Decision Tree Function Approximation in Reinforcement Learning

Larry D. Pyeatt
Texas Tech University
Lubbock, Texas 79409
larry.pyeatt@ttu.edu

Adele E. Howe
Colorado State University
Fort Collins, CO 80523
howe@cs.colostate.edu

Abstract

The goal in reinforcement learning is to learn the value of taking each action from each possible state in order to maximize the total reward. In scaling reinforcement learning to problems with large numbers of states and/or actions, the representation of the value function becomes critical. 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

Reinforcement learning is an approach to learning by interacting with the environment. It is modeled on the type of learning that occurs in nature. When we do something that brings a positive reward (pleasure) we tend to do the same thing again. Likewise, actions that bring negative reward are avoided. The ability to link cause and effect (sometimes incorrectly) is a basic ability shared by even the simplest animals. The central idea of reinforcement learning algorithms is Temporal Difference (TD) learning.

An agent based on reinforcement learning receives inputs from the environment and uses them to select what action to take. In the simplest case, the inputs are assumed to give complete information about the state of the agent in its world. As shown in Figure 1, the agent receives a reinforcement signal from the environment along with the current state at each time step. At each time step, the reinforcement can be positive, negative or zero. The goal is for the agent to maximize the average reinforcement that it receives over time by creating an optimal action selection *policy*. A policy is a function that maps states to actions. The reinforcement learning agent finds a policy by learning an estimate of the value of each possible state. Another way to say this is that the agent learns to approximate the *value function*. A value function is simply some function that maps inputs representing the current state s to a real number v that is the expected long-term reward that can be earned when starting from that state.

A popular approach for representing 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 [11]. However, table lookup does not scale well with the number of inputs. Some variations of this approach, such as sparse coarse coding and hashing [10], have been used to improve scalability somewhat. 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 [2]. Our alternative is to use a decision tree to represent the value function.

We began investigating this problem because we are building a reinforcement learning based agent for two simulated robotic environments: Robot Automobile Racing Simulator (RARS) and Khepera, a desktop robot. 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 behavior is a direct result of the fact that backpropagation networks using sigmoidal transfer functions perform non-local updates to the function that they are learning. This leads to a cycle where it learns to perform well for a time and then begins to perform poorly. In this paper, we show empirically that our approach avoids this problem by providing stable and reliable convergence to the estimated value function.

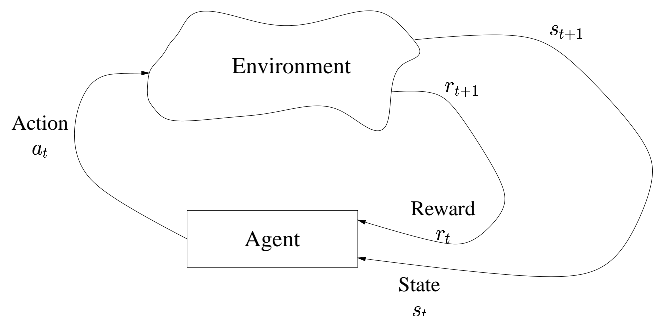


Figure 1: An agent using reinforcement learning interacts with its environment by receiving states and rewards from the environment, and generating actions that change the environment.

2 Approaches to Value Function Estimation

The goal in reinforcement learning is to find the *optimal* policy. The optimal policy is the mapping from inputs (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 is the simplest approach to reinforcement learning. Each dimension is quantized into a number of bins. If there are n inputs and each input is quantized into m bins, then there are n^m entries in the table that is used to store the value function approximation. Unfortunately, table lookup does not scale well with the number of dimensions in the space. For example, even the simplest implementation for our RARS robot has six inputs. Table lookup with each input dimension divided into 20 regions would result in 20^6 table entries. This is obviously not feasible. Each action requires a table, so it would take several megabytes of storage to represent the value functions. Also, the rate of learning would be very slow, since each table entry must be visited many times before the approximation becomes accurate.

2.2 Neural Network

A back-propagation neural network with a sigmoidal activation function can be used to learn the value function. This approach can solve larger problems than table lookup but it is not guaranteed to converge. Although there have been some impressive successes using neural networks as function approximators for reinforcement learning [12], the neural network approach often performs poorly even on relatively simple problems [2]. The problems associated with using standard backpropagation neural networks in reinforcement learning stem from the fact that these networks perform non-local changes to the value function, while reinforcement learning requires that updates to the value function be local. When updating the value of a specific state or state action pair, the network can destroy the learned value of some other state. Thrun and Schwartz [13] discuss another problem with using back-propagation networks. Due to the nature of back-propagation and reinforcement learning, it is very easy for the system to consistently over-estimate the utility of state-action pairs, resulting in improper credit assignment. For these and other reasons, the back-propagation approach with sigmoidal activation functions is not guaranteed to converge. This problem is alleviated somewhat by using experience replay and other techniques, but at the cost of much higher computation.

One promising approach is the use of radial basis functions (RBFs) instead of sigmoidal activation functions [5]. This approach alleviates the problems of non-local update, but tends to have problems representing the value function at the edge of the state space and introduces some waviness in the value function. Also, selecting the widths and centers of the RBFs can be difficult.

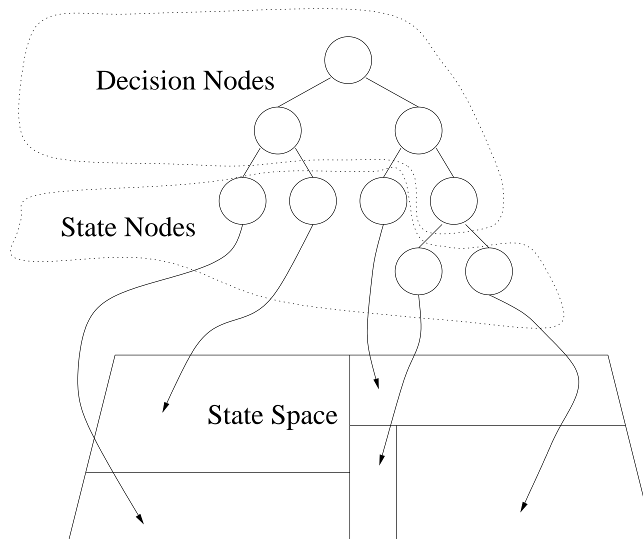


Figure 2: Dividing the state space with a decision tree.

2.3 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 [2]. Decision trees [7] allow the space to be divided with varying levels of resolution. Figure 2 shows an example of a decision tree that divides the state space into five 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 learn the values of taking each action in each region.

Using a decision tree to approximate the value function is not a new idea. Utile Distinction Memory (UDM) [6] uses batched analysis of statistics gathered over many steps to partition the state space by splitting states of a finite state machine. The resulting state space representation can be represented as a decision tree. UDM uses the t-test to determine when to split a state.

The G-learning algorithm [4] uses a decision tree to generate a variable resolution discretization for a space with binary inputs. This algorithm starts by representing the world as a single state and then recursively splits states where necessary. G-learning determines when to split a node by performing a t-test on historical data.

The U Tree [15] algorithm, which was developed by Uther and Veloso, extends G-learning to work in continuous state space. Like G-learning, our algorithm keeps historical infor-

mation and occasionally performs batch processing to determine what states need to be split. Uther and Veloso evaluated the Kolmogorov-Smirnov test and sum-squared error as a means to determine where and when to split a leaf node, but did not evaluate using the t-test.

Our approach also extends G-learning to work in a continuous state space, while requiring less historical information than UDM. The goal is to provide robust convergence along with scalability. The decision tree is learned along with the policy. Our work is very similar to the U Tree algorithm, but we explore some alternative metrics for determining when and where to subdivide the space and we analyze performance in different domains. Also, the U Tree algorithm uses batched updating of the state space representation while we do continuous updating. With batch updating, all leaf nodes are inspected periodically. Continuous updating inspects leaf nodes whenever they are visited, which results in a more uniform distribution of evaluations over time, and avoids evaluating leaf nodes that have not been visited recently.

2.4 Overview of Algorithm

At the core of our algorithm, we use a variation of reinforcement learning known as Q-learning [16, 17], which maps state-action pairs instead of states. At each time step, the change in the estimated value $Q(s_t, a_t)$ of a state-action pair is calculated as

$$\Delta \leftarrow \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right], \qquad (1)$$

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. The value of $Q(s_t, a_t)$ is then updated by $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \Delta$.

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 a history of the inputs and rewards that have been received. The history list is 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 the list reaches a threshold length, 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,

1. Receive input vector v and reward r_t for time t.
2. Use input vector v to find a leaf node representing state s_t.
3. Select the action a with the largest value of $Q(s_t, a)$, 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 $\Delta Q(s_{t-1}, a_{t-1})$ and v 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 3: Algorithm for decision tree based reinforcement learning.

the tree grows from the root downward, continually subdividing the input space into smaller regions. Figure 3 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 3), but they use different algorithms to choose a decision boundary.

Information Gain This is the classic method used in Quinlan's ID3 [9]. It measures the information gained from a particular split.

Gini Index This metric is based on the Gini Criterion by Breiman [3], but modified as in OC1 by Murthy [8]. The Gini Index measures the probability of misclassifying a set of instances.

Twoing Rule This metric, which was also used in Murthy's OC1 and proposed by Breiman, 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 in G-learning, although we remove the restriction that all inputs be binary, allowing our algorithm to learn in a continuous state space.

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. How quickly does each algorithm learn?
2. Which reinforcement learning algorithm performs best after training?
3. Is the decision tree based approach less prone to the over-training 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 [2]. 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 [14]. This domain is more difficult than the others, since there are two control signals and more inputs.

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 problem domain was determined by how many trials were needed for the algorithms to reach a stable policy. In the mountain car problem domain, 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. In the pole balancing problem domain, 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 problem domain, each algorithm was run for 300 laps around the track, regardless of how many time steps were required.

All algorithms were run 10 times in each domain, and we calculated the average performance for each trial. For the mountain car domain, we collected the number of steps taken to reach the goal. For pole balancing, we collected the time that the pole remained balanced. For the RARS domain, we collected the time taken to complete a lap and the number of times that the car crashed. We processed the data using a 9 mean smooth to show longer trends and constructed graphs.

3.2.1 Time to learn

We are interested in how each algorithm performed during learning. In particular, how long does it take for each algorithm to reach a good, stable solution? Figures 4–6 show the performance of the various algorithms.

Mountain car: This is the easiest of the three problem domains, having only two inputs and three possible actions. As shown in Figure 4, all of the decision tree methods converged to about the same level of performance, with little difference in the time that they took to find a good solution.

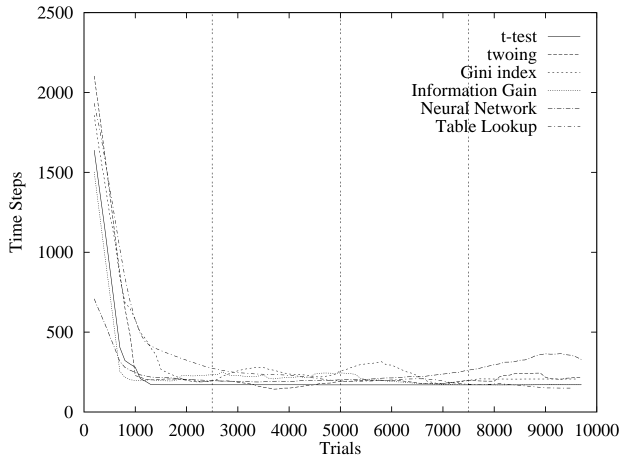


Figure 4: Smoothed learning performance on the mountain car problem.

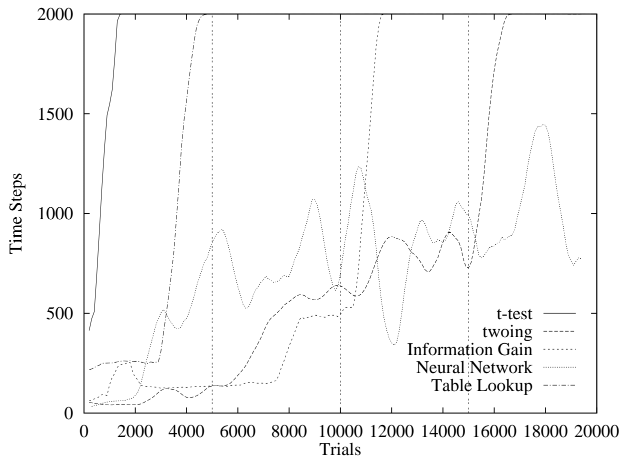
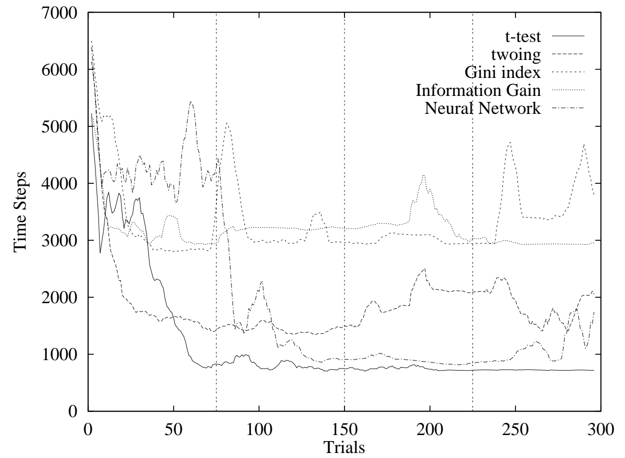
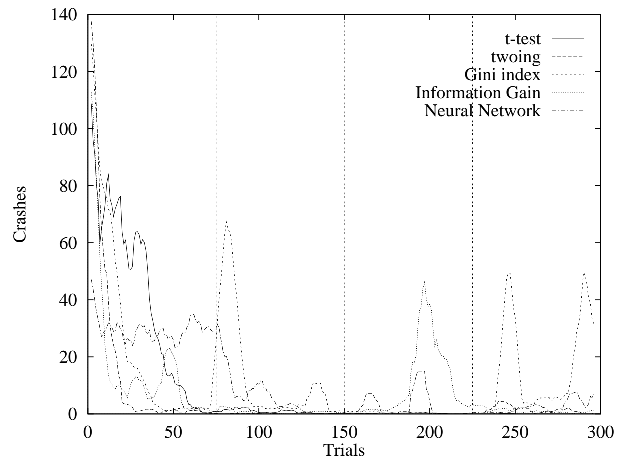


Figure 5: Smoothed learning performance on the pole balance problem.



(a) time steps per lap.



(b) number of crashes per lap.

Figure 6: Smoothed learning performance for RARS.

Pole balance: This domain is a little more difficult than the mountain car problem. Figure 5 shows the relative performance of the various algorithms on this problem. The t-test decision tree method converged to a stable solution after less than 2,000 trials and successfully balanced the pole thereafter. The table lookup method took a little longer, but still found a solution in only 5,000 trials. Only the neural network approach failed to find a stable solution within the time limit.

RARS: This problem is too large to be solved using the table lookup method; so we only compare the performance of the various decision tree methods and the neural network approach. Figure 6 shows the performance of these algorithms. 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.

Overall, it appears that the t-test decision tree method performs best. The neural network approach worked well in two of the three domains, but performed poorly for pole balancing.

3.2.2 Performance after learning

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. ANOVA is a statistical technique which tests differences in mean values of a dependent variable between two or more categories of independent variables. We calculated a one way ANOVA on each period with algorithm as the independent variable and performance as the dependent variable. For all problems, $P < 0.01$, indicating statistically significant performance differences (mountain car: $F = 113.45$, pole balance: $F = 219.67$, RARS lap time: $F = 506.13$, and RARS crashes: $F = 29.43$). Table 1 shows the performance for each algorithm over the fourth period, where learning should be complete and performance should be stable.

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 ($t = 1.35, p < 0.18$). The table lookup method had higher standard deviation than either information gain or t-test, indicating that the state space representation of the decision tree approaches may have been better suited to the problem than the fixed grid representation used in table lookup.

Pole balance: Information gain, table lookup and t-test all achieved perfect performance and were able to balance the pole for 2,000 time steps for every trial in period four. The neural network approach did not converge well and was rather unstable. The twoing decision tree converged to a stable solution during period four.

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.

Overall, the t-test method performed best. It provided the best lap times and the lowest number of crashes in the RARS domain, and gave perfect performance in pole balancing. On the mountain car problem, the t-test method was slightly worse on average than table lookup, but performance had far less variation from one trial to the next.

3.2.3 Over-Training cycle

The major motivation for this work was to overcome the over-training cycles that we had encountered using the neural network approach. We assessed whether or not a method suffered from the over-training 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 method initially found a good policy, but became over-trained after about 6,000 trials. The high standard deviations show that the neural network and twoing value approaches were unstable. Figure 4 shows that the performance of the neural network method actually deteriorated noticeably during period four. The t-test, Gini, and information gain methods had low standard deviations, indicating that they had reached a stable solution.

Pole balance: The t-test, table lookup, and information gain methods all had zero standard deviation and perfect performance. The twoing method found a good solution during period four, as shown in Figure 5. The neural network approach never found a good solution to this problem.

RARS: The t-test method had the lowest standard deviation in both performance measures indicating that it had found a stable solution. 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 over-trained. Its high standard deviation indicates that it did not converge to a stable solution at that time. Figure 6a shows that the performance of the neural network approach was good throughout period three and actually decreased during period four.

In two of the three domains, the neural network approach showed signs of over-training, and in the remaining domain it failed to find a good solution at all. The t-test approach

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

performed best in two of the domains and was outperformed by table lookup in one domain. The table lookup method gave the best performance in the mountain car and second best on pole balancing, but could not be used on the RARS domain. Overall, the t-test approach was the clear winner.

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 [8]. 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. In particular, we will compare our work to the methods and domains used by Chapman and Kaelbling [4]. 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. The t-test method for selecting the decision boundary gave the best performance overall in the domains that we studied.

5 Acknowledgements

This research was supported partly by National Science Foundation Career Award IRI-9624058. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation herein.

Bibliography

[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] 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.

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

[4] 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.

[5] R. Matthew Kretchmar and Charles W. Anderson. Comparison of cmacs and radial basis functions for local function approximators in reinforcement learning. In *Proceedings of the International Conference on Neural Networks*, pages 834–7, IEEE Services Center 445 Hoes Lane P.O. BOX 1331 Picataway, NJ 08855-1331, June 1997. IEEE, IEEE Press.

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

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

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

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

[10] 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.

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

[12] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, March 1995.

[13] Sebastian Thrun and Anton Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the Fourth Connectionist Models Summer School*. Lawrence Erlbaum, Hillsdale, NJ, December 1993.

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

[15] William T. B. Uther and Manuela M. Veloso. Tree based discretization for continuous state space reinforcement learning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-98)*, Madison, WI, July 1998. AAAI.

[16] Christopher J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Kings College, Cambridge, UK, 1989.

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