# IMPROVING THE PERFORMANCE OF Q-LEARNING WITH LOCALLY WEIGHTED REGRESSION

By

HALIM ALJIBURY

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF ENGINEERING

UNIVERSITY OF FLORIDA

2001

ACKNOWLEDGMENTS

This work would not have been completed without the guidance of the members of my thesis committee, Prof. Antonio Arroyo, Prof. Mike Nechyba, and Prof. Eric Schwartz. Prof. Keith Doty also deserves credit for helping me formulate the original question regarding the improvement of Q-learning.

I am grateful for the facilities at the Machine Intelligence Laboratory here at the University of Florida, which gave me the ability to carry out my research. My fellow members of the lab also deserve thanks for providing an enriching experience during my years as a graduate student.

TABLE OF CONTENTS


| | | Page |
|---|---|---|
| ACKNOWLEDGMENTS | | ii |
| ABSTRACT | | v |
| CHAPTERS | | |
| 1 | INTRODUCTION | 1 |
| | Problem Description | 1 |
| | Thesis Organization | 2 |
| 2 | BACKGROUND OF REINFORCEMENT LEARNING | 4 |
| | Description of Reinforcement Learning | 4 |
| | Definition of a Markov Decision Process (MDP) | 5 |
| | Policies in MDPs | 6 |
| | Optimizing a Policy | 6 |
| | Optimality Criteria | 7 |
| | Value Functions | 8 |
| | Derivation of Policy and Value Iteration Rules from the Bellman Equations | 9 |
| | Optimization with Q-Learning | 10 |
| | Limitations of Discrete-state Q-Learning | 11 |
| | Progression of RL | 13 |
| | Q-Learning Advancements | 13 |
| | SARSA | 13 |
| | Getting around the Dimensionality Factor | 14 |
| | Approximating the Value Function | 14 |
| | Applying MDP Methods to POMDP Problems | 16 |
| | Description of Perceptual Aliasing | 17 |
| | Dealing with POMDPs | 17 |
| | Approximation after Learning is Complete | 19 |
| | Locally Weighted Regression for Approximation | 20 |
| 3 | SIMULATOR DESIGN AND IMPLEMENTATION | 22 |
| | Choice of Physical Platform | 22 |
| | Description of the Platform | 22 |
| | Simulator Considerations | 23 |

Simulation Implementation Generalities……... 24
Arena and Obstacles……..…….. 24
Infrared Calculation…….……..….. 25
Infrared Algorithm…….…….. 26
Collision Detection Algorithm……..…….…. 26
Robot Extraction Algorithm…… 26
Other Simulator Features……..……... 27
Ease of Porting Code……..……..…… 37

4 THE EXPERIMENTATION…… 28
Q-Learning Parameters…….…… 28
Q-Table Generation……... 30
Approximating the Value Function with a Neural Network…….. 32
Approximating the Value Function with Locally Weighted Regression…… 32
Other Weighting Functions……..……....… 33
Operation in Different Environments……..… 34

5 DISCUSSION OF RESULTS……..…. 36
Predicted vs. Observed Q-Table Convergence……..…… 36
Vector Quantization……..…… 36
Perceptual Aliasing……...…… 39
Neural Networks Cannot Map this Problem's Value Function……..…… 41
Basis for the Performance Comparison……..... 42
Mechanism of Performance Enhancement……..… 44

6 CONCLUSIONS AND FURTHER DIRECTIONS……...…. 46
Summary of Contributions…… 46
Future Directions…… 46

REFERENCES……..… 48

APPENDIX…….……...….. 52

BIOGRAPHIC SKETCH…….……...…. 53

Abstract of Thesis Presented to the Graduate School
Of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Engineering

IMPROVING THE PERFORMANCE OF Q-LEARNING WITH
LOCALLY WEIGHTED REGRESSION

By

Halim Aljibury

December 2001

Chair: A. Antonio Arroyo
Major Department: Electrical Engineering

This thesis describes a method of dealing with the large or continuous state spaces encountered in large reinforcement learning problems. The problem is first approached via learning with a coarse quantization of states. The value function is then approximated to extend the results to the original large or continuous state space. The second step in this method improves the problem performance by allowing the agent to use all of the originally available state information.

Performance of this two-step method is significantly improved compared to performance on the coarsely quantized problem. The nature of this method makes it directly applicable to practical reinforcement learning problems where perceptual aliasing or interminable learning times are a problem.

A simulator is used to demonstrate this method on the task of obstacle avoidance in a complicated environment, in which a moderate amount of perceptual aliasing is present. Significant performance enhancements are observed, as well as a learned policy that is not

dependent on state history. This differentiates the method described here from most current methods of dealing with perceptual aliasing that involve a memory of previous states to distinguish two otherwise identical states from each other.

# CHAPTER 1
# INTRODUCTION

## 1.1 Problem Description

Oftentimes, the problem faced by researchers applying reinforcement learning to a nontrivial robotics problem is that they run head-on into the curse of dimensionality. This is a particular problem for those researchers using discrete-state algorithms, as the number of states exponentially increases with the complexity of the problem. Various methods of dealing with large state spaces have been proposed and experimentally demonstrated, but none of these methods have effectively dealt with practical situations where the difference between states is sometimes not perceivable by the robot. It can be shown that many useful reinforcement learning algorithms experience theoretical difficulties under these conditions, and that other algorithms which explicitly account for incomplete perception either lack generality, or are too computationally awkward for effective use.

This thesis demonstrates a method by which the performance of a discrete-state algorithm can be improved when applied to a continuous-state problem in combination with a function approximator. The method consists of two steps. The first step consists of learning the value function over a small number of discrete states. The second step involves using the function approximator to generalize from those discrete states to a continuous state space, allowing previously discarded state information to be used effectively in the problem solution.

## 1.2 Thesis Organization

This thesis has been divided into 6 chapters: introduction, problem background & related work, simulator design and implementation, experimentation, discussion, and conclusions & future directions. An appendix containing information about how to obtain the simulator source code and associated user's guide concludes the thesis.

The problem background covers the relevant background of reinforcement learning while leading to my approach to the problem. The background material is delved into in order to specifically work up to and define value functions for Markov decision processes (MDPs), which measure the utility of any particular policy mapping states to actions. I then go over the theoretical backing and useful applications of Q-learning so that the dimensionality problem is made evident. Several methods from the literature are presented which approach this topic, leading into the utility of my specific approach of increasing the real-world performance of Q-learning.

The simulator design and implementation chapter describes why the simulator was necessary, as well as the specifications and implementation details. In the interest of keeping the thesis focused on experimentation, implementation details are confined to describing the more novel elements of the code.

The chapter on experimentation covers generation of the Q-table, followed by descriptions of the methods used to approximate the value function. Results on the approximation by a neural network (NN) are presented and then followed up by the positive results of an approximation using locally weighted regression.

The discussion chapter analyzes the experimental results and issues surrounding them. A summation of the results is presented here, followed by the conclusion chapter,

which takes a look at the practical utility of this method. Finally, a short exploration is made of further directions in which this work could be taken.

# CHAPTER 2
# BACKGROUND OF REINFORCEMENT LEARNING

## 2.1 Description of Reinforcement Learning

The principle behind reinforcement learning (RL) is easy to conceptualize. In the general, observable RL problem, the agent is able to fully perceive its current state $s_t$. That state $s_t$ is the determination of the state of the agent's environment and the agent's position in that environment at a time or step t. For this part of the discussion, it is assumed that the state is a member of the set of discrete states S. Suppose an agent ( a robot) determines itself to be in a particular state. The agent will then choose and execute some action $a_t$ from some set A of (discrete) possible actions. The agent will then observe its inputs, and determine the new state $s_{t+1}$. After each action, the agent perceives its next state $s_{t+1}$, and receives a bounded reinforcement signal (or reward $r_{t+1)}$ that reflects the value to the agent of executing action $a_t$ in state $s_t$.

The goal of the agent is to find a policy $\pi$ mapping states to probabilities of executing actions that maximize the long-term reward of the agent. Reinforcement learning methods generally describe those algorithms by which an agent learns a behavior policy through feedback generated by trial-and-error. Reinforcement learning methods of solving for a behavior policy particularly lend themselves to situations where the world model is not known in advance [1].

In a reinforcement learning problem, the agent is only given an immediate reward signal, but is not told what actions are best in the long run. Actions that give large

immediate rewards may not be optimal in the long run. Consider, for example, the problem of driving a truck out of a steep valley, as shown in Figure 2-1.

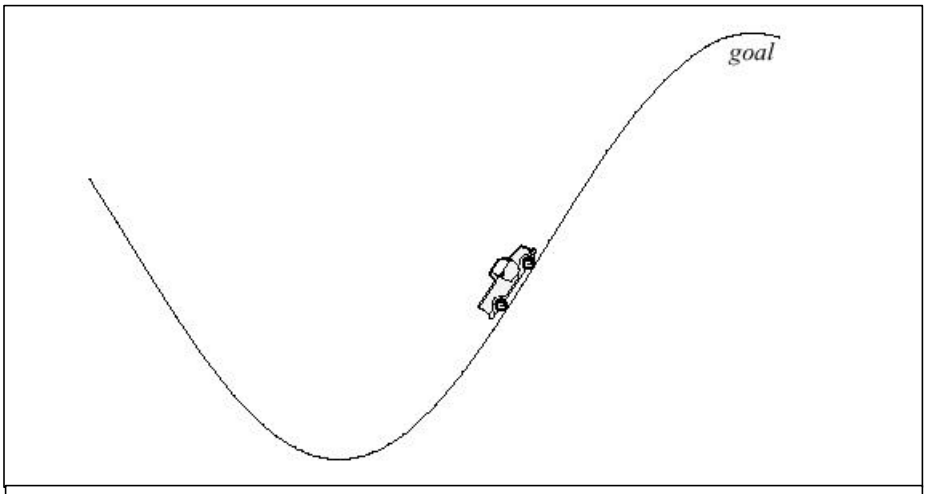


Figure 2-1. Truck on a Hill

Suppose the engine in the truck is not powerful enough to accelerate up the steep valley side to get to the goal. In order to reach the goal, the only solution is to first move away from the goal and up the opposite slope on the left. Then, by flooring it, the car can build up enough momentum to carry it up the steep slope on the right. Thus, the agent must first choose a sequence of actions that look worse in terms of getting closer to the final goal. If the agent does not choose these actions with negative immediate rewards, it will never be able to achieve its ultimate objective. Problems with delayed rewards, such as the above example, can be modeled as Markov decision processes (MDPs).

## 2.2 Definition of a Markov Decision Process (MDP).

An MDP consists of the tuple {S,A,R,T}, where

S = a finite set of states,
A = a finite set of actions,
R: S x A -> R = bounded reward function, where

$$R(s,a) = E(r_{t+1} \mid s_t = s, a_t = a) \quad (2\text{-}1)$$

T: S x A -> Π(S) = state transition function, where

$$T(s,a,s') = P(s_{t+1} = s' \mid s_t = s, a_t = a). \quad (2\text{-}2)$$

## 2.3 Policies in MDPs

In RL the goal of the agent is to maximize its long-term reward from the environment. In each state, the agent has a choice of possible actions. Which action the agent elects to execute in each state affects the agent's long-term reward, and influences future action selections. In order to discuss this method of action selection, it is necessary to define the concept of a policy.

An agent's policy $\pi(s,a)$ maps states to probabilities of selecting different actions where $\pi(s,a)$ denotes the probability of selecting an action a given that the agent is in state s at time t:

$$\pi(s,a) = P(a_t = a \mid s_t = s) \quad (2\text{-}3)$$

The object of RL is to find that policy $\pi^*(s,a)$ which maximizes the agent's long-term reward in other words, we want to find the optimal policy $\pi^*(s,a)$. There are two possible scenarios for this problem. In the first scenario, the agent can access the complete description of the MDP: i.e., all state transition probabilities, values of each state, and so forth. A collection of methods known as dynamic programming address this paradigm. In the second scenario, we will assume that we do not have a priori knowledge of the underlying MDP, which is the scenario that is the primary focus of RL. Dynamic programming methods useful in the first case, with some adaptations, form the basis of many RL methods.

## 2.4 Optimizing a Policy

When looking at a state-action policy covering an MDP, the primary concerns are these two questions:

What is the value (goodness) of a policy $\pi(s,a)$?
What is the optimal policy $\pi^*(s,a)$?

2.4.1 Optimality Criteria

What is meant by 'optimal long-term policy'? There are several ways one can set about defining the optimal policy. In general, the definition of 'optimal' can generate policies that differ greatly from each other. The most common methods of defining optimality are shown below:

$$J_1 = E\left[\sum_{t=0}^{h} r_t\right] \tag{2-4}$$

This criterion considers only h future rewards, and ignores all rewards beyond h time steps ahead. Here, $E[\cdot]$ denotes the expected value under policy $\pi$. The second optimality criterion $J_2$ is the infinite-horizon, discounted model:

$$J_2 = E\left[\sum_{t=0}^{\infty} \tilde{a}^t r_t\right] \quad 0 \le \tilde{a} < 1 \tag{2-5}$$

The discount parameter $\gamma$ discounts future rewards, obtaining a greater value for temporally close rewards, and is mathematically necessary to ensure that the sum is bounded. Finally, the third optimality criterion $J_3$ is the average-reward model:

$$J_3 = E\left[\frac{1}{h}\sum_{t=0}^{h} r_t\right] \tag{2-6}$$

For the average-reward optimality criterion, it is difficult to distinguish policies with different reward profiles over time. That is, immediate rewards are not weighed any more heavily than rewards in the distant future. Of the three optimality criteria, the infinite-horizon, discounted criterion $J_2$ is the most tractable analytically; therefore, it is most often used in practice. When mathematically analyzing RL problems, it is

important to realize that the varying optimality criteria will place different emphasis on the desirability of delayed rewards while determining the optimal policy. For this reason, the discussion of 'optimality' is restricted to that of the infinite-horizon discounted case $J_2$.

2.4.2 Value Functions

To find the value of a particular policy, a value function $V^{\pi}(s)$ must be defined such that $V^{\pi}(s)$ is the expected long-term reward for the MDP under the policy $\pi$.

$$V^{\pi}(s) = E_{\pi}[J_t \mid s_t = s] \tag{2-7}$$

In other words, $V^{\pi}(s)$ denotes the expected long-term reward under policy $\pi$ given that the agent starts in state s at time step t, and subsequently follows policy $\pi$. Since we are considering the infinite-horizon discounted case of the optimality criterion $J_t = \sum_{k=0}^{\infty} \tilde{a}^k r_{t+k+1}$, we obtain from Eq. (2-7)

$$V^{\delta}(s) = E_{\delta}\left[\sum_{k=0}^{\infty} \tilde{a}^k r_{t+k+1} \middle| s_t = s\right] \tag{2-8}$$

Let the action-value function $Q^{\pi}(s,a)$ be defined as

$$Q^{p}(s,a) = E_{\delta}\left[\sum_{k=0}^{\infty} \tilde{a}^k r_{t+k+1} \middle| s_t = s, a_t = a\right] \tag{2-9}$$

In other words, $Q^{\pi}(s,a)$ denotes the expected long-term reward under policy $\pi$ given that the agent starts in state s and executes action a at time step t, and subsequently follows policy $\pi$. Let us now define Q*(s,a) as the optimal action-value function where $Q^*(s,a) = \max_{\delta} Q^{\delta}(s,a)$ over all possible states and actions. As Bellman showed in [2],

a stationary deterministic optimal policy always exists. The problem then faced in RL is how to efficiently find that policy.

2.4.3 Derivation of Policy and Value Iteration Rules from the Bellman Equations

When deriving the optimal policy, it is important to realize that the best policy $\pi^*$ will have the value of all states s equal to their expected long-term reward. The following steps show the derivation of an iterative policy-update equation that converges to the optimal policy $\pi^*$.

$$V^*(s) = \max_a Q^*(s,a) \tag{2-10}$$

$$V^*(s) = \max_a E_{\delta*}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a\right] \tag{2-11}$$

$$V^*(s) = \max_a E_{\delta*}\left[r_{t+1} + \gamma \sum_{k=2}^{\infty} \gamma^k r_{t+k+2} \middle| s_t = s, a_t = a\right] \tag{2-12}$$

then, using the notation defined in the discussion of Markov problems:

$$V^*(s) = \max_a \left[R(s,a) + \gamma \sum T(s,a,s')V^*(s')\right], \forall S \in s \tag{2-13}$$

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q*(s',a'), \forall s \in S, \forall a \in A \tag{2-14}$$

Note that Bellman showed that the optimal policy is both stationary and deterministic. Using the above equations, we can define our optimal policy $\pi^*$ as:

$$\delta*(s) = \arg\max_a Q*(s,a), \forall s \in S \tag{2-15}$$

where

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} \left[T(s,a,s')V^*(s')\right], \forall s \in S \tag{2-16}$$

To compare any arbitrary policy against another, one must determine the value (or utility) of that policy. In the case of the optimal policy, that value can be determined iteratively by the following method [1]:

- Initialize V(s) to some value.
- Repeatedly perform the following:

- $Q(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s')V(s'), \forall s \in S, \forall a \in A$ (2-17)
- $V(s) = \max_a Q(s,a)$ (2-18)

When the maximum change in V(s) is epsilon-close to the previous value, then you have a value function that has converged as close as desired to the optimal value function V*(s). At this point, the optimal policy can be easily calculated from Eq. (2-15).

Conversely, an optimal policy can be iteratively obtained from an initial arbitrary policy $\pi(s)$ [3].

Repeat:

- Let $\pi(s) = \pi'(s), \forall s \in S$ (2-19)
- Evaluate $V\pi(s)$:
- $Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s')V^{\pi}(s')$, (2-20)
- Improve the policy $\pi(s)$:
- $\pi'(s) = \arg\max_a Q^{\pi}(s,a)$ (2-21)

until the improvement from $\pi$ to $\pi'$ is epsilon-small. At this point, $\pi'(s)$ is arbitrarily close to the optimal policy.

2.4.4 Optimization with Q-Learning

Watkins [4, 5] combined the two steps into a single-step update for the algorithm he named Q-learning. As a result, Q-learning is more computationally tractable, and has been proved to converge, while convergence proof is lacking for the separated steps [6].

Q-Learning Algorithm:

- Initialize Q(s,a) arbitrarily for all states and actions.
- Set s to some initial start state

Repeat:

- Select action a using the policy derived from current Q(s,a) function.
- Take action a and observe the reward r and next state s'.
- Update Q(s,a) based on the step just taken. (Eq. 2-22)
- Set the state s to the new state s'

The quantity $Q^{\pi}(x_i,a)$ is the predicted value of any particular state/action pair. Selecting the $\max_{a \in A} Q(x, a)$ over all states x will result in the ideal value function $V^{\pi}(x)$. Of course, since the method begins with an imperfect estimate of the Q-values of each state, an iterative evaluation and prediction update algorithm must be applied.

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma[\max_{a'} Q(s',a')] - Q(s,a)) \qquad (2\text{-}22)$$

The above algorithm has been proven to converge with probability 1 to the true Q-value $Q^{\pi}$, given that the problem is a MDP, all states are visited infinitely often, and that the Q-values are represented as discrete values in a look-up table [7, 5]

## 2.5 Limitations of Discrete-State Q-Learning

Kearns & Singh have demonstrated convergence bounds for a finite sequence in which the transition probabilities are well-mixed [8]. While Q-learning has been proven to converge to optimality, nothing has been proven about the convergence rate under non-optimal conditions. In practice, depending on what learning parameters are used, convergence to optimality (or near-optimality) can be extremely slow. Often, the central difficulty with the convergence rate is dependent on the number of discrete states. Reinforcement learning using discrete states has an inherent problem, arising from what is known as the 'curse of dimensionality.' As the number of state variables increases, the number of discrete states increases exponentially. To make matters worse, even if sufficient memory is available to store state values, the frequency with which the agent visits any particular state will proportionately decrease. Since RL methods generally require repeat visits to states in order to have the estimated value converge to the true value, the convergence time will also generally increase exponentially. This difficulty

tends to inhibit meaningful experimentation with discrete-state RL problems having more than a few free state variables.

For example, consider a robot with two infrared sensors each capable of 8-bit precision, plus being able to turn in any of 360 different directions and move at 8 different speeds. In order to apply Q-learning, we need to assume that the inputs will completely distinguish states, and that no hidden states remain in the problem. This assumption is rarely true in practice, but implications of this will be dealt with in Section 2.7.

The set of states for this problem would be every possible combination of sensor values times the number of possible actions, or $2^8 * 2^8 * .7*2^9 * 2^3$, or $.7*2^{28}$ [1] different states. Obtaining a value function covering every state in this problem's Q-table with the above Q-learning algorithm would be quite impossible. Even if all the state values could be stored (somewhat reasonable with current technology, roughly 200 megs of memory required), the sheer size of the problem would inhibit convergence, due to long sequences of actions needing many passes to propagate their values back to the start of the sequence. In fact, according to the bounds established by Kearns & Singh (Eq. 2-23), it would take nearly a trillion steps to approach within .01 of the true Q-values. And that's assuming well-mixed transition probabilities. If this weren't the case, the number of transitions would be greater.

$$Transitions = \left(\frac{N * \log(1/e)}{e^2}\right)(\log(N) + \log(\log(1/e))) \qquad (2\text{-}23)$$

[1] Values resulted from the following considerations: 2 IR sensors with 8-bit resolution = $2^8*2^8$ combinations, 360 possible rotation states: $360 = .7*2^9$, 8 possible actions = $2^3$

## 2.6 Progression of RL

### 2.6.1 Q-Learning Advancements

In the years since Watkins originated the Q-learning algorithm, it has received much attention from researchers due to its ease of implementation and sound theoretical backing. Encouraging results have also come from efforts to reduce the storage space and number of updates required.

First, Q-learning was proven under certain conditions to converge to an ideal model and policy [5]. Work from that point generally proceeded in two directions. The first direction was focused on reducing the number of updates (or state–transitions) required per state to approach near-optimal values. Watkins also proved convergence of $Q(\lambda)$ when $\lambda = 1$, as well as an alternate Q-learning algorithm which updates the values of multiple states at each iteration [9].

### 2.6.2 SARSA

Rummery & Niranjan came up with an algorithm named SARSA [10], whose difference from standard Q-learning is that the value used is for the next state/action actually taken, not the value selected with the max operator. This changes the algorithm from an off-line algorithm to an on-line algorithm. Jaakola and others proved that online TD methods converged, though the results only hold for MDPs [7, 11]. Littman used their results to show that SARSA converges to a defined value function, and that its performance is asymptotically optimal [12].

Gordon showed that SARSA can easily fail to converge when given a partially observable MDP (POMDP)[13]. In general, POMDP problems have proven very difficult to analyze, and few researchers have done other than decrease the PO character

of the problem by introducing a state-sequence memory. For examples of this, refer to [14, 15, 16]. Loch and Singh [17] however, established the utility of SARSA(λ) with eligibility traces to certain classes of POMDP problems for which near-optimal memoryless policies exist. Jordan et al. [18] also presented results regarding memoryless policies.

2.6.3 Getting Around the Dimensionality Factor

The other general area of work has focused on tackling the dimensionality problem by updating multiple states with information from a single action. This approach presupposes that states that are very similar to each other will likely have the same values for the state-action pairs. At each update, the algorithm updates the actual Q-value as well as the Q-values of all neighboring states [19]. Work in this area has been performed by, among others, Singh et al. [20].

An alternative approach to the discrete-state approach has been to simply abandon the practice of finely dividing the states to approximate continuous-valued conditions, and simply adopt a continuous-valued approach. The difficulty with this approach is that any technique which depends on a policy based on a memory of what happened when a specific state was previously visited will fail because of the utter improbability of a prior visit unless other information is available to generate the action values for that state. If a continuous value function can be produced which would map states to values, this would greatly simplify the generation of an optimal policy.

2.6.4 Approximating the Value Function

Considerable effort has been spent on generating a useful approximation of the value function from observed state transitions, so that an estimate can be obtained for the

value of an unvisited state. The reasoning behind such attempts can be described in the following manner: In a finely discretized state space, it seems logical to reason that for states which closely resemble each other, their values likewise also resemble each other. Unfortunately, such efforts do not rest on as sound a theoretical background, and in general, these efforts have met with mixed results, with successes in some specific applications, but failing in others. Tesauro met with eyebrow-raising success in using neural net methods [21]. However, I suspect that his particular application (backgammon) is abnormally well-suited to NN methods, because the game depends on the probabilities of dice combinations, the probabilities of which smoothly vary. Other researchers have shown that linear approximations are capable of approximating value functions, but no such proofs exist for nonlinear methods, despite some positive experimental results [22]. Linden [23] has pointed out some examples where NN methods are incapable of mapping some value functions that contain discontinuities. One example of Linden's is that of obstacle-avoidance, where the forbidden regions (inside the obstacles) produces discontinuities in the value function at the borders of the obstacles. It is interesting to note the difference between Linden's approach, that of using a gridworld, and the approach in this thesis of a reactive policy based on sensory inputs. While his approach guarantees the existence of forbidden regions in state space and discontinuities in his value function, my approach does not, and assumes that every point in state space is reachable.

Finally, Tsitsiklis and Roy [24] describe a proof of the convergence of on-line function approximators in finite as well as non-terminating Markov processes. In addition, they reconcile varying results in the literature regarding convergence (or lack

thereof) of function approximators by showing that the convergence depends on the sampling frequency of the various states. Convergence is only guaranteed if the states are sampled according to the distribution in their steady-state Markov chain. Doing otherwise tends to generate inaccurate state-action values, leading to non-optimal policies, and so forth.

## 2.7 Applying MDP Methods to POMDP Problems

Since the convergence properties of Q-learning and related algorithms are currently fairly well understood, one might expect wide application of them to real robot situations. However, there is a slight problem in moving from simulation to reality. It stems from the fact that the various RL algorithms are dependent on the fact that the Markov process being learned is entirely observable, i.e., that there are no hidden states. In real applications this becomes difficult to ensure.

In applying an algorithm such as Q-learning to a real problem, the researcher is typically hoping that acceptable results will still be obtained, despite some partially observable characteristics of the situation. The common sense reasoning (hope) is that the performance of MDP methods will decline gracefully as the PO character increases. As demonstrated by Gordon [13], and other researchers, this reasoning is proved false. Even one hidden state in an MDP can cause divergence. Still, although it is easy to contrive mathematical situations where MDP methods perform miserably, such examples show the 'worst case' performance of the algorithms [17]. Actual applications of MDP methods in experiments where problems exist that have good memory-less policies, have, in the main returned what the experimenters consider 'acceptable' results.

'Acceptable' here is understood to be not necessarily perfect behavior, but at least a significant increase in performance during the course of the experiment.

A common factor in many RL simulations is a 'gridworld', where the agent knows exactly what state it is in, since the state is equivalent to its location. In contrast, a typical experimental setup involves distinguishing states on the basis of the agent's sensory input (e.g., the infrared reflections measured by the agent.) Outside of some mechanism such as beacon triangulation, or a state transition memory to ensure that one location cannot be mistaken for another, there is always a risk of perceptual aliasing.

## 2.8 Description of Perceptual Aliasing

Ballard and Whitehead [25] showed that when using an agent's observational inputs to define states, it introduces the possibility of two different situations appearing the same to the agent. This is quite an issue when doing experiments with discrete-valued states, as the perceptual resolution suffers due to state-minimization. For instance, when quantizing an IR sensor to 4 different input values, as shown in Figure 2-2, a significant amount of information is lost. Figures 5-1 and 5-2 also demonstrate examples of perceptual aliasing.

## 2.9 Dealing with POMDPs

One method of dealing with POMDPs has been to add some memory to the agent so that two different states may be distinguished by the state sequence taken to reach them [15, 26] In my view, such solutions are undesirable when working towards a reactive policy, since they tend to make the agent more dependent on the exact environment. For instance, suppose an agent has generated a Q-table for an object-avoidance problem that possesses considerable POMDP character, but nevertheless has a

good memoryless policy. If the obstacles are moved around, a purely reactive method should perform about the same as before, while the performance of a memory-based method would likely be degraded.

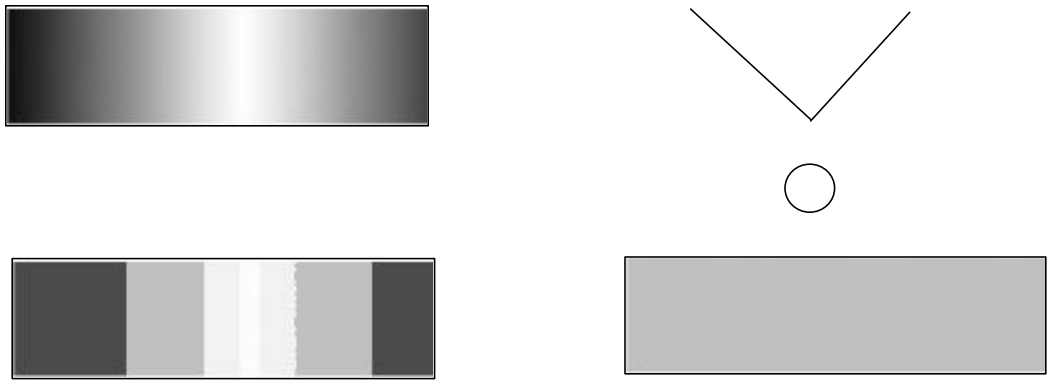

Figure 2-2. Proceeding counterclockwise from the upper right, we see the situation of a robot approaching a corner. In the diagram at upper left, we see how a robot would detect the corner with ideal sensors. In the diagram just below that, we see how the robot would observe the corner, when forced to quantize the inputs into 4 levels. Then at the lower right, we have gone one step further and reduced the number of sensors from many to only 2, where each sensor gives the average reading over its detection area. It is now unclear what, if anything, lies ahead of the robot.

Another way of thinking exists about POMDP algorithms in which a probabilistic estimate is made of the state an agent occupies. The drawback of this type of technique is that it is computationally very expensive, and reliant on problem information regarding the exact number of hidden states [16]. In addition, Whitehead and Ballard [25] suggest a method of accessing additional sensory data, however in practice their method is somewhat impractical since typically an agent is either already using all of its sensors, or is computationally unable to handle the extra data.

The control policy followed by algorithms when solving POMDPs affects the probabilities that an agent will occupy a particular state [18]. This problem is not present

in Q-learning, where the control policy followed by the agent does not affect eventual convergence. However, if all states in the POMDP are persistently exciting, then it is possible to establish convergence. In this case, the Q-learning algorithm becomes a function of the Q-values of the underlying states, combined with their probability of occupancy, given a particular observation:

$$\forall x \in X,$$
$$Q(x,a) = \sum_{s \in S} P^p(s \mid x,a)[R^a(s) + g \sum_{x' \in X} Pa(s,x') \max_{a' \in A} Q(x',a')] \qquad (2\text{-}24)$$

The meaning one can draw here is that the learned value for a state is the summed value of each of the underlying states, weighted by their probabilities. As stated earlier, treating the agent's sensory inputs as state values generally adds hidden states. Since any particular state is defined by a particular range of sensor values, a good approximation of the learned value function could conceivably have a better estimation for the values of the hidden states than the overall learned value for the observable state, assuming that the true value function which covers all states is well-behaved.

## 2.10 Approximation after Learning is Complete

Most work in approximating value functions has focused on maintaining an approximation of the function during learning, and using this approximation for control. For examples, refer to [27, 28, 29]. While some positive results have been reported, all such work has been performed on fully observable MDPs. No results have been reported where sensor values have been mapped to states, introducing some POMDP character to the problem. In order to avoid treading on theoretically shaky ground, I have decided to perform Q-learning on this problem, since it has been shown to at least converge in the mean on this type of problem [7]. Performing the approximation after the learning is

complete avoids the theoretical problem generated by altering the state occupancy probabilities.

## 2.11 Locally Weighted Regression for Approximation

Locally weighted regression (LWR) is well suited for generating a smoothed line from data points having a non-uniform distribution [30, 31, 32, 28]. Kernel regression is not as well suited for problems having an irregular distribution of data points. K-nearest neighbor methods are also unsuited ($K > 1$), due to the irregularity in the data density in the Q-table. Neural networks, although having produced good results in some applications [21], are not expected to function well here, approximating a Q-function, which varies so widely with comparatively few data points.

LWR has been successfully used in a variety of RL applications. In general, LWR is used to create a local model of a function around a query point. The particular advantage of using only a local model is that only simple linear functions are required to approximate the global function in the neighborhood of the query point. This avoids the often difficult task of generating a global model of a nonlinear function. In regards to RL applications, LWR has been found quite useful for problems in which a value function used for control is required to be continuous, but only discrete samples are available to estimate the value function.

Forbes and Andre [28] obtained good results using LWR in an automobile lane-centering task, along with a form of prioritized sweeping. Moore, Schneider and Deng authored a paper covering some algorithmic enhancements to LWR [33]. Nikovski [26] compared LWR favorably to other techniques regarding a problem of estimating the distance to a door. Tadepalli and Ok use local linear regression to improve the

performance of H-learning [34]. Atkeson, Moore, and Schaal [35] have a number of practical results in using LWR in billiards and devil-sticking, showing that LWR is well-suited for control applications that demand fine precision.

# CHAPTER 3
# SIMULATOR DESIGN AND IMPLEMENTATION

## 3.1 Choice of Physical Platform

A TJ Pro robot was chosen as the platform to be simulated. The TJ Pro has been used in our lab in several previous RL experiments [36], and it has proved a versatile, yet simple system. The small number of sensors and actuators lends it to easy simulation, while the robot itself has shown itself perfectly capable as a physical RL platform [36, 37].

## 3.2 Description of the Platform

A TJ Pro robot, as seen in Figure 3-1 is circular, roughly 17 cm in diameter and about 8 cm tall. It has two 40 KHz modulated IR emitters and detectors in front. Collision detection is accomplished by a bump sensor composed of 4 microswitches (3 in front, 1 in back), and a bumper that rides over the switches. Two gearhead motors mated to rubber wheels are mounted to either side of the body. The TJ Pro generally runs off a 6-pack of AA batteries, which are mounted in a holder inside the body. The brains of the robot are provided by a Motorola HC11 microcontroller and 32K of RAM. The speed at which the robot moves is controlled by PWM signals to the motors. The maximum speed with fully charged batteries is around 10 cm/sec.

The practical length of an experiment is limited by the battery lifespan of roughly a half-hour. While the RAM contents remain stable with a battery recharger in place, swapping out batteries in the middle of an experiment is an invitation to problems. One

addition to the platform, a pair of side-looking IR detectors, was made to the simulated TJ Pro. This was necessary to reduce the amount of chattering in the Q-values. This modification can also be performed to the physical robot with a minor amount of electronics work.

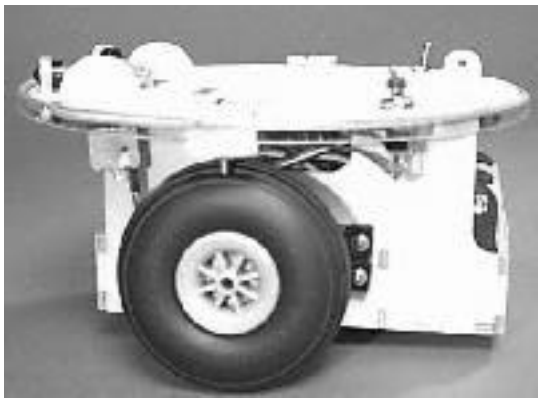

Figure 3-1. The TJ Pro

## 3.3 Simulator Considerations

A quick comparison of the predicted convergence rate with the number of table updates per second makes it obvious that the converged Q-table needed for the project could not be obtained in a practical manner on a physical platform. A simulator was needed in order to shorten the duration of the experiments. The simulation must accurately model IR sensing (for proximity). The simulation must also detect collisions of the simulated robot with obstacles and the sides of the arena. In most respects, it must also model the correct movement of the robot. Two simplifications were made with respect to the motion. First, the physical robots don't move in a perfectly straight line. Instead, one of the wheels is generally somewhat off-center, or a slightly greater diameter, or there is some other imperfection in the motors. The net result isn't a straight line, but somewhat of an arc. This motion was simplified to straight-line motion, since it

was much faster to calculate. This simplification should have no effect on the results, since other movement actions will swamp out this deviation. The other simplification deals with the behavior of the robot when it is in actual physical contact with an obstacle. When the robot contacts an obstacle at an oblique angle, it tends to turn more perpendicular during the collision due to friction-torque at the contact point. During the collision, the robot tends to slide across the face of the obstacle, sometimes turning, sometimes not—depending on whether the wheels slip. Since this behavior is somewhat difficult to model accurately, I decided to leave it out of the simulator.

The first version of the simulator was a Dos-based C++ program that I used for the evaluation of genetically optimized Q-learning parameter selection [38]. Tracking down bugs in this simulator proved problematic, because there was no convenient interface to determine how the robot was behaving. After discovering several bugs where the robot was able to escape from the arena, or enter obstacles, I decided to develop a graphical interface. I then learned how to program in a windows-based environment, and developed an application that would visually simulate the robot's motion, as well as providing a means of recording data about the simulation runs.

## 3.4 Simulation Implementation Generalities

### 3.4.1 Arena and Obstacles

A 5 meter by 5 meter arena shaped as is seen in Figure 4-1. There is no functional difference between the arena walls and the sides of the obstacles. They are simply contained in the program as a list of line segments that the robot cannot cross. Reshaping the arena is trivial, as it merely involves changing the list of line segments. The obstacle list will handle any length of 4-cornered obstacles, although increasing the number will,

of course, make the program run more slowly. Once the corners of the arena and obstacles have been specified, they are rotated 5 degrees clockwise. This is done to decrease the occurrence of infinite slopes during the IR calculation section.

### 3.4.2 Infrared Calculation

The infrared emissions of the TJ pro robot are simulated by an array of line segments that extend outwards a simulated 2 meters—somewhat beyond what will produce a detectable reflection on a real robot. The guiding principle here was 'better too long than too short'. The actual length of the IR line segments doesn't really matter in terms of processing overhead, so I drew them long, and then calibrated the return to what was observed from white objects at various distances. The manner in which IR is calculated lends itself to simulating various IR reflectivities, as well as simulating specular & diffuse reflection. Since the obstacles in the physical arena have fairly coarse surfaces, only the diffuse reflection was calculated. It would be only a simple extension of the simulation to calculate the increased IR return from a smooth perpendicular surface.

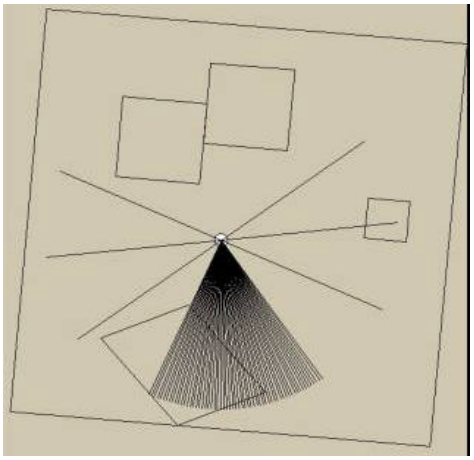

Figure 3-2. The IR lines can be seen here. Note the difference in the resolution in the side IR sensors compared to the front IR sensors. If the full resolution is needed, it is straightforward to alter the IR routines to have full resolution on all sensors.

### 3.4.3 Infrared Algorithm

Each obstacle segment is checked for an intersection with the entire IR segment list, and if an intersection point is found, the distance between obstacle and robot is stored in memory. If multiple obstacle segments intersect with the same IR segment list, then the closest intersection point is used. The simulated return value is determined by applying the inverse square law. The simulation agrees well with reality, with only a constant intensity scaling coefficient needed. (the exact value was determined during the calibration in order to match reality.) The good agreement shows that it was ok to only calculate the diffuse reflection, since the reflection return from perpendicular obstacles matched that in reality.

### 3.4.4 Collision Detection Algorithm

The robot was checked for a collision with each of the obstacle segments in the following manner. Since the robot was circular, with an equation of the form $X^2 + Y^2 = R^2$, and the segment equation was of the form $Y = m * X + b$, intersections were determined in the following manner. The two equations were set equal to each other, then a little mathematical massage put it in classic quadratic form. Upon application of the quadratic solution equation, it was straightforward to determine whether an intersection existed by seeing whether either or both roots were imaginary.

### 3.4.5 Robot Extraction Algorithm

Once the robot found itself partially inside an object, it was moved so that its edge was tangent to the obstacle. The direction of retreat was opposite to the initial direction of motion, which allows for extraction following a backwards collision. At first, the code allowed the possibility of multiple contacts occurring. However, after observing how the

program dealt with difficult situations (such as within the very acute angle in Figure 4-1), I decided to drop this portion of the algorithm since it was only used rarely in several very long runs. Instead, I had the program check for more than one collision. If more than one was found, then the robot was returned to the old location, with its bump detector triggered.

### 3.4.6 Other Simulator Features

Other interesting program features, more completely described in the users guide accompanying the simulator:

- It is possible to pause the simulation and place the robot elsewhere in the arena – useful while testing how the robot behaves while approaching corners, for instance.
- The simulation speed can be slowed down to a snails pace, if so desired.
- The visible IR lines can be toggled off to increase speed and visual appeal.
- Lastly, a filename progression feature enabling the use of copies of the program on multiple computers and have them write back their results to a central location w/o having to worry about overwriting the data from the previous runs.

## 3.5 Ease of Porting Code

The code that would run on a physical robot can be ported almost without change to the simulator. The simulator framework performs the various IR & movement functions in a form little different than that used in actual robot code. The physical robot's control code performs exactly the same when on the simulator, with only the slight differences in the format of the functions dealing with the simulated arena.

# CHAPTER 4
# THE EXPERIMENTATION

## 4.1 Q-Learning Parameters

A representation of the robot arena used in most of the experimental procedures can be seen in Fig. 4-1. The arena was designed to be a complex environment, where a composite collision-avoidance behavior would be required. There were wide-open regions, narrow passages, and acute angles intended to trap the unwary robot. Such an environment is intended to duplicate the complexity of a real-world environment. A collision-avoidance behavior that stayed as far as possible from any wall or obstacle would fail to explore the confined areas, while a wall-following behavior would fail to explore the open space.

An acceptable collision-avoidance behavior would have the robot covering all of the reachable area in the arena, while colliding as little as possible (as seen in Figure 4-2). This specification rules out otherwise ideal collision-avoidance behaviors such as spinning in place, advance-retreat, etc. Determining the required reinforcement schedule involved tweaking the rewards until the desired behavior emerged. The reward values were -20 for a collision, +5 for a forward step, +2 for a turn, and +1 for backing up. The other Q-learning parameters were learning rate alpha = 0.4, and discounting factor gamma = 0.7 Also, to ensure sufficient exploration of states, a random move was selected 10% of the time, while a greedy (maximum value) selection was made for the remaining 90% of the actions.

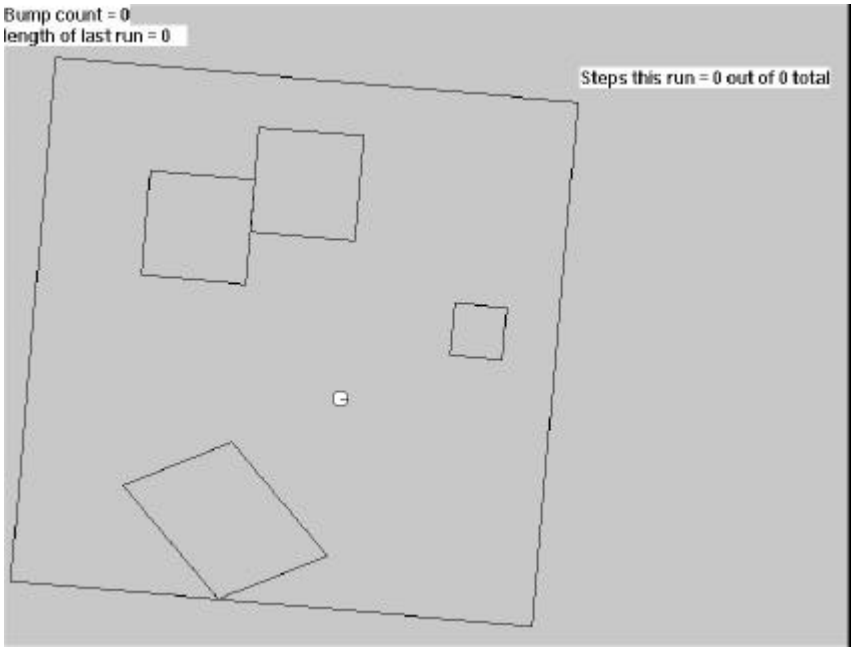


Figure 4-1. The main robot arena.

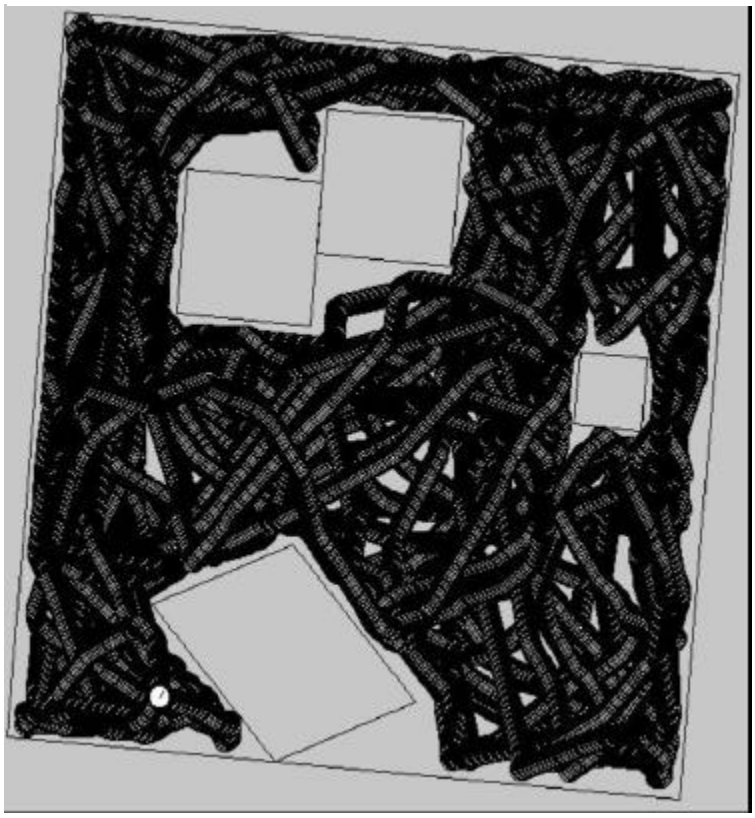

Figure 4-2. A picture showing the area covered by the simulated robot over 300,000 steps. Note the unpopular areas near the upper left and bottom center.

At first, attempting to avoid generating the backup-forward behavior, the reward for backing up was set to be negative. However, though the generated behavior appeared to be acceptable at first, the collision-avoidance performance ceased to improve noticeably as the experiment wore on. In general, it was noticed that reinforcement schedules that had penalties for any particular action resulted in quick collision-avoidance results, the long-term performance suffered in comparison to the schedules with all-positive rewards. This is likely due to the low Q-values being unrepresentative of the true utility of the actions, and prematurely suppressing further exploration.

## 4.2 Q-Table Generation

Once a reinforcement schedule was found that generated an acceptable behavior, the simulator was run for as long as reasonably possible to generate a Q-table that had approached convergence. Over a 10-day run, the simulator performed 650 million updates, saving the Q-table every 50,000 steps. When the graph of the absolute value sum of the Q-table states was plotted, it appears as if the Q-values had come reasonably close to convergence. Because of the chattering of the Q-values, the Q-tables of the 10 last saves were averaged together to obtain a representative Q-table.

As can be seen in Figure 4-3, the Q-table sum does not smoothly approach convergence, but instead chatters. Gordon has described this state-value behavior [13], and its genesis lies in the problem of perceptual aliasing, where two different robot positions are both determined to be the same state. Ordinarily, this wouldn't be a problem, since the same action would be appropriate for both positions. However,

perceptual aliasing causes difficulties when different actions would be appropriate. See Chapter 5 for discussion and illustration of this issue.

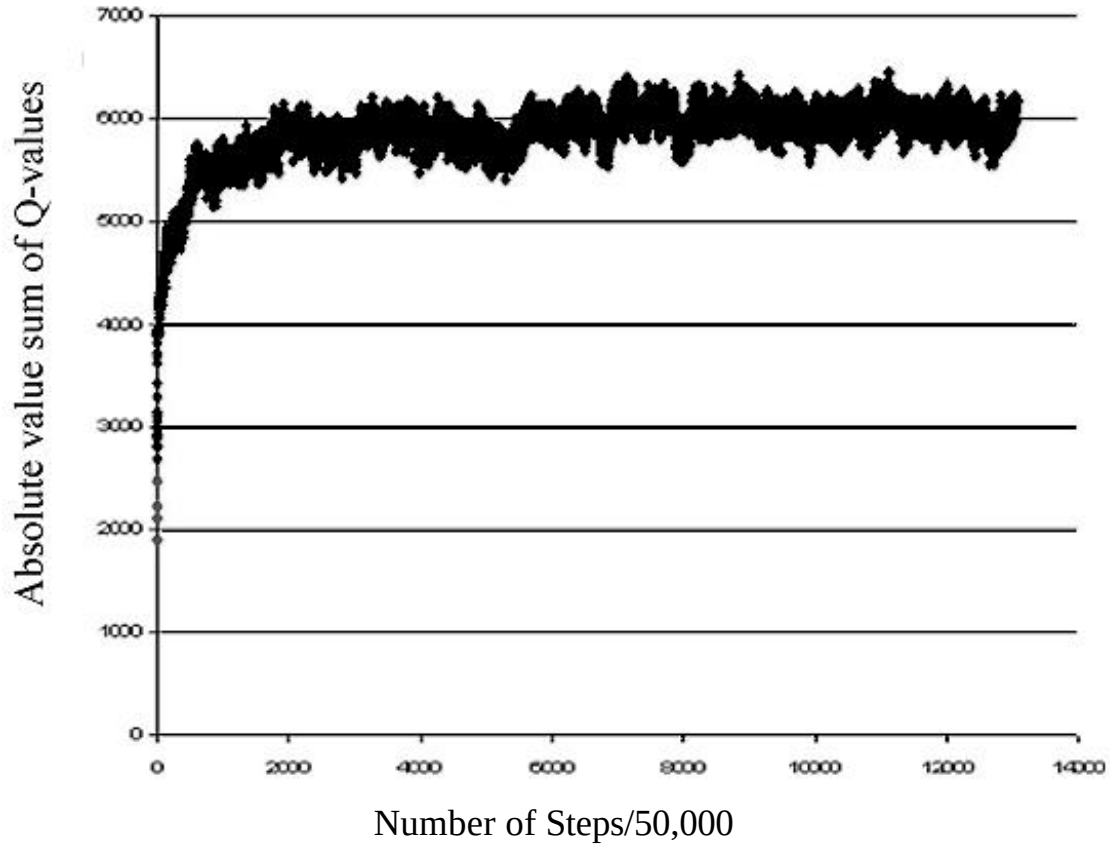


Figure 4-3. Graph of Q-value convergence, showing the absolute Q-value sum over all states.

Once the Q-table was generated, it was necessary to convert the quantized input ranges to specific values so that they could be used as sample points for function approximation. Instead of merely taking the middle of the state, vector quantization was performed. This was accomplished by taking the mean of the actual input value for each state over a large number of steps. This accounted for the possibility that the actual average of the inputs might be displaced from the middle of the quantized ranges. See section 5.2 for a more detailed analysis of the vector quantization step.

## 4.3 Approximating the Value Function with a Neural Network

The first step in extending the collision-avoidance behavior with continuous-valued inputs was to implement a neural network that approximated a value function from the Q-table data points. The neural net used was modified from the freely available NevProp v1.16 neural network simulator (authored by Phil Goodman, David Rosen and Allen Plummer). However, despite extensive effort, a useable approximation could not be made. Even the best performance was only marginally better than having no controller at all. See section 5.4 for further discussion on why the neural network failed.

## 4.4 Approximating the Value Function with Locally Weighted Regression

The next step was to approximate the value function with locally weighted regression (LWR). Because of the paucity of the sample points and the roughness of the value function, it was unnecessary to include values distant from the query point in the regression calculation. Instead, only the nearest 10 points were used, and various distance-weighting formulas were tested for performance. Figure 4-4 shows a graph of the collision-avoidance performance plotted against the weighting scheme.

The experimental runs were evaluated on a 'time until task failure' basis, i.e., by measuring the number of steps to the first collision. For comparison, the best neural network controller had an average run to first collision of roughly several hundred steps before a wall was contacted (which basically amounts to the distance to the nearest obstacle), and a random controller had a run of several thousand steps before it drifted into a wall. The increased run length of the random controller is primarily attributable to the fact that, on average, the actions largely cancel each other out, leaving the robot in

roughly the same location. An analogous example would be the movement of a dust speck in a glass of water because of Brownian motion.

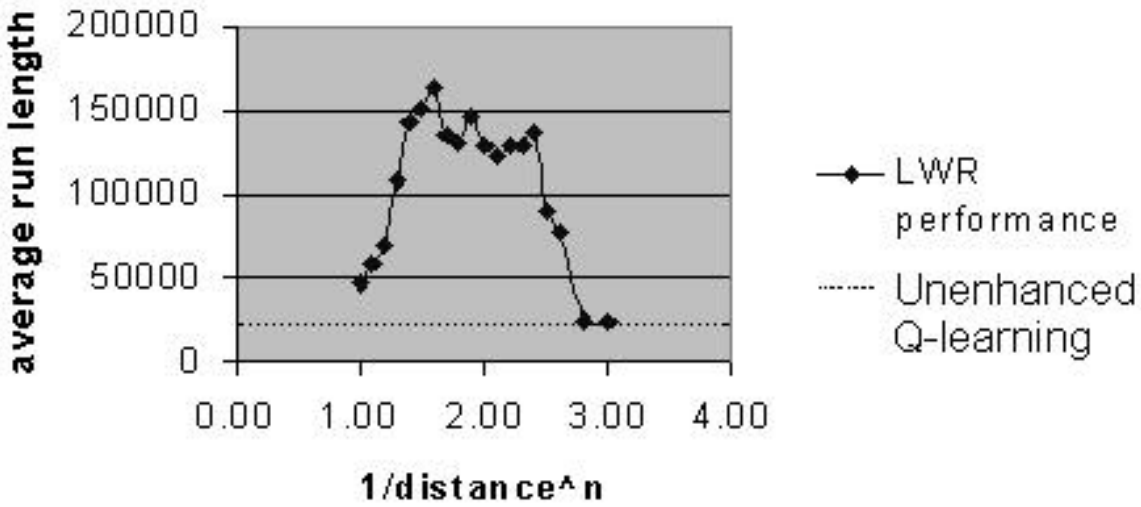


Figure 4-4. Performance of LWR compared to standard Q-learning for the arena shown in Figure 4-1

The runs were carried out on multiple computers to generate as many samples of runs as reasonably possible, with a minimum of 300 samples. Each set of runs had a different weighting function. The functions were all of the form $1/\text{distance}^n$, where n could be any positive real number, though the range investigated was from 1.0 to 3.0. Values of n greater than 3 increasingly reproduced the quantized Q-values of the original table, and thus the identical action-selection. Values of n less than one resulted in rapid performance decrease, for reasons that will be discussed in section 5.6.

## 4.5 Other Weighting Functions

Other distance-weighting functions were not systematically investigated, though spot-checks of other types of functions showed that good object-avoidance performance depended on weights becoming extremely large near Q-table data points, as well as not

having broad shape such as is found with a Gaussian distribution. This is somewhat contrary to what is reported by Atkeson et al. [32], who found that the choice of the weighting function was not generally critical to performance, though did note the existence of some exceptions. The high degree of roughness in the value function seems to demand a narrow weighting function in order to diminish the influence of irrelevant data distant from the query point.

## 4.6 Operation in Different Environments

Once experimentation had showed that the LWR improves the collision-avoidance performance for a specific environment, the next step was to examine whether the learned value function was specific to that environment. Such would be the case for RL algorithms that use a state-sequence memory to distinguish states that would otherwise appear identical. If the Q-learning step has learned a good memoryless policy, then this policy, though probably not perfectly adapted for other environments, will still usefully serve as a controller without having to re-do the computationally expensive Q-learning step.

As can be seen in Figure 4-5, the controller performed quite well in comparison to the original performance in the learned environment. In fact, it actually performed better. The reasons why will be examined in Chapter 5, but what the explanation boils down to is that the original environment had more of a problem with hard-to-detect obstacle corners. Furthermore, using continuous-valued inputs in concert with the LWR-generated approximation of the value function resulted in substantial improvement in every case, showing that this method has practical utility in the task of using discretized Q-table information to control a continuous-valued problem.

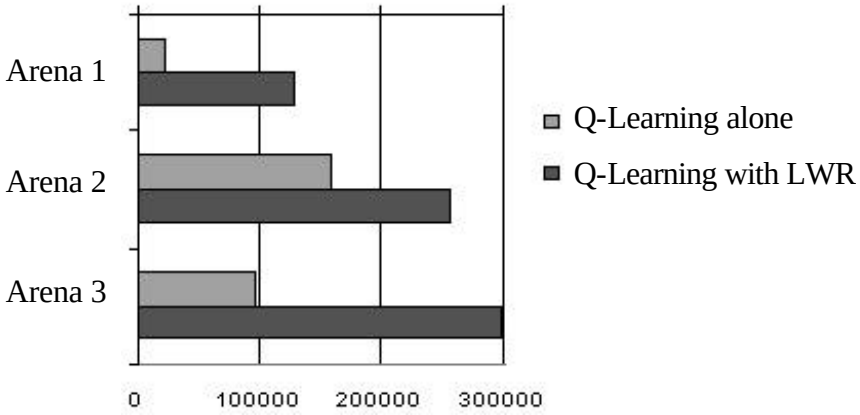


Figure 4-5. A comparison of the collision avoidance performance in different environments. Arena 1 is the arena from Figure 4-1, arena 2 is shown in Figure 4-6, and Arena 3 is shown in Figure 4-7. In all three environments the use of LWR significantly enhanced the performance.

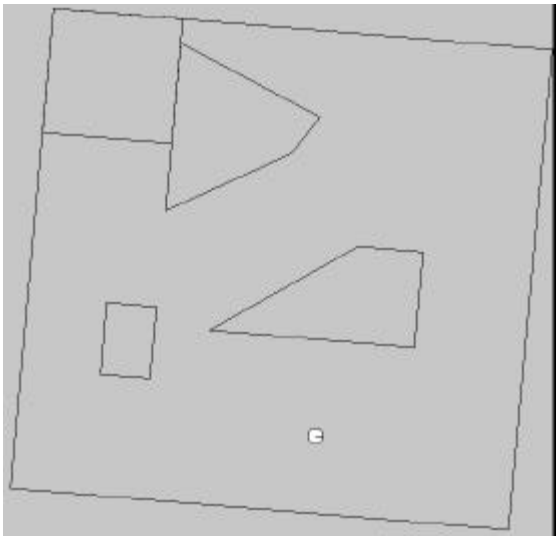

Figure 4-6. Arrangement of the second arena.

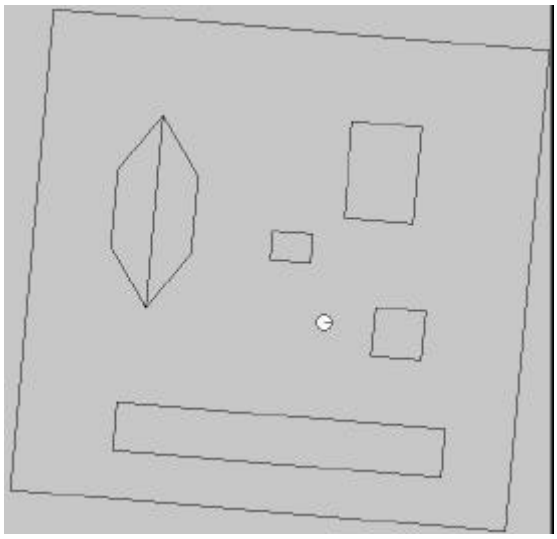

Figure 4-7. Arrangement of the third arena.

# CHAPTER 5
# DISCUSSION OF RESULTS

## 5.1 Predicted vs. Observed Q-table Convergence

According to the formula derived by Kearns and Singh [8], with well-mixed transition probabilities, the Q-values for this experimental setup should approach within a factor of .01 of the true values within about 6 million steps. Referring to Figure 4-2, it is apparent that convergence proceeds somewhat more slowly (6 million steps corresponds to 120 on the X-axis). Indeed, it takes nearly 100 million steps (2000 on the X-axis) before the Q-table sum visibly levels off. This extended convergence length seems reasonable, considering that some of the robots states are much less likely to be occupied than others (for instance, an IR return of 0 on all sensors while the bump detector is triggered). The chattering of the Q-table values does not seem to noticeably impact the rate of convergence, since the chattering occurs on a much faster time scale than the convergence rate.

## 5.2 Vector Quantization

Vector quantization describes a process in which a state space is partitioned into regions, each defined by a representative vector. The representative vectors are chosen to minimize the total mean squared distance between the sample points and the nearest vector. Vector quantization is generally an iterative process, starting with division of the state space into arbitrary regions, calculating the centroid of the data distribution in each region, and then redefining the regions based on the location of the nearest centroid.

Figure 5-1 shows a random distribution of data over a 2-dimensional state space, along with an arbitrary estimate of the region borders and centroids of those regions (marked by black dots). The region borders are defined by being equidistant from the two nearest centroids. The next step, seen in Figure 5-2, is the recalculation of the centroid of each region, and the region boundaries are redrawn in accordance with the changed centroid positions. Since the data distribution in this example was random, we see little change in the original centroids and boundaries.

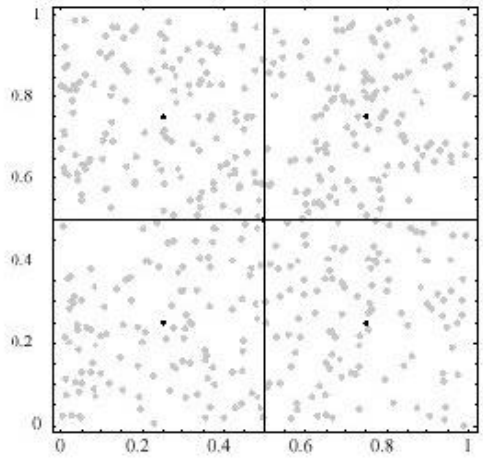


Figure 5-1. A random distribution of points over two dimensions, with the estimated centroid of each region shown as a black dot.

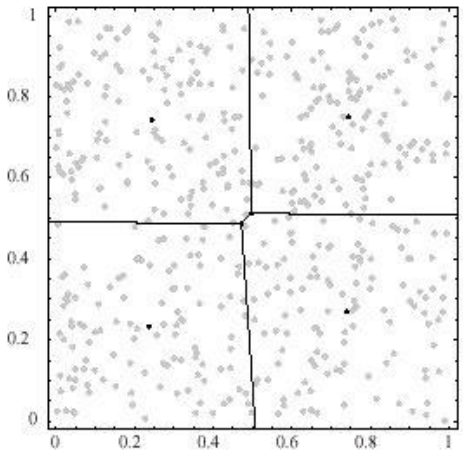


Figure 5-2. The centroids of the regions in Fig. 5-1 have been calculated, and the region boundaries redrawn in accordance.

In the next example, however, the data distribution is not random. As can be seen in Figure 5-3, the original guesses regarding the location of the centroids is wildly inaccurate, and the recalculation of the centroids results in substantial change in the position of the boundaries (Figure 5-4.)

The Q-values obtained during the Q-learning phase were action values for each input range that defined a state. In order to obtain data points from the Q-table, it was necessary to condense the state-defining input ranges into discrete points. A difficulty was found however, in that simply taking the mean of the input range resulted in the regression performance being worse than expected. It became apparent that the distribution of the (continuous-valued) inputs was non-uniform, resembling Figure 5-3 more than Figure 5-1.

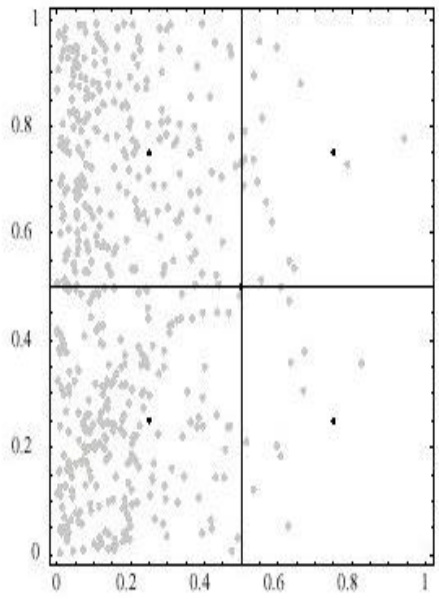


Figure 5-3. A starting estimate of the regions.

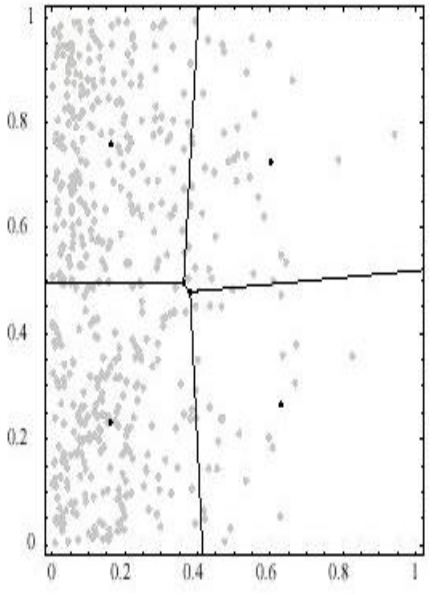


Figure 5-4. The four centroids have been recalculated, and the boundaries moved in accordance.

For example, the input range of the left front IR sensor that corresponded to the lowest IR level extended over a range of 0 to 8. When the robot was roaming in wide-open areas, the input value of 0 was very common, which meant that the Q-value corresponding to the state was based more on the actions taken at the low end of the input

range. When the weighted average was used, instead of the median of the input range, it shifted the average from 4 towards the lower end of the range.

This simple step performed one step of the vector quantization procedure, that of recalculating the centroids. Performing multiple iterations would have involved determining the new Q-value for the altered state, which would have been too computationally expensive. Still, even one iteration was enough to markedly improve the performance in combination with the LWR.

## 5.3 Perceptual Aliasing

As stated earlier in this thesis, pretending that sensory input values define discrete states and then using MDP solution methods opens oneself to problems. The problems lie in the fact that identical input values are in reality different states that might or might not demand different actions.

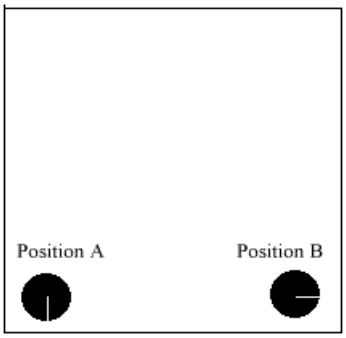


Figure 5-5. Same inputs, same action

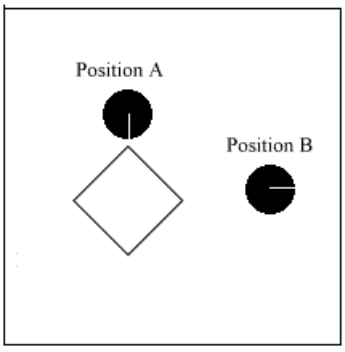


Figure 5-6. Same inputs, different action

Figures 5-5 and 5-6 both illustrate situations where the robots at positions A and B have identical inputs, and therefore believe themselves to be in identical states. In Figure 5-5, this doesn't matter, since the same action is obviously appropriate – a turn to the left. In Figure 5-6 however, different actions are appropriate. The robot at position B can

easily proceed forward, however, the robot at position A cannot. In fact, the robot at position A is likely to collide with the corner, as this situation (approaching a corner so that the IR returns are balanced) is much less likely to occur than that for position B. Therefore the Q-values of the state will be heavily biased towards the expected reward from actions taken from position B versus those actions taken from position A. There is also no guarantee that the learned Q-value will be stable, since as Gordon showed [13], perceptual aliasing can lead to the indefinite oscillation or chattering of the Q-values associated with the indistinguishable states. In the example above, the average learned Q-value will primarily reflect experiences taken from position B, however the Q-value will never quite converge to what it would be if all experiences were only from position B (and none from A). Every time position A is encountered, the learned Q-value will hiccup away from the value learned for B, so therefore the Q-values over a long period of time must be averaged in order to obtain a true estimate for the observed state.

Figures 5-7, 5-8, and 5-9 show that this is exactly what happened during the simulation. These figures show the position for every collision during a long run. Note

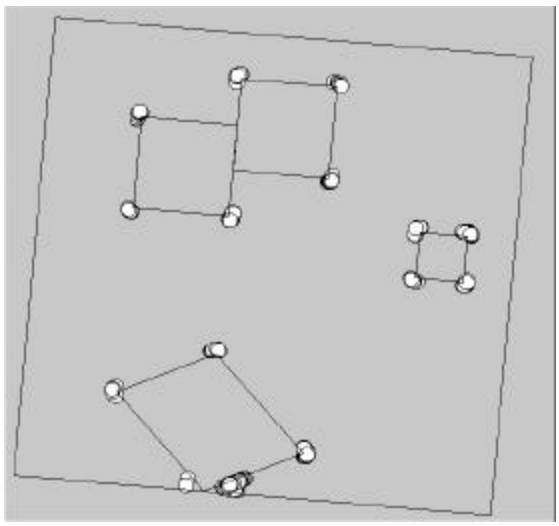

Figure 5-7. Collisions in the first arena

that the only collisions are against corners—exactly what would be expected given the reasoning above.

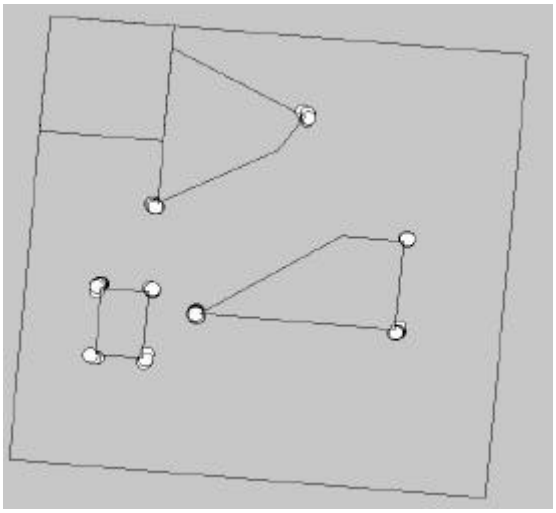

Figure 5-8. Collisions in the second arena.

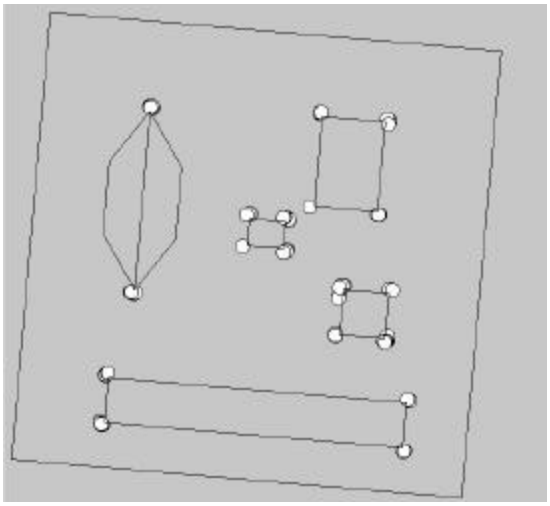

Figure 5-9. Collisions in the third arena.

## 5.4 Neural Networks Cannot Map this Value Function

The size of the neural network that will approximate a function defined by sample points is related to the complexity of the function. A figure that is often used as a rule of thumb when determining the maximum number of hidden units that will not result in overfitting is that there must be 10 sample points for every degree of freedom (weight) within the network [39]. With fewer sample points, or more hidden units, the likelihood of overfitting increases. Even if the neural network successfully reproduces the sample points, it will generally overfit and have an invalid approximation between the sample points.

The problem that is faced when using a neural network to approximate this Q-function is that too few sample points exist to define any function that is not relatively smooth. A cursory inspection of the Q-values shows that they are samples from an extremely jagged function–one which is not subject to any useful approximation by a

neural network with a constrained number of weights. Despite trying a variety of neural network morphologies, it proved impossible to generate a neural network with fewer degrees of freedom than the number of sample points that would exactly reproduce the sample points. It likewise proved impossible to accomplish the less-rigorous task of selecting the correct actions over the states defined in the Q-learning process.

Faced with the failure of a reasonably-sized neural network to generate an acceptable approximation, an attempt was made with an unreasonably-sized network that had somewhat more weights than sample inputs. This network was able to reproduce the Q-values. However, when this network was used as a controller, it was, as expected, incapable of generating correct actions for any input that was not equal to the sample points used to train it. However, this was a predicted consequence, given the overfitting that was virtually assured.

## 5.5 Basis for the Performance Comparison

One might question the basis on which the various runs were compared. The 'time until task failure' was selected in order to rule out the presence of artifacts due to feature-weighting. When in a state where the bumper was depressed, it can be seen that the neighboring states with untouched bumpers have an influence on the outcome of the regression. How much of an influence depends on how the feature is weighted. Clearly, simply taking the Euclidean distance between bump = 1 and bump = 0 states invites the 'adding apples to oranges' analogy by overlooking the fact that this distance is not the same type of distance as between IR sensor points.

The easiest way to deal with this problem was to add a scaling factor to the bump distance, and then add it into the overall distance between query point and sample points.

On inspection however, this raises the question of 'what is the best scaling factor'? Rather than attempt to deal with determining the ideal scaling factor at the same time as determining the correct distance weighting factor, I decided to only determine the best distance weighting, and that meant that the behavior dealing with the aftermath of a collision stood a good chance of being poor, unless the feature scaling interacted favorably with the distance weighting formula. In an effort to compare the distance weighting functions on an even basis, it therefore proved necessary to time the runs until the first collision happened, and then repeat the procedure. Since the experiment wouldn't test the behavior of any actions after a bump, it proved convenient to consider the un-bumped states separately from the bumped states; in effect giving infinite feature weighting.

After determining the useful range of distance weighting factors, an experiment was performed to test whether the LWR controller could recover from a collision. The distance weighting equation used for this experiment was $1/d^2$, since that was the center of the range for which improvement was noted. The two conditions of 'bumped' and 'un-bumped' were considered separately, with the regression only including un-bumped states for un-bumped queries, and vice versa. Essentially, this was an infinite feature weighting of the bump value. As the experiment unfolded, the controller showed that it could indeed recover from bumping into an object without performing action sequences such as spinning in place, retreating and then advancing into a collision repeatedly, and so forth.

## 5.6 Mechanism of Performance Enhancement

Figure 5-10 shows a simplified interpretation of how LRW smoothes the values along the boundary between two states B and C. As shown in Figure 5-10 by the curved lines, the LWR approximation can affect the action selection (on the basis of highest action value) near the boundaries of the Q-table states.

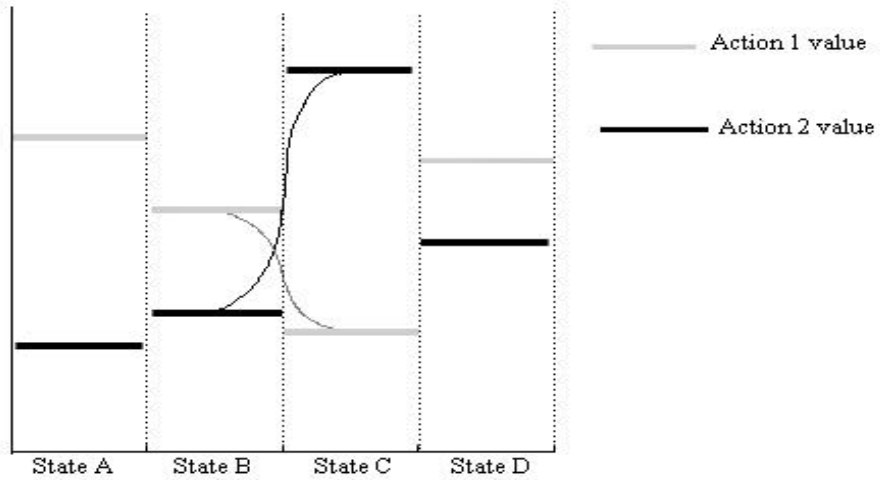


Figure 5-10. Boundary smoothing

When the collision points of all three arenas are compared, it can be seen that all collisions were against corners, with more acutely angled corners having more collisions, and obtusely angled corners having much fewer collisions. The self-evident statement that could therefore be made is that any change in action selection which resulted in improved performance would result in fewer collisions against corners.

Using Figure 5-10 as an analogy, let us say that State B represents the condition of a robot nearing a corner. The robot has no idea that it is getting dangerously close to a corner, as the inputs which it receives mimic the condition of being a safe distance from a perpendicular wall. The state-action values learned for State C represent the composite

conditions of the robot colliding with a corner, as well as the robot coming somewhat close to a perpendicular wall. In States B and C, Action 1 would be to go forward, and Action 2 would be to turn away.

When a discrete-state action selection method is used in this example, and the inputs are in a range belonging to State B, it can be seen that action 1 is selected. By the time the input values have risen to the threshold of State C, it is too late to turn away—a collision has already occurred. However, if LWR had been used to smooth the values along the boundary between the states, the robot would have "known" it was approaching an undesirable state as the input values neared the state boundary, and turned away before a collision occurred.

Since the value function was presumed to be a well-behaved function (i.e., no discontinuities, etc..) smoothing the state boundaries would reproduce the value function more exactly. Unfortunately, there was no a priori method to determine how much smoothing was too much, so it seemed wisest to start with little smoothing (by a large distance-weighting factor), and then slowly decrease the weighting factor until the collision-avoidance performance began to decline again—which implied that there was now too much smoothing.

# CHAPTER 6
# CONCLUSIONS AND FURTHER DIRECTIONS

## 6.1 Summary of Contributions

This thesis demonstrated the use of an MDP algorithm, Q-learning, to a POMDP problem in which a good memory-less policy was determined to exist. The performance of this algorithm was enhanced after the completion of learning by the generation of a value function over a continuous state-space via LWR. The significance of this method lies in that a procedure was demonstrated which allows a continuous-valued problem to be approached as a discrete-state problem in order to minimize the computational complexity without sacrificing all of the possible accuracy obtainable with additional state information. The benefits to this method are straightforward—it becomes possible to attempt the characterization of complex continuous-state problems that were previously immune to everything but theoretical massage.

## 6.2 Future Directions

The next step to be taken with this method is to apply the function approximation for control during learning. Though convergence has not yet been proven, the possibility exists that good results could be obtained experimentally on this class of problem. Gordon [40] has done some investigation of online fitted approximations, but his results, while encouraging, show that large classes of approximators, including LWR, are demonstratively divergent under some situations. This is somewhat discouraging, as LWR seems to be the most fitting type of approximator for this task.

Another method that could be used to improve performance is to perform multiple learning episodes, with intervening recalculation of the state classification boundaries via vector quantization. This would help compensate for the fact that the researcher-chosen boundaries are almost never at the optimal locations. Krose and Dam have performed some related work in this area, using a self-organizing input state quantization procedure that obtained good results [41].

REFERENCES


[1] L. P. Kaebling, M. L. Littman, and A.W. Moore, “Reinforcement learning: A Survey,” *Journal of Artificial Intelligence Research*, vol. 4, pp. 237-285, 1996.

[2] R. Bellman, *Dynamic Programming.* Princeton, New Jersey: Princeton University Press, 1957.

[3] P. Cichosz, *Reinforcement Learning Algorithms Based on the Methods of Temporal Differences*. Master's Thesis, Institute of Computer Science, Warsaw University of Technology, 1994.

[4] C. J. C. H. Watkins, *Learning from Delayed Rewards.* Ph.D. Thesis, King's College, Cambridge, 1989.

[5] C. J. C. H. Watkins and P. Dayan, “Technical Note: Q-Learning,” *Machine Learning* vol. 8, pp. 279-292, 1992.

[6] M. B. Ring, *Continual Learning in Reinforcement Environments*. PhD thesis, University of Texas at Austin, 1994.

[7] T. Jaakkola, M. I. Jordan, and S. P. Singh. “On the Convergence of Stochastic Iterative Dynamic Programming Algorithms,” *Neural Computation*, 6(6): pp. 1185-1201, 1994.

[8] M. Kearns and S. Singh, “Finite-sample Convergence Rates for Q-learning and Indirect Algorithms,” in *Advances in Neural Information Processing Systems 12*, M. Kearns, S. A. Solla, and D. Cohn, Eds., Cambridge, MA: MIT Press, 1999, pp 996-1002.

[9] A. G. Barto, S. J. Bradtke, and S. P. Singh, “Learning to Act Using Real-time Dynamic Programming,” *Artificial Intelligence,* vol. 72, (1):81-138, 1995.

[10] G. A. Rummery and M. Niranjan, “On-line Q-learning Using Connectionist Systems,” Cambridge University Engineering Dept., Technical Report CUED/F-INFENG/TR 166, 1994.

[11] S. Singh, T. Jaakkola, M. L. Littman, and C. Szpesvari, "Convergence Results for Single-Step On-policy Reinforcement-learning Algorithms," Univ. Col., Dept. Comp. Sci., Boulder, CO, Tech. Rep., 1998.

[12] M. L. Littman. (1996). Combining Exploration and Control in Reinforcement Learning: The Convergence of SARSA. Available: <http://www.cs.duke.edu/~mlittman>.

[13] G. J. Gordon, "Chattering in SARSA()," CMU Learning Lab, Pittsburgh, PA, Internal Report, 1996.

[14] M. Wiering and J. Schmidhuber, "HQ-learning," *Adaptive Behavior*, vol. 6.2, pp. 219-246, 1998.

[15] L. J. Lin and Tom Mitchell, "Memory Approaches to Reinforcement Learning in Nonmarkovian Domains," Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CS-92-138, 1992.

[16] L. Chrisman, "Reinforcement Learning with Perceptual Aliasing: The Perceptual Distinctions Approach," in *Proceedings of the Tenth International Conference on Artificial Intelligence*, 1992, pp. 183-188.

[17] J. Loch and S. Singh, "Using Eligibility Traces to Find the Best Memoryless Policy in Partially Observable Markov Decision Processes," in *Proc. ICML.*, 1998, pp. 141-150.

[18] M. I. Jordan, S.P. Singh, T. Jaakkola, "Learning Without State-estimation in Partially Observable Markovian Decision Processes," in *Proceedings of the Eleventh Machine Learning Workshop*, 1994, pp. 284-292.

[19] C. Szepesvári and M. L. Littman, "A Unified Analysis of Value-function-based Reinforcement-learning Algorithms," *Neural Computation*, 11:8, pp. 2017-2059, 1999.

[20] S. Singh, T. Jaakkola, and M. I. Jordan, "Reinforcement Learning with Soft State Aggregation," *Advances in Neural Information Processing Systems*, G. Tesauro and D. Touretzky, Eds., 1995, vol. 7, pp 361-368.

[21] G. Tesauro, "Practical Issues in Temporal Difference Learning," *Machine Learning*, vol. 8, pp. 257–277, 1992.

[22] G. A. Rummery, *Problem Solving With Reinforcement Learning*. PhD thesis, University of Cambridge, 1995.

[23] A. Linden. (1993). On Discontinuous Q-functions in Reinforcement Learning. Available: anonymous ftp from <archive.cis.ohio-state.edu> in directory /pub/neuroprose.

[24] J. N. Tsitsiklis and B. Van Roy, "An Analysis of Temporal-difference Learning with Function Approximation," M.I.T., Cambridge, MA, Tech. Rep. LIDS-P-2322, 1996.

[25] S. D. Whitehead and D. H. Ballard, "Learning to Perceive and Act by Trial and Error," *Machine Learning*, vol. 7, pp. 45-83, 1991.

[26] D. Nikovski, "Visual Memory-based Learning for Mobile Robot Navigation," in *Proceedings of the Second International Conference on Computational Intelligence and Neurosciences*, 1997, vol. 2, pp.1-4.

[27] J. Boyan and A. Moore, "Generalization in Reinforcement Learning: Safely Approximating the Value Function," in *Advances in Neural Information Processing Systems 7*, 1995, pp. 369-376.

[28] J. Forbes and D. Andre, "Practical Reinforcement Learning in Continuous Domains," Computer Science Division, University of California, Berkeley, Tech. Rep. UCB/CSD-00-1109, 2000.

[29] P. Sabes, "Approximating Q-values with Basis Function Representations," presented at the Fourth Connectionist Models Summer School, Hillsdale, NJ, 1993.

[30] Cleveland, W.S. and Loader, C. "Smoothing by Local Regression: Principles and Methods" (with discussion). in *Statistical Theory and Computational Aspects of Smoothing*, W. Hardle and M.G. Schimek, Eds. Heidelberg: Physica-Verlag, 1996, pp. 10-49; pp. 80-102; pp. 113-120.

[31] T. Hastie and C. Loader, "Local Regression: Automatic Kernel Carpentry," *Statistical Science*, vol. 8, pp.120-143, 1993.

[32] C. G. Atkeson, A. W. Moore, and S. Schaal, "Locally Weighted Learning," *Artificial Intelligence Review*, 11:(1-5), pp. 11-73, 1997.

[33] A. Moore, J. Schneider, and K. Deng, "Efficient Locally Weighted Polynomial Regression Predictions," in *Proceedings of the 14th International Conference on Machine Learning*, D. Fisher, Ed., 1997, pp. 236-244.

[34] P. Tadepalli and D. Ok, "Scaling up Average Reward Reinforcement Learning by Approximating the Domain Models and the Value Function," in *Proc. 13th Int. Conf. on Machine Learning*, 1996, pp. 471-479.

[35] C. G. Atkeson, A. W. Moore, and S. Schaal, “Locally Weighted Learning for Control,” *Artificial Intelligence Review Special Issue on Lazy Learning Algorithms*, 11:75-113, 1997.

[36] D. Bagnell, K. Doty, and A. Arroyo, "Comparison of Reinforcement Learning Techniques for Automatic Behavior Programming," in *AAAI 98 POMDP*, Oct. 1998, pp. 1-6.

[37] S. D. Jantz, K. L. Doty, J. A. Bagnell and I. R. Zapata, “Kinetics of Robotics: The Development of Universal Metrics in Robot Swarms,” presented at the Florida Conference on Recent Advances in Robotics, Miami, FL., 1997.

[38] H. Aljibury and A. Antonio Arroyo, “Creating Q-table Parameters Using Genetic Algorithms,” presented at the Florida Conference on Recent Advances in Robotics, Gainesville, Florida, 1999.

[39] L. Fu, *Neural Networks in Computer Intelligence*. 1st ed., New York, NY: McGraw-Hill, Inc., 1994.

[40] G. J. Gordon, *Approximate Solutions to Markov Decision Processes*. Ph.D. Thesis, Carnegie Mellon University, 1999.

[41] B. J. Krose and J.W. van Dam, “Adaptive State Space Quantisation for Reinforcement Learning of Collision-free Navigation,” in *Proc. of the IEEE Int. Workshop on Intelligent Robots and Systems '92*, 1992, pp. 9-14.

# APPENDIX

A copy of the simulator used in this thesis (and user's guide) can be obtained from <http://mil.ufl.edu/publications>. The simulator was compiled under Visual C++ 6.0, and is known to work under Windows 95 and 98. It will probably work on other versions of Windows, but this hasn't been tested. The source code will also probably compile under other windows compilers, but supplementary code libraries might be needed. The user's guide contains instructions on how to set up the files, and how to edit the simulator code in order to adapt it to your own uses.

## BIOGRAPHICAL SKETCH

Halim Aljibury was born in Fresno, CA in 1974, and attended Dartmouth College and the University of California at San Diego as an undergraduate. He received a Bachelor of Science degree in biochemistry from UCSD in 1996, and then switched career paths by becoming an electrical engineering grad student at the University of Florida, where he has been active with the Machine Intelligence Laboratory since 1997.