Efficient Value Function Approximation Using Regression Trees

Xin Wang and Thomas G. Dietterich
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3202

Abstract

Value function approximation is a problem central to reinforcement learning. Many applications of reinforcement learning have relied on neural network function approximators, which are very slow to train and require substantial parameter tweaking to obtain good performance. Other reinforcement learning studies have applied nearest neighbor and CMAC function approximators, but these cannot scale to problems with many features, especially if some features are irrelevant. We describe initial work on a new function approximation method that uses regression trees to represent value functions. A novel aspect of our method is its error criterion, which combines three terms: the supervised training error, a Bellman error term, and an advantage error term. By using this composite error criterion, we are able to combine many of the benefits of fitted value iteration, $TD(0)$, and advantage updating. The new method is compared experimentally to previous work that employed $TD(\lambda)$ to solve job-shop scheduling problems (Zhang & Dietterich, 1996). The results show that the new method performs as well as the neural network method employed in that work, and that it can be trained in much less time. Our new method shows promise of providing a function approximator that is much more efficient and much easier to apply than neural network methods.

1 Introduction

Most reinforcement learning algorithms are based on estimating the value function (or the action-value function). For problems with very large or infinite state spaces, some form of value function approximation method must be employed. Existing function approximation methods fall into three major classes.

In many well-known applications, such as Tesauro's TD-Gammon (1992), Zhang's space shuttle payload scheduling system (1995, 1996), and Crites' elevator scheduling system (1996), neural network function approximators were employed. These approximators can represent a very large class of smooth functions, and they can handle correlated and irrelevant features quite well. However, careful tuning of the parameters and the training procedures is required in order to get these methods to work well. Worst of all, these methods require very long training times. Zhang's space shuttle system required nearly a CPU week to train successfully.

A second class of function approximation methods are the linear methods (e.g., (Tsitsiklis & van Roy, 1996)). These train much faster, but they have difficulty with correlated features, and they require the programmer to carefully design the input features so that the value function is approximately linear.

The third class of function approximation methods are local methods, such as nearest-neighbor regression (Peng, 1995), local linear regression (Atkeson, Moore, & Schaal, 1996), and the CMAC (Albus, 1975; Sutton, 1996). These methods work well in small-to-medium-sized spaces, but as the number of features grows large, these methods fail. They also fail if there are irrelevant or correlated features. Hence, successful application of these methods requires the programmer to identify a small set of nearly-independent features.

Our long-term goal is to develop a function approximation methodology that is fast, gives good performance, requires little-or-no tuning of parameters, and works well when the number of features is very large and when the features may be irrelevant or correlated. This paper describes a step toward this goal. We present a function approximation based on regression trees and show that it gives good performance on a job-shop scheduling task and that it can be trained very quickly. Although we have only tested the method on job-shop scheduling problems, we believe it can be extended to work in many Markov decision problems.

The remainder of the paper is organized as follows. First, we describe our overall approach. Second, we give the details of the method. Third, we evaluate the method experimentally on a job-shop scheduling problem. Finally, we discuss the strengths and weaknesses of the approach.

1.1 Notation

We will assume that we are solving a Markov decision problem over a finite set of states S, a finite set of actions A, a transition probability function $P(s'|s,a)$ (which gives the probability of landing in state s' after performing action a in state s), and a reward function $R(s'|s,a)$ (which gives the reward received when action a is performed in state s and results in state s'). The value function will be denoted $V(s)$, and the state-action function will be denoted $Q(s,a)$. Approximations of V and Q will be denoted by $\hat{V}$ and $\hat{Q}$.

2 General Approach: Batch Function Approximation

Zhang and Dietterich (1995) introduced a general framework for applying reinforcement learning to combinatorial optimization problems. The key assumption of this framework is that the user desires to solve many instances of combinatorial optimization problems drawn from a single general class of problems. For example, Zhang and Dietterich studied instances of Space Shuttle Payload Processing. Learning proceeds by analyzing a set of training problem instances and constructing a value function. This value function is then applied to solve new ("test") problem instances.

We have developed and tested our regression tree function approximation method within this combinatorial optimization framework. Our method is divided into three phases. In the first phase, an offline search is performed on each of the training problem instances to find good trajectories through the state space. The result of this first phase is the construction of a subtree of the entire search space for each problem instance, with the hope that this subtree contains good solutions to that instance.

In the second phase, we construct estimated values for each of the nodes in the subtrees. Starting at the leaf nodes of the subtrees, values are propagated backwards through the subtrees using Bellman's equation. The result is that state s visited during the offline search is assigned a value $V(s)$. Because each state is represented as a feature vector $\mathbf{x}(s)$, this gives us an ordered pair $(\mathbf{x}(s), V(s))$ for each state.

In the third phase, we fit a value function approximator to these ordered pairs. This paper focuses on this third phase, and shows how a regression tree algorithm can be applied. After the regression tree has been trained, it can then be applied to solve new problem instances.

In this paper, we do not consider the possibility of iterating this batch training procedure. One can imagine using the learned value function to perform a new offline search in the hope of finding even better trajectories for training. These improved trajectories could be used to train another

Table 1: Pseudo-code for the regression tree algorithm.

```
GrowRegressionTree(S, w_S):
    where S is a set of training examples of the form (x(s), V(s))
    w_S is a linear function that minimizes the composite error E(S, w_S).
        Choose best candidate splitting plane p for S.
        Split S into S_L and S_R according to p.
        Fit a linear function w_L to S_L to minimize the composite error E(S_L, w_L)
        Fit a linear function w_R to S_R to minimize the composite error E(S_R, w_R)
        If E(S, w_S) - [E(S_L, w_L) + E(S_R, w_R)] <= 0.05 · E(S, w_S)
            return new leaf(S, w_S)
        else return new node(p,
                             GrowRegressionTree(S_L, w_L),
                             GrowRegressionTree(S_R, w_R)).
```

regression tree, and the process could be repeated until convergence. We show below that for the problems we studied, the first regression tree that we construct already performs as well as Wei Zhang's neural network value function approximator.

3 A Regression Tree Function Approximator

3.1 Structure

Our regression trees are binary trees. Each internal node contains a splitting plane that divides the feature space into two half-spaces corresponding the node's left and right child nodes. Each leaf node in the tree contains a linear function defined over the feature space. Given feature vector $\mathbf{x}$, its value is predicted by "dropping" it through the tree, obeying the splitting plane at each internal node, and evaluating the linear function at the leaf node.

3.2 Rationale

We had two reasons for investigating regression trees. First, like decision trees, regression trees can be trained quickly. Existing algorithms for regression trees employ a top-down divide-and-conquer strategy, which rapidly breaks the overall problem into a modest number of individual linear fitting problems, which are easy to solve.

Second, regression trees show promise of being good representations for value functions in combinatorial optimization problems. Our job-shop scheduling domain contains irreversible actions, and this gives rise to value functions with discontinuities. To see why, consider two states, s_1 and s_2 such that s_2 can be reached by applying an operator in s_1, but from s_2 there is no way to reach a good solution. Then $V(s_2) < V(s_1)$. If the feature vectors for s_1 and s_2 are similar, then the value function will change discontinuously between these two points. Neural network and local linear methods can't represent this, because they always produce continuous function approximations, but regression trees can use splitting nodes to capture these discontinuities.

3.3 Growing the Regression Tree

Table 1 shows our training algorithm in pseudo-code. There are two key steps in any regression tree algorithm: (a) choosing the splitting planes at the internal nodes, and (b) fitting the planes at the leaf nodes.

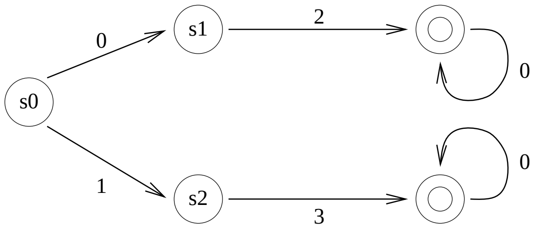

Figure 1: A simple Markov Decision Process. All transitions are deterministic. Each arc is labeled with the reward obtained from traversing that arc.

Choosing Splitting Planes. We wish to put splitting planes where there are discontinuities in the value function. To avoid searching the infinite space of splitting planes, we borrowed an idea from Hinton and Revow's (1996) decision-tree algorithm in which splitting planes are defined by taking the plane that lies mid-way between two training examples. They considered all pairs of training examples belonging to different classes, and evaluated each of these candidate planes (by one-step lookahead) to choose the best one.

Our analysis above suggests that discontinuities in the value function will be found primarily between pairs of states (s_1, s_2) where s_2 can be reached in one step from s_1. Our algorithm proceeds by randomly drawing a sample of 50 such pairs. For each candidate pair, let $\mathbf{x}_1$ and $\mathbf{x}_2$ be the corresponding feature vectors. Then $\mathbf{m}_{1,2} = (\mathbf{x}_1 - \mathbf{x}_2)/2$ is the point that lies midway between $\mathbf{x}_1$ and $\mathbf{x}_2$. A new instance $\mathbf{x}$ is closer to $\mathbf{x}_1$ if $(\mathbf{x} - \mathbf{m}) \cdot (\mathbf{x}_1 - \mathbf{x}_2) > 0$ or, equivalently, if $\mathbf{x} \cdot (\mathbf{x}_1 - \mathbf{x}_2) > \mathbf{m} \cdot (\mathbf{x}_1 - \mathbf{x}_2)$. This defines the splitting plane.

Each candidate split is evaluated by one-step lookahead. The training examples at the current node are partitioned according to the splitting plane, and the two resulting sets of points are fitted with linear surfaces as described below. The split that gives the best overall fit is chosen.

Fitting Linear Functions to the Leaf Nodes. The other key step in our algorithm is the fitting of leaf nodes to the training data. In supervised learning, the usual practice is to find the linear surface that minimizes the squared error between the predicted and the actual values. We will call this the supervised error:

$$E_1 = \sum_s \left[V(s) - \hat{V}(s) \right]^2. \tag{1}$$

This method has been applied in reinforcement learning (e.g., (Boyan & Moore, 1995)), but it is easy to see that a least squares fit may not give the correct policy. Consider the simple MDP shown in Figure 3.3. State s_0 has two child states, s_1 and s_2. Suppose the states are represented by one feature with the values 0, 1, and 2, respectively. The target values for V are 4, 2, and 3 respectively, so s_2 is the preferred child of s_0. Figure 2 shows a least-squares fit to this data. This line incorrectly predicts that s_1 has a higher value than s_2. In contrast, consider the line labeled "Composite Fit". The squared error of this line is larger, but it correctly predicts that $\hat{V}(s_2) > \hat{V}(s_1)$, so it gives the correct policy. The problem arises because of the bias of linear functions. The linear function cannot exactly fit the three points, and the trade-offs it makes among the errors at s_0, s_1, and s_2 cause it to give the wrong policy.

An alternative to minimizing the supervised error is to search for a $\hat{V}$ that satisfies the Bellman

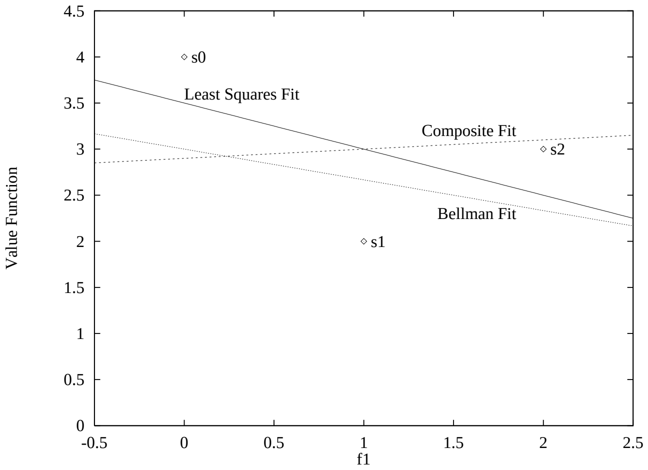

Figure 2: Three different approximations of a value function.

equation. To do this, we can define the Bellman error,

$$E_2 = \sum_s \left(\hat{V}(s) - \max_a \sum_{s'} P(s'|s,a)[R(s'|s,a) + \hat{V}(s')] \right)^2. \quad (2)$$

and then search for a linear $\hat{V}$ that minimizes it. Unfortunately, the high bias of linear approximations may also prevent them from satisfying the Bellman equation too. Figure 2 shows the best fit for our problem (labelled "Bellman Fit"); it still has a negative slope, so it still gives the wrong policy.

Our approach takes inspiration from Leemon Baird's (1993) Advantage Updating algorithm. Let $Q(s,a)$ be the return of performing action a in state s:

$$Q(s,a) = \sum_{s'} P(s'|s,a)[R(s'|s,a) + V(s')]. \quad (3)$$

Let a_{best} be the action that has the highest Q value. Then the *advantage* of performing action a in state s is defined as

$$A(s,a) = Q(s,a) - Q(s,a_{best}). \quad (4)$$

The advantages of all actions except a_{best} are negative. We can then define the *advantage error* to be

$$E_3 = \sum_s \sum_a \left(\left[\hat{Q}(s,a) - \hat{Q}(s,a_{best}) \right]_+ \right)^2 \quad (5)$$

where the notation $(x)_+$ is 0 if x is negative, and x if x is positive. Hence, if the predicted advantage of a is positive, then we have an error (since all advantages should be negative), and we square that error.

To train the leaves of the regression tree, we combine all three error terms to give a weighted composite error

$$E = \omega_1 E_1 + \omega_2 E_2 + \omega_3 E_3. \quad (6)$$

Here, the ω_i are all non-negative, and they control the relative importance of the supervised, Bellman, and Advantage error terms. Our hope is that by choosing appropriate values for the ω_i, we can obtain a fit such as the "Composite Fit" shown in Figure 2, which will give the correct policy.

The central claim of this paper is that all three terms in this error formula are important. The supervised term helps get the predicted values into the right general range. The Bellman term ensures that the predictions are mutually consistent. And the Advantage term ensures that the value function will yield a good policy.

Many popular reinforcement algorithms, including TD(0), SARSA(0), and Q-learning, are so-called "bootstrapping" algorithms, where the training signal for one state depends on the current predicted values of other states. One reason these algorithms are slow to learn is that they must bootstrap off of an initial value function that may be very far from the true value function. The supervised error term may help avoid this by forcing reasonable initial values for the bootstrapping process. Furthermore, because the best action a_{best} in the advantage term is defined using the supervised values (rather than bootstrapped values), it should be more stable and reliable.

We employ a simple stopping rule to terminate the growth of the regression tree. If the improvement in the composite error is less than 5%, when a parent node is split into two leaf nodes, then growth is terminated.

4 Experimental Test of the Method

To test this method, we conducted a series of experiments with resource-constrained scheduling problems. We employed the ART-1 data set developed by Wei Zhang (1996), which is a set of synthetic scheduling problems designed to be similar to space shuttle payload processing problems. A complete description of the data set along with the actual problems used can be obtained from the authors. Here is a brief summary: The data set contains 100 problem instances. Each instance has 3 or 4 jobs, and each job consists of a partially-ordered set of 6 to 10 tasks. This partial order represents the prerequisite relationships among tasks (these are generated at random such that 60% of all possible pair-wise prerequisite constraints are present). Each task also has a duration (6 to 15 time units) and a list of required resources. There are two types of resources, and each task requires 0 to 6 units of each resource. There are 14 units of each resource available, but for each resource type, these 14 units are divided into two resouce pools of sizes 6 and 8. A task must draw all of its units of a resource from one of these two pools. Finally, each job has a due time by which time all of its tasks must complete. The separation in due times between the jobs within a problem instance is between 8 and 15 time units. The goal of the scheduling process is to find the shortest overall schedule that completes all jobs before their deadlines without violating any resource or prerequisite constraints.

Wei Zhang formulated resource-constrained scheduling as a Markov Decision Problem in which the states correspond to complete schedules (i.e., every task has an assigned start time), but those schedules may contain violations of the resource constraints. The starting state is a critical-path schedule (i.e., a schedule in which all tasks are scheduled as late as possible, subject to prerequisite constraints; resource constraints are ignored). Three actions are available to move from one state to another: (a) Pool Reassignment (which changes which resource pool is used by a task to acquire its resources), (b) Move Forward (which moves a task to the next later time at which all of its resource requirements can be satisfied), and (c) Move Backward (which moves a task to the next earlier time at which all of its resource requirements can be satisfied). When a task is moved, all of its temporal dependents (i.e., its prerequisites and their prerequisites, recursively) are also moved and re-scheduled in a critical-path fashion.

To formulate a reward function, Wei Zhang defined a normalized measure of schedule length called the Resource Dilation Factor (RDF, see (Zhang & Dietterich, 1995; Zhang, 1996)). The RDF provides a normalized measure of the length of a schedule that takes into account the difficulty of individual scheduling problems. The reward function gives a reward of -0.001 for each move that results in a schedule that still contains violations, and a reward of $-RDF$ for a move that results in a violation-free schedule.

The goal of Zhang's research was to learn an approximately optimal policy for choosing operators in order to move from the starting schedule to a violation-free final schedule that minimizes the RDF. He applied a very complex neural network training procedure based on the $TD(\lambda)$ algorithm. The online training process required a great deal of parameter tuning, and consumed approximately 7 CPU days. Of the 100 problem instances in the ART-1 data set, Zhang used 20 for training, 30 for cross-validation, and 50 for the test set. The learned evaluation function was able to find very good final schedules using one-step lookahead greedy search. At each state s, the program generates all possible successor states $s_1, s_2, \ldots, s_n$, computes their estimated values using the neural network, and chooses the successor state with the highest estimated value.

The neural network used 22 input features, which were carefully designed by Wei Zhang. In our experiments, we used these same input features and the same one-step greedy algorithm for applying the learned value function approximation. However, we employed the batch training process and the regression tree function approximator described above. We employed 25 of the problems for training, 25 for cross-validation (to choose ω_1, ω_2, and ω_3, and the learning rate for gradient descent), and 50 for testing.

The initial offline search was performed by using a beam search with a beam width of 20 and using the RDF of the current state as the heuristic evaluation function. When the beam search for each problem instance is complete, we have between 20 and 80 complete paths from the start state to a terminal state for each problem instance. The state space can contain loops, but these are detected and removed by the beam search. After the search, the backed up value of each visited state is computed as follows. Each terminal state (i.e., either a solution state or a state that was not expanded further) is evaluated and assigned a value equal to $-RDF$ for that state. Then the values of non-terminal states are computed recursively by taking the minimum of the values of their successor states (plus 0.001 for the "cost" of applying the search operator). To the extent that the beam search is poorly informed and incomplete, these backed-up values will be wrong. But they provide a basis for training a value function approximator.

In preliminary experiments, we computed the backed up values of all states visited during the beam search and used them all for training the regression trees. However, the results on the cross-validation set were quite bad. We discovered, however, that if we used only the states along the k best paths, for various small values of k, the performance was much better. One explanation for this is that our value function approximator is quite biased, and it is not able to represent the value function over the entire state space. However, for good performance, we only need to get a good approximation of the value function in the neighborhood of the optimal policy. Interestingly, several researchers have suspected that something similar occurs with neural networks and the $TD(\lambda)$ algorithm. If ϵ-greedy exploration is combined with $TD(\lambda)$, the effect is to maintain a "current policy" and collect training examples that consist of "local" departures from that current policy. Hence, at any point in time, the neural network is only trained on a set of paths that are slight variations from the current "best" path. This means that the neural network does not need to fit the value function over the entire state space, but only in the neighborhood of the current best policy.

Hence, in our experiments, we only trained on the k best paths for each problem instance (where k is chosen by cross-validation). To compute the k best paths, we sort the terminal (leaf) states by their RDF values, and choose the k best states. All paths that lead from the initial state to these

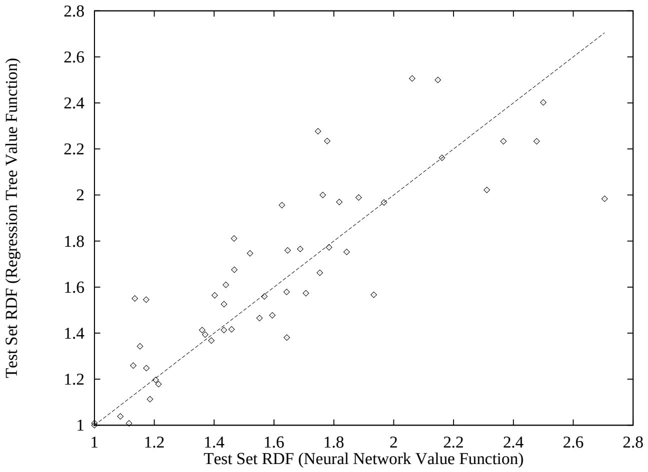

Figure 3: Regression tree (1 best path and using RDF) vs Neural net on test set

leaf states are marked. In addition, all states that can be reached by applying one operator to any of these marked states are also marked. Then all marked states are used to train the regression tree.

To choose the values for the various parameters, we performed some initial training runs to find a learning rate for gradient descent that assured convergence without overly long training times. We choose the following learning rates: for $k = 1$, 0.01, for $k = 6$, 0.015, and for $k = 30$, 0.006. We then evaluated all combinations of the following parameters: $k = 1$, 6, 30, $\omega_1 = 0.6$, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2, $\omega_2 = 0$, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, and $\omega_3 = 0$, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9. This required a total of 1,200 runs of the regression tree training algorithm. The best values of the parameters were $k = 1$, $\omega_1 = 1.2$, $\omega_2 = 0.7$, and $\omega_3 = 0.5$.

4.1 Results

On the test set, the chosen configuration produces performance very similar to the performance of the neural network. On the 50 test problems, the regression tree gives better results for 23, gives the same result for 3, and gives worse results for 24 problems. A 2-sided t test cannot reject the null hypothesis that the two algorithms are giving the same performance ($p = .84$). Figure 3 plots the RDF values for the 50 test problems along with the diagonal line $y = x$. Points lying above the line correspond to problems where the neural network gave a smaller (better) RDF; points below the line correspond to problems where the regression tree gave the larger RDF. From this, we conclude that the regression trees are giving essentially the same performance as the neural network function approximator.

The results also show that it was important to include the Bellman and Advantage terms in the objective function for fitting the regression tree. If we train using only the Supervised term (E_1), the regression tree is clearly worse than the neural network (it finds better solutions in 4 problems but worse solutions in 46 problems). Similarly, if we train using only the Bellman term (E_2), the regression tree is better in 3 problems and worse in 47. And if we train using only the Advantage term, the regression tree is better in 6 problems and worse in 44.

What happens if we train using two of the three terms? To test this, we alternately set each of the three terms to zero while leaving the other terms at their best values. If the Supervised term is

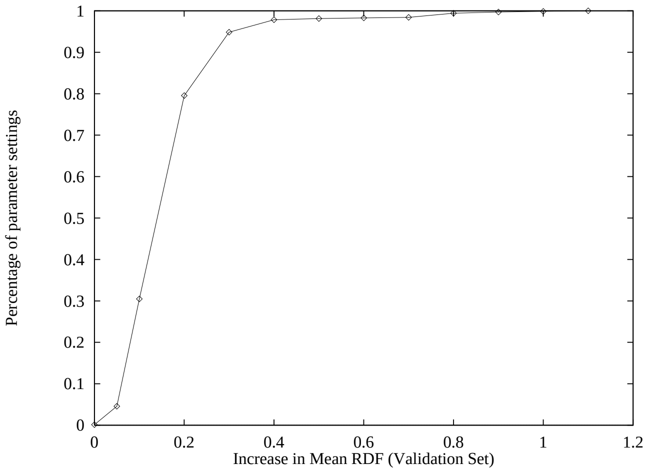

Figure 4: Cumulative distribution of the mean validation set RDF for various settings of the ω's

omitted, the regression tree is better in 1 problem and worse in the remaining 49. If the Bellman term is omitted, the regression tree is better in 14, tied in 3, and worse in 34 problems. And if the Advantage term is removed, the regression ree is better in 11, tied in 1, and worse in 38. This provides clear evidence that all three terms are playing an essential role. It also suggests that the Supervised term is the most important, the Advantage term is the second most important, and the Bellman term is the least important.

Given that the three terms are all important, how sensitive are the results to the exact settings of ω_1, ω_2, and ω_3? Figure 4 attempts to answer this question by plotting the cumulative distribution of the validation results. The horizontal axis shows the RDF increasing (with zero defined to be the RDF of the best parameter settings on the validation set). The vertical axis shows the percentage of parameter settings having a mean RDF less than or equal to the RDF on the horizontal axis. We can see that 30% of the parameter vectors that we tested give mean RDF's within 0.1 RDF units of the best. The fact that this curve rises steeply shows that there are many parameter settings that give very good performance. An examination of the individual data points suggets that a good setting for the ω parameters should give at least half of the total weight to the Supervised term, and the rest of the weight should be divided more-or-less equally among the other two terms.

The final question we address is training time. The training time for most parameter settings is approximately 1450 seconds on an Intel i686 processor under Linux (g++ implementation). The time to perform the offline search is very short (approximately 10 seconds). In comparison, the training time for the neural network method was approximately 1 CPU week. However, that was measured on hardware that we estimate was 6 times slower, so this can be very roughly translated to 28 CPU hours or approximately 100,000 seconds. Hence, we have a speedup of at least a factor of 50. This comparison ignores the amount of time required with both methods to tune the various parameters controlling the learning process. Nonetheless, we can see that the regression tree method is clearly much faster than the neural network function approximation scheme.

One interesting observation is that if ω_2 is set too large relative to ω_3, the training time increases to appoximately 5000 seconds. This shows that there is also a CPU-time benefit to combining the Bellman and Advantage terms.

5 Discussion

There are four issues that must be addressed to make this method generally applicable. First, the method requires reasonable estimates of the values of the states in order to evaluate the Supervised term. In deterministic combinatorial optimization problems, such values are easy to obtain by applying standard search algorithms. But for highly-stochastic Markov decision problems, such values are harder to find. In domains where a simulator is available, Monte Carlo methods (e.g., roll-outs or more sophisticated methods, see (Boyan & Moore, 1996; Bertsekas & Tsitsiklis, 1996; Tesauro & Galperin, 1997)) can provide reasonable estimates. But in situations where learning is entirely online, our approach will not be applicable.

Second, the method currently requires internal cross-validation to set the various parameters (learning rate, relative weighting of the error terms). More experience is needed across several domains to discover a general method for setting these parameters. It may be that a fixed setting of the parameters will work well across many domains, but we expect that the optimal setting will depend on the degree of stochasticity of the domain and the accuracy of the initial supervised values. Hence, we believe that some adaptive method for setting the parameters will need to be developed.

Third, the method is currently a one-pass batch algorithm. Many reinforcement learning algorithms require continuing, incremental learning. Regression trees are not well-suited to incremental learning (although Utgoff (Utgoff, 1989) has developed good incremental algorithms for decision trees). So we believe the most fruitful direction for future research is to alternate between exploratory search (guided by the current approximate value function) and batch fitting of a new regression tree.

Fourth, our current splitting rule assumes that all features are equally relevant (and uncorrelated). We need to improve the rule to perform some kind of feature selection during splitting so that irrelevant and correlated features can be detected and ignored.

6 Conclusions

This paper has introduced a new method for value function approximation based on regression trees. The key aspect of the new method is the combination of three terms in the error function: a Supervised term, a Bellman term, and an Advantage term. The experiments show that all three terms are essential to achieve good performance.

This paper has also shown that regression trees can provide a powerful and efficient value function approximation method. On a resource-constrained scheduling task, the regression tree method is able to match the performance of a highly tuned neural network function approximator while reducing the amount of CPU time required by a factor of at least 50.

Acknowledgements

The authors gratefully acknowledge the support of NSF Grant 9626584-IRI and AFOSR Grant F49620-98-1-0375.

References

Albus, J. S. (1975). A new approach to manipulator control: The cerebellar and articulation controller (CMAC). *Journal of Dynamic Systems*, *97*, 220–227.

Atkeson, C. G., Moore, A. W., & Schaal, S. (1996). Locally weighted learning. Tech. rep., Dept of Computer Science, GATECH.

Baird, L. C. (1993). Advantage updating. Tech. rep. WL-TR-93-1146, Wright-Patterson Air Force Base.

Bertsekas, D. P., & Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming.* Athena Scientific, Belmont, MA.

Boyan, J. A., & Moore, A. W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In Tesauro, G., Touretzky, D. S., & Leen, T. K. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 7, pp. 369–376. The MIT Press, Cambridge.

Boyan, J. A., & Moore, A. W. (1996). Learning evaluation functions for large acyclic domains. In *Proc. 13th International Conference on Machine Learning*, pp. 63–70. Morgan Kaufmann.

Crites, R. H., & Barto, A. G. (1996). Improving elevator performance using reinforcement learning.. In Touretzky et al. (Touretzky, Mozer, & Hasselmo, 1996), pp. 1017–1023.

Hinton, G. E., & Revow, M. (1996). Using pairs of data-points to define splits for decision trees.. In Touretzky et al. (Touretzky et al., 1996), pp. 507–513.

Peng, J. (1995). Efficient memory-based dynamic programming. In Prieditis, A., & Russell, S. (Eds.), *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 438–446 San Francisco, CA. Morgan Kaufmann.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding.. In Touretzky et al. (Touretzky et al., 1996), pp. 1038–1044.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, *8*, 257–277.

Tesauro, G., & Galperin, G. R. (1997). On-line policy improvement using Monte-Carlo search. In Mozer, M. C., Jordan, M. I., & Petsche, T. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 9, p. 1068. The MIT Press.

Touretzky, D. S., Mozer, M. C., & Hasselmo, M. E. (Eds.). (1996). *Advances in Neural Information Processing Systems*, Vol. 8. The MIT Press, Cambridge.

Tsitsiklis, J. N., & van Roy, B. (1996). Feature-based methods for large scale dynamic programming. *Machine Learning*, *22*, 59–94.

Utgoff, P. E. (1989). Incremental induction of decision trees. *Machine Learning*, *4*, 161–186.

Zhang, W., & Dietterich, T. G. (1995). A reinforcement learning approach to job-shop scheduling. In *1995 International Joint Conference on Artificial Intelligence*, pp. 1114–1120. Morgan Kaufmann, San Francisco, CA.

Zhang, W. (1996). *Reinforcement Learning for Job-Shop Scheduling.* Ph.D. thesis, Oregon State University, Department of Computer Science.

Zhang, W., & Dietterich, T. G. (1996). High-performance job-shop scheduling with a time-delay TD(λ) network. In Touretzky, D. S., Mozer, M. C., & Hasselmo, M. E. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 8, pp. 1024–1030. MIT Press, Cambridge, MA.