Computer Science
Technical Report

Colorado State University

Decision Tree Function Approximation in Reinforcement Learning

Larry D. Pyeatt Adele E. Howe
Colorado State University
Fort Collins, CO 80523
email: {pyeatt,howe}@cs.colostate.edu
URL: http://www.cs.colostate.edu/~{pyeatt,howe}

October 15, 1998

Technical Report CS-98-112

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

Decision Tree Function Approximation in Reinforcement Learning

Larry D. Pyeatt Adele E. Howe
Colorado State University
Fort Collins, CO 80523
email: {pyeatt,howe}@cs.colostate.edu
URL: http://www.cs.colostate.edu/~{pyeatt,howe}

October 15, 1998

Abstract

We present a decision tree based approach to function approximation in reinforcement learning. We compare our approach with table lookup and a neural network function approximator on three problems: the well known mountain car and pole balance problems as well as a simulated automobile race car. We find that the decision tree can provide better learning performance than the neural network function approximation and can solve large problems that are infeasible using table lookup.

1 Motivation

A popular approach for estimating the value function in reinforcement learning is the table lookup method. This approach is guaranteed to converge, subject to some restrictions on the learning parameters [2]. However, table lookup does not scale well with the number of inputs, although some variations of this approach, such as sparse coarse coding and hashing [9], have been used to improve scalability. Another approach is to use a neural network to learn the value function. That approach scales better, but is not guaranteed to converge and often performs poorly even on relatively simple problems [3]. We propose an approach that exploits decision trees for learning to estimate the value function.

We started to investigate this problem because we are building a reinforcement learning based agent for two simulated robotic environments: Robot Automobile Racing Simulator (RARS) and Khepera. The dimensionality of these problems was too large for a table lookup method. we found that a major drawback to a standard neural network based reinforcement learning implementation was its tendency to over-train on the portion of the state space that it visits often and forget the value function for portions of the state space that it has not visited recently. This leads to a cycle where it learns to perform well for a time and then begins to perform poorly. In this paper, we

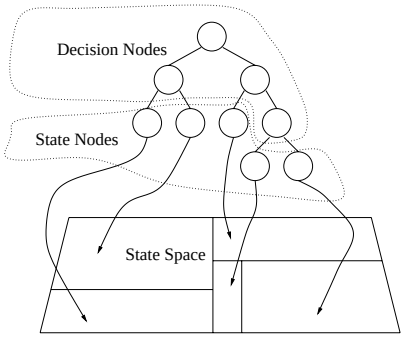


Figure 1: Dividing the state space with a decision tree.

show empirically that our new approach avoids this problem by providing stable and reliable convergence to the estimated value function.

2 Approaches to Estimating the Value Function

The goal in reinforcement learning is to find the optimal policy. The *optimal* policy is the mapping from states to actions that maximizes the sum of the rewards. The value of a state is defined as the sum of the rewards received when starting in that state and following the policy to a terminal state. The value function can be approximated using any general function approximator such as neural network, look-up table, or decision tree.

2.1 Table Lookup

Table lookup does not scale well with the number of dimensions in the space. Even the simplest implementation for the RARS robot has six inputs. Table lookup with each input dimension divided into 20 regions would result in 20^6 table entries.

2.2 Neural Network

The neural network approach can solve larger problems than table lookup but it is not guaranteed to converge. In practice, the neural network approach often performs poorly even on relatively simple problems [3].

2.3 Our Approach: Decision Tree-Based

The straightforward table lookup method subdivides the input space into equal intervals. Each part of the state space has the same resolution. A better approach would allow high resolution only where needed. Some attempts have been made to use variable resolution tables, with limited success [3]. Decision trees [6] allow the space to be divided with varying levels of resolution. Figure 1 shows an example of a decision tree that divides

the state space into 5 regions. The tree can be used to map an input vector to one of the leaf nodes, which corresponds to a region in the state space. Reinforcement learning can be used to associate a value with each region.

G-learning [5] uses a decision tree to learn compact representations of the value function for problems with binary inputs. Our approach extends the method to an algorithm that handles real values. The goal is to provide robust convergence along with scalability. The decision tree is learned along with the policy.

2.4 Overview of Algorithm

We use a variation of reinforcement learning known as Q-learning [11, 12], which maps state-action pairs instead of states. The estimated value $Q(s_t, a_t)$ of a state-action pair is updated according to the equation

$$\Delta Q(s_t, a_t) = \alpha[r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

where t is the current time step, r_{t+1} is the immediate reward received at time $t+1$, s_t is the state at time t, and a_t is the action performed at time t. α is a learning parameter such that $0 \leq \alpha \leq 1$. γ controls the ratio of immediate reward to return from future states.

Our decision tree contains two types of nodes: decision nodes and leaf nodes. A *decision node* represents a single decision about one input variable. This decision determines which branch is taken to find the next node. Each *leaf node* stores the estimated values for its corresponding region in the state space. We use Q-learning, so each leaf node stores one value for each possible action that can be taken, along with statistics which are used to decide whether the region represented by the node should be split.

The decision tree starts out with only one leaf node that represents the entire input space. As the algorithm runs, the leaf node gathers information in its history list. When it has enough information, a test is performed to determine whether the leaf node should be split (see section 2.5). If a split is required, the test also determines the decision boundary. A new decision node is created to replace the leaf node, and two new leaf nodes are created and attached to the decision node. The old leaf node is then deleted. In this manner, the tree grows from the root downward, continually subdividing the input space into smaller regions. Figure 2 summarizes our algorithm.

2.5 When and Where to Split a Leaf Node

The decision about when a node should be split and where to place each decision boundary is crucial. We investigated three methods from the decision tree literature plus a new method that is similar to G-learning. All four of the methods use mean and standard deviation of $\Delta Q(s_{t-1}, a_{t-1})$ to determine if a node should be split (see Figure 2), but they use different algorithms to choose a decision boundary.

Information Gain This is the classic method used in Quinlan's ID3 [8]. It measures the information gained from a particular split.

1. Receive input vector and reward r_t for time t.
2. Use input vector to find a leaf node representing state s_t.
3. Select the action a_t with the largest value of $Q(s_t, a_t)$, or select a random action with some small probability.
4. If the action was not chosen at random, calculate $\Delta Q(s_{t-1}, a_{t-1})$ and update $Q(s_{t-1}, a_{t-1})$.
5. Add $\Delta Q(s_{t-1}, a_{t-1})$ to the history list for the leaf node corresponding to s_{t-1}.
6. Decide if s_{t-1} should be divided into two states by examining the history list for s_{t-1}.
   (a) if history_list_length $<$ history_list_min_size then split $:=$ False
   (b) else
      i. calculate average μ and standard deviation σ of $\Delta Q(s_{t-1}, a_{t-1})$ in the history list
      ii. if $|\mu| < 2\sigma$ then split $:=$ True
      iii. else split $:=$ False
7. Perform split, if required
8. Save r_t, a_t and s_t so that they can be used for training on the next iteration.
9. Return a_t.

Figure 2: Algorithm for decision tree based reinforcement learning.

Gini Index This metric is based on the Gini Criterion by Breiman [4], but modified as in OC1 by Murthy [7]. The Gini Index measures the probability of misclassifying a set of instances.

Twoing Rule This metric, also proposed by Breiman and used in Murthy's OC1, compares the number of examples in each category on each side of the proposed split.

T-statistic Our approach is based on the T-statistic. The algorithm calculates the means and variances for each input variable. If the node has not received any positive $\Delta Q(s_{t-1}, a_{t-1})$ in its history list, then the input variable with the highest variance is chosen as the decision variable for the new decision node. Otherwise, the decision is made by calculating the T statistic for each variable and selecting the variable with the highest T statistic. This approach is similar to that used by Chapman and Kaelbling [5], although we remove the restriction that all inputs be binary.

3 Empirical Performance Study

To assess the performance of our decision tree based reinforcement learning algorithms, we compared them to table lookup and neural network reinforcement learning on three problem domains. Our study focused on answering the following questions:

1. Which reinforcement learning algorithm performs best after training?
2. How quickly does each algorithm learn?

Table 1: Performance during the fourth period: each column shows average and standard deviation.

	Mountain Car		Pole Balance		RARS Crashes		RARS Times	
	Ave.	Sd	Ave.	Sd	Ave.	Sd	Ave.	Sd
Gini	206	0.7	No result		15.7	22.4	3617	751.8
Info Gain	170	2.2	2000	0	1.1	1.9	2956	82.8
NN	328	45.4	1008	225.6	2.8	4.6	1172	566.0
Table	160	13.6	2000	0	No result		No result	
T-test	170	0.3	2000	0	0.01	0.1	723	14.8
Twoing	219	53.4	1848	360.2	1.4	2.7	1869	418.0

3. Is the decision tree based approach less prone to the learn/forget cycle than the neural network approach?

3.1 Problem Domains

Mountain car is a classic reinforcement learning task where the goal is to learn the proper acceleration to get out of a valley and up a mountain [3]. The car does not have enough power to simply climb up the mountain, so it has to rock back and forth across the valley until it gains enough momentum to carry it up the mountain.

Pole balance is another classic problem where the goal is to balance a pole that is affixed to a cart by a hinge [1]. The cart moves in one dimension on a finite track. At each time step, the controller decides whether to push the cart to the left or to the right.

RARS is an environment where a simulated race car driver is responsible for controlling acceleration and steering as the car races against other cars [10].

3.2 Results

For each problem domain, we ran all of the reinforcement learning algorithms and then divided the total run time for each algorithm into four periods. The number of iterations in each domain was determined by how long it took the algorithms to reach a stable policy. On the Mountain car problem, we ran each algorithm for a total of 10,000 trials; each trial lasted for 10,000 time steps or until the car reached the goal state. On the pole balancing problem, we ran each algorithm for 20,000 trials of 2,000 time steps or until the controller failed to maintain the pole in a balanced position. In the RARS domain, each algorithm was run for 300 laps around the track, regardless of how many time steps were required.

3.2.1 Performance after learning

We calculated a one way ANOVA on each period with algorithm as the independent variable and performance as the dependent variable. The F ratio for the Mountain car

problem was $F = 113.45$. For the pole balance problem, $F = 219.67$. For RARS lap time, $F = 506.13$ and for crashes, $F = 29.43$. For all tests, $P < 0.01$, indicating statistically significant differences. Table 1 shows the performance for each algorithm over the fourth period.

Mountain car: The table lookup method performed better than any other method. However, information gain and T-test decision tree methods also performed well. We performed a one tailed T-test of information gain vs. T-test methods and found no significant difference ($t = 1.35, p < 0.18$).

Pole balance: Information gain, table lookup and T-test all achieved perfect performance and were able to balance the pole for 2000 time steps for every trial in period four. The neural network approach did not converge well and was rather unstable.

RARS: The T-test approach shows significantly better performance than the other methods. The neural network approach performed well, but crashed a great deal more often than any of the top three decision tree methods.

3.2.2 Time to learn

We processed the data using a 9 mean smooth to show longer trends (see Figure 3).

Mountain car: This is the easiest of the three problem domains, having only two inputs and three possible actions. All of the decision tree methods converged to about the same level of performance.

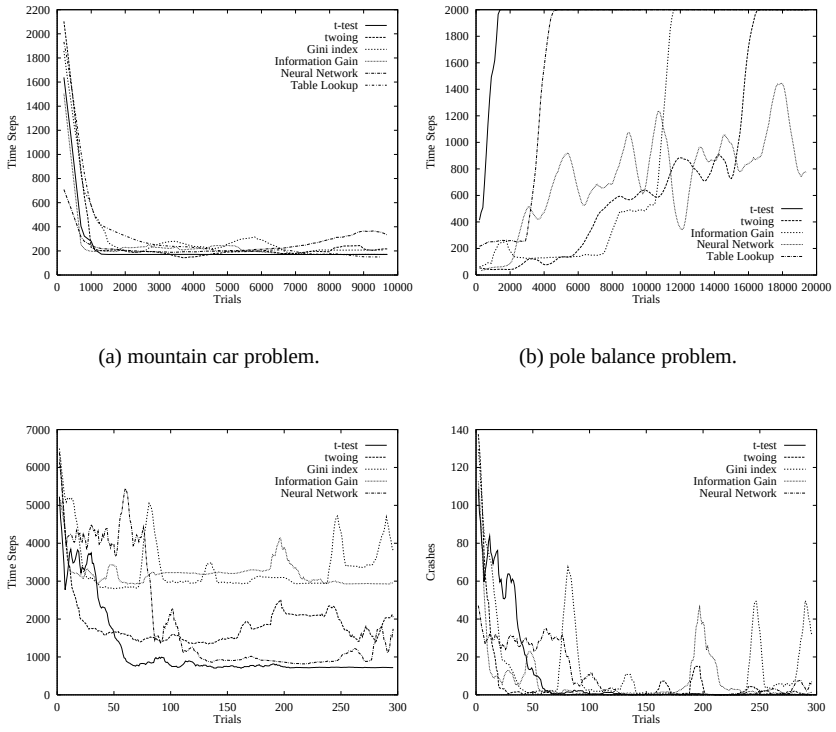
Pole balance: This domain is a little more difficult than the Mountain car problem. The T-test decision tree method converged to a stable solution after less than 2,000 trials and successfully balanced the pole thereafter. Only the neural network approach failed to find a stable solution within the time limit.

RARS: The T-test decision tree method quickly finds a good state-space representation and achieves good performance. The neural network approach also finds a good policy. The other decision tree methods do not perform as well; they were still searching for suitable state-space representations at the end of the run.

3.2.3 Learn/Forget cycle

The main motivation for this work was to overcome the learn/forget cycles that we had encountered using the neural network approach. We assessed whether a method suffered from a learn/forget cycle by examining the learning curves and comparing standard deviation (shown in Table 1) in its ultimate (period four) performance.

Mountain car: The neural network approach initially found a good policy, but became overtrained after about 6,000 trials. The high standard deviations show that the neural network and Twoing value approaches were unstable. The T-test, Gini, and information gain methods had low standard deviations.



(a) mountain car problem.

(b) pole balance problem.

(c) time steps per lap in RARS.

(d) number of crashes per lap in RARS.

Figure 3: Smoothed learning performance.

Pole balance: The T-test, table lookup, and information gain methods all had zero standard deviation and perfect performance.

RARS: The T-test method had the lowest standard deviation in both performance measures. The neural network approach performed almost as well as the T-test based decision tree method for some time. However, after about 250 trials, the neural network became overtrained. Its high standard deviation indicates that it did not converge to a stable solution at that time.

4 Future Work and Conclusions

Following recent work in the decision tree literature, we will augment our approach to use oblique instead of axis parallel decision boundaries. Oblique boundaries lead to smaller decision trees by allowing each node to use several input variables.

We have evaluated four methods for *selecting* the decision boundaries. In our future work, we plan to explore some alternative approaches with a view to characterizing how

well each approach works in a given domain. We do not expect to find a single method that works best in all cases.

Decision tree based reinforcement learning provides good learning performance and meets our needs for more reliable convergence than the neural network approach. It also has lower memory requirements than the table lookup method, and scales better to large input spaces.

References

[1] A.G. Barto, R.S. Sutton, and C.W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:834–846, 1983.

[2] Andrew G. Barto and Richard S. Sutton. *An Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, 1997.

[3] Justin A. Boyan and Andrew W. Moore. Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D.S. Touretsky, and T. K. Leen, editors, *Advances in Neural Information Processing Systems 7*, Cambridge, MA, 1995. MIT Press.

[4] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and regression trees. Technical report, Wadsworth International, Monterey, CA, 1984.

[5] David Chapman and Leslie Pack Kaelbling. Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In John Mylopoulos and Ray Reiter., editors, *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 726–731, San Mateo, Ca., 1991. Morgan Kaufmann.

[6] Kolluru Venkata Sreerama Murthy. *On Growing Better Decision Trees from Data*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, 1996.

[7] Sreerama K. Murthy, Simon Kasif, and Steven Salzberg. A system for induction of oblique decision trees. *JAIR*, 2:1–33, 1994.

[8] J R Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[9] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, pages 1038–1044, Cambridge, MA, 1996. MIT Press.

[10] Mitchell E. Timin. Robot automobile racing simulator (RARS). Anonymous ftp `ftp.ijs.com:/rars`, 1995.

[11] Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.

[12] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.