other points that have not yet been seen.

Gradient-descent methods try to minimize error by adjusting the weights on each step in the direction that reduces error the most. In the linear case, the gradient descent update for adjusting the weights is given by

$$w_i(t+1) = w_i(t) + \alpha[v(t) - V(x_t)]\nabla_{w_i} V(x_t)$$

where $\nabla_{w_i} V(x_t) = \phi_i(x_t)$ is the gradient of the linear function with respect to weight parameter w_i and $v(t)$ is the true function value for x_t.

Linear gradient-descent methods are simple and they are particularly well-suited to reinforcement learning [12]. A key aspect determining how well these methods work in practice, though, is the quality of the features they use. The features must represent whatever task-relevant qualities of the state may be needed to discriminate one state from another, as well as any associated feature interactions that may be important.

3.1 Tile coding

Coarse coding [7] is a general approach to defining a set of adequate features. In this form of representation, each feature corresponds to some subset of the state space (the feature's "receptive field"). For a given state, a feature is said to be activated if the state belongs to that receptive field. The representation of state is coarse coded in the sense that the receptive fields overlap to produce a distributed representation whose acuity is proportional to the number of features activated in a given state. One general purpose way to define receptive fields suitable for efficient on-line learning is called tile coding [12].

Tile coding is a particular form of coarse coding in which the receptive fields for all features are organized into exhaustive partitions of the input space. The features are assumed to be binary, the receptive fields are called *tiles*, and each partition is called a *tiling*. The tilings are offset from each other in order to achieve the overlap needed for local generalizations. For a single input dimension, the offsets typically used in tile coding are given by $i(w/n)$ where i is the index of the tiling, w is the tile width, and n is the number of tilings ($0 \leq i < n$). There are several advantages to organizing the receptive fields in this way. Every point in the input space activates the same number of tiles, so there is strict control over the density of tiles and the resulting precision of the approximation. It is also easy to set the learning rate for a linear gradient-descent function approximator based on tile coding. Since the number of features active for each point is equal to the number of tilings m, the learning rate can be expressed intuitively as a fraction of the rate $1/m$ which gives exact one-trial learning. The weight update for activated features is given by

$$w_i(t+1) = w_i(t) + \frac{\alpha}{m}[v(t) - V(x_t)]$$

where α is the desired fraction.

Tile coding has been been used extensively for reinforcement learning, and the overall coarse coding approach is known to be capable of computing high quality approximations [9]. It is not clear how well classifier system methods compare to these approaches from the standpoint of function approximation. We try to answer that question with an empirical comparison of tile coding with the widely used classifier system mechanisms in XCS [4] for predicting expected payoff.

3.2 Comparing tile coding with XCS predictions

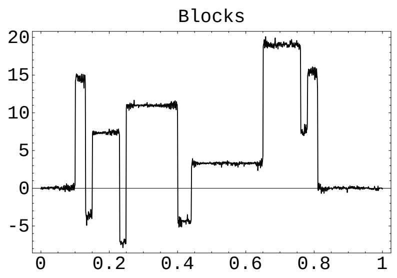
The effectiveness of classifier system methods for function approximation can be assessed by using function values as rewards [13] and allowing the system to generate outputs in the usual way. In order to test the XCS prediction mechanism, a skeletal classifier system was implemented. This skeletal system has traditional ternary rules with no actions and no rule discovery mechanisms. On every step the system is presented with a data point x, and the reward received is the function value $f(x)$. The system forms a match set and proceeds to update the basic XCS parameters: experience, prediction, prediction error, and fitness. The system prediction is calculated in the usual way and that prediction becomes the system's estimate for the value of x. The parameter settings were consistent with those used for XCS in the literature [13]: learning rate 0.2, error threshold 0.2, fitness power 5.0, and fitness scale (i.e., α) 0.1. See Butz and Wilson [4] for details about these parameters and computations.

The test function suite was taken from a set of functions proposed by Donoho and Johnstone [5] that has been widely used in the literature on statistical estimation and reconstruction of signals from data. We use four one-dimensional functions — Blocks, Bumps, Doppler, and HeaviSine — that provide a good variety of spatial variability and smoothness (see definitions in the Appendix). The training data for each function was drawn from a set of 2048 equally spaced sample points. A separate distinct set of 2000 equally spaced sample points was set aside to use as a test set. The quality of an approximation is measured in terms of the average squared error at those sample points. More specifically, the performance measure is

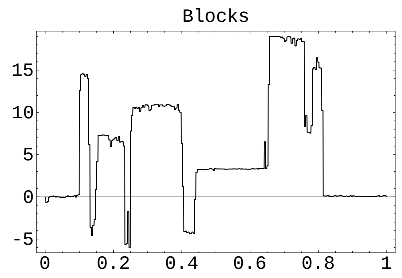
$$R = n^{-1} \sum_{i=0}^{n-1} (\hat{f}(x_i) - f(x_i))^2$$

where $\hat{f}$ is the approximation and f is the true function. In all of the experiments reported here, learning proceeded over 100 trials with 10,000 steps per trial, and with a random data point selected from the training set on each step. This gave the function approximators ample time to converge to their most accurate output. Results were averaged over 10 replications, and statistical significance was assessed using a Student's t-test with significance level 0.05.

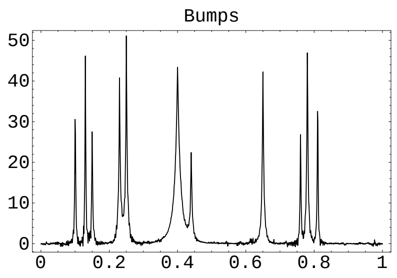
The goal of this comparison is to assess how well each approach makes use of a fixed allocation of approximation resources. For tile coding this means that the



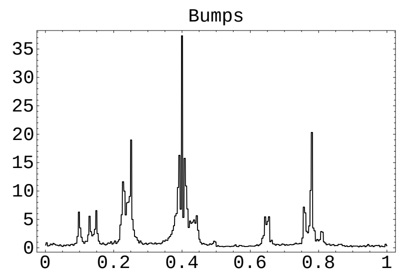
(a) Tile coding of Blocks



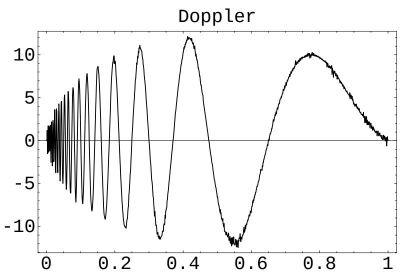
(b) XCS prediction of Blocks



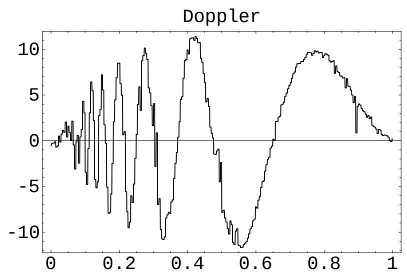
(c) Tile coding of Bumps



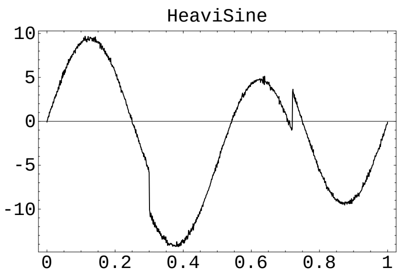
(d) XCS prediction of Bumps



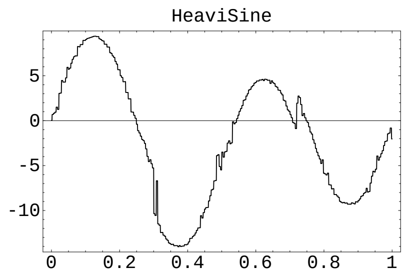
(e) Tile coding of Doppler



(f) XCS prediction of Doppler



(g) Tile coding of HeaviSine



(h) XCS prediction of HeaviSine

Fig. 1. Reconstructions computed by tile coding and XCS prediction

Algorithm	Approximation Error							
	Blocks		Bumps		Doppler		HeaviSine	
	Train	Test	Train	Test	Train	Test	Train	Test
Tile coding	0.06988	1.7535	0.16809	0.93068	0.03579	0.08922	0.02327	0.06458
XCS prediction	3.4697	3.2368	25.111	25.977	2.1472	2.1360	0.08345	0.08865

Table 1. Average square errors for tile coding and XCS prediction

number of tiles and the way they are organized is fixed. On these test functions, we use 2048 grid-like tiles each having width 1/256. The tiles are organized into 8 tilings that are offset as described previously. The learning rate is specified by the assignment $\alpha = 0.2$. For the XCS prediction mechanism, the population of classifiers is fixed at 2048 rules generated randomly using a probability of 1/3 for placing the # symbol at any given position in a rule condition. Each classifier condition is 8 bits, giving every classifier the same input resolution as one of the grid-like tiles.

The results on the suite of test functions are summarized in Table 1. All of the differences in performance between the tile coding approximation and the XCS prediction are statistically significant. Tile coding is substantially more effective than the XCS prediction on these functions. Tile coding shows an impressive ability to reconstruct functions with respect to the training data. Its performance on the four test functions compares favorably with results on the same data achieved by more sophisticated approximation techniques like a discrete wavelet transform [5]. The reconstructions shown in Figure 1 show that the tile coding representation has enough precision to pinpoint the location of abrupt changes in function values. Moreover, tile coding also has sufficient local generalization properties that the approximations are fairly smooth.

The XCS prediction, on the other hand, does poorly from the standpoint of both precision and smoothness. There is a sense in which this is not surprising, since the mechanisms were intended to be used in combination with rule discovery to compute a good approximation of the value function. There is a dilemma with that arrangement, however. Rule discovery depends on guidance from the prediction computations in order to know what type of rules to generate. If that guidance is poor, then rule discovery will have to thrash around somewhat randomly until it discovers something that improves the approximation.

It should be possible to take the information in a population of classifiers, even if that population is random, and reliably compute good approximations that provide useful guidance for rule discovery. What aspects of the tile coding approach can be leveraged to improve the value function approximations computed in classifier systems? One straightforward approach would be to restrict our attention somehow to hyperplane features that define an exhaustive partition. This would allow the tile coding computational mechanisms to be used directly, but would be overly restrictive from the standpoint of typical classifier system operating principles. For many reasons, the heterogeneity of the classifier population is a feature not a bug. An alternative approach is to use that heterogeneity to our advantage by devising a variation of tile coding that relies more on the strengths of distributed representations. The next section introduces a new alternative based on this idea called hyperplane coding.

3.3 Hyperplane coding

Hyperplane coding is a closely related variation of tile coding in which classifier rule conditions fill the role of tiles, and there are few restrictions on the way those "tiles" are organized. The hypothesis behind this idea is that classifier rules can be more effective as function approximators if they collectively implement a distributed representation of the value function. The distributed representation is realized by treating individual rules as features rather than as independent function approximators whose estimates are pooled to compute an overall result.

In a random population, the classifier conditions serving as tiles do not cover the space like an exhaustive partition. Nevertheless, a population of classifiers does richly cover the space with a collection of overlapping coordinate hyperplanes. Each point in the input space is covered by an expected number of tiles (or matching conditions) k given by

$$k = N \left(\frac{p_{\#} + 1}{2} \right)^{l}$$

where N is the population size, $p_{\#}$ is the probability of the `#` symbol appearing at any position in the condition, and l is the length of the condition. For the population sizes typically used in classifier system applications, this expected value is much larger than the fixed number of tilings most often used for tile coding. This bodes well for the resolution of approximations based on hyperplane coding, since greater tile density usually means higher precision.

The coarse coding idea requires the ability to represent patterns of contiguous inputs (the tiles arranged in a tiling) that can be offset from each other by arbitrary amounts. This requirement is trivial to fulfill in tile coding. The tiles are fixed sized intervals in each dimension, and the interval endpoints can be adjusted as needed. The standard syntax for the input condition of a classifier rule does not provide this kind of flexibility. It is not clear how to adjust that syntax to represent hyperplanes offset by arbitrary amounts in input space, while preserving the simple matching operations between rules and messages. For example, it is easy to represent the lower half of the input range $[0, 1]$ with the condition `0#...#`, which corresponds to the interval $[0, 0.5]$ using the standard binary encoding. How do we represent the hyperplane corresponding to the offset interval $[0 + \epsilon, 0.5 + \epsilon]$?

One obvious way to manage this issue is to apply the offset to the input space, then define hyperplanes on that transformed space in the usual way. Looking at the offset interval $[0 + \epsilon, 0.5 + \epsilon]$ again, we can determine if some input value x belongs to that interval by checking if a message encoding the translated value $x - \epsilon$ matches the condition `0#...#`. This leads to the following ideas for the way a population of classifiers is organized to implement coarse coding. Each classifier is assigned to a specific tiling[2], just like each tile belongs to a specific tiling under tile coding. In this case, though, there is no specific organization imposed on the tiling. Continuing with the analogy, we do associate a fixed offset with each tiling. The classifier system operating principles are also adjusted somewhat. Instead of having a single message matched against all rules on each cycle, we generate a separate message for each tiling. Each message is computed from the raw input by applying the offset associated with the tiling in question.

The only remaining details needing attention have to do with tile width and offsets. Since hyperplanes in general do not correspond to simple contiguous regions of the input space, some thought must be given to the issue of how to define tile width. There are several possibilities and we choose one of the simplest. The smallest possible contiguous region defined by a hyperplane is one that corresponds to a single binary value. The width of this region is simply the resolution size used to discretize the raw input. The width of every contiguous region matched completely by some hyperplane is a multiple of this resolution size. Consequently, the resolution size can be used as the tile width for all hyperplanes[3].

As noted previously, the offsets typically used in tile coding are given by $i(w/n)$ where i is the index of the tiling, w is the tile width, and n is the number of tilings ($0 \leq i < n$). Under this arrangement, every point has at least one tile in common with all points that are within a tile width away in each direction. This translation scheme uses only positive offsets that translate tiles to the right. A point gets grouped with its neighboring points on the left when the adjacent tile on the left (that does not originally contain all the points) gets translated to cover those points. This scheme does not work well in the classifier system setting, however. An unmodified input message matches classifiers representing the base tiles (i.e., tiling $i = 0$) covering a point x. If we adhere to the usual concept of a match set, the only way that x will be grouped into a tile with any other point is if the match set contains a classifier that matches both points. Offsets can change the groupings by excluding some points, but there is no way to include points that are not covered by the base match set. This makes it important to group the matched points in as many ways as possible. Accordingly, we use a more symmetric set of offsets given by $i(w/n)$ with $-n/2 \leq i < n/2$ so that points get grouped in both directions.

[2] We will call each major grouping of classifiers a tiling, even though the set does not partition the input space (i.e., the elements are not disjoint, and they may not span the entire space).

[3] Each hyperplane has its own smallest width determined by the position of the lowest order specific bit in the classifier condition. Giving each condition its own tile width and offset would lead to a potentially unmanageable number of messages on each cycle, though, so that option is not considered here.

Algorithm Variant	Approximation Error	
	Train	Test
Baseline	0.24656	0.34647
Gray code	0.22560	0.32736
Salience	0.19654	0.30969
Better offsets	0.13746	0.30148

Table 2. Average square errors for variations of hyperplane coding on the Blocks function

4 Experiments With Hyperplane Coding

In order to evaluate this idea, the skeletal classifier system described previously for the experiment with XCS prediction mechanisms was modified to implement the hyperplane coding algorithm described above. This section briefly describes that implementation, empirically evaluates its performance, then describes a series of modifications that improve performance.

4.1 Baseline implementation

The initial implementation of linear function approximation based on hyperplane coding starts with the algorithm described above and uses parameters taken from the previous experiments with tile coding. We use a population of 2048 random classifiers organized into 8 tilings of 256 classifiers each. The classifier conditions were 8 bits long to provide the same resolution for discretizing the input as the tile coding approach. Each classifier has a weight parameter w that is adjusted by gradient descent just as in tile coding. The learning parameter α for gradient decent was set to 0.2, again in agreement with the tile coding experiments. These choices give the linear approximator based on hyperplane coding roughly the same amount of approximation resources to work with as the tile coding version had.

We begin by focusing our attention on how the algorithm performs on the Blocks function. Performance on Blocks is summarized in Table 2. The average square error on the training data was 0.24656, which is a statistically significant drop in performance from the error of 0.06988 for tile coding in Table 1. Interestingly, the roles were reversed on the testing data. The average square error for hyperplane coding was 0.34647, a statistically significant improvement over tile coding's value of 1.7535. The relatively large number of hyperplanes covering each point apparently gives the hyperplane coding scheme a huge advantage in generalization. Note that the performance advantage of hyperplane coding over the XCS prediction is statistically significant on both the test and the training data.

4.2 Gray coded inputs

One of the important properties of approximation techniques like tile coding is that the generalizations they compute are localized. Points that are sufficiently close in input space will produce output values that are close. Moreover, values in widely spaced regions can be learned with relatively little interference. This property is compromised somewhat with hyperplane coding since hyperplanes are not restricted to contain localized collections of points. Some of the approximation error we observe in the results so far can probably be attributed to this effect.

If this is true, then a representation that provides more localized collections of points should boost performance. The Gray code is known to be such a representation for bit strings [6]. In order to see why, consider the classifier condition `##10`. The bit strings matching that condition are `0010`, `0110`, `1010`, and `1110`. None of these points are contiguous under a binary coding. A binary reflected Gray code, however, groups these points into two clusters of consecutive points: (`0010, 0110`) and (`1110, 1010`). This example is illustrative of a more general phenomenon. A Gray code will never group bit strings matching some condition into more clusters of consecutive points than a binary code does. Furthermore, for some conditions, the Gray code will organize the points into fifty percent fewer clusters than the binary code (as in our simple example). See Faloutsos [6] for more details.

This analysis suggests that significant improvements should be obtained by using a binary reflected Gray code to encode the inputs for our function approximator. Table 2 shows that those improvements do indeed occur. The performance improvement is statistically significant for both the test data and the training data.

4.3 Feature salience

Under tile coding, every point belongs to exactly one tile in every tiling. As noted previously, a point belongs to many tiles in each tiling under hyperplane coding. Because the hyperplanes in a tiling are so diverse, they may not all be equally useful for approximating the function. It might matter that some are more specific than others, some may correspond more closely to key regularities in the function, and so on. This presents the approximation algorithm with a feature selection problem that does not occur with tile coding. The problem is important because irrelevant features are a source of noise that can slow down learning of relevant features.

One way to address this feature selection problem is to use dynamically adjusted learning rates to identify which features are most relevant to the task at hand. The idea is to give small learning rates to weights for irrelevant features and large learning rates to weights for relevant features. The individual learning rates thereby become a source of bias that make learning and generalization more efficient. Sutton [11] describes an algorithm — called the Incremental Delte-Bar-Delta (IDBD) method — that uses experience to incrementally adjust learning rates in a linear learning system. That algorithm is well suited to this setting and was incorporated into our hyperplane-based function approximator.

The intuition behind the IDBD algorithm is to adjust rates based on the correlation between successive weight changes: if the weight changes have all been in the same direction, the rate was too small; if the weight changes have been in opposite directions, the rate was too large. The algorithm has one free parameter, the meta learning rate θ. It also uses two parameters for each feature ξ: a learning rate parameter β_ξ and a memory parameter h_ξ that stores a trace of the cumulative sum of recent errors. Each h_ξ is initialized to zero. Given a match set with binary features ξ, weights w_ξ, and approximation error $\delta(t)$, the version of IDBD used here performs the following updates in the order indicated:

1. $\beta_\xi(t+1) = \beta_\xi(t) + \theta\delta(t)h_\xi(t)$
2. $\alpha_\xi(t+1) = e^{\beta_\xi(t+1)}$
3. $w_\xi(t+1) = w_\xi(t) + \alpha_\xi(t+1)\delta(t)$
4. $h_\xi(t+1) = h_\xi(t)[1 - \alpha\xi(t+1)]^+ + \alpha_\xi(t+1)\delta(t)$ where the notation $[x]^+$ indicates a quantity equal to x if $x > 0$ and 0 otherwise.

See Sutton [11] for more details about this algorithm and the reasons why it works.

The linear function approximator based on hyperplane coding was augmented with the IDBD algorithm using meta parameter $\theta = 0.01$. Following Sutton's advice about implementation details, bounds were enforced on each β_ξ to prevent arithmetic underflow. The lower bound was $\ln(\alpha)$ so that the adjusted rates never fell below the global rate α specified for the function approximator. We also enforced an upper bound of 1.0 and limited the change in β_ξ on any one step to ± 1 to help ensure that the weight adjustments remain stable. The results in Table 2 show that these changes had the anticipated effect. Statistically significant performance improvements were seen on both the test data and the training data.

4.4 Improved feature offsets

The large number of overlapping hyperplanes in a match set help to provide a strong generalization capability for linear function approximation based on hyperplane coding. Since the performance on the training data still lags far behind the levels achieved by tile coding, there appears to be room for improvement in the acuity of this approximation. Some performance improvements might be achieved if the available features could be reorganized to cut through the input space in a larger variety of ways. Two changes were implemented to test this hypothesis. First, we change the number of tilings. The number of tilings was set to 8 in the baseline implementation simply to be consistent with the parameters used in tile coding. On closer examination, though, this setting does not achieve the desired effect. In the tile coding implementation, every point has at least one tile in common with all points that are within a tile width away in each direction. The hyperplane coding using the symmetric offsets in the match set is

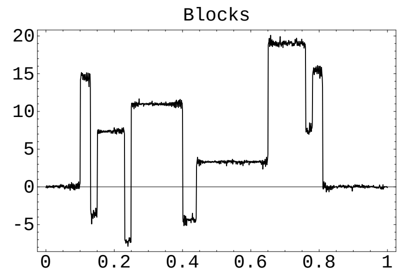
Algorithm	Approximation Error							
	Blocks		Bumps		Doppler		HeaviSine	
	Train	Test	Train	Test	Train	Test	Train	Test
Tiles	0.06988	1.7535	0.16809	0.93068	0.03579	0.08922	0.02327	0.06458
Hyperplanes	0.13746	0.30148	0.35245	0.98615	0.04709	0.08056	0.01444	0.02288

Table 3. Average square errors for tile coding and hyperplane coding

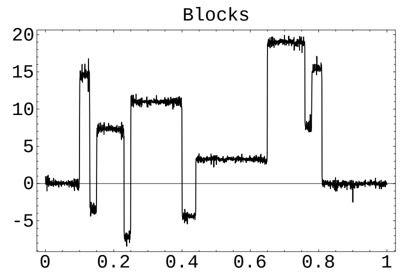
more limited. Every point can potentially be grouped only with points that are within half a tile width away in each direction. This deficiency is easily remedied by doubling the number of tilings to 16. The size of the population of classifiers remains the same, so the algorithm still has the same amount of approximation resources to work with.

Second, while the choice of the resolution size as the tile width for all tilings was convenient, it does not take full advantage of the possibilities for tile offsets. The resolution size is the simplest tile width that makes sense for classifier conditions with a specific bit at the lowest order bit position, and those classifiers occupy a large fraction of a random population. Larger tile widths are possible for the remaining classifiers, however, and the increased overlap could improve the acuity of the overall approximation. This possibility can be easily tested by organizing the classifiers into two types of features, coarse and fine, stored in two separate groups of tilings. The fine features are those classifiers with a specific bit at the lowest order bit position. These classifiers use the resolution size as the tile width. The remaining classifiers are all treated as coarse features, which use a tile width equal to twice the resolution size. As a consequence of these changes, on each cycle the system generates up to $4n$ potentially distinct messages where n is the number of tilings used in a comparable tile coding scheme.

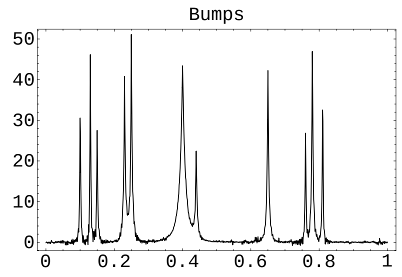
The results in Table 2 show that these changes had the desired effect. There was a statistically significant and large improvement on the training data, with no significant change in performance on the test data. The overall performance of this version of hyperplane coding is summarized in Table 3. Tile coding still has a statistically significant performance advantage on the training data for all of the test functions except HeaviSine, where hyperplane coding is far superior. Hyperplane coding has a statistically significant performance advantage on the test data for all of the test functions except Bumps, where tile coding has a small but significant advantage. The reconstructions computed by each type of coding for all of the test functions are shown in Figure 2. The precision and smoothness properties of the two representations are remarkably similar. It appears that hyperplane coding offers an alternative for linear approximations that is comparable in performance to what can be achieved with a more conventional approach like tile coding.



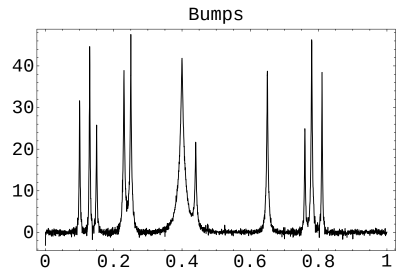
(a) Tile coding of Blocks



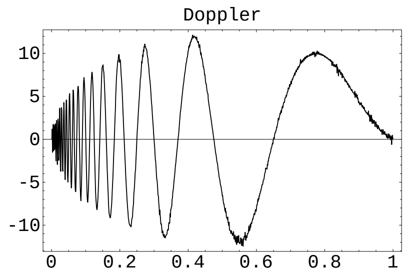
(b) Hyperplane coding of Blocks



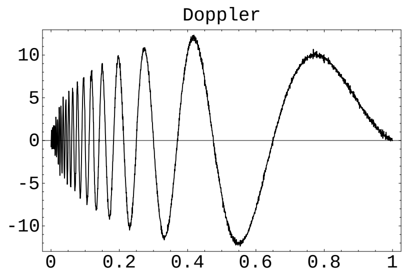
(c) Tile coding of Bumps



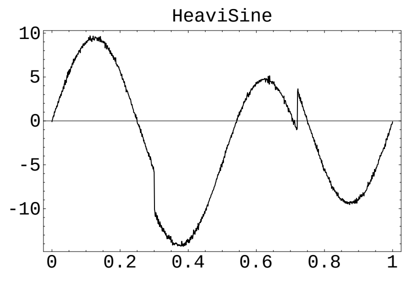
(d) Hyperplane coding of Bumps



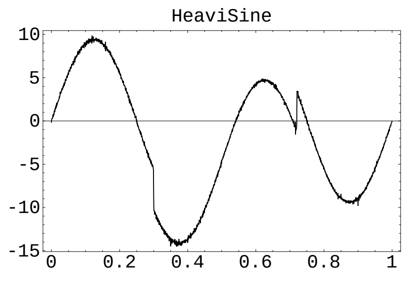
(e) Tile coding of Doppler



(f) Hyperplane coding of Doppler



(g) Tile coding of HeaviSine



(h) Hyperplane coding of HeaviSine

Fig. 2. Reconstructions computed by tile coding and hyperplane coding

5 Conclusions

This paper has shown that by carefully using the resources available in a random population of classifiers, continuous value functions can be approximated with a high degree of accuracy. The results demonstrate that hyperplane coding can achieve levels of performance comparable to those achieved by more well-known approaches such as tile coding. Hyperplane coding treats classifier system rules as features that contribute to a distributed representation of the value function. This approach computes much better approximations than more conventional classifier system methods in which individual rules compute approximations independently. High quality value function approximations that provide both data recovery and generalization are a critically important component of most approaches to solving reinforcement learning problems. Because these results substantially improve the quality of the approximations that can be computed by a classifier system using relative small populations of classifiers, this work provides the foundation for significant improvements in classifier system performance.

Conventional approaches such as linear gradient-descent function approximation based on tile coding are faster, but the hyperplane coding approach seems to offer more opportunities for increasing precision without incurring significantly greater computational costs. The density of tiles in hyperplane coding is naturally higher than the density in tile coding. This contributes to more resolution in the final approximation. The precision of the approximation can also be increased by increasing the length of the classifier input conditions instead of by adding more tiles. Moreover, the hyperplane coding scheme makes it possible to adapt the collection of tiles to achieve more precision. The obvious next step in this research is to use the approximation resources available in a random population as a starting point for a more refined approach to approximation that reallocates resources adaptively to gain greater precision in those regions of the input space where it is needed.

Finally, we note that in hyperplane coding the classifier conditions serve the role of value-based generalizations [14] in the way they organize inputs according to similar function values. While this clearly allows for the specification of decision policies for solving reinforcement learning problems, it ignores the attribute-based generalizations that have been a key feature of the rule-based policies produced by learning classifier systems. Future work will show how attribute-based rule conditions can be learned along with value-based generalizations in a tightly coupled fashion.

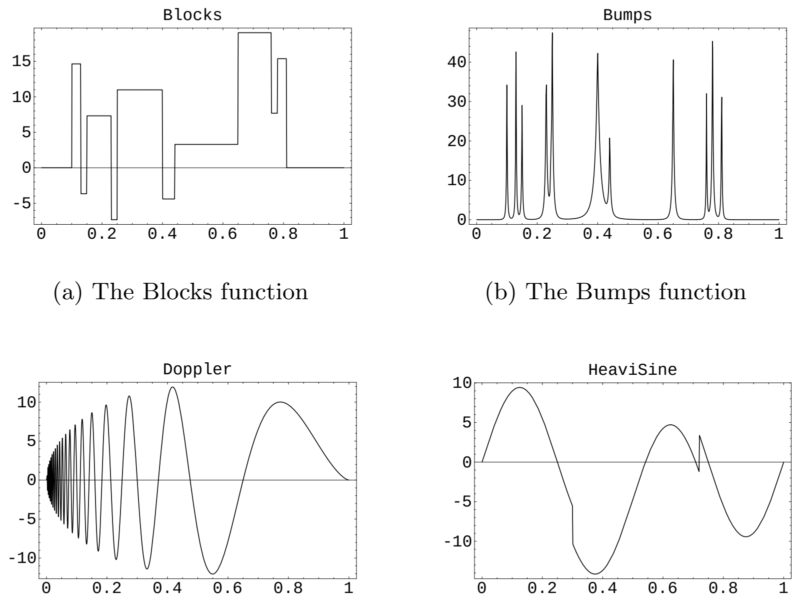
† Acknowledgments

This work is based on research originally funded by the MITRE Sponsored Research program. That support is gratefully acknowledged. The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions or viewpoints expressed by the author.

References

1. Lashon B. Booker. Viewing Classifier Systems as an Integrated Architecture. In *Collected Abstracts for the First International Workshop on Learning Classifier System (IWLCS-92)*, 1992. October 6–8, NASA Johnson Space Center, Houston, Texas.
2. Lashon B. Booker, David E. Goldberg, and John H. Holland. Classifier Systems and Genetic Algorithms. *Artificial Intelligence*, 40:235–282, 1989.
3. Larry Bull and Toby O'Hara. Accuracy-based neuro and neuro-fuzzy classifier systems. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 905–911. Morgan Kaufmann Publishers, 9-13 July 2002.
4. Martin V. Butz and Stewart W. Wilson. An Algorithmic Description of XCS. In Pier Luca Lanzi, Wolfgang Stolzmann, and Stewart W. Wilson, editors, *Advances in Learning Classifier Systems*, volume 1996 of *LNAI*, pages 253–272. Springer-Verlag, Berlin, 2001.
5. David L. Donoho and Iain M. Johnstone. Ideal spatial adaptation by wavelet shrinkage. *Biometrika*, 81:425–455, 1994.
6. Christos Faloutsos. Gray codes for partial match and range queries. *IEEE Transactions on Software Engineering*, 14(10):1381–1393, October 1988.
7. Geoffrey E. Hinton, James L. McClelland, and David E. Rumelhart. Distributed representations. In David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group, editors, *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations*, pages 77–109. MIT Press, 1986.
8. David LeRoux and Michael Littman. Reinforcement learning using lcs in continuous state space. Seventh International Workshop on Learning Classifier Systems (IWLCS-2004), Extended Abstract, 2004.
9. W. Thomas Miller, Filson H. Glanz, and L. Gordon Kraft. CMAC: An associative neural network alternative to backpropagation. *Proceedings of the IEEE*, 78(10):1561–1567, October 1990.
10. Satinder P. Singh, Tommi Jaakkola, and Michael I. Jordan. Reinforcement learning with soft state aggregation. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems*, volume 7, pages 361–368. The MIT Press, 1995.
11. Richard S. Sutton. Adapting bias by gradient descent: An incremental version of delta-bar-delta. In *AAAI*, pages 171–176, 1992.
12. Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge,MA, 1998.
13. Stewart W. Wilson. Classifiers that approximate functions. *Natural Computing*, 1(2-3):211–234, 2002.
14. Richard C. Yee. Abstraction in control learning. Technical Report COINS Technical Report 92-16, Department of Computer and Information Science. University of Massachusetts, Amherst, MA 10003, 1992. A dissertation proposal.

Appendix - Test Functions



(a) The Blocks function (b) The Bumps function

(c) The Doppler function (d) The HeaviSine function

Fig. 3. The Four Donoho Test Functions

Blocks.

$$f(t) = 3.65948 * \sum h_j K(t - t_j) \text{ where } K(t) = (1 + \mathrm{sgn(t)})/2$$

$(t_j) = (0.1, 0.13, 0.15, 0.23, 0.25, 0.4, 0.44, 0.65, 0.76, 0.78, 0.81)$
$(h_j) = (4, -5, 3, -4, 5, -4.2, 2.1, 4.3, -3.1, 2.1, -4.2)$

Bumps.

$$f(t) = 10.5174 * \sum h_j K((t - t_j)/w_j) \text{ ,where } K(t) = (1 + |t|)^{-4}$$

$(t_j) = (0.1, 0.13, 0.15, 0.23, 0.25, 0.4, 0.44, 0.65, 0.76, 0.78, 0.81)$
$(h_j) = (4, 5, 3, 4, 5, 4.2, 2.1, 4.3, 3.1, 5.1, 4.2)$
$(w_j) = (0.005, 0.005, 0.006, 0.01, 0.01, 0.03, 0.01, 0.01, 0.005, 0.008, 0.005)$

Doppler.

$$f(t) = 24.2158 * \sin(2\pi(1 + \epsilon)/(t + \epsilon))\sqrt{t(1 - t)} \text{ , where } \epsilon = 0.05$$

HeaviSine.

$$f(t) = 2.3564 * [4\sin(4\pi t) - \mathrm{sgn(t - 0.3)} - \mathrm{sgn(0.72 - t)}]$$